The LEDA User Manual

Version 4.1

Kurt Mehlhorn
Max-Planck-Institut für Informatik,
66123 Saarbrücken, Germany

Stefan Näher
Fachbereich Mathematik und Informatik
Martin-Luther Universität Halle-Wittenberg
06099 Halle, Germany

Michael Seel
Max-Planck-Institut für Informatik,
66123 Saarbrücken, Germany

Christian Uhrig
Algorithmic Solutions Software GmbH,
Postfach 151101
66041 Saarbrücken, Germany

Contents

Acknowledgement

Many people contributed to this new version of LEDA. Their names are listed in the *Acknowledgements* file in the LEDA root directory. The authors want to thank all of them for their bug reports, helpful suggestions and valuable comments.

License Terms and Availability

The LEDA research version can be used free of charge for academic research and teaching. However, it is not in the public domain and its use requires a *research license* which is distributed by the Max-Planck-Institut für Informatik in Saarbrücken. Please visit the LEDA homepage

`http://www.mpi-sb.mpg.de/LEDA`

and go to the *Download* section for downloading LEDA and for information on how to obtain a research license. Any commercial use of LEDA requires a *commercial license*

which is distributed by Algorithmic Solutions Software GmbH, Postfach 151101, 66041 Saarbrücken, (fax: +49 681 842502, email: contact@algorithmic-solutions.com)

Any use by a company, government institution, or any other non-academic institution is considered to be commercial use.
It is not allowed to use or distribute any code being part of or based on the research version of LEDA commercially.

Chapter 0

Preface

One of the major differences between combinatorial computing and other areas of computing such as statistics, numerical analysis and linear programming is the use of complex data types. Whilst the built-in types, such as integers, reals, vectors, and matrices, usually suffice in the other areas, combinatorial computing relies heavily on types like stacks, queues, dictionaries, sequences, sorted sequences, priority queues, graphs, points, segments, ... In the fall of 1988, we started a project (called **LEDA** for Library of Efficient Data types and Algorithms) to build a small, but growing library of data types and algorithms in a form which allows them to be used by non-experts. We hope that the system will narrow the gap between algorithms research, teaching, and implementation. The main features of LEDA are:

1. LEDA provides a sizable collection of data types and algorithms in a form which allows them to be used by non-experts. This collection includes most of the data types and algorithms described in the text books of the area.

2. LEDA gives a precise and readable specification for each of the data types and algorithms mentioned above. The specifications are short (typically, not more than a page), general (so as to allow several implementations), and abstract (so as to hide all details of the implementation).

3. For many efficient data structures access by position is important. In LEDA, we use an item concept to cast positions into an abstract form. We mention that most of the specifications given in the LEDA manual use this concept, i.e., the concept is adequate for the description of many data types.

4. LEDA contains efficient implementations for each of the data types, e.g., Fibonacci heaps for priority queues, skip lists and dynamic perfect hashing for dictionaries, ...

5. LEDA contains a comfortable data type graph. It offers the standard iterations such as "for all nodes v of a graph G do" or "for all neighbors w of v do", it allows to add and delete vertices and edges and it offers arrays and matrices indexed by nodes and edges,... The data type graph allows to write programs for graph problems in a form close to the typical text book presentation.

6. LEDA is implemented by a C++ class library. It can be used with almost any C++ compiler that supports templates.

7. LEDA is available via www from **http://www.mpi-sb.mpg.de/LEDA**. The distribution contains all sources, installation instructions, technical reports, and the user manual.

8. LEDA is not in the public domain, but can be used freely for research and teaching. Information on a commercial license is available from the authors or leda@mpi-sb.mpg.de.

This manual contains the specifications of all data types and algorithms currently available in LEDA. Users should be familiar with the C++ programming language (see [74] or [50]).

The manual is structured as follows: In chapter 1, which is a prerequisite for all other chapters, we discuss the basic concepts and notations used in LEDA. New users of LEDA should carefully read section 1.3 to avoid problems when plugging in self defined parameter types. If you want to get information about the LEDA documentation scheme please read section 1.7. For technical information concerning the installation and usage of LEDA concerning the compilation and linkage of programs users should refer to chapter 2. There's also a section describing the prefixing scheme of LEDA and some remarks about interaction with other Software libraries. The other chapters define the data types and algorithms available in LEDA and give examples of their use. These chapters can be consulted independently from one another.

There are also other sources of information. On the LEDA web page you can get the online html version of this manual. There are preliminary chapters of the forthcoming LEDA book available and there are also documentation reports covering internal topics of LEDA and the implementation of diverse data types. Please have also a look into the directory `LEDAROOT/Manual/contrib` which contains more comprehensive information about packages like *ps_file*.

Finally there's a tool called `xlman` which allows online help and demonstration on all unix platforms having a LaTeX package installed.

New in Version 4.1

Version Macro `__LEDA__` (defined to 400 in this version)

the LEDA book – all demos and runtime tests which are documented in the LEDA book are part of this LEDA package. They are accessible via the xlman tool for documentation and demo presentation.

standard headers – setting the flag `LEDA_STD_HEADERS` will cause LEDA to use the new std header files (without ".h"). This can be done by using `-DLEDA_STD_HEADERS` on the compiler command line or by uncommenting the corresponding definition at the end of `<LEDA/system.h>`. Note that all libraries have to be compiled with this flag enabled and that old and new header files cannot be used simultaneously.

memory allocator – the LEDA memory manager is now accessible via an allocator interface according to the C++ standard allocator interface. Please refer to *leda_allocator* manual page.

new geometry – there are several new algorithms constructing so called constrained triangulations of polygons, embedded graphs and segments. There are also new types for three dimensional geometry.

geowin – there is a new editor for visulization and animation of geometric algorithms called geowin. The editor manages geometric objects in so-called scenes. Please refer to the manual pages.

Please read the CHANGES and FIXES files in the LEDA root directory for details.

Chapter 1

Basics

An extended version of this chapter is available as chapter Foundations of [58]

1.1 Getting Started

Please use your favourite text editor to create a file *prog.c* with the following program:

```
#include <LEDA/d_array.h>

main()
{
  d_array<string,int> N(0);
  string s;
  while (cin >> s) N[s]++;
  forall_defined (s,N)
    cout << s << " " << N[s] << endl;
}
```

If you followed the installation guidelines (see chapter 2.1), you can compile and link it with LEDA's basic library *libL.a* (cf. section 2.2.2). For example on a Unix machine where g++ is installed you can type

```
g++ -o prog prog.c -lL -lm
```

When executed it reads a sequence of strings from the standard input and then prints the number of occurrences of each string on the standard output. More examples of LEDA programs can be found throughout this manual.

The program above uses the parameterized data type dictionary array (*d_array*$<I,E>$) from the library. This is expressed by the include statement (cf. section 2.2.1 for more details). The specification of the data type *d_array* can be found in section 6.5. We use it also as a running example to discuss the principles underlying LEDA in the following sections.

1.2 The LEDA Manual Page (the type specification)

In general the specification of a LEDA data type consists of four parts: a definition of the set of objects comprising the (parameterized) abstract data type, a description of how to create an object of the data type, the definition of the operations available on the objects of the data type, and finally, information about the implementation. The four parts appear under the headers **definition**, **creation**, **operations**, and **implementation** respectively. Sometimes there is also a fifth part showing an **example**.

- **Definition**

  This part of the specification defines the objects (also called instances or elements) comprising the data type using standard mathematical concepts and notation.

  Example

  The generic data type dictionary array:

  An object a of type *d_array*<I,E> is an injective function from the data type I to the set of variables of data type E. The types I and E are called the index and the element type respectively, a is called a dictionary array from I to E.

  Note that the types I and E are parameters in the definition above. Any built-in, pointer, item, or user-defined class type T can be used as actual type parameter of a parameterized data type. Class types however have to provide several operations listed in chapter 1.3.

- **Creation**

  A variable of a data type is introduced by a C++ variable declaration. For all LEDA data types variables are initialized at the time of declaration. In many cases the user has to provide arguments used for the initialization of the variable. In general a declaration

  ```
  XYZ<t1, ... ,tk>  y(x1, ... ,xt);
  ```

  introduces a variable y of the data type XYZ< t1, ... ,tk > and uses the arguments x1, ... ,xt to initialize it. For example,

  ```
  h_array<string,int> A(0);
  ```

  introduces A as a dictionary array from strings to integers, and initializes A as follows: an injective function a from *string* to the set of unused variables of type *int* is constructed, and is assigned to A. Moreover, all variables in the range of a are initialized to 0. The reader may wonder how LEDA handles an array of infinite size. The solution is, of course, that only that part of A is explicitly stored which has been accessed already.

  For all data types, the assignment operator (=) is available for variables of that type. Note however that assignment is in general not a constant time operation, e.g., if $L1$ and $L2$ are variables of type *list*$< T >$ then the assignment $L1 = L2$ takes time proportional to the length of the list $L2$ times the time required for copying an object of type T.

Remark: For most of the complex data types of LEDA, e.g., dictionaries, lists, and priority queues, it is convenient to interpret a variable name as the name for an object of the data type which evolves over time by means of the operations applied to it. This is appropriate, whenever the operations on a data type only "modify" the values of variables, e.g., it is more natural to say an operation on a dictionary D modifies D than to say that it takes the old value of D, constructs a new dictionary out of it, and assigns the new value to D. Of course, both interpretations are equivalent. From this more object-oriented point of view, a variable declaration, e.g., *dictionary<string,int>* D, is creating a new dictionary object with name D rather than introducing a new variable of type *dictionary<string,int>*; hence the name "Creation" for this part of a specification.

- **Operations**

In this section the operations of the data types are described. For each operation the description consists of two parts

1. The interface of the operation is defined using the C++ function declaration syntax. In this syntax the result type of the operation (*void* if there is no result) is followed by the operation name and an argument list specifying the type of each argument. For example,

```
list_item   L.insert (E x, list_item it, int dir = LEDA::after)
```

   defines the interface of the insert operation on a list L of elements of type E (cf. section 5.7). Insert takes as arguments an element x of type E, a *list_item it* and an optional relative position argument *dir*. It returns a *list_item* as result.

```
E&  A[I x]
```

   defines the interface of the access operation on a dictionary array A. It takes an element x of type I as an argument and returns a variable of type E.

2. The effect of the operation is defined. Often the arguments have to fulfill certain preconditions. If such a condition is violated the effect of the operation is undefined. Some, but not all, of these cases result in error messages and abnormal termination of the program (see also section 3.13). For the insert operation on lists this definition reads:

   > A new item with contents x is inserted after (if $dir = LEDA::after$) or before (if $dir = LEDA::before$) item *it* into L. The new item is returned.
   > *Precondition*: item *it* must be in L.

   For the access operation on dictionary arrays the definition reads:

   > returns the variable $A(x)$.

- **Implementation**

  The implementation section lists the (default) data structures used to implement the data type and gives the time bounds for the operations and the space requirement. For example,

  Dictionary arrays are implemented by randomized search trees ([2]). Access operations $A[x]$ take time $O(\log \mathrm{dom}(A))$. The space requirement is $O(\mathrm{dom}(A))$.

1.3 User Defined Parameter Types

If a user defined class type T shall be used as actual type parameter in a container class, it has to provide the following operations:

a) a constructor taking no arguments	$T :: T()$
b) a copy constructor	$T :: T(const T\&)$
c) an assignment operator	$T\&\ T :: \textbf{operator} = (const T\&)$

and if required by the parameterized data type

d) an input operator	$istream\&\ \textbf{operator}{>>}(istream\&, T\&)$
e) an output operator	$ostream\&\ \textbf{operator}{<<}(ostream\&, const\ T\&)$
f) a compare function	$int\ \textbf{compare}(const\ T\&, const\ T\&)$
g) a hash function	$int\ \textbf{Hash}(const\ T\&)$

In the following two subsections we explain the background of the required compare and hash function. Section 1.3.3 concerns a very special parameter type, namely implementation parameters.

1.3.1 Linear Orders

Many data types, such as dictionaries, priority queues, and sorted sequences require linearly ordered parameter types. Whenever a type T is used in such a situation, e.g. in *dictionary*<T,...> the function

```
int   compare(const T&, const T&)
```

must be declared and must define a linear order on the data type T.

A binary relation *rel* on a set T is called a linear order on T if for all x,y,z in T:

1) x *rel* x
2) x *rel* y and y *rel* z implies x *rel* z
3) x *rel* y or y *rel* x
4) x *rel* y and y *rel* x implies $x = y$

A function *int* compare(const T&, const T&) defines the linear order *rel* on T if

$$\text{compare}(x, y) \quad \begin{cases} < 0, & \text{if } x \ rel \ y \text{ and } x \neq y \\ = 0, & \text{if } x = y \\ > 0, & \text{if } y \ rel \ x \text{ and } x \neq y \end{cases}$$

For each of the data types *char*, *short*, *int*, *long*, *float*, *double*, *integer*, *rational*, *bigfloat*, *real*, *string*, and *point* a function *compare* is predefined and defines the so-called default ordering on that type. The default ordering is the usual $\leq$ - order for the built-in numerical types, the lexicographic ordering for *string*, and for *point* the lexicographic ordering of the cartesian coordinates. For all other types T there is no default ordering, and the user has to provide a *compare* function whenever a linear order on T is required.

Example: Suppose pairs of real numbers shall be used as keys in a dictionary with the lexicographic order of their components. First we declare class *pair* as the type of pairs of real numbers, then we define the I/O operations *operator>>* and *operator<<* and the lexicographic order on *pair* by writing an appropriate *compare* function.

```
class pair {
  double  x;
  double  y;

public:
 pair() { x = y = 0; }
 pair(const pair& p) { x = p.x; y = p.y; }


 friend istream& operator>> (istream& is, pair& p)
 { is >> p.x >> p.y; return is; }
 friend ostream& operator<< (ostream& os, const pair& p)
 { os << p.x << " " << p.y; return os; }

 friend int compare(const pair&, const pair&);

};

int compare(const pair& p, const pair& q)
{
  if (p.x < q.x) return  -1;
  if (p.x > q.x) return   1;
  if (p.y < q.y) return  -1;
  if (p.y > q.y) return   1;
  return 0;
}
```

Now we can use dictionaries with key type *pair*, e.g.,

```
dictionary<pair,int> D;
```

Sometimes, a user may need additional linear orders on a data type T which are different from the order defined by *compare*. In The following example a user wants to order points in the plane by the lexicographic ordering of their cartesian coordinates and by their polar coordinates. The former ordering is the default ordering for points. The user can introduce an alternative ordering on the data type *point* (cf. section 11) by defining an appropriate comparing function

```
int   pol_cmp(const point& x, const point& y)
{ /* lexicographic ordering on polar coordinates */ }
```

Now she has several possibilities:

1. First he can call the macro

   ```
   DEFINE_LINEAR_ORDER(point, pol_cmp, pol_point)
   ```

   After this call *pol_point* is a new data type which is equivalent to the data type *point*, with the only exception that if *pol_point* is used as an actual parameter e.g. in *dictionary<pol_point,...>*, the resulting data type is based on the linear order defined by *pol_cmp*. Now, dictionaries based on either ordering can be used.

   ```
   dictionary<point,int>     D0; // default ordering
   dictionary<pol_point,int> D1; // polar ordering
   ```

   In general the macro call

   ```
   DEFINE_LINEAR_ORDER(T, cmp, T1)
   ```

   introduces a new type $T1$ equivalent to type T with the linear order defined by the compare function *cmp*.

2. As a new feature all order based data types like dictionaries, priority queues, and sorted sequences offer a constructor which allows a user to set the internally used ordering at construction time.

   ```
   dictionary<point,int> D0;          // default ordering
   dictionary<point,int> D1(pol_cmp); // polar ordering
   ```

   This alternative handles the cases where two or more different orderings are needed more elegantly.

3. Instead of passing a compare function $cmp(const\ T\&, const\ T\&)$ to the sorted type one can also pass an object (a so-called *compare object*) of a class that is derived from the class *leda_cmp_base* and that overloads the function-call operator $int\ operator()(const\ T\&, const\ T\&)$ to define a linear order for T.

   This variant is helpful when the compare function depends on a global parameter. We give an example. More examples can be found in several sections of the LEDA book [58]. Assume that we want to compare edges of a graph $GRAPH<point, int>$ (in this type every node has an associated point in the plane; the point associated with a node v is accessed as $G[v]$) according to the distance of their endpoints. We write

```
class cmp_edges_by_length: public leda_cmp_base<edge> {
  const GRAPH<point,int>& G;
public:
  cmp_edges_by_length(const GRAPH<point,int>& g): G(g){}

  int operator()(const edge& e, const edge& f) const
  { point pe = G[G.source(e)]; point qe = G[G.target(e)];
    point pf = G[G.source(f)]; point qf = G[G.target(f)];
    return compare(pe.sqr_dist(qe),pf.sqr_dist(qf));
  }
};

main(){
  GRAPH<point,int> G;
  cmp_edges_by_length cmp(G);
  list<edge> E = G.all_edges();
  E.sort(cmp);
}
```

The class *cmp_edges_by_length* has a function operator that takes two edges e and f of a graph G and compares them according to their length. The graph G is a parameter of the constructor. In the main program we define $cmp(G)$ as an instance of *cmp_edges_by_length* and then pass *cmp* as the compare object to the sort function of *list<edge>*. In the implementation of the sort function a comparison between two edges is made by writing $cmp(e, f)$, i.e., for the body of the sort function there is no difference whether a function or a compare object is passed to it.

1.3.2 Hashed Types

LEDA also contains parameterized data types requiring a *hash function* and an *equality test* (operator==) for the actual type parameters. Examples are dictionaries implemented by hashing with chaining (*_dictionary<K,I,ch_hashing>*) or hashing arrays (*h_array<I,E>*). Whenever a type T is used in such a context, e.g., in *h_array<T,...>* there must be defined

1. a hash function *int* **Hash**(*const T*&)

2. the equality test *bool* **operator** == (*const T*&, const*T*&)

Hash maps the elements of type T to integers. It is not required that *Hash* is a perfect hash function, i.e., it has not to be injective. However, the performance of the underlying implementations very strongly depends on the ability of the function to keep different elements of T apart by assigning them different integers. Typically, a search operation in a hashing implementation takes time linear in the maximal size of any subset whose elements are assigned the same hash value. For each of the simple numerical data types char, short, int, long there is a predefined *Hash* function: the identity function.

We demonstrate the use of *Hash* and a data type based on hashing by extending the example from the previous section. Suppose we want to associate information with values of the *pair* class by using a hashing array *h_array<pair,int> A*. We first define a hash function that assigns each pair (x, y) the integral part of the first component x

```
int  Hash(const pair& p) { return int(p.x); }
```

and then can use a hashing array with index type *pair*

```
h_array<pair, int>  A;
```

1.3.3 Implementation Parameters

For many of the parameterized data types (in the current version: dictionary, priority queue, d_array, and sortseq) there exist variants taking an additional data structure parameter for choosing a particular implementation (cf. chapter 6.2, 6.4, 6.6 and 7.2). Since C++ does not allow to overload templates we had to use different names: the variants with an additional implementation parameters start with an underscore, e.g., *_d_array<I,E,impl>*. We can easily modify the example program from section 1.1 to use a dictionary array implemented by a particular data structure, e.g., skip lists ([**?**]), instead of the default data structure (cf. section 15.1).

```
#include <LEDA/_d_array.h>
#include <LEDA/impl/skiplist.h>

main()
{
  _d_array<string, int, skiplist>  N(0);
  string s;

  while (cin >> s) N[s]++;

  forall_defined(s,N)
    cout << s << " " << N[s] << endl;
}
```

Any type *_XYZ*<T1, ... ,Tk,*xyz_impl*> is derived from the corresponding “normal” parameterized type *XYZ*<T1, ...,Tk>, i.e., an instance of type *_XYZ*<T1, ...,Tk,*xyz_impl*> can be passed as argument to functions with a formal parameter of type *XYZ*<T1, ...,Tk>&. This provides a mechanism for choosing implementations of data types in pre-compiled algorithms. See “prog/graph/dijkstra.c” for an example.
LEDA offers several implementations for each of the data types. For instance, skip lists, randomized search trees, and red-black trees for dictionary arrays. Users can also provide their own implementation. A data structure *xyz_impl* can be used as actual implementation parameter for a data type *_XYZ* if it provides a certain set of operations and uses certain virtual functions for type dependent operations (e.g. compare, initialize, copy, ...). Chapter 15 lists all data structures contained in the current version and gives the exact requirements for implementations of dictionaries, priority_queues, sorted sequences and dictionary arrays. A detailed description of the mechanism for parameterized data types and implementation parameters used in LEDA will be published soon.

1.4 Arguments

- **Optional Arguments**

  The trailing arguments in the argument list of an operation may be optional. If these trailing arguments are missing in a call of an operation the default argument values given in the specification are used. For example, if the relative position argument in the list insert operation is missing it is assumed to have the value *LEDA::after*, i.e., L.insert(it, y) will insert the item $<y>$ after item it into L.

- **Argument Passing**

  There are two kinds of argument passing in C++, by value and by reference. An argument x of type *type* specified by "*type* x" in the argument list of an operation or user defined function will be passed by value, i.e., the operation or function is provided with a copy of x. The syntax for specifying an argument passed by reference is "*type*& x". In this case the operation or function works directly on x (the variable x is passed not its value).

  Passing by reference must always be used if the operation is to change the value of the argument. It should always be used for passing large objects such as lists, arrays, graphs and other LEDA data types to functions. Otherwise a complete copy of the actual argument is made, which takes time proportional to its size, whereas passing by reference always takes constant time.

- **Functions as Arguments**

  Some operations take functions as arguments. For instance the bucket sort operation on lists requires a function which maps the elements of the list into an interval of integers. We use the C++ syntax to define the type of a function argument f:

  ```
  T  (*f)(T1, T2, ..., Tk)
  ```

  declares f to be a function taking k arguments of the data types $T1, \ldots, Tk$, respectively, and returning a result of type T, i.e,

  $$f : T1 \times \ldots \times Tk \longrightarrow T$$

1.5 Items

Many of the advanced data types in LEDA (dictionaries, priority queues, graphs, ...), are defined in terms of so-called items. An item is a container which can hold an object relevant for the data type. For example, in the case of dictionaries a *dic_item* contains a pair consisting of a key and an information. A general definition of items is given at the end of this section.

We now discuss the role of items for the dictionary example in some detail. A popular specification of dictionaries defines a dictionary as a partial function from some type K

to some other type I, or alternatively, as a set of pairs from $K \times I$, i.e., as the graph of the function. In an implementation each pair (k, i) in the dictionary is stored in some location of the memory. Efficiency dictates that the pair (k, i) cannot only be accessed through the key k but sometimes also through the location where it is stored, e.g., we might want to lookup the information i associated with key k (this involves a search in the data structure), then compute with the value i a new value i', and finally associate the new value with k. This either involves another search in the data structure or, if the lookup returned the location where the pair (k, i) is stored, can be done by direct access. Of course, the second solution is more efficient and we therefore wanted to provide it in LEDA.

In LEDA items play the role of positions or locations in data structures. Thus an object of type *dictionary*<K,I>, where K and I are types, is defined as a collection of items (type *dic_item*) where each item contains a pair in $K \times I$. We use $<k, i>$ to denote an item with key k and information i and require that for each k in K there is at most one i in I such that $<k, i>$ is in the dictionary. In mathematical terms this definition may be rephrased as follows: A dictionary d is a partial function from the set *dic_item* to the set $K \times I$. Moreover, for each k in K there is at most one i in I such that the pair (k, i) is in d.

The functionality of the operations

```
dic_item  D.lookup(K k)
I         D.inf(dic_item it)
void      D.change_inf(dic_item it, I i')
```

is now as follows: $D.lookup(K\ k)$ returns an item it with contents (k, i), $D.inf(it)$ extracts i from it, and a new value i' can be associated with k by $D.change\_inf(it, i')$.

Let us have a look at the insert operation for dictionaries next:

```
dic_item  D.insert(K k, I i)
```

There are two cases to consider. If D contains an item it with contents (k, i') then i' is replaced by i and it is returned. If D contains no such item, then a new item, i.e., an item which is not contained in any dictionary, is added to D, this item is made to contain (k, i) and is returned. In this manual (cf. section 6.1) all of this is abbreviated to

dic_item	D.insert($K\ k, I\ i$)	associates the information i with the key k. If there is an item $<k, j>$ in D then j is replaced by i, else a new item $<k, i>$ is added to D. In both cases the item is returned.

We now turn to a general discussion. With some LEDA types XYZ there is an associated type *XYZ_item* of items. Nothing is known about the objects of type *XYZ_item* except that there are infinitely many of them. The only operations available on *XYZ_items* besides the one defined in the specification of type XYZ is the equality predicate "==" and the assignment operator "=" . The objects of type XYZ are defined as sets or sequences of *XYZ_items* containing objects of some other type Z. In this situation an

XYZ_item containing an object z in Z is denoted by $<z>$. A new or unused *XYZ_item* is any *XYZ_item* which is not part of any object of type XYZ.

Remark: For some readers it may be useful to interpret a *dic_item* as a pointer to a variable of type $K \times I$. The differences are that the assignment to the variable contained in a *dic_item* is restricted, e.g., the K-component cannot be changed, and that in return for this restriction the access to *dic_items* is more flexible than for ordinary variables, e.g., access through the value of the K-component is possible.

1.6 Iteration

For many (container) types LEDA provides iteration macros. These macros can be used to iterate over the elements of lists, sets and dictionaries or the nodes and edges of a graph. Iteration macros can be used similarly to the C++ **for** statement. Examples are

- for all item based data types:

  forall_items(it, D) { the items of D are successively assigned to variable it }

  forall_rev_items(it, D) { the items of D are assigned to it in reverse order }

- for lists and sets:

  forall(x, L) { the elements of L are successively assigned to x}

  forall_rev(x, L) { the elements of L are assigned to x in reverse order}

- for graphs:

  forall_nodes(v, G) { the nodes of G are successively assigned to v}

  forall_edges(e, G) { the edges of G are successively assigned to e}

  forall_adj_edges(e, v) { all edges adjacent to v are successively assigned to e}

PLEASE NOTE:
Inside the body of a forall loop insertions into or deletions from the corresponding container are not allowed, with one exception, the current item or object of the iteration may be removed, as in

```
forall_edges(e,G) {
  if (source(e) == target(e)) G.del_edge(e);
} // remove selfloops
```

The item based data types `list`, `array`, and `dictionary` provide now also an STL compatible iteration scheme. The following example shows STL iteration on lists. Note that not all LEDA supported compilers allow the usage of this feature.

```
list<int> L;
// fill list somehow
list<int>::iterator it;
for ( it = L.begin(); it != L.end(); it++ )
  cout << *it << endl;
```

`list<int>::iterator` defines the iterator type, `begin()` delivers access to the first list item via an iterator. `end()` is the past the end iterator and serves as an end marker. The increment operator `++` moves the iterator one position to the next item, and `*it` delivers the content of the item to which the iterator is pointing. For more information on STL please refer to the standard literature about STL.

For a more flexible access to the LEDA graph data type there are graph iterators which extent the STL paradigm to more complex container types. To make use of these features please refer to 10.

1.7 The LEDA Tools for Manual Production and Documentation

The Lman and Fman utilities give online access to the manual pages of the LEDA system. Lman produces LaTeX-output and Fman produces low quality ASCII-output for quick reference. The usage is

```
Lman T[.h] options
Fman T[.h] filter
```

where T is the name of a LEDA type, e.g., list, sortseq, or point. Without options Lman produces manual pages as contained in the LEDA manual. Thus, `Lman list` produces the manual page for the LEDA type list and `Fman list` produces a low quality version of it. Please try out Lman and Fman before proceeding. If they do not work the error is very likely to be one of the following.

- You don't have a current version of *perl* installed and accessible.
- You don't have a current version of *latex* installed and accessible.
- The environment variable `LEDAROOT` is not set to the root directory of the LEDA system.
- `$LEDAROOT/Manual/cmd` is not part of your `PATH`.
- `$LEDAROOT/Manual/tex` is not part of your `TEXINPUTS`.

The behavior of Lman and Fman can be fine-tuned by options. A call `Lman` without arguments gives a short survey of all available options. Options are specified in assignment syntax `variable=value`. There must be no blank on either side of the equality sign. In the list of options to follow we list the default value of each option first.

size={12, 11, 10} Determines the font size.

constref={no, yes} Determines how const-ref parameters are displayed. With the no-option a const-ref parameter `const T& x` is displayed as a value parameter `T x` and with the yes-option it is displayed in full.

partypes={no, yes} Determines how parameters of unary and binary operators are displayed. Consider for example an operator `+` of a class number. With the no-option the operator `operator+(number x, number y)` is displayed as `x + y` and with the yes-option it is displayed as `number x + number y`.

numbered={no, yes} Determines whether the headerline of the manual page is numbered. You probably want it numbered when the manual page becomes part of a larger document.

warnings={yes, no} Determines whether Lman gives warnings. We recommend to use the option warnings=yes and to deal with individual warnings as later in the chapter.

verbosity={yes, no} Determines whether Lman gives status information.

ack={no, yes} Determines whether Lman asks for acknowledgments of warnings.

usesubscripts={yes, no} Determines whether variables consisting of a single character followed by a number are displayed as subscripted variables.

filter={all, signatures, definition, creation, operations, implementation, example, opname} Determines which part of the manual page is shown. The all-option shows the complete manual page, the signature-options shows the signatures of all operations of the data type, the next five options show only the corresponding section of the manual page, and the opname-option shows only the operation with the same name. Thus `Fman list insert` shows the insert operation for lists and `Lman point filter=operator+` extracts the + operator of the point type (write arrop to inquire about the array operator and funop to inquire about the function operator).

outfile={"",string} Determines whether the TEX-file generated is written on a temporary file (the default option) or on the file with name string. Only available for Lman.

latexruns={1, 0, 2} Determines the number of LATEX runs used to produce the manual page. You need to run LATEX twice if the manual page contains cross references.

xdvi={yes, no} Determines whether the manual page is displayed by xdvi.

Lman can be customized by putting options in a file Lman.cfg in either the home directory or the working directory. Command line options take precedence over options in the working directory which in turn take precedence over options in the home directory.

Fman is the utility for quick reference. It takes two arguments, a typename and one of the filters, and directly displays its output. For example, the call `Fman list signatures` will display the signatures of all operations of the list data type.

Lman and Fman are not only applicable to the header files of the LEDA system, they can be applied to any file containing (possibly among other things) C++-code suitably augmented by so-called manual comments. A manual comment is any comment of the form `/*{\Mcommand ...}*/` where Mcommand is one of so-called manual commands. We

```
#ifndef LEDA_STACK_H
#define LEDA_STACK_H

#include <LEDA/basic.h>
#include <LEDA/impl/slist.h>

/*{\Manpage {stack} {E} {Stacks} {S}}*/

template<class E> class _CLASSTYPE stack : private SLIST
{
/*{\Mdefinition
An instance |S| of the parameterized data type |\Mname| is a sequence of
elements of data type |E|, called the element type of |S|. Insertions or
deletions of elements take place only at one end of the sequence, called
the top of |S|. The size of |S| is the length of the sequence, a stack
of size zero is called the empty stack.}*/

public:

/*{\Mcreation}*/

  stack() {}
/*{\Mcreate creates an instance |\Mvar| of type |\Mname| and initializes
it to the empty stack.}*/

  stack(const stack<E>& S) : SLIST(S) {}
 ~stack() { clear(); }
  stack<E>& operator=(const stack<E>& S)
  { return (stack<E>&)SLIST::operator=(S); }

/*{\Moperations 2.5 4}*/

E top()   const { return ACCESS(E,SLIST::head());}
/*{\Mop      returns the top element of |\Mvar|.\\
             \precond $S$ is not empty.}*/

void push(E x)  { SLIST::push(Copy(x)); }
/*{\Mop      adds $x$ as new top element to |\Mvar|.}*/

E pop()         { E x=top(); SLIST::pop(); return x; }
/*{\Mop      deletes and returns the top element of |\Mvar|.\\
             \precond $S$ is not empty.}*/

\*{\Mimplementation
Stacks are implemented by singly linked linear lists. All operations take
time $O(1)$.}*/
};
#endif
```

Figure 1.1: The header file for the stack data type.

discuss manual commands in detail below. Every manual comment produces part of the manual page.

Stacks (stack)

1. Definition

An instance S of the parameterized data type *stack*<*E*> is a sequence of elements of data type E, called the element type of S. Insertions or deletions of elements take place only at one end of the sequence, called the top of S. The size of S is the length of the sequence, a stack of size zero is called the empty stack.

2. Creation

stack<*E*> S;	creates an instance S of type *stack*<*E*> and initializes it to the empty stack.

3. Operations

E	S.top( )	returns the top element of S. *Precondition*: S is not empty.
void	S.push(E x)	adds x as new top element to S.
E	S.pop( )	deletes and returns the top element of S. *Precondition*: S is not empty.

4. Implementation

Stacks are implemented by singly linked linear lists. All operations take time $O(1)$, except clear which takes time $O(n)$, where n is the size of the stack.

Figure 1.2: The manual page for the stack data type

Figure 1.1 shows a header file augmented by manual comments and figure 1.2 shows the manual page produced from it. The body of a manual comment is an ordinary LaTeX-text augmented by the possiblity to *quote code*. Code is quoted by enclosing it either between vertical bars (|...|) or between double square brackets (`[[...]]`). Code quoted by double square brackets is typeset using typewriter font and code quoted by vertical bars is given a math-like appearance. So `[[`S.insert(E x)`]]` produces `S.insert(E x)` and |S.insert(E x)| produces $S.insert(E\ x)$.

Lman is based on the command lextract. See figure 1.3.

```
lextract infile outfile [options]
```

reads infile, processes the manual commands in it, and writes the core of a TeX-file on outfile. All Lman options apply. Lman itself works in three phases. It first produces a tempory LaTeX-file

```
\documentclass[a4paper,size pt]{article}
```

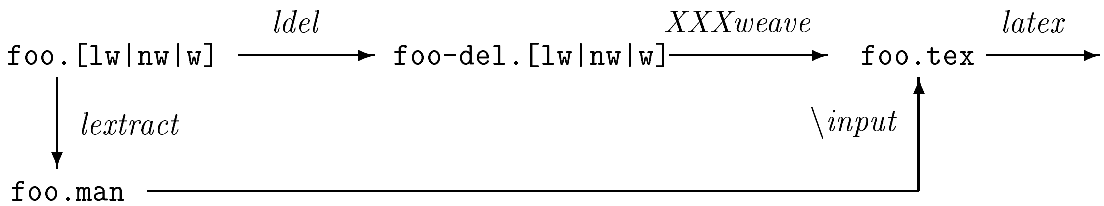

Figure 1.3: The Manual Production Process

```
\usepackage{Lweb}
\begin{document}
  output of lextract
\end{document}
```

and then applies latex and xdvi to it.

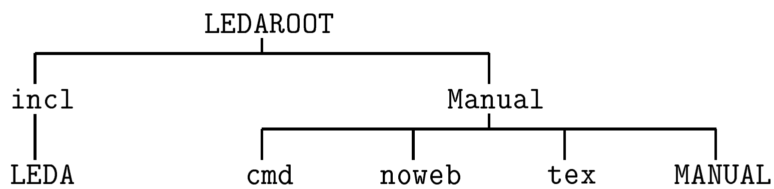

Figure 1.4: Manual relevant directories

Ldoc is the tool to produce full documentations consisting of manual page, implementation, and test routines. Ldoc is based on noweb [72], and Lweb. Lweb is a dialect of noweb which we developed for the production of the LEDA book and the documentation of the LEDA system. You should have a working knowledge of noweb before proceeding. Ldoc is based on the commands *lextract*, *ldel*, and *weave*. We discussed lextract already.

```
ldel infile outfile
```

reads infile, deletes all manual comments from it, and writes the result on outfile. Weave is the weave-command of the literate programming tool in use, i.e., noweave for noweb, and lweave for Lweb. Ldoc works on noweb-, and Lweb-files. It first uses lextract to extract the manual and ldel to extract a file pure of manual comments, it then applies the appropriate weave command to the output of ldel, and it finally applies latex and xdvi to the resulting file. All Lman options apply.

Figure 1.4 shows the parts of the LEDAROOT directory that are relevant for manual production. Subdirectory `Manual` splits into cmd, noweb, tex, and MANUAL: `cmd` contains the commands Lman, Fman, ..., `noweb` contains the full documentation (written in noweb) of all commands, `tex` contains the style-files for manual production, and `MANUAL` contains the actual LEDA manual.

For more information about the LEDA tools for manual production and documentation we refer the reader to the corresponding chapter of [58].

Chapter 2

Technical Information

This chapter provides information about installation, usage, interaction with other software packages, and an overview about all supported system platforms.

2.1 Installation

As mentioned before, you can either get the source code package or the object code package from the LEDA download site. Both have to be unpacked as follows. Suppose the downloaded file is `LEDA-4.1.tgz`. Place it into a directory for which you have write permission and type

```
gunzip LEDA-4.1.tgz
```

and

```
tar xvf LEDA-4.1.tar
```

If you have the source code of LEDA just continue reading, otherwise go to section 2.1.2

2.1.1 Source Code Package

The source code package is only distributed for unix installation (see section 2.4 for details). In this subsection we use `leda-root-path` to denote the path name of the LEDA main directory (in the source code package the name usually consists of the prefix *LEDA* followed by the version number, for instance *LEDA-4.1*). It should contain the following files and subdirectories:

README	information about LEDA
INSTALL	this file
CHANGES (please read !)	most recent changes
FIXES	bug fixes since last release
Manual/	user manual
Makefile	make script
confdir/	configuration directory
lconfig	configuration command
incl/	include directory
src/	source files
test/	example and test programs
demo/	demo programs

Configuration

1. Go to `leda-root-path`.

2. Type: `lconfig <cc> [static | shared]`

   where `<cc>` is the name (or command) of your C++compiler and the optional second parameter defines the kind of libraries to be generated. Note that in the current release shared libraries can be made only under SunOS 5.x (Solaris), Linux, IRIX, and HP-UX.

   Examples: `lconfig sunpro`, `lconfig g++`, `lconfig CC shared`

   lconfig without arguments prints a list of known compilers.
   If your compiler is not in the list you might have to edit the `<LEDA/sys/unix.h>` header file.

Compilation

Now just type

```
make
```

to make the following libraries (cf. section 2.2.2):

libL.a (libL.so)	main library
libG.a (libG.so)	graph library
libP.a (libP.so)	d2-geo library
libD3.a (libP.so)	d3-geo library
libW.a (libW.so)	window library

(*.so if shared libraries are used)

The graphics library libW is based on X11. It can be made only if the main X11 include directory is located in one of the standard include directories of your compiler (e.g.

/usr/include/X11). If this is not the case you have to tell the compiler where it is by using the -I option in src/Make.src, e.g., -I/usr/openwin/include should work on most SUN workstations.

Now you are in a situation similar to people having downloaded the object code package. So please read on!

2.1.2 Object Code Packages

If you have unpacked an object code package, then your current working directory should contain the files libL.a, libG.a, libP.a, libW.a and lib3D.a.

To procede you have to modify your environment as follows:

1. set the the environment variable LEDAROOT to the path leda-root-path containing the extracted package. Use setenv LEDAROOT leda-root-path for csh/tcsh and LEDAROOT = leda-root-path; export LEDAROOT; for sh/bash.

2. Extend the command search path by including $LEDAROOT/Manual/cmd into your command search path (environment variable path (csh) or PATH (sh)) and call rehash (if required by your system).

3. You can produce shared libraries for the systems solaris, linux, irix, and hpux. Include $LEDAROOT into the LD_LIBRARY_PATH search path. Then goto $LEDAROOT and type make shared. This will construct the shared libraries libL.so,libG.so,libP.so,libW.so, and libD3.so (*.sl for hpux).

4. For the online documentation tool go to $LEDAROOT and type make xlman to compile and link LEDA's interactive manual reader "xlman" and some demo programs. Now you can start "xlman" for reading and printing manual pages, starting demo programs and browsing more release notes.

You may want to ask your system adminstrator to install the header files and libraries in the system's default directories. Then you no longer have to specify header and library search paths on the compiler command line.

Remark: Of course you can also use LEDA if you do not have access to the standard directories of the compiler you're using. Section 2.2.2 will tell you how.

2.2 Usage

2.2.1 Header Files

LEDA data types and algorithms can be used in any C++ program as described in this manual (for the general layout of a manual page please see chapter 1.2). The specifications (class declarations) are contained in header files. To use a specific data type its header file has to be included into the program. In general the header file for data type xyz is <LEDA/xyz.h>. The header file names are also printed at the end of the definition section of each data type.

2.2.2 Libraries

The implementions of most LEDA data types and algorithms are precompiled and contained in the 5 libraries mentioned above (libL.a, libG.a, libP.a, libW.a, lib3D.a). They can be linked with C++ application programs. In the following description it is assumed that these libraries are installed in one of the systems default library directories (e.g. `/usr/lib`), which allows to use the "`-l...`" compiler option.

libL.a is the main LEDA library, it contains the implementations of all simple data types (chapter 3), basic data types (chapter 5), dictionaries and priority queues (chapter 6 and 7). A program *prog.c* using any of these data types has to be linked with the libL.a library like this:

```
CC prog.c -lL -lm
```

libG.a is the LEDA graph library. It contains the implementations of all graph data types and algorithms (chapter 8). To compile a program using any graph data type or algorithm the libG.a and libL.a library have to be used:

```
CC prog.c -lG -lL -lm
```

libP.a is the LEDA library for geometry in the plane. It contains the implementations of all data types and algorithms for geometry (chapter 11 and 12). To compile a program using geometric data types or algorithms the libP.a, libG.a, libL.a and maths libraries have to be used:

```
CC prog.c -lP -lG -lL -lm
```

libW.a is the LEDA library for graphic windows under the X11 window system. Application programs using data type *window* (cf. section 14.3) have to be linked with this library:

```
CC prog.c -lW -lP -lG -lL -lX11 -lm
```

libD3.a is the LEDA library for three-dimensional geometry. Application programs using three-dimensional data types (cf. chapter 13) have to be linked with this library:

```
CC prog.c -lD3 -lW -lP -lG -lL -lX11 -lm
```

Remarks:

- Note that the libraries must be given in the above order.
- If using windows on solaris systems you might have to link with the socket and thread libraries as well:

  ```
  CC (g++) prog.c -lW -lP -lG -lL -lX11 -lsocket -lthread -lm
  ```

- If the include files and libraries are not copied to `/usr/(local/)include` and `/usr/(local/)lib` on your system, you have to use the `-I` and `-L` compiler flags to compile and link LEDA programs:

  ```
  CC (g++) -Ileda-root-path/incl -Lleda-root-path...
  ```

  The `-I` flag tells the compiler where to find the header files, the `-L` tells the compiler where to find the libraries.

- If using shared libraries you either have to include `leda-root-path` into the dynamic library search path for ld:

  `setenv LD_LIBRARY_PATH leda-root-path:$LD_LIBRARY_PATH`

  or you have to use the `-R` option

  `CC (g++) -Ileda-root-path/incl -Lleda-root-path -Rleda-root-path ...`

  The `-R` flag tells the executable where to find the shared libraries for dynamic linking (`-rpath` in case of Irix).

2.3 Namespaces, Standard Compliance, and Interaction with other Libraries

If users want to use other software packages like ABACUS, CGAL, or STL together with LEDA in one project avoiding naming conflicts is an issue. As name spaces are not yet generally available LEDA offers a new prefixing mechanism to avoid name conflicts. The libraries from all object code packages as well as the ones produced from the source code package contain class name symbols with the common prefix "`leda_`". A macroing package allows users to use unprefixed class names as used in the manual pages in chapters 3 to 14.

All unprefixed type names are transfered to prefixed ones at compile time by the macros provided automatically by the inclusion of any LEDA header file. There's two consequences to this.

- if you debug your code you'll always see prefixed symbols from within debuggers like *gdb*.

- if you use other libraries like STL in the same project you have to avoid LEDA's automatic prefixing being valid in the general program scope. You can disable automatic prefixing by compiling your program including the flag `-DLEDA_PREFIX`, e.g.

  `CC -DLEDA_PREFIX prog.c -lP -lG -lL -lm`

  But then of course you have to use explicitly prefixed class names in your code like:

  ```
  leda_list<int> L;
  L.append(3);
  leda_graph G;
  ...
  ```

Remember that you have to prefix all LEDA class types yourself if you use `-DLEDA_PREFIX`.

The second issue of interaction concerns the data type *bool* which is part of the new C++ standard. However not all compilers currently support a bool type. LEDA offers *bool* either compiler provided or defined within LEDA if the compiler lacks the support. Some STL packages follow a similar scheme. To solve the existance conflict of two different bool type definitions we suggest to use LEDA's bool as STL is a pure template library only provided by header files and its defined bool type can be easily replaced.

Finally LEDA now supports also standard header names without the trailing ".h". Often mixed usage of old and new headers create link problems. LEDA can be forced to use standard header names by setting the flag `LEDA_STD_HEADERS`. If you compile LEDA using `-DLEDA_STD_HEADERS` on the compiler command line or by uncommenting the corresponding definition at the end of `<LEDA/system.h>` you will have symbols based on standard header files in the libraries. Note that all libraries have to be compiled with this flag enabled and that old and new header files cannot be used simultaneously.

2.4 Platforms

When this manual was printed LEDA was available as object code for the following platforms (please read the web pages for the current details):

Platforms		
Machine	**Operating System**	**Compiler**
SUN Sparc	SunOS, Solaris	SunPRO CC, g++, CSet++
Silicon Graphics	IRIX	CC, g++
Silicon Graphics	IRIX64	CC
80486	Linux	g++
Hewlett Packard	HP-UX	HP C++, g++
DEC	Digital Unix	cxx, g++
IBM	AIX	CSet++, g++

If the platform you're using is not listed here, please contact us.

For the following platforms LEDA is available from Algorithmic Solutions Software GmbH. Note, that they sell student licenses and university licenses to special prices.

Platforms		
Machine	**Operating System**	**Compiler**
80x86	Windows NT or Windows 95 DLLs	Borland 5.0,5.2 Microsoft Visual 5.0,6.0 Watcom 11.0

Chapter 3

Simple Data Types and Basic Support Operations

This section describes simple data types like strings, streams and gives some information about error handling, memory management and file system access. The stream data types described in this section are all derived from the C++ stream types *istream* and *ostream*. They can be used in any program that includes the <LEDA/stream.h> header file. Some of these types may be obsolete in combination with the latest versions of the standard C++ I/O library.

3.1 Strings (string)

1. Definition

An instance s of the data type *string* is a sequence of characters (type *char*). The number of characters in the sequence is called the length of s. A string of length zero is called the empty string. Strings can be used wherever a C++ *const char∗* string can be used.

Strings differ from the C++ type *char∗* in several aspects: parameter passing by value and assignment works properly (i.e., the value is passed or assigned and not a pointer to the value) and *strings* offer many additional operations.

$\#include < LEDA/string.h >$

2. Creation

string s; introduces a variable s of type *string*. s is initialized with the empty string.

string $s(const\ char * p)$;

introduces a variable s of type *string*. s is initialized with a copy of the C++ string p.

string $s(char\ c)$; introduces a variable s of type *string*. s is initialized with the one-character string "c".

string $s(const\ char * format, ...)$;
introduces a variable s of type *string*. s is initialized with the string produced by printf($format$,...).

3. Operations

int s.length() returns the length of string s.

char& $s[int\ i]$ returns the character at position i.
Precondition: $0 \le i \le s$.length()-1.

string $s(int\ i,\ int\ j)$ returns the substring of s starting at position $\max(0, i)$ and ending at position $\min(j, s.\text{length}()-1)$.
If $\min(j, s.\text{length}()-1) < \max(0, i)$ then the empty string is returned.

string s.head($int\ i$) returns the first i characters of s.

string s.tail($int\ i$) returns the last i characters of s.

int s.pos($string\ s1,\ int\ i$) returns the minimum j such that $j \ge i$ and s_1 is a substring of s starting at position j (returns -1 if no such j exists).

int s.pos($string\ s1$) returns pos$(s1, 0)$.

string s.insert($int\ i,\ string\ s1$) returns $s(0, i-1) + s_1 + s(i, s.\text{length}()-1)$.

string s.replace($string\ s1,\ string\ s2,\ int\ i = 1$)
returns the string created from s by replacing the i-th occurrence of s_1 in s by s_2.

string s.replace($int\ i,\ int\ j,\ string\ s1$)
returns the string created from s by replacing $s(i, j)$ by s_1.
Precondition: $i \le j$.

string s.replace($int\ i,\ string\ s1$) returns the string created from s by replacing $s[i]$ by s_1.

string	s.replace_all(*string s1*, *string s2*)	returns the string created from s by replacing all occurrences of s_1 in s by s_2. *Precondition*: The occurrences of s_1 in s do not overlap (it's hard to say what the function returns if the precondition is violated.).
string	s.del(*string s1*, *int* $i = 1$)	returns s.replace(s_1, "", i).
string	s.del(*int i*, *int j*)	returns s.replace(i, j, "").
string	s.del(*int i*)	returns s.replace(i, "").
string	s.del_all(*string s1*)	returns s.replace_all(s_1, "").
void	s.read(*istream& I*, *char delim* = ' ')	reads characters from input stream I into s until the first occurrence of character *delim*.
void	s.read(*char delim* = ' ')	same as to s.read(*cin*,*delim*).
void	s.read_line(*istream& I*)	same as to s.read(I,'\ n').
void	s.read_line( )	same as to s.read_line(*cin*).
void	s.read_file(*istream& I*)	same as to s.read(I,'EOF').
void	s.read_file( )	same as to s.read_file(*cin*).
string&	s += x	appends x to s and returns a reference to s.
string	$x + y$	returns the concatenation of x and y.
bool	$x == y$	true iff x and y are equal.
bool	x != y	true iff x and y are not equal.
bool	$x < y$	true iff x is lexicographically smaller than y.
bool	$x > y$	true iff x is lexicographically greater than y.
bool	$x \leq y$	returns $(x < y) \mid (x == y)$.
bool	$x \geq y$	returns $(x > y) \mid (x == y)$.
istream&	*istream& I* $\gg$ *string& s*	same as s.read(I,' ').

$ostream\&\ \ ostream\&\ O \ll s$	writes string s to the output stream O.

4. Implementation

Strings are implemented by C++ character vectors. All operations involving the search for a pattern $s1$ in a string s take time $O(s.lenght() * s1.length())$, [ ] takes constant time and all other operations on a string s take time $O(s.length())$.

3.2 File Input Streams (file_istream)

1. Definition

The data type *file_istream* is equivalent to the *ifstream* type of C++.

#include < *LEDA/stream.h* >

3.3 File Output Streams (file_ostream)

1. Definition

The data type *file_istream* is equivalent to the *ofstream* type of C++.

#include < *LEDA/stream.h* >

3.4 String Input Streams (string_istream)

1. Definition

An instance I of the data type *string_istream* is an C++istream connected to a string s, i.e., all input operations or operators applied to I read from s.

#include < *LEDA/stream.h* >

2. Creation

string_istream $I(const\ char * s)$;

creates an instance I of type string_istream connected to the string s.

3. Operations

All operations and operators (>>) defined for C++istreams can be applied to string input streams as well.

3.5 String Output Streams (string_ostream)

1. Definition

An instance O of the data type *string_ostream* is an C++ostream connected to an internal string buffer, i.e., all output operations or operators applied to O write into this internal buffer. The current value of the buffer is called the contents of O.

#include < *LEDA/stream.h* >

2. Creation

string_ostream O; creates an instance O of type string_ostream.

3. Operations

string O.str( ) returns the current contents of O.

All operations and operators (<<) defined for C++ostreams can be applied to string output streams as well.

3.6 Command Input Streams (cmd_istream)

1. Definition

An instance I of the data type *cmd_istream* is an C++istream connected to the output of a shell command *cmd*, i.e., all input operations or operators applied to I read from the standard output of command *cmd*.

#include < *LEDA/stream.h* >

2. Creation

cmd_istream $I(char * cmd)$;

creates an instance I of type cmd_istream connected to the output of command *cmd*.

3. Operations

All operations and operators (>>) defined for C++istreams can be applied to command input streams as well.

3.7 Command Output Streams (cmd_ostream)

1. Definition

An instance O of the data type *cmd_ostream* is an C++ostream connected to the input of a shell command *cmd*, i.e., all output operations or operators applied to O write into the standard input of command *cmd*.

#include < *LEDA/stream.h* >

2. Creation

cmd_ostream $O(char * cmd)$;

creates an instance O of type cmd_ostream connected to the input of command *cmd*.

3. Operations

All operations and operators (<<) defined for C++ostreams can be applied to command output streams as well.

3.8 Random Sources (random_source)

1. Definition

An instance of type *random_source* is a random source. It allows to generate uniformly distributed random bits, characters, integers, and doubles. It can be in either of two modes: In bit mode it generates a random bit string of some given length p ($1 \leq p \leq 31$) and in integer mode it generates a random integer in some given range $[low..high]$ ($low \leq high < low + 2^{31}$). The mode can be changed any time, either globally or for a single operation. The output of the random source can be converted to a number of formats (using standard conversions).

#include < *LEDA/random_source.h* >

2. Creation

random_source S; — creates an instance S of type *random_source*, puts it into bit mode, and sets the precision to 31.

random_source $S(int\ p)$;

creates an instance S of type *random_source*, puts it into bit mode, and sets the precision to p ($1 \leq p \leq 31$).

random_source $S(int\ low,\ int\ high)$;

creates an instance S of type *random_source*, puts it into integer mode, and sets the range to $[low..high]$.

3. Operations

unsigned long	S.get( )	returns a random unsigned integer of maximal precision (32 bits on 32-bit systems and 64 bits on 64-bit systems).
void	S.set_seed(*int s*)	resets the seed of the random number generator to s.
void	S.set_range(*int low*, *int high*)	sets the mode to integer mode and changes the range to $[low..high]$.
int	S.set_precision(*int p*)	sets the mode to bit mode, changes the precision to p bits and returns previous precision.
int	S.get_precision( )	returns current precision of S.

random_source&	$S \gg$ *char& x*	extracts a character x of default precision or range and returns S, i.e., it first generates an unsigned integer of the desired precision or in the desired range and then converts it to a character (by standard conversion).
random_source&	$S \gg$ *unsigned char& x*	extracts an unsigned character x of default precision or range and returns S.
random_source&	$S \gg$ *int& x*	extracts an integer x of default precision or range and returns S.
random_source&	$S \gg$ *long& x*	extracts a long integer x of default precision or range and returns S.
random_source&	$S \gg$ *unsigned int& x*	extracts an unsigned integer x of default precision or range and returns S.
random_source&	$S \gg$ *unsigned long& x*	extracts a long unsigned integer x of default precision or range and returns S.
random_source&	$S \gg$ *double& x*	extracts a double precision floating point number x in $[0,1]$, i.e, $u/(2^{31}-1)$ where u is a random integer in $[0..2^{31}-1]$, and returns S.
random_source&	$S \gg$ *float& x*	extracts a single precision floating point number x in $[0,1]$, i.e, $u/(2^{31}-1)$ where u is a random integer in $[0..2^{31}-1]$, and returns S.
random_source&	$S \gg$ *bool& b*	extracts a random boolean value (true or false).
int	$S(\,)$	returns an integer of default precision or range.
int	S(*int prec*)	returns an integer of supplied precision *prec*.
int	S(*int low*, *int high*)	returns an integer from the supplied range [*low..high*].

3.9 Random Variates (random_variate)

1. Definition

An instance R of the data type *random_variate* is a non-uniform random number generator. The generation process is governed by an *array*<*int*> w. Let $[l \mathinner{..} r]$ be the index range of w and let $W = \sum_i w[i]$ be the total weight. Then any integer $i \in [l \mathinner{..} h]$ is generated with probability $w[i]/W$. The weight function w must be non-negative and W must be non-zero.

#include < *LEDA/random_variate.h* >

2. Creation

random_variate R(*array*<*int*> w);

creates an instance R of type *random_variate*.

3. Operations

int	R.generate( )	generates $i \in [l \mathinner{..} h]$ with probability $w[i]/W$.

3.10 Dynamic Random Variates (dynamic_random_variate)

1. Definition

An instance R of the data type *dynamic_random_variate* is a non-uniform random number generator. The generation process is governed by an *array*<*int*> w. Let $[l \mathinner{..} r]$ be the index range of w and let $W = \sum_i w[i]$ be the total weight. Then any integer $i \in [l \mathinner{..} h]$ is generated with probability $w[i]/W$. The weight function w must be non-negative and W must be non-zero. The weight function can be changed dynamically.

The include file is `<LEDA/random_variate.h>`

#include < *LEDA/random_variate.h* >

2. Creation

dynamic_random_variate R(*array*<*int*> w);

creates an instance R of type *dynamic_random_variate*.

3. Operations

int	R.generate( )	generates $i \in [l \mathinner{..} h]$ with probability $w[i]/W$.

int R.set_weight(*int* i, *int* g)

sets $w[i]$ to g and returns the old value of $w[i]$.
Precondition: $i \in [l..h]$.

3.11 Memory Management

LEDA offers an efficient memory management system that is used internally for all node, edge and item types. This system can easily be customized for user defined classes by the "LEDA_MEMORY" macro. You simply have to add the macro call "LEDA_MEMORY(T)" to the declaration of a class T. This redefines new and delete operators for type T, such that they allocate and deallocate memory using LEDA's internal memory manager.

```
struct pair {
  double x;
  double y;

  pair() { x = y = 0; }
  pair(const pair& p) { x = p.x; y = p.y; }

  friend ostream& operator<<(ostream&, const pair&) { ... }
  friend istream& operator>>(istream&, pair&)       { ... }
  friend int compare(const pair& p, const pair& q)  { ... }

  LEDA_MEMORY(pair)

};

dictionary<pair,int> D;
```

3.12 Memory Allocator (leda_allocator)

1. Definition

An instance A of the data type *leda_allocator*<T> is a memory allocator according to the C++standard. *leda_allocator*<T> is the standard compliant interface to the LEDA memory management.

#include < *LEDA/allocator.h* >

2. Types

Local types are *size_type*, *difference_type*, *value_type*, *pointer*, *reference*, *const_pointer*, and *const_reference*.

leda_allocator<T>:: *rebind* allows the construction of a derived allocator: *leda_allocator*<T>:: *template rebind*<*T1*>:: *other* is the type *leda_allocator*<*T1*>.

3. Creation

leda_allocator<T> A; introduces a variable A of type *leda_allocator*<T>.

4. Operations

pointer	A.allocate(*size_type* n, *const_pointer* $= 0$)	returns a pointer to a newly allocated memory range of size $n * sizeof(T)$.
void	A.deallocate(*pointer* p, *size_type* n)	deallocates a memory range of $n * sizeof(T)$ starting at p. *Precondition*: the memory range was obtained via $allocate(n)$.
pointer	A.address(*reference* r)	returns $\&r$.
const_pointer	A.address(*const_reference* r)	returns $\&r$.
void	A.construct(*pointer* p, *const_reference* r)	makes an inplace new $new(\ (void*)p\ )\ T(r)$.
void	A.destroy(*pointer* p)	destroys the object referenced via p by calling $p \rightarrow \sim T(\ )$.
size_type	A.max_size( )	the largest value n for which the call $allocate(n, 0)$ might succeed.

5. Implementation

Note that the above class template uses all kinds of modern compiler technology like member templates, partial specialization etc. It runs only on a subset of LEDA's general supported platforms like $g{+}{+} > 2.95$, $SGI\ CC > 7.3$.

3.13 Error Handling (error)

LEDA tests the preconditions of many (not all!) operations. Preconditions are never tested, if the test takes more than constant time. If the test of a precondition fails an error handling routine is called. It takes an integer error number i and a $char*$ error message string s as arguments. It writes s to the diagnostic output (cerr) and terminates the program abnormally if $i \neq 0$. Users can provide their own error handling function *handler* by calling

```
set_error_handler(handler)
```

After this function call *handler* is used instead of the default error handler. *handler* must be a function of type $void\ handler(int,\ char*)$. The parameters are replaced by the error number and the error message respectively.

1. Operations

$\#include < LEDA/error.h >$

void error_handler($int\ err\_no,\ const\ char * msg$)

reports error messages by passing err_no and msg to the default error handler.

LedaErrorHandler set_error_handler($void\ (*err\_handler)(int,\ const\ char*)$)

sets the default error handler to function $err\_handler$. Returns a pointer to the previous error handler.

void catch_system_errors($bool\ b = true$)

after a call to this function system errors (e.g. bus errors and segmentation faults) are handled by LEDA's error handler.

bool leda_assert($bool\ cond,\ const\ char * err\_msg,\ int\ err\_no = 0$)

calls $error\_handler(err\_no, err\_msg)$ if $cond = false$ and returns $cond$.

3.14 Files and Directories (file)

The following functions are declared in <LEDA/file.h>.

string set_directory(*string new_dir*)
: sets the current working directory to *new_dir* and returns the name of the old cwd.

string get_directory( )
: returns the name of the current working directory.

list<string> get_directories(*string dir*)
: returns the list of names of all sub-directories in directory *dir*.

list<string> get_files(*string dir*)
: returns the list of names of all regular files in directory *dir*.

list<string> get_files(*string dir*, *string pattern*)
: returns the list of names of all regular files in directory *dir* matching pattern.

list<string> get_entries(*string dir*)
: returns the list of all entries (directory and files) of directory *dir*.

bool is_directory(*string fname*)
: returns true if *fname* is the path name of a directory and false otherwise.

bool is_file(*string fname*)
: returns true if *fname* is the path name of a regular file and false otherwise.

bool is_link(*string fname*)
: returns true if *fname* is the path name of a symbolic link and false otherwise.

int size_of_file(*string fname*)
: returns the size of file *fname* in bytes.

string tmp_file_name( )
: returns a unique name for a temporary file.

bool delete_file(*string fname*)
: deletes file *fname* returns true on success and false otherwise.

string first_file_in_path(*string fname*, *string path*, *char sep* = ' : ')
: searches all directories in string *path* (separated by *sep*) for the first directory *dir* that contains a file with name $fname$ and returns $dir/fname$ (the empty string if no such directory is contained in path).

3.15 Some Useful Functions (misc)

The following functions and macros are defined in <LEDA/basic.h>.

int	read_int(*string s*)	prints s and reads an integer from *cin*.
double	read_real(*string s*)	prints s and reads a real number from *cin*.
string	read_string(*string s*)	prints s and reads a line from *cin*.
char	read_char(*string s*)	prints s and reads a character from *cin*.
int	Yes(*string s*)	returns (read_char(s) == 'y').
bool	get_environment(*string var*)	returns *true* if variable *var* is defined in the current environment and *false* otherwise.
bool	get_environment(*string var*, *string& val*)	if variable *var* is defined in the current environment its value is assigned to *val* and the result is *true*. Otherwise, the result is *false*.
float	used_time( )	returns the currently used cpu time in seconds.
float	used_time(*float& T*)	returns the cpu time used by the program from time T up to this moment and assigns the current time to T.
void	print_statistics( )	prints a summary of the currently used memory.
void	leda_wait(*float sec*)	suspends execution for *sec* seconds.
double	truncate(*double x*, *int k* = 10)	returns a double whose mantissa is truncated after $k-1$ bits after the binary point, i.e, if $x \neq 0$ then the binary representation of the mantissa of the result has the form d.dddddddd, where the number of d's is equal to k. There is a corresponding function for *integers*; it has no effect.
T	leda_min(T a, T b)	returns the minimum of a and b.
T	leda_max(T a, T b)	returns the maximum of a and b.
void	leda_swap(T& a, T& b)	swaps values of a and b.

3.16 Two Tuples (two_tuple)

1. Definition

An instance p of type *two_tuple*$<A,B>$ is a two-tuple (a,b) of variables of types A, and B, respectively.

Related types are *two_tuple*, *three_tuple*, and *four_tuple*.

#include $<$ *LEDA/tuple.h* $>$

2. Creation

two_tuple$<A,B>$ p; — creates an instance p of type *two_tuple*$<A,B>$. All components are initialized to their default value.

two_tuple$<A,B>$ $p(A\ u,\ B\ v)$;
creates an instance p of type *two_tuple*$<A,B>$ and initializes it with the value (u,v).

3. Operations

$A\&$	p.first( )	returns the A-component of p. If p is a const-object the return type is A.
$B\&$	p.second( )	returns the B-component of p. If p is a const-object the return type is B.
bool	$p == q$	equality test for *two_tuples*. Each of the component types must have an equality operator.
int	compare(*two_tuple*$<A,B>$ p, *two_tuple*$<A,B>$ q)	lexicographic ordering for *two_tuples*. Each of the component types must have a compare function.
int	Hash(*two_tuple*$<A,B>$ p)	hash function for *two_tuples*. Each of the component types must have a Hash function.

4. Implementation

The obvious implementation is used.

3.17 Three Tuples (three_tuple)

1. Definition

An instance p of type *three_tuple*$<A, B, C>$ is a three-tuple (a, b, c) of variables of types A, B, and C, respectively.

Related types are *two_tuple*, *three_tuple*, and *four_tuple*.

#include < *LEDA/tuple.h* >

2. Creation

three_tuple$<A, B, C>$ p;

creates an instance p of type *three_tuple*$<A, B, C>$. All components are initialized to their default value.

three_tuple$<A, B, C>$ $p(A\ u,\ B\ v,\ C\ w)$;

creates an instance p of type *three_tuple*$<A, B, C>$ and initializes it with the value (u, v, w).

3. Operations

A&	p.first( )	returns the A-component of p. If p is a const-object the return type is A.
B&	p.second( )	returns the B-component of p. If p is a const-object the return type is B.
C&	p.third( )	returns the C-component of p. If p is a const-object the return type is C.
bool	$p == q$	equality test for *three_tuples*. Each of the component types must have an equality operator.
int	compare(*three_tuple*$<A, B, C>$ p, *three_tuple*$<A, B, C>$ q)	lexicographic ordering for *three_tuples*. Each of the component types must have a compare function.
int	Hash(*three_tuple*$<A, B, C>$ p)	hash function for *three_tuples*. Each of the component types must have a Hash function.

4. Implementation

The obvious implementation is used.

3.18 Four Tuples (four_tuple)

1. Definition

An instance p of type $four\_tuple<A,B,C,D>$ is a four-tuple (a,b,c,d) of variables of types A, B, C, and D, respectively.

Related types are *two_tuple*, *three_tuple*, and *four_tuple*.

#include < *LEDA/tuple.h* >

2. Creation

$four\_tuple<A,B,C,D>$ p;

creates an instance p of type $four\_tuple<A,B,C,D>$. All components are initialized to their default value.

$four\_tuple<A,B,C,D>$ $p(A\ u,\ B\ v,\ C\ w,\ D\ x)$;

creates an instance p of type $four\_tuple<A,B,C,D>$ and initializes it with the value (u,v,w,x).

3. Operations

$A\&$	p.first( )	returns the A-component of p. If p is a const-object the return type is A.
$B\&$	p.second( )	returns the B-component of p. If p is a const-object the return type is B.
$C\&$	p.third( )	returns the C-component of p. If p is a const-object the return type is C.
$D\&$	p.fourth( )	returns the D-component of p. If p is a const-object the return type is D.
bool	$p == q$	equality test for *four_tuples*. Each of the component types must have an equality operator.
int	compare($four\_tuple<A,B,C,D>$ p, $four\_tuple<A,B,C,D>$ q)	lexicographic ordering for *four_tuples*. Each of the component types must have a compare function.
int	Hash($four\_tuple<A,B,C,D>$ p)	hash function for *four_tuples*. Each of the component types must have a Hash function.

4. Implementation

The obvious implementation is used.

5. Example

We customize *four_tuples* and define a *h_array* for them.

```
#define prio() first()
#define inf()  second()
#define pq_item() third()
#define part_item() fourth()
typedef four_tuple<int,int,int,int> my_qu;

my_qu q;
my_qu q1(2,2,0,0);
q.prio() = 5;

h_array<my_qu,int> M;
M[my_qu(2,2,nil,nil)] = 5;
```

Chapter 4

Number Types and Linear Algebra

4.1 Integers of Arbitrary Length (integer)

1. Definition

An instance a of the data type *integer* is an integer number of arbitrary length.

#include < *LEDA/integer.h* >

2. Creation

integer a;	creates an instance a of type *integer* and initializes it with zero.
integer a(*int* n);	creates an instance a of type *integer* and initializes it with the value of n.
integer a(*unsigned int* i);	creates an instance a of type *integer* and initializes it with the value of n.
integer a(*long* l);	creates an instance a of type *integer* and initializes it with the value of l.
integer a(*unsigned long* i);	creates an instance a of type *integer* and initializes it with the value of i.
integer a(*double* x);	creates an instance a of type *integer* and initializes it with the integral part of x.
integer a(*int* sz, *const digit* $*$ *vec*);	creates an instance a of type *integer* and initializes it with the value represented by the first sz digits *vec*.

integer $a(char * s)$;	a creates an instance a of type *integer* from its decimal representation given by the string s.
integer $a(number\_string\ s)$;	a creates an instance a of type *integer* from its decimal representation given by the string s.

3. Operations

The arithmetic operations $+$, $-$, $*$, $/$, $+=$, $-=$, $*=$, $/=$, $-$(unary), $++$, $--$, the modulus operation ($\%$, $\%=$), bitwise AND ($\&$, $\&=$), bitwise OR ($|$, $|=$), the complement (˜), the shift operations ($<<$, $>>$), the comparison operations $<$, $<=$, $>$, $>=$, $==$, $!=$ and the stream operations all are available.

int	a.length( )	returns the number of bits of the representation of a.
bool	a.is_long( )	returns whether a fits in the data type *long*.
long	a.to_long( )	returns a *long* number which is initialized with the value of a. *Precondition*: a.is_long() is *true*.
double	a.to_double( )	returns a double floating point approximation of a.
number_string	a.to_string( )	returns the decimal representation of a.
integer&	a.from_string(*number_string s*)	sets a to the number that has decimal respresentation s.
sz_t	a.used_words( )	returns the length of the digit vector that represents a.
digit	a.highword( )	returns the most significant digit of a.
digit	a.contents(*int i*)	returns the i-th digit of a.
void	a.hex_print(*ostream& o*)	prints the digit vector that represents a in hex format to the output stream o.
bool	a.iszero( )	returns whether a is equal to zero.

Non-member functions

double	to_double(*integer a*)	returns a double floating point approximation of a.
integer	sqrt(*integer a*)	returns the largest *integer* which is not larger than the square root of a.
integer	abs(*integer a*)	returns the absolute value of a.

integer	gcd(*integer* a, *integer* b)	returns the greatest common divisor of a and b.
int	log(*integer* a)	returns the logarithm of a to the basis 2.
int	sign(*integer* a)	returns the sign of a.
integer	sqr(*integer* a)	returns a^2.
integer	*integer* :: random(*int* n)	returns a random integer of length n bits.

4. Implementation

An *integer* is essentially implemented by a vector *vec* of *unsigned long* numbers. The sign and the size are stored in extra variables. Some time critical functions are also implemented in sparc assembler code.

4.2 Rational Numbers (rational)

1. Definition

An instance q of type *rational* is a rational number where the numerator and the denominator are both of type *integer*.

#include < *LEDA/rational.h* >

2. Creation

rational q; creates an instance q of type *rational*.

rational q(*integer* n);

creates an instance q of type *rational* and initializes it with the integer n.

rational q(*integer* n, *integer* d);

creates an instance q of type *rational* and initializes its the rational number n/d.

rational q(*double* x);

creates an instance q of type *rational* and initializes it with the value of x.

3. Operations

The arithmetic operations $+$, $-$, $*$, $/$, $+=$, $-=$, $*=$, $/=$, $-$(unary), $++$, $--$, the comparison operations $<$, $<=$, $>$, $>=$, $==$, $!=$ and the stream operations are all available.

integer	q.numerator( )	returns the numerator of q.
integer	q.denominator( )	returns the denominator of q.
rational&	q.simplify(*integer* a)	simplifies q by a. *Precondition*: a divides the numerator and the denominator of q.
rational&	q.normalize( )	normalizes q.
void	q.negate( )	negates q.
void	q.invert( )	inverts q.
rational	q.inverse( )	returns the inverse of q.
double	to_double( )	returns a double floating point approximation of q.

string	q.to_string( )	returns a string representation of q.

Non-member functions

int	sign(*rational* q)	returns the sign of q.
rational	abs(*rational* q)	returns the absolute value of q.
rational	sqr(*rational* q)	returns the square of q.
integer	trunc(*rational* q)	returns the *integer* with the next smaller absolute value.
rational	pow(*rational* q, *int* n)	returns the n-th power of q.
rational	pow(*rational* q, *integer* a)	returns the a-th power of q.
integer	floor(*rational* q)	returns the next smaller *integer*.
integer	ceil(*rational* q)	returns the next bigger *integer*.
integer	round(*rational* q)	rounds q to the nearest *integer*.
double	Double(*rational* q)	returns a double floating point approximation of q.
rational	small_rational_between(*rational* p, *rational* q)	returns a rational number between p and q whose denominator is as small as possible.
rational	small_rational_near(*rational* p, *rational* *eps*)	returns a rational number between $p - eps$ and $p + eps$ whose denominator is as small as possible.

4. Implementation

A *rational* is implemented by two *integer* numbers which represent the numerator and the denominator. The sign is represented by the sign of the numerator.

4.3 Big Floatingpoint Numbers (bigfloat)

1. Definition

In general a *bigfloat* is given by two integers s and e where s is the significant and e is the exponent. The tuple (s, e) represents the real number $s \cdot 2^e$. In addition, there are the special *bigfloat* values *NaN* (not a number), *pZero*, *nZero* $(= +0, -0)$, and *pInf*, *nInf* $(= +\infty, -\infty)$. These special values behave as defined by the IEEE floating point standard.

In particular, $5/+0 = \infty$, $-5/+0 = -\infty$, $\infty + 1 = \infty$, $5/\infty = +0$, $+\infty + (-\infty) = NaN$ and $0 \cdot \infty = NaN$.

Arithmetic on *bigfloat*s uses two parameters. The precision *prec* of the result (in number of binary digits) and the rounding mode *mode*. Possible rounding modes are:

- *TO_NEAREST*: round to the closest representable value
- *TO_ZERO*: round towards zero
- *TO_INF*: round away from zero
- *TO_P_INF*: round towards $+\infty$
- *TO_N_INF*: round towards $-\infty$
- *EXACT*: compute exactly for $+, -, *$ and round to nearest otherwise

Operations $+$, $-$, $*$ work as follows. First, the exact result z is computed. If the rounding mode is EXACT then z is the result of the operation. Otherwise, let s be the significant of the result; s is rounded to *prec* binary places as dictated by *mode*. Operations / and $\sqrt{\ }$ work accordingly except that EXACT is treated as TO_NEAREST.

The parameters *prec* and *mode* are either set directly for a single operation or else they are set globally for every operation to follow. The default values are 53 for *prec* and TO_NEAREST for *mode*.

#include < *LEDA/bigfloat.h* >

2. Creation

A *bigfloat* may be constructed from data types *double*, *long*, *int* and *integer*, without loss of accuracy. In addition, an instance of type *bigfloat* can be created as follows.

bigfloat x(*integer* s, *integer* e);

introduces a variable x of type bigfloat and initializes it to $s \cdot 2^e$

3. Operations

The arithmetical operators $+$, $-$, $*$, $/$, $+=$, $-=$, $*=$, $/=$, *sqrt*, the comparison operators $<$, $\leq$, $>$, $\geq$, $=$, $\neq$ and the stream operators $\ll$ and $\gg$ are available. Addition, subtraction, multiplication, division and square root are implemented by the functions *add*, *sub*, *mul*, *div* and *sqrt*, respectively. For example, the call

$$add(x,\ y,\ prec,\ mode,\ is\_exact)$$

computes the sum of bigfloats x and y with *prec* binary digits, in rounding mode *mode*, and returns it. The optional last parameter *is_exact* is a boolean variable that is set to *true* if and only if the returned bigfloat exactly equals the sum of x and y. The parameters *prec* and *mode* are also optional and have the global default values *global_prec* and *round_mode* respectively, that is, the three calls $add(x,\ y,\ global\_prec,\ round\_mode)$, $add(x,\ y,\ global\_prec)$, and $add(x,\ y)$ are all equivalent. The syntax for functions *sub*, *mul*, *div*, and *sqrt* is analogous.

The operators $+$, $-$, $*$, and $/$ are implemented by their counterparts among the functions *add*, *sub*, *mul* and *div*. For example, the call $x + y$ is equivalent to $add(x, y)$.

A *bigfloat* x can be rounded by the call $round(x, prec, mode, is\_exact)$. The optional boolean variable *is_exact* is set to true if and only if the rounding operation did not change the value of x.

*bigfloat*s offer a small set of mathematical functions (e.g. *abs*, *log2*, *ceil*, *floor*, *sign*), functions to test for special values, conversions to *double*s and *integer*s, functions to access *significant* and *exponent*, and functions to set the global precision, the rounding mode and the output mode.

bool	isNaN(*bigfloat* x)	returns *true* if and only if x is in special state *NaN*
bool	isnInf(*bigfloat* x)	returns *true* if and only if x is in special state *nInf*
bool	ispInf(*bigfloat* x)	returns *true* if and only if x is in special state *pInf*
bool	isnZero(*bigfloat* x)	returns *true* if and only if x is in special state *nZero*
bool	ispZero(*bigfloat* x)	returns *true* if and only if x is in special state *pZero*
bool	isZero(*bigfloat* x)	returns *true* if and only if $ispZero(x)$ or $isnZero(x)$
bool	isInf(*bigfloat* x)	returns *true* if and only if $ispInf(x)$ or $isnInf(x)$
bool	isSpecial(*bigfloat* x)	returns *true* if and only if x is in a special state
long	sign(*bigfloat* x)	returns the sign of x.

bigfloat abs(*bigfloat* x) returns the absolute value of x

bigfloat pow2(*integer* p) returns 2^p

bigfloat sqrt_d(*bigfloat* x, *long* p, *int* $d =$ 2)

returns $\sqrt[d]{x}$, with relative error $\leq 2^{-p}$ but not necessarily exactly rounded to p binary digits

integer ilog2(*bigfloat* x) returns the binary logarithm of abs(x), rounded up to the next integer.
Precondition: $x \neq 0$

integer ceil(*bigfloat* x) returns x, rounded up to the next integer

integer floor(*bigfloat* x) returns x, rounded down to the next integer

integer to_integer(*bigfloat* x, *rounding_modes rmode* = *TO_NEAREST*)

returns the integer value next to x (in the given rounding mode)

double to_double(*bigfloat* x)

returns the double value next to x (in rounding mode *TO_NEAREST*

ostream& *ostream*& *os* $\ll$ x

writes x to output stream *os*

istream& *istream*& *is* $\gg$ *bigfloat*& x

reads x from input stream *is* in decimal format

string to_string( ) returns a string representation of x.

long x.get_significant_length(*void*)

returns the length of the significant of x

integer x.get_exponent(*void*)

returns the exponent of x

integer x.get_significant(*void*)

returns the significant of x

long *bigfloat*::set_precision(*long* p)

sets the global precision to p and returns the old precision

rounding_modes *bigfloat*::set_rounding_mode(*rounding_modes* m)

sets the global rounding mode to m and returns the old rounding mode *output_modes bigfloat* :: set_output_mode(*output_modes o_mode*)

sets the output mode to *o_mode* and returns the old output mode

4.4 Algebraic Real Numbers (real)

1. Definition

An instance x of the data type real is an algebraic real number. There are many ways to construct a real: either by conversion from *double*, *bigfloat*, *integer* or *rational* or by applying one of the arithmetic operators $+, -, *, /$ or $\sqrt[d]{}$ to real numbers. One may test the sign of a real number or compare two real numbers by any of the comparison relations $=, \neq, <, \leq, >$ and $\geq$. The outcome of such a test is *exact*. There is also a non–standard version of the sign function: the call $x.sign(integer\ q)$ computes the sign of x under the precondition that $|x| \leq 2^{-q}$ implies $x = 0$. This version of the sign function allows the user to assist the data type in the computation of the sign of x, see the example below.

There are several functions to compute approximations of reals. The calls $to\_bigfloat(x)$ and $x.get\_bigfloat\_error(\ )$ return *bigfloat*s $xnum$ and $xerr$ such that $|xnum - x| \leq xerr$. The user may set a bound on $xerr$. More precisely, after the call $x.improve\_approximation\_to(integer\ q)$ the data type guarantees $xerr \leq 2^{-q}$. One can also ask for *double* approximations of a real number x. The calls $to\_double(x)$ and $x.get\_double\_error(\ )$ return *double*s $xnum$ and $xerr$ such that $|xnum - x| \leq |xnum| * xerr$. Note that $xerr = \infty$ is possible.

#include < *LEDA/real.h* >

2. Creation

reals may be constructed from data types *double*, *bigfloat*, *long*, *int* and *integer*. The default constructor *real*( ) initializes the real to zero.

3. Operations

double to_double(*real*) returns the current double approximation of x.

bigfloat to_bigfloat(*real*) returns the current bigfloat approximation of x.

double *x*.get_double_error( )

returns the relative error of the current double approximation of x, i.e., $|x - to\_double(x)|$ is bounded by $x.get\_double\_error(\ ) \cdot |to\_double(x)|$.

bigfloat *x*.get_bigfloat_error( )

returns the absolute error of the current bigfloat approximation of x, i.e., $|x - to\_bigfloat(x)|$ is bounded by $x.get\_bigfloat\_error(\ )$.

int sign(*real x*) returns the sign of (the exact value of) x.

int x.sign(*integer* q) as above.
Precondition: if $|x| \le 2^{-q}$ then $x = 0$.

void x.improve_approximation_to(*integer* q)
(re-)computes the approximation of x such that *x.get_bigfloat_error*( ) $\le 2^{-q}$ after the call *x.improve_approximation_to*(q).

void x.compute_with_precision(*long* k)
(re-)computes the approximation of x; each numerical operation is carried out with k binary places.

void x.guarantee_relative_error(*long* k)
(re-)computes an approximation of x such that the relative error of the approximation is less than 2^{-k}.

ostream& *ostream*& $O \ll x$
writes the closest interval that is known to contain x to the output stream O

istream& *istream*& $I \gg$ *real*& x
reads *real* number x from the output stream I (in *double* format)

real sqrt(*real* x) $\sqrt{x}$

real root(*real* x, *int* d) $\sqrt[d]{x}$
Precondition: $d \ge 2$

real abs(*real* x) absolute value of x

real sqr(*real* x) square of x

real dist(*real* x, *real* y) euclidean distance of vector (x,y) to the origin

real powi(*real* x, *int* n)
n.th power of x

4. Implementation

A *real* is represented by the expression which defines it and a *double* approximation x' together with a relative error bound ϵ_x. The arithmetic operators $+$, $-$, $*$, $/$, $\sqrt[d]{\ }$ take constant time. When the sign of a *real* number needs to be determined, the data type first computes a number q, if not already given as an argument to *sign*, such that $|x| \le 2^{-q}$ implies $x = 0$. The bound q is computed as described in [55].

Using *bigfloat* artihmetic, the data type then computes an interval I of maximal length

2^{-q} that contains x. If I contains zero, then x itself is equal to zero. Otherwise, the sign of any point in I is returned as the sign of x.

Two shortcuts are used to speed up the computation of the sign. Firstly, if the *double* approximation already suffices to determine the sign, then no *bigfloat* approximation is computed at all. Secondly, the *bigfloat* approximation is first computed only with small precision. The precision is then doubled until either the sign can be decided (i.e., if the current approximation interval does not contain zero) or the full precision 2^{-q} is reached.

5. Example

We give two examples of the use of the data type *real*. The examples deal with the Voronoi diagram of line segments and the intersection of line segments, respectively.

The following incircle test arises in the computation of Voronoi diagrams of line segments [15, 13]. For i, $1 \leq i \leq 3$, let $l_i : a_i x + b_i y + c_i = 0$ be a line in two-dimensional space and let $p = (0,0)$ be the origin. In general, there are two circles passing through p and touching l_1 and l_2. The centers of these circles have homogeneuos coordinates (x_v, y_v, z_v), where

$$\begin{aligned} x_v &= a_1c_2 + a_2c_1 \pm \mathrm{sign}(s)\sqrt{2c_1c_2(\sqrt{N}+D)} \\ y_v &= b_1c_2 + b_2c_1 \pm \mathrm{sign}(r)\sqrt{2c_1c_2(\sqrt{N}-D)} \\ z_v &= \sqrt{N} - a_1a_2 - b_1b_2 \end{aligned}$$

and

$$\begin{aligned} s &= b_1D_2 - b_2D_1, \quad N = (a_1^2+b_1^2)(a_2^2+b_2^2) \\ r &= a_1D_2 - a_2D_1, \quad D = a_1a_2 - b_1b_2. \end{aligned}$$

Let us concentrate on one of these (say, we take the plus sign in both cases). The test whether l_3 intersects, touches or misses the circle amounts to determining the sign of

$$E := dist^2(v, l_3) - dist^2(v, p) = \frac{(a_3x_v + b_3y_v + c_3)^2}{a_3^2 + b_3^2} - (x_v^2 + y_v^2).$$

The following program computes the sign of $\tilde{E} := (a_3^2 + b_3^2) \cdot E$ using our data type real.

int INCIRCLE(**real** a_1, **real** b_1, **real** c_1, **real** a_2, **real** b_2, **real** c_2, **real** a_3, **real** b_3, **real** c_3)
{
  real $RN = \mathrm{sqrt}((a_1 * a_1 + b_1 * b_1) * (a_2 * a_2 + b_2 * b_2));$

real $RN_1 = \text{sqrt}(a_1 * a_1 + b_1 * b_1)$;
real $RN_2 = \text{sqrt}(a_2 * a_2 + b_2 * b_2)$;
real $A = a_1 * c_2 + a_2 * c_1$;
real $B = b_1 * c_2 + b_2 * c_1$;
real $C = 2 * c_1 * c_2$;
real $D = a_1 * a_2 - b_1 * b_2$;
real $s = b_1 * RN_2 - b_2 * RN_1$;
real $r = a_1 * RN_2 - a_2 * RN_1$;
int $sign_x = \text{sign}(s)$;
int $sign_y = \text{sign}(r)$;
real $x_v = A + sign_x * \text{sqrt}(C * (RN + D))$;
real $y_v = B - sign_y * \text{sqrt}(C * (RN - D))$;
real $z_v = RN - (a_1 * a_2 + b_1 * b_2)$;
real $P = a_3 * x_v + b_3 * y_v + c_3 * z_v$;
real $D_3^2 = a_3 * a_3 + b_3 * b_3$;
real $R^2 = x_v * x_v + y_v * y_v$;
real $E = P * P - D_3^2 * R^2$;
return $\text{sign}(E)$;
}

We can make the above program more efficient if all coefficients a_i, b_i and c_i, $1 \leq i \leq 3$, are k bit integers, i.e., integers whose absolute value is bounded by $2^k - 1$. In [15, 13] we showed that for $\tilde{E} \neq 0$ we have $|\tilde{E}| \geq 2^{-24k-26}$. Hence we may add a parameter *int* k in the above program and replace the last line by

$$\textbf{return } E.\text{sign}(24 * k + 26).$$

Without this assistance, *real*s automatically compute a weaker bound of $|\tilde{E}| \geq 2^{-56k-161}$ for $\tilde{E} \neq 0$ by [14].

We turn to the line segment intersection problem next. Assume that all endpoints have k–bit integer homogeneous coordinates. This implies that the intersection points have homogeneous coordinates (X, Y, W) where X, Y and W are (4 k + 3) - bit integers. The Bentley–Ottmann plane sweep algorithm for segment intersection [59] needs to sort points by their x–coordinates, i.e., to compare fractions X_1/W_1 and X_2/W_2 where X_1, X_2, W_1, W_2 are as above. This is tantamount to determining the sign of the $8k + 7$ bit integer $X_1 * W_2 - X_2 * W_1$. If all variables X_i, W_i are declared **real** then their sign test will be performed quite efficiently. First, a *double* approximation is computed and then, if necessary, *bigfloat* approximations of increasing precision. In many cases, the *double* approximation already determines the sign. In this way, the user of the data type *real* gets the efficiency of a floating point filter [31, 60] without any work on his side. This is in marked contrast to [31, 60] and will be incorporated into [59].

4.5 A Floating Point Filter (floatf)

1. Definition

The type $floatf$ provides a clean and efficient way to approximately compute with large integers. Consider an expression E with integer operands and operators $+, -$, and $*$, and suppose that we want to determine the sign of E. In general, the integer arithmetic provided by our machines does not suffice to evaluate E since intermediate results might overflow. Resorting to arbitrary precision integer arithmetic is a costly process. An alternative is to evaluate the expression using floating point arithmetic, i.e., to convert the operands to doubles and to use floating-point addition, subtraction, and multiplication.

Of course, only an approximation E' of the true value E is computed. However, E' might still be able to tell us something about the sign of E. If E' is far away from zero (the forward error analysis carried out in the next section gives a precise meaning to "far away") then the signs of E' and E agree and if E' is zero then we may be able to conclude under certain circumstances that E is zero. Again, forward error analysis can be used to say what 'certain circumstances' are.

The type $floatf$ encapsulates this kind of approximate integer arithmetic. Any integer (= object of type *integer*) can be converted to a $floatf$; $floatf$s can be added, subtracted, multiplied, and their sign can be computed: for any $floatf\ x$ the function $Sign(x)$ returns either the sign of x (-1 if $x < 0$, 0 if $x = 0$, and $+1$ if $x > 0$) or the special value NO_IDEA. If x approximates X, i.e., X is the integer value obtained by an exact computation, then $Sign(x)! = NO\_IDEA$ implies that $Sign(x)$ is actually the sign of X if $Sign(x) = NO\_IDEA$ then no claim is made about the sign of X.

#include < *LEDA/floatf.h* >

2. Creation

floatf x;	introduces a variable x of type *floatf* and initializes it with zero.
floatf x(*integer* i);	introduces a variable x of type *floatf* and initializes it with integer i.

3. Operations

floatf	$a + b$	Addition.
floatf	$a - b$	Subtraction.
floatf	$a * b$	Multiplication.
int	Sign(*floatf* f)	as described above.

4. Implementation

A $floatf$ is represented by a double (its value) and an error bound. An operation on $floatf$s performs the corresponding operation on the values and also computes the error bound for the result. For this reason the cost of a $floatf$ operation is about four times the cost of the corresponding operation on doubles. The rules used to compute the error bounds are described in ([59]).

5. Example

see [59] for an application in a sweep line algorithm.

4.6 Real-Valued Vectors (vector)

1. Definition

An instance of data type *vector* is a vector of variables of type *double*.

#include < *LEDA/vector.h* >

2. Creation

vector v; creates an instance v of type *vector*; v is initialized to the zero-dimensional vector.

vector v(*int* d); creates an instance v of type *vector*; v is initialized to the zero vector of dimension d.

vector v(*double* a, *double* b);
creates an instance v of type *vector*; v is initialized to the two-dimensional vector (a, b).

vector v(*double* a, *double* b, *double* c);
creates an instance v of type *vector*; v is initialized to the three-dimensional vector (a, b, c).

vector v(*vector* w, *int prec*);
creates an instance v of type *vector*; v is initialized to a copy of w. The second argument is for compatibility with *rat_vector*.

3. Operations

int	v.dim( )	returns the dimension of v.
double&	v[*int* i]	returns i-th component of v. *Precondition*: $0 \le i \le v.\mathrm{dim}()-1$.
double	v.hcoord(*int* i)	for compatibility with *rat_vector*.
double	v.coord(*int* i)	for compatibility with *rat_vector*.
double	v.sqr_length( )	returns the square of the Euclidean length of v.
double	v.length( )	returns the Euclidean length of v.
vector	v.norm( )	returns v normalized.
double	v.angle(*vector* w)	returns the angle between v and w.

vector	v.rotate90( )	returns the v rotated by 90 degrees. *Precondition*: $v.dim() = 2$
vector	v.rotate(*double* a)	returns the v rotated by an angle of a. *Precondition*: $v.dim() = 2$
vector	$v + v1$	Addition. *Precondition*: v.dim() $=$ $v1$.dim().
vector	$v - v1$	Subtraction. *Precondition*: v.dim() $=$ $v1$.dim().
double	$v * v1$	Scalar multiplication. *Precondition*: v.dim() $=$ $v1$.dim().
vector	$v *$ *double* r	Componentwise multiplication with double r.
bool	$v == w$	Test for equality.
bool	v != w	Test for inequality.
void	v.print(*ostream*& O)	prints v componentwise to ostream O.
void	v.print( )	prints v to *cout*.
void	v.read(*istream*& I)	reads $d = v.dim(\,)$ numbers from input stream I and writes them into $v[0] \ldots v[d-1]$.
void	v.read( )	reads v from *cin*.
ostream&	*ostream*& $O \ll v$	writes v componentwise to the output stream O.
istream&	*istream*& $I \gg$ *vector*& v	reads v componentwise from the input stream I.

Additional Operations for vectors in two and three-dimensional space

double	v.xcoord( )	returns the zero-th cartesian coordinate of v.
double	v.ycoord( )	returns the first cartesian coordinate of v.
double	v.zcoord( )	returns the second cartesian coordinate of v.
int	compare_by_angle(*vector* $v1$, *vector* $v2$)	For a non-zero vector v let $\alpha(v)$ be the angle by which the positive x-axis has to be turned counterclockwise until it aligns with v. The function compares the angles defined by $v1$ and $v2$, respectively. The zero-vector precedes all non-zero vectors in the angle-order.

4. Implementation

Vectors are implemented by arrays of real numbers. All operations on a vector v take time $O(v.dim())$, except for dim and [ ] which take constant time. The space requirement is $O(v.dim())$.

Be aware that the operations on vectors and matrices incur rounding errors and hence are not completely reliable. For example, if M is a matrix, b is a vector, and x is computed by $x = M.solve(b)$ it is not necessarily true that the test $b == M * b$ evaluates to true. The types *integer_vector* and *integer_matrix* provide exact linear algebra.

4.7 Real-Valued Matrices (matrix)

1. Definition

An instance of the data type *matrix* is a matrix of variables of type *double*.

#include < *LEDA/matrix.h* >

2. Creation

matrix $M(int\ n = 0,\ int\ m = 0);$

creates an instance M of type *matrix*, M is initialized to the $n \times m$ - zero matrix.

matrix $M(int\ n,\ int\ m,\ double * D);$

creates the $n \times m$ matrix M with $M(i,j) = D[i * m + j]$ for $0 \leq i \leq n-1$ and $0 \leq j \leq m-1$. *Precondition*: D points to an array of at least $n * m$ numbers of type *double*.

3. Operations

int	M.dim1( )	returns n, the number of rows of M.
int	M.dim2( )	returns m, the number of columns of M.
vector&	M.row(*int* i)	returns the i-th row of M (an m-vector). *Precondition*: $0 \leq i \leq n-1$.
vector	M.col(*int* i)	returns the i-th column of M (an n-vector). *Precondition*: $0 \leq i \leq m-1$.
matrix	M.trans( )	returns M^T ($m \times n$ - matrix).
matrix	M.inv( )	returns the inverse matrix of M. *Precondition*: M.det() $\neq 0$.
double	M.det( )	returns the determinant of M. *Precondition*: M is quadratic.
vector	M.solve(*vector* b)	returns vector x with $M \cdot x = b$. *Precondition*: M.dim1() == M.dim2() = =b.dim() and M.det() $\neq 0$.
double&	M(*int* i, *int* j)	returns $M_{i,j}$. *Precondition*: $0 \leq i \leq n-1$ and $0 \leq j \leq m-1$.
matrix	M + *M1*	Addition. *Precondition*: M.dim1() == $M1$.dim1() and M.dim2() == $M1$.dim2().

matrix	$M - M1$	Subtraction. *Precondition*: M.dim1() == $M1$.dim1() and M.dim2() == $M1$.dim2().
matrix	$M * M1$	Multiplication. *Precondition*: M.dim2() == $M1$.dim1().
vector	M * *vector vec*	Multiplication with vector. *Precondition*: M.dim2() == *vec*.dim().
matrix	M * *double x*	Multiplication with double x.
void	M.print(*ostream*& O)	prints M row by row to ostream O.
void	M.print( )	prints M *cout*.
void	M.read(*istream*& I)	reads $M.dim1(\ ) \times M.dim2(\ )$ numbers from input stream I and writes them row by row into matrix M.
void	M.read( )	prints M from *cin*.
ostream&	*ostream*& $O \ll M$	writes matrix M row by row to the output stream O.
istream&	*istream*& $I \gg$ *matrix*& M	reads a matrix row by row from the input stream I and assigns it to M.

4. Implementation

Data type *matrix* is implemented by two-dimensional arrays of double numbers. Operations det, solve, and inv take time $O(n^3)$, dim1, dim2, row, and col take constant time, all other operations take time $O(nm)$. The space requirement is $O(nm)$.

Be aware that the operations on vectors and matrices incur rounding error and hence are not completely reliable. For example, if M is a matrix, b is a vector, and x is computed by $x = M.solve(b)$ it is not necessarly true that the test $b == M * b$ evaluates to true. The types *integer_vector* and *integer_matrix* provide exact linear algebra.

4.8 Vectors with Integer Entries (integer_vector)

1. Definition

An instance of data type *integer_vector* is a vector of variables of type *integer*, the so called ring type. Together with the type *integer_matrix* it realizes the basic operations of linear algebra. Internal correctness tests are executed if compiled with the flag `LA_SELFTEST`.

#include < *LEDA/integer_vector.h* >

2. Creation

integer_vector v;

creates an instance v of type *integer_vector*. v is initialized to the zero-dimensional vector.

integer_vector v(*int* d);

creates an instance v of type *integer_vector*. v is initialized to a vector of dimension d.

integer_vector v(*integer* a, *integer* b);

creates an instance v of type *integer_vector*. v is initialized to the two-dimensional vector (a, b).

integer_vector v(*integer* a, *integer* b, *integer* c);

creates an instance v of type *integer_vector*. v is initialized to the three-dimensional vector (a, b, c).

integer_vector v(*integer* a, *integer* b, *integer* c, *integer* d);

creates an instance v of type *integer_vector*; v is initialized to the four-dimensional vector (a, b, c, d).

3. Operations

int	v.dim( )	returns the dimension of v.
integer&	v[*int* i]	returns i-th component of v. *Precondition*: $0 \le i \le v.dim(\,) - 1$.
integer_vector&	v += *v1*	Addition plus assignment. *Precondition*: $v.dim(\,) == v1.dim(\,)$.
integer_vector&	v −= *v1*	Subtraction plus assignment. *Precondition*: $v.dim(\,) == v1.dim(\,)$.
integer_vector	$v + v1$	Addition. *Precondition*: $v.dim(\,) == v1.dim(\,)$.
integer_vector	$v - v1$	Subtraction. *Precondition*: $v.dim(\,) == v1.dim(\,)$.

integer	$v * v1$	Inner Product. *Precondition*: $v.dim(\,) == v1.dim(\,)$.
integer_vector	*integer* $r * v$	Componentwise multiplication with number r.
integer_vector	$v *$ *integer* r	Componentwise multiplication with number r.
ostream&	*ostream&* $O \ll v$	writes v componentwise to the output stream O.
istream&	*istream&* $I \gg$ *integer_vector&* v	reads v componentwise from the input stream I.

4. Implementation

Vectors are implemented by arrays of type *integer*. All operations on a vector v take time $O(v.dim(\,))$, except for *dimension* and [ ] which take constant time. The space requirement is $O(v.dim(\,))$.

4.9 Matrices with Integer Entries (integer_matrix)

1. Definition

An instance of data type *integer_matrix* is a matrix of variables of type *integer*, the so called ring type. The arithmetic type *integer* is required to behave like integers in the mathematical sense.

The types *integer_matrix* and *integer_vector* together realize many functions of basic linear algebra. All functions on integer matrices compute the exact result, i.e., there is no rounding error. Most functions of linear algebra are *checkable*, i.e., the programs can be asked for a proof that their output is correct. For example, if the linear system solver declares a linear system $Ax = b$ unsolvable it also returns a vector c such that $c^T A = 0$ and $c^T b \neq 0$. All internal correctness checks can be switched on by the flag `LA_SELFTEST`. Preconditions are checked by default and can be switched off by the compile flag `LEDA_CHECKING_OFF`.

#include < *LEDA/integer_matrix.h* >

2. Creation

integer_matrix $M(int\ n,\ int\ m);$

creates an instance M of type *integer_matrix* of dimension $n \times m$.

integer_matrix $M(int\ n\ =\ 0);$

creates an instance M of type *integer_matrix* of dimension $n \times n$.

integer_matrix $M($*array*< *integer_vector* > $A);$

creates an instance M of type *integer_matrix*. Let A be an array of m column - vectors of common dimension n. M is initialized to an $n \times m$ matrix with the columns as specified by A.

integer_matrix *integer_matrix*::identity(*int* n)

returns an identity matrix of dimension n.

3. Operations

int	M.dim1( )	returns n, the number of rows of M.
int	M.dim2( )	returns m, the number of columns of M.
integer_vector&	M.row(*int* i)	returns the i-th row of M (an m - vector). *Precondition*: $0 \leq i \leq n-1$.
integer_vector	M.col(*int* i)	returns the i-th column of M (an n - vector). *Precondition*: $0 \leq i \leq m-1$.

integer& $M(int\ i,\ int\ j)$ returns $M_{i,j}$.
Precondition: $0 \leq i \leq n-1$ and $0 \leq j \leq m-1$.

Arithmetic Operators

integer_matrix $M + M1$ Addition.
Precondition:
M.dim1() == $M1$.dim1() and M.dim2() == $M1$.dim2().

integer_matrix $M - M1$ Subtraction.
Precondition:
M.dim1() == $M1$.dim1() and M.dim2() == $M1$.dim2().

integer_matrix $M * M1$ Multiplication.
Precondition:
M.dim2() == $M1$.dim1().

integer_vector $M *$ *integer_vector vec*
Multiplication with vector.
Precondition:
M.dim2() == *vec*.dim().

integer_matrix $M *$ *integer* x Multiplication of every entry with integer x.

integer_matrix *integer* $x * M$ Multiplication of every entry with integer x.

Non-Member Functions

integer_matrix transpose(*integer_matrix M*)
returns M^T ($m \times n$ - matrix).

integer_matrix inverse(*integer_matrix M*, *integer& D*)
returns the inverse matrix of M. More precisely, $1/D$ times the matrix returned is the inverse of M.
Precondition: determinant(M) $\neq 0$.

bool inverse(*integer_matrix M*, *integer_matrix& inverse*, *integer& D*, *integer_vector& c*)
determines whether M has an inverse. It also computes either the inverse as $(1/D) \cdot inverse$ or a vector c such that $c^T \cdot M = 0$.

integer determinant(*integer_matrix* M, *integer_matrix*& L, *integer_matrix*& U, *array*<*int*>& q, *integer_vector*& c)

returns the determinant D of M and sufficient information to verify that the value of the determinant is correct. If the determinant is zero then c is a vector such that $c^T \cdot M = 0$. If the determinant is non-zero then L and U are lower and upper diagonal matrices respectively and q encodes a permutation matrix Q with $Q(i,j) = 1$ iff $i = q(j)$ such that $L \cdot M \cdot Q = U$, $L(0,0) = 1$, $L(i,i) = U(i-1,i-1)$ for all i, $1 \leq i < n$, and $D = s \cdot U(n-1,n-1)$ where s is the determinant of Q.
Precondition: M is quadratic.

bool verify_determinant(*integer_matrix* M, *integer* D, *integer_matrix*& L, *integer_matrix*& U, *array*<*int*> q, *integer_vector*& c)

verifies the conditions stated above.

integer determinant(*integer_matrix* M)

returns the determinant of M.
Precondition: M is quadratic.

int sign_of_determinant(*integer_matrix* M)

returns the sign of the determinant of M.
Precondition: M is quadratic.

bool linear_solver(*integer_matrix* M, *integer_vector* b, *integer_vector*& x, *integer*& D, *integer_matrix*& *spanning_vectors*, *integer_vector*& c)

determines the complete solution space of the linear system $M \cdot x = b$. If the system is unsolvable then $c^T \cdot M = 0$ and $c^T \cdot b \neq 0$. If the system is solvable then $(1/D)x$ is a solution, and the columns of *spanning_vectors* are a maximal set of linearly independent solutions to the corresponding homogeneous system.
Precondition: M.dim1() == b.dim().

bool linear_solver(*integer_matrix* M, *integer_vector* b, *integer_vector*& x, *integer*& D, *integer_vector*& c)

determines whether the linear system $M \cdot x = b$ is solvable. If yes, then $(1/D)x$ is a solution, if not then $c^T \cdot M = 0$ and $c^T \cdot b \neq 0$.
Precondition: M.dim1() == b.dim().

bool linear_solver(*integer_matrix* M, *integer_vector* b, *integer_vector*& x, *integer*& D)

as above, but without the witness c
Precondition: M.dim1() == b.dim().

bool is_solvable(*integer_matrix M*, *integer_vector b*)

determines whether the system $M \cdot x = b$ is solvable
Precondition: M.dim1() == b.dim().

bool homogeneous_linear_solver(*integer_matrix M*, *integer_vector& x*)

determines whether the homogeneous linear system $M \cdot x = 0$ has a non - trivial solution. If yes, then x is such a solution.

int homogeneous_linear_solver(*integer_matrix M*, *integer_matrix& spanning_vecs*)

determines the solution space of the homogeneous linear system $M \cdot x = 0$. It returns the dimension of the solution space. Moreover the columns of *spanning_vecs* span the solution space.

void independent_columns(*integer_matrix M*, *array<int>& columns*)

returns the indices of a maximal subset of independent columns of M. The index range of *columns* starts at 0.

int rank(*integer_matrix M*)

returns the rank of matrix M

ostream& *ostream& O* $\ll$ *M*

writes matrix M row by row to the output stream O.

istream& *istream& I* $\gg$ *integer_matrix& M*

reads matrix M row by row from the input stream I.

4. Implementation

The datatype *integer_matrix* is implemented by two-dimensional arrays of variables of type *integer*. Operations *determinant*, *inverse*, *linear_solver*, and *rank* take time $O(n^3)$, *column* takes time $O(n)$, *row*, *dim1*, *dim2*, take constant time, and all other operations take time $O(nm)$. The space requirement is $O(nm)$.

All functions on integer matrices compute the exact result, i.e., there is no rounding error. The implemenation follows a proposal of J. Edmonds (J. Edmonds, Systems of distinct representatives and linear algebra, Journal of Research of the Bureau of National Standards, (B), 71, 241 - 245). Most functions of linear algebra are *checkable* , i.e., the programs can be asked for a proof that their output is correct. For example, if the linear system solver declares a linear system $Ax = b$ unsolvable it also returns a vector c such that $c^T A = 0$ and $c^T b \neq 0$.

4.10 Rational Vectors (rat_vector)

1. Definition

An instance of data type *rat_vector* is a vector of rational numbers. A d-dimensional vector $r = (r_0, \ldots, r_{d-1})$ is represented in homogeneous coordinates $(h_0, \ldots, h_d)$, where $r_i = h_i/h_d$ and the h_i's are of type *integer*. We call the r_i's the cartesian coordinates of the vector. The homogenizing coordinate h_d is positive.

This data type is meant for use in computational geometry. It realizes free vectors as opposed to position vectors (type *rat_point*). The main difference between position vectors and free vectors is their behavior under affine transformations, e.g., free vectors are invariant under translations.

rat_vector is an item type.

#include < *LEDA/rat_vector.h* >

2. Creation

rat_vector $v(int\ d = 2)$; — introduces a variable v of type *rat_vector* initialized to the zero vector of dimension d.

rat_vector $v(integer\ a,\ integer\ b,\ integer\ D)$;

introduces a variable v of type *rat_vector* initialized to the two-dimensional vector with homogeneous representation (a, b, D) if D is positive and representation $(-a, -b, -D)$ if D is negative.
Precondition: D is non-zero.

rat_vector $v(rational\ x,\ rational\ y)$;

introduces a variable v of type *rat_vector* initialized to the two-dimensional vector with homogeneous representation (a, b, D), where $x = a/D$ and $y = b/D$.

rat_vector $v(integer\ a,\ integer\ b,\ integer\ c,\ integer\ D)$;

introduces a variable v of type *rat_vector* initialized to the two-dimensional vector with homogeneous representation (a, b, c, D) if D is positive and representation $(-a, -b, -c, -D)$ if D is negative.
Precondition: D is non-zero.

rat_vector $v(rational\ x,\ rational\ y,\ rational\ z)$;

introduces a variable v of type *rat_vector* initialized to the two-dimensional vector with homogeneous representation (a, b, c, D), where $x = a/D$, $y = b/D$ and $z = c/D$.

rat_vector v(*integer* a, *integer* b);

introduces a variable v of type *rat_vector* initialized to the two-dimensional vector with homogeneous representation $(a, b, 1)$.

rat_vector v(*integer_vector* c, *integer* D);

introduces a variable v of type *rat_vector* initialized to the vector with homogeneous coordinates $(\pm c_0, \ldots, \pm c_{d-1}, \pm D)$, where d is the dimension of c and the sign chosen is the sign of D.
Precondition: D is non-zero.

rat_vector v(*integer_vector* c);

introduces a variable v of type *rat_vector* initialized to the direction with homogeneous coordinate vector $\pm c$, where the sign chosen is the sign of the last component of c.
Precondition: The last component of c is non-zero.

rat_vector v(*vector* w, *int* *prec*);

introduces a variable v of type *rat_vector* initialized to $(\lfloor P * w_0 \rfloor, \ldots, \lfloor P * w_{d-1} \rfloor, P)$, where d is the dimension of w and $P = 2^{prec}$.

3. Operations

3.1 Initialization, Access and Conversions

rat_vector *rat_vector*::d2(*integer* a, *integer* b, *integer* D)

returns a *rat_vector* of dimension 2 initialized to a vector with homogeneous representation (a, b, D) if D is positive and representation $(-a, -b, -D)$ if D is negative.
Precondition: D is non-zero.

rat_vector *rat_vector*::d3(*integer* a, *integer* b, *integer* c, *integer* D)

returns a *rat_vector* of dimension 3 initialized to a vector with homogeneous representation (a, b, c, D) if D is positive and representation $(-a, -b, -c, -D)$ if D is negative.
Precondition: D is non-zero.

rat_vector *rat_vector*::unit(*int* i, *int* $d = 2$)

returns a *rat_vector* of dimension d initialized to the i-th unit vector.
Precondition: $0 \leq i < d$.

rat_vector	*rat_vector*::zero(*int* $d = 2$)	returns the zero vector in d-dimensional space.
int	v.dim( )	returns the dimension of v.
integer	v.hcoord(*int* i)	returns the i-th homogeneous coordinate of v.
rational	v.coord(*int* i)	returns the i-th cartesian coordinate of v.
rational	v[*int* i]	returns the i-th cartesian coordinate of v.
rational	v.sqr_length( )	returns the square of the length of v.
vector	v.to_float( )	returns a floating point approximation of v.

Additional Operations for vectors in two and three-dimensional space

rational	v.xcoord( )	returns the zero-th cartesian coordinate of v.
rational	v.ycoord( )	returns the first cartesian coordinate of v.
rational	v.zcoord( )	returns the second cartesian coordinate of v.
integer	v.X( )	returns the zero-th homogeneous coordinate of v.
integer	v.Y( )	returns the first homogeneous coordinate of v.
integer	v.Z( )	returns the second homogeneous coordinate of v.
integer	v.W( )	returns the homogenizing coordinate of v.
rat_vector	v.rotate90( )	returns the v rotated by 90 degrees. *Precondition*: $v.dim() == 2$.
int	compare_by_angle(*rat_vector v1*, *rat_vector v2*)	For a non-zero vector v let $\alpha(v)$ be the angle by which the positive x-axis has to be turned counter-clockwise until it aligns with v. The function compares the angles defined by *v1* and *v2*, respectively. The zero-vector precedes all non-zero vectors in the angle-order.

3.2 Tests

bool	$v == w$	Test for equality.
bool	$v\ != w$	Test for inequality.

3.3 Arithmetical Operators

rat_vector	*integer* $n * v$	multiplies all cartesian coordinates by n.
rat_vector	*rational* $r * v$	multiplies all cartesian coordinates by r.
rat_vector&	v *= *integer* n	multiplies all cartesian coordinates by n.
rat_vector&	v *= *rational* r	multiplies all cartesian coordinates by r.
rat_vector	v / *integer* n	divides all cartesian coordinates by n.
rat_vector	v / *rational* r	divides all cartesian coordinates by r.
rat_vector&	v /= *integer* n	divides all cartesian coordinates by n.
rat_vector&	v /= *rational* r	divides all cartesian coordinates by r.
rational	*const* $v * w$	scalar product, i.e., $\sum_{0 \le i < d} v_i w_i$, where v_i and w_i are the cartesian coordinates of v and w respectively.
rat_vector	$v + w$	adds cartesian coordinates.
rat_vector&	v += w	addition plus assignment.
rat_vector	$v - w$	subtracts cartesian coordinates.
rat_vector&	v −= w	subtraction plus assignment.
rat_vector	$-v$	returns -v.

3.4 Input and Output

ostream&	*ostream*& $O \ll v$	writes v's homogeneous coordinates componentwise to the output stream O.
istream&	*istream*& $I \gg$ *rat_vector*& v	reads v's homogeneous coordinates componentwise from the input stream I. The operator uses the current dimension of v.

3.5 Linear Hull, Dependence and Rank

bool	contained_in_linear_hull(*array*<*rat_vector*> A, *rat_vector* x)	determines whether x is contained in the linear hull of the vectors in A.

int linear_rank(*array*<*rat_vector*> A)

computes the linear rank of the vectors in A.

bool linearly_independent(*array*<*rat_vector*> A)

decides whether the vectors in A are linearly independent.

array<*rat_vector*> linear_base(*array*<*rat_vector*> A)

computes a basis of the linear space spanned by the vectors in A.

4. Implementation

Vectors are implemented by arrays of integers as an item type. All operations like creation, initialization, tests, vector arithmetic, input and output on an vector v take time $O(v.dim(\ ))$. $dim(\ )$, coordinate access and conversions take constant time. The operations for linear hull, rank and independence have the cubic costs of the used matrix operations. The space requirement is $O(v.dim(\ ))$.

4.11 Functions of Numerical Analysis ( )

We collect some functions of numerical analysis. *The algorithms in this section are not the best known and are not recommended for serious use.* We refer the reader to the book "Numerical Recipes in C: The Art of Scientific Computing" by B.P. Flannery, W.H. Press, S.A. Teukolsky, and W.T. Vetterling, Cambridge University Press for better algorithms.

The functions in this section become available by including *numerical_analysis.h*.

4.11.1 Minima and Maxima

double minimize_function(*double* (*f)(*double*), *double*& *xmin*, *double tol* = 1.0e − 10)

finds a local minimum of the function f of one argument. The minimizing argument is returned in *xmin* and the minimal function value is returned as the result of the function. *xmin* is determined with tolerance *tol*, i.e., the true value of the minimizing argument is contained in the interval $[xmin(1-\epsilon), xmin(1+\epsilon)]$, where $\epsilon = \max(1, xmin) \cdot tol$. Please do not choose *tol* smaller than 10^{-15}.
Precondition: : If $+\infty$ or $-\infty$ is a local minimum of f, then the call of *minimize_function* may not terminate.
The algorithm is implemented as follows: First three arguments are determined such that $a < b < c$ (or $a > b > c$) and $f(a) \geq f(b) \leq f(c)$, i.e., a and c bracket a minimum. The interval is found by first taking two arbitrary arguments and comparing their function values. The argument with the larger function value is taken as a. Then steps of larger and larger size starting at b are taken until a function value larger than $f(b)$ is found. Once the bracketing interval is found, golden-ratio search is applied to it.

double minimize_function(*F f*, *double*& *xmin*, *double tol* = 1.0e − 10)

a more flexible version of the above. It is assumed that class F offers the operator
double operator( )(*double x*). This operator is taken as the function f.

4.11.2 Integration

double integrate_function(*double* (*f)(*double*), *double l*, *double r*,
double delta = 1.0e − 2)

Computes the integral of f in the interval $[l, r]$ by forming the sum $delta * \sum_{0 \leq i < K} f(l + i \cdot delta)$, where $K = (r - l)/delta$.
Precondition: $l \leq r$ and $delta > 0$.

double integrate_function(*F f*, *double l*, *double r*, *double delta* $= 1.0e-2$)

a more flexible version of the above. It is assumed that class F offers the operator *double operator*( )(*double x*). This operator is taken as the function f.

4.11.3 Useful Numerical Functions

double binary_entropy(*double x*)

returns the binary entropy of x, i.e., $-x \cdot \log x - (1-x) \cdot \log(1-x)$.
Precondition: $0 \leq x \leq 1$.

4.11.4 Root Finding

double zero_of_function(*double* $(*f)$(*double*), *double l*, *double r*, *double tol* $= 1.0e-10$)

returns a zero x of f. We have either $|f(x)| \leq 10^{-10}$ or there is an interval $[x_0, x_1]$ containing x such that $f(x_0) \cdot f(x_1) \leq 0$ and $x_1 - x_0 \leq tol \cdot \max(1, |x_1| + |x_1|)$.
Precondition: $l \leq r$ and $f(l) \cdot f(r) \leq 0$.

double zero_of_function(*F f*, *double l*, *double r*, *double tol* $= 1.0e-10$)

a more flexible version of the above. It is assumed that class F offers the operator *double operator*( )(*double x*). This operator is taken as the function f.

Chapter 5

Basic Data Types

5.1 One Dimensional Arrays (array)

1. Definition

An instance A of the parameterized data type *array*<E> is a mapping from an interval $I = [a..b]$ of integers, the index set of A, to the set of variables of data type E, the element type of A. $A(i)$ is called the element at position i.

#include < *LEDA/array.h* >

2. Types

array<E>:: *item* — the item type.

array<E>:: *value_type* — the value type.

3. Creation

array<E> A(*int* a, *int* b);

creates an instance A of type *array*<E> with index set $[a..b]$.

array<E> A(*int* n);

creates an instance A of type *array*<E> with index set $[0..n-1]$.

array<E> A; creates an instance A of type *array*<E> with empty index set.

Special Constructors

array<E> A(*int* *low*, E x, E y);

creates an instance A of type *array*<E> with index set $[low, low+1]$ initialized to $[x, y]$.

array<*E*> A(*int low*, *E x*, *E y*, *E w*);

creates an instance A of type *array*<*E*> with index set $[low, low + 2]$ initialized to $[x, y, w]$.

array<*E*> A(*int low*, *E x*, *E y*, *E z*, *E w*);

creates an instance A of type *array*<*E*> with index set $[low, low + 3]$ initialized to $[x, y, z, w]$.

4. Operations

Basic Operations

const E&	$A[$*int x*$]$	returns $A(x)$. *Precondition*: $a \leq x \leq b$.
void	A.resize(*int a*, *int b*)	sets the index set of A to $[a..b]$ such that for all $i \in [a..b]$ which are not contained in the old index set $A(i)$ is set to the default value of type E.
void	A.resize(*int n*)	same as $A.resize(0, n-1)$.
int	A.low( )	returns the minimal index a of A.
int	A.high( )	returns the maximal index b of A.
int	A.size( )	returns the size $(b - a + 1)$ of A.
void	A.init(*E x*)	assigns x to $A[i]$ for every $i \in \{\ a \ldots b\ \}$.
bool	A.C_style( )	returns *true* if the array has "C-style", i.e., the index set is $[0..size - 1]$.
void	A.swap(*int i*, *int j*)	swaps the values of $A[i]$ and $A[j]$.
void	A.permute( )	the elements of A are randomly permuted.
void	A.permute(*int l*, *int h*)	the elements of A$[l..h]$ are randomly permuted.

Sorting and Searching

void	A.sort(*int* $(*cmp)(E, E\ )$)	sorts the elements of A, using function *cmp* to compare two elements, i.e., if $(in_a, \ldots, in_b)$ and $(out_a, \ldots, out_b)$ denote the values of the variables $(A(a), \ldots, A(b))$ before and after the call of sort, then $cmp(out_i, out_j) \leq 0$ for $i \leq j$ and there is a permutation π of $[a..b]$ such that $out_i = in_{\pi(i)}$ for $a \leq i \leq b$.

void	A.sort( )	sorts the elements of A according to the linear order of the element type E. *Precondition*: A linear order on E must have been defined by $compare(const E\&, const E\&)$.
void	A.sort(*int* (*cmp)(E, E), *int* l, *int* h)	sorts sub-array A[$l..h$] using compare function *cmp*.
void	A.sort(*int* l, *int* h)	sorts sub-array A[$l..h$] using the linear order on E.
int	A.binary_search(*int* (*cmp)(E, E), E x)	performs a binary search for x. Returns i with $A[i] = x$ if x in A, A.low() $-$ 1 otherwise. Function *cmp* is used to compare two elements. *Precondition*: A must be sorted according to *cmp*.
int	A.binary_search(E x)	as above but uses the default linear order on E.
int	A.binary_locate(*int* (*cmp)(E, E), E x)	Returns the maximal i with $A[i] \leq x$ or $A.low($ $)$-1 if $x < A[low]$. Function *cmp* is used to compare elements. *Precondition*: A must be sorted according to *cmp*.
int	A.binary_locate(E x)	as above but uses the default linear order on E.

Input and Output

void	A.read(*istream*& I)	reads $b-a+1$ objects of type E from the input stream I into the array A using the $operator \gg (istream\&, E\&)$.
void	A.read( )	calls A.read(*cin*) to read A from the standard input stream *cin*.
void	A.read(*string* s)	As above, uses string s as prompt.
void	A.print(*ostream*& O, *char* *space* = ' ')	prints the contents of array A to the output stream O using $operator \ll (ostream\&, const\ E\&)$ to print each element. The elements are separated by character *space*.
void	A.print(*char* *space* = ' ')	calls A.print(*cout*, *space*) to print A on the standard output stream *cout*.
void	A.print(*string* s, *char* *space* = ' ')	As above, uses string s as header.

Iteration

STL compatible iterators are provided when compiled with `-DLEDA_STL_ITERATORS` (see

LEDAROOT/demo/stl/array.c for an example).

5. Implementation

Arrays are implemented by C++vectors. The access operation takes time $O(1)$, the sorting is realized by quicksort (time $O(n \log n)$) and the binary_search operation takes time $O(\log n)$, where $n = b - a + 1$. The space requirement is $O(I * sizeof(E))$.

5.2 Two Dimensional Arrays (array2)

1. Definition

An instance A of the parameterized data type *array2*<*E*> is a mapping from a set of pairs $I = [a..b] \times [c..d]$, called the index set of A, to the set of variables of data type E, called the element type of A, for two fixed intervals of integers $[a..b]$ and $[c..d]$. $A(i, j)$ is called the element at position (i, j).

#include < *LEDA/array2.h* >

2. Creation

array2<*E*> A(*int* a, *int* b, *int* c, *int* d);

creates an instance A of type *array2*<*E*> with index set $[a..b] \times [c..d]$.

array2<*E*> A(*int* n, *int* m);

creates an instance A of type *array2*<*E*> with index set $[0..n-1] \times [0..m-1]$.

3. Operations

E&	A(*int* i, *int* j)	returns $A(i, j)$. *Precondition*: $a \le i \le b$ and $c \le j \le d$.
int	A.low1( )	returns a.
int	A.high1( )	returns b.
int	A.low2( )	returns c.
int	A.high2( )	returns d.

4. Implementation

Two dimensional arrays are implemented by C++vectors. All operations take time $O(1)$, the space requirement is $O(I * sizeof(E))$.

5.3 Stacks (stack)

1. Definition

An instance S of the parameterized data type *stack*<E> is a sequence of elements of data type E, called the element type of S. Insertions or deletions of elements take place only at one end of the sequence, called the top of S. The size of S is the length of the sequence, a stack of size zero is called the empty stack.

#include < *LEDA/stack.h* >

2. Creation

stack<E> S; creates an instance S of type *stack*<E>. S is initialized with the empty stack.

3. Operations

E	S.top( )	returns the top element of S. *Precondition*: S is not empty.
void	S.push(E x)	adds x as new top element to S.
E	S.pop( )	deletes and returns the top element of S. *Precondition*: S is not empty.
int	S.size( )	returns the size of S.
bool	S.empty( )	returns true if S is empty, false otherwise.
void	S.clear( )	makes S the empty stack.

4. Implementation

Stacks are implemented by singly linked linear lists. All operations take time $O(1)$, except clear which takes time $O(n)$, where n is the size of the stack.

5.4 Queues (queue)

1. Definition

An instance Q of the parameterized data type *queue*<E> is a sequence of elements of data type E, called the element type of Q. Elements are inserted at one end (the rear) and deleted at the other end (the front) of Q. The size of Q is the length of the sequence; a queue of size zero is called the empty queue.

#include < *LEDA/queue.h* >

2. Creation

queue<E> Q;	creates an instance Q of type *queue*<E>. Q is initialized with the empty queue.

3. Operations

E	Q.top( )	returns the front element of Q. *Precondition*: Q is not empty.
E	Q.pop( )	deletes and returns the front element of Q. *Precondition*: Q is not empty.
void	Q.append(E x)	appends x to the rear end of Q.
int	Q.size( )	returns the size of Q.
bool	Q.empty( )	returns true if Q is empty, false otherwise.
void	Q.clear( )	makes Q the empty queue.

4. Implementation

Queues are implemented by singly linked linear lists. All operations take time $O(1)$, except clear which takes time $O(n)$, where n is the size of the queue.

5.5 Bounded Stacks (b_stack)

1. Definition

An instance S of the parameterized data type *b_stack*<*E*> is a stack (see section 5.3) of bounded size.

#include < *LEDA/b_stack.h* >

2. Creation

b_stack<*E*> $S(int\ n)$;

creates an instance S of type *b_stack*<*E*> that can hold up to n elements. S is initialized with the empty stack.

3. Operations

E	S.top( )	returns the top element of S. *Precondition*: S is not empty.
E	S.pop( )	deletes and returns the top element of S. *Precondition*: S is not empty.
void	S.push(E x)	adds x as new top element to S. *Precondition*: S.size() $< n$.
void	S.clear( )	makes S the empty stack.
int	S.size( )	returns the size of S.
bool	S.empty( )	returns true if S is empty, false otherwise.

4. Implementation

Bounded stacks are implemented by C++vectors. All operations take time $O(1)$. The space requirement is $O(n)$.

5.6 Bounded Queues (b_queue)

1. Definition

An instance Q of the parameterized data type *b_queue*<E> is a queue (see section 5.4) of bounded size.

#include < *LEDA/b_queue.h* >

2. Creation

b_queue<E> Q(*int* n);

creates an instance Q of type *b_queue*<E> that can hold up to n elements. Q is initialized with the empty queue.

3. Operations

E	Q.top( )	returns the front element of Q. *Precondition*: Q is not empty.
E	Q.pop( )	deletes and returns the front element of Q. *Precondition*: Q is not empty.
void	Q.append(E x)	appends x to the rear end of Q. *Precondition*: Q.size() $< n$.
void	Q.clear( )	makes Q the empty queue.
int	Q.size( )	returns the size of Q.
bool	Q.empty( )	returns true if Q is empty, false otherwise.

4. Implementation

Bounded queues are implemented by circular arrays. All operations take time $O(1)$. The space requirement is $O(n)$.

5.7 Linear Lists (list)

1. Definition

An instance L of the parameterized data type *list*<*E*> is a sequence of items (*list_item*). Each item in L contains an element of data type E, called the element type of L. The number of items in L is called the length of L. If L has length zero it is called the empty list. In the sequel $\langle x \rangle$ is used to denote a list item containing the element x and $L[i]$ is used to denote the contents of list item i in L.

#include < *LEDA/list.h* >

2. Types

list<*E*>:: *item* — the item type.

list<*E*>:: *value_type* — the value type.

3. Creation

list<*E*> L; — creates an instance L of type *list*<*E*> and initializes it to the empty list.

4. Operations

Access Operations

int	L.length( )	returns the length of L.
int	L.size( )	returns L.length().
bool	L.empty( )	returns true if L is empty, false otherwise.
list_item	L.first( )	returns the first item of L (nil if L is empty).
list_item	L.last( )	returns the last item of L. (nil if L is empty)
list_item	L.get_item(*int* i)	returns the item at position i (the first position is 0). *Precondition*: $0 \leq i < L$.length(). **Note** that this takes time linear in i.
list_item	L.succ(*list_item* it)	returns the successor item of item it, nil if $it = L$.last(). *Precondition*: it is an item in L.
list_item	L.pred(*list_item* it)	returns the predecessor item of item it, nil if $it = L$.first(). *Precondition*: it is an item in L.

list_item	L.cyclic_succ(*list_item it*)	returns the cyclic successor of item it, i.e., L.first() if $it = L$.last(), L.succ(it) otherwise.
list_item	L.cyclic_pred(*list_item it*)	returns the cyclic predecessor of item it, i.e, L.last() if $it = L$.first(), L.pred(it) otherwise.
E	L.contents(*list_item it*)	returns the contents $L[it]$ of item it. *Precondition*: it is an item in L.
E	L.inf(*list_item it*)	returns L.contents(it).
E	L.front( )	returns the first element of L, i.e. the contents of L.first(). *Precondition*: L is not empty.
E	L.head( )	same as $L.front(\,)$.
E	L.back( )	returns the last element of L, i.e. the contents of L.last(). *Precondition*: L is not empty.
E	L.tail( )	same as $L.back(\,)$.
int	L.rank($E\ x$)	returns the rank of x in L, i.e. its first position in L as an integer from $[1 \ldots \lvert L \rvert]$ (0 if x is not in L). **Note** that this takes time linear in $rank(x)$. *Precondition*: compare has to be defined for type E.

Update Operations

list_item	L.push($E\ x$)	adds a new item $\langle x \rangle$ at the front of L and returns it ($L.insert(x, L.first(\,), LEDA::before)$).
list_item	L.push_front($E\ x$)	same as $L.push(x)$.
list_item	L.append($E\ x$)	appends a new item $\langle x \rangle$ to L and returns it ($L.insert(x, L.last(\,), LEDA::after)$).
list_item	L.push_back($E\ x$)	same as $L.append(x)$.
list_item	L.insert($E\ x$, *list_item pos*, *int dir* $= LEDA::after$)	inserts a new item $\langle x \rangle$ after (if $dir = LEDA::after$) or before (if $dir = LEDA::before$) item pos into L and returns it (here $LEDA::after$ and $LEDA::before$ are predefined constants). *Precondition*: it is an item in L.
E	L.pop( )	deletes the first item from L and returns its contents. *Precondition*: L is not empty.

E	*L*.pop_front( )	same as $L.pop(\,)$.
E	*L*.Pop( )	deletes the last item from L and returns its contents. *Precondition*: L is not empty.
E	*L*.pop_back( )	same as $L.Pop(\,)$.
E	*L*.del_item(*list_item it*)	deletes the item it from L and returns its contents $L[it]$. *Precondition*: it is an item in L.
E	*L*.del(*list_item it*)	same as $L.del\_item(it)$.
void	*L*.erase(*list_item it*)	deletes the item it from L. *Precondition*: it is an item in L.
void	*L*.remove(*E x*)	removes all items with contents x from L. *Precondition*: compare has to be defined for type E.
void	*L*.move_to_front(*list_item it*)	moves it to the front end of L.
void	*L*.move_to_rear(*list_item it*)	moves it to the rear end of L.
void	*L*.move_to_back(*list_item it*)	same as $L.move\_to\_rear(it)$.
void	*L*.assign(*list_item it*, *E x*)	makes x the contents of item it. *Precondition*: it is an item in L.
void	*L*.conc(*list<E>& L1*, *int dir = LEDA::after*)	appends ($dir = LEDA::after$ or prepends ($dir = LEDA::before$) list L_1 to list L and makes L_1 the empty list. *Precondition*: : $L \neq L_1$
void	*L*.swap(*list<E>& L1*)	swaps lists of items of L and $L1$;
void	*L*.split(*list_item it*, *list<E>& L1*, *list<E>& L2*)	splits L at item it into lists $L1$ and $L2$. More precisely, if $it \neq nil$ and $L = x_1, \ldots, x_{k-1}, it, x_{k+1}, \ldots, x_n$ then $L1 = x_1, \ldots, x_{k-1}$ and $L2 = it, x_{k+1}, \ldots, x_n$. If $it = nil$ then $L1$ is made empty and $L2$ a copy of L. Finally L is made empty if it is not identical to $L1$ or $L2$. *Precondition*: it is an item of L or nil.

void L.split(*list_item it*, *list*<*E*>& *L1*, *list*<*E*>& *L2*, *int dir*)

splits L at item it into lists $L1$ and $L2$. Item it becomes the first item of $L2$ if $dir{=}{=}LEDA::before$ and the last item of $L1$ if $dir = LEDA::after$. *Precondition*: it is an item of L.

void L.apply(*void* $(*f)(E\&\ x)$) — for all items $\langle x \rangle$ in L function f is called with argument x (passed by reference).

void L.reverse_items( ) — reverses the sequence of items of L.

void L.reverse_items(*list_item it1*, *list_item it2*)

reverses the sub-sequence $it1, \ldots, it2$ of items of L. *Precondition*: $it1 = it2$ or $it1$ appears before $it2$ in L.

void L.reverse( ) — reverses the sequence of entries of L.

void L.reverse(*list_item it1*, *list_item it2*)

reverses the sequence of entries $L[it1] \ldots L[it2]$.

void L.permute( ) — randomly permutes the items of L.

void L.permute(*list_item* $*\ I$) — permutes the items of L into the same order as stored in the array I.

void L.clear( ) — makes L the empty list.

Sorting and Searching

void L.sort(*int* $(*cmp)(E,\ E\ )$, *int* $dd = 1$)

sorts the items of L using the ordering defined by the compare function $cmp : E \times E \longrightarrow int$, with

$$cmp(a,b) \quad \begin{cases} < 0, & \text{if } a < b \\ = 0, & \text{if } a = b \\ > 0, & \text{if } a > b \end{cases}$$

More precisely, if $(in_1, \ldots, in_n)$ and $(out_1, \ldots, out_n)$ denote the values of L before and after the call of sort, then $cmp(L[out_j], L[out_{j+1}]) \leq 0$ for $1 \leq j < n$ and there is a permutation π of $[1..n]$ such that $out_i = in_{\pi_i}$ for $1 \leq i \leq n$.

void L.sort( ) — sorts the items of L using the default ordering of type E, i.e., the linear order defined by function *int* compare(*const* $E\&$, *const* $E\&$).

void L.merge_sort(*int* (**cmp*)(*E*, *E*))

sorts the items of L using merge sort and the ordering defined by *cmp*. The sort is stable, i.e., if $x = y$ and $\langle x \rangle$ is before $\langle y \rangle$ in L then $\langle x \rangle$ is before $\langle y \rangle$ after the sort. *L.merge_sort*( ) is more efficient than *L.sort*( ) if L contains large pre-sorted intervals.

void L.merge_sort( ) — as above, but uses the default ordering of type E.

void L.bucket_sort(*int* i, *int* j, *int* $(*f)(E)$)

sorts the items of L using bucket sort, where $f : E \longrightarrow int$ with $f(x) \in [i..j]$ for all elements x of L. The sort is stable, i.e., if $f(x) = f(y)$ and $\langle x \rangle$ is before $\langle y \rangle$ in L then $\langle x \rangle$ is before $\langle y \rangle$ after the sort.

void L.bucket_sort(*int* $(*f)(E)$)

same as $bucket\_sort(i, j, f)$ where i and j are the minimal and maximal value of $f(e)$ as e ranges over all elements of L.

void L.merge(*list*<*E*>& *L1*, *int* (**cmp*)(*E*, *E*))

merges the items of L and $L1$ using the ordering defined by *cmp*. The result is assigned to L and $L1$ is made empty.
Precondition: L and $L1$ are sorted incresingly according to the linear order defined by *cmp*.

void L.merge(*list*<*E*>& *L1*) — merges the items of L and $L1$ using the default linear order of type E.

void L.unique(*int* (**cmp*)(*E*, *E*))

removes duplicates from L.
Precondition: L is sorted incresingly according to the ordering defined by *cmp*.

void L.unique( ) — removes duplicates from L.
Precondition: L is sorted incresingly according to the default ordering of type E.

list_item L.search(*E* x) — returns the first item of L that contains x, nil if x is not an element of L.
Precondition: compare has to be defined for type E.

list_item L.min(*int* (**cmp*)(*E*, *E*)) — returns the item with the minimal contents with respect to the linear order defined by compare function *cmp*.

list_item L.min( ) — returns the item with the minimal contents with respect to the default linear order of type E.

list_item L.max(*int* (∗*cmp*)(*E*, *E*)) — returns the item with the maximal contents with respect to the linear order defined by compare function *cmp*.

list_item L.max( ) — returns the item with the maximal contents with respect to the default linear order of type E.

Input and Output

void L.read(*istream*& *I*, *char delim* = (*char*)*EOF*)

reads a sequence of objects of type E terminated by the delimiter *delim* from the input stream I using *operator* ≫ (*istream*&, *E*&). L is made a list of appropriate length and the sequence is stored in L.

void L.read(*char delim* = '\n')

calls L.read(*cin*, *delim*) to read L from the standard input stream *cin*.

void L.read(*string s*, *char delim* = '\n')

As above, but uses string s as a prompt.

void L.print(*ostream*& *O*, *char space* = ' ')

prints the contents of list L to the output tream O using *operator* ≪ (*ostream*&, *const E*&) to print each element. The elements are separated by character *space*.

void L.print(*char space* = ' ') — calls L.print(*cout*, *space*) to print L on the standard output stream *cout*.

void L.print(*string s*, *char space* = ' ')

As above, but uses string s as a header.

Operators

list<*E*>& L = L1 — The assignment operator makes L a copy of list L_1. More precisely if L_1 is the sequence of items $x_1, x_2, \ldots, x_n$ then L is made a sequence of items $y_1, y_2, \ldots, y_n$ with $L[y_i] = L_1[x_i]$ for $1 \leq i \leq n$.

E& L[*list_item it*] — returns a reference to the contents of *it*.

list_item L[*int i*] — an abbreviation for *L.get_item*(i).

list_item	L += E x	same as $L.append(x)$; returns the new item.
ostream&	*ostream*& *out* $\ll$ L	same as $L.print(out)$; returns *out*.
istream&	*istream*& *in* $\gg$ *list*<*E*>& L	same as $L.read(in))$; returns *in*.

Iteration

forall_items(it, L) { "the items of L are successively assigned to *it*" }

forall(x, L) { "the elements of L are successively assigned to x" }

STL compatible iterators are provided when compiled with `-DLEDA_STL_ITERATORS` (see *LEDAROOT/demo/stl/list.c* for an example).

5. Implementation

The data type list is realized by doubly linked linear lists. All operations take constant time except for the following operations: search and rank take linear time $O(n)$, item(i) takes time $O(i)$, bucket_sort takes time $O(n + j - i)$ and sort takes time $O(n \cdot c \cdot \log n)$ where c is the time complexity of the compare function. n is always the current length of the list.

5.8 Singly Linked Lists (slist)

1. Definition

An instance L of the parameterized data type *slist*<E> is a sequence of items (*slist_item*). Each item in L contains an element of data type E, called the element type of L. The number of items in L is called the length of L. If L has length zero it is called the empty list. In the sequel $\langle x \rangle$ is used to denote a list item containing the element x and $L[i]$ is used to denote the contents of list item i in L.

#include < *LEDA/slist.h* >

2. Creation

slist<E> L;	creates an instance L of type *slist*<E> and initializes it to the empty list.
slist<E> $L(E\ x)$;	creates an instance L of type *slist*<E> and initializes it to the one-element list $\langle x \rangle$.

3. Operations

int	L.length( )	returns the length of L.
int	L.size( )	returns L.length().
bool	L.empty( )	returns true if L is empty, false otherwise.
slist_item	L.first( )	returns the first item of L.
slist_item	L.last( )	returns the last item of L.
slist_item	L.succ(*slist_item it*)	returns the successor item of item *it*, nil if $it = L$.last(). *Precondition*: *it* is an item in L.
slist_item	L.cyclic_succ(*slist_item it*)	returns the cyclic successor of item *it*, i.e., L.first() if $it = L$.last(), L.succ(it) otherwise.
E	L.contents(*slist_item it*)	returns the contents $L[it]$ of item *it*. *Precondition*: *it* is an item in L.
E	L.inf(*slist_item it*)	returns L.contents(it). *Precondition*: *it* is an item in L.
E	L.head( )	returns the first element of L, i.e. the contents of L.first(). *Precondition*: L is not empty.

E	*L*.tail( )	returns the last element of *L*, i.e. the contents of *L*.last(). *Precondition*: *L* is not empty.
slist_item	*L*.push(*E x*)	adds a new item $\langle x \rangle$ at the front of *L* and returns it.
slist_item	*L*.append(*E x*)	appends a new item $\langle x \rangle$ to *L* and returns it.
slist_item	*L*.insert(*E x*, *slist_item pos*)	inserts a new item $\langle x \rangle$ after item *pos* into *L* and returns it. *Precondition*: *it* is an item in *L*.
E	*L*.pop( )	deletes the first item from *L* and returns its contents. *Precondition*: *L* is not empty.
void	*L*.del_succ_item(*slist_item it*)	deletes the successor of item *it* from *L*. *Precondition*: *it* is an item in *L* and has a successor.
void	*L*.conc(*slist*<*E*>& *L1*)	appends list L_1 to list *L* and makes L_1 the empty list. *Precondition*: $L \mathrel{!}= L_1$.
E&	*L*[*slist_item it*]	returns a reference to the contents of *it*.
slist_item	*L* += *E x*	appends a new item $\langle x \rangle$ to *L* and returns it.

5.9 Sets (set)

1. Definition

An instance S of the parameterized data type *set*<E> is a collection of elements of the linearly ordered type E, called the element type of S. The size of S is the number of elements in S, a set of size zero is called the empty set.

#include < LEDA/set.h >

2. Creation

set<E> S;	creates an instance S of type *set*<E> and initializes it to the empty set.

3. Operations

void	S.insert(E x)	adds x to S.
void	S.del(E x)	deletes x from S.
bool	S.member(E x)	returns true if x in S, false otherwise.
E	S.choose( )	returns an element of S. *Precondition*: S is not empty.
set<E>	S.join(*set*<E> T)	returns $S \cup T$.
set<E>	S.diff(*set*<E> T)	returns $S - T$.
set<E>	S.intersect(*set*<E> T)	returns $S \cap T$.
set<E>	S.symdiff(*set*<E> T)	returns the symetric difference of S and T.
set<E>	$S + T$	returns $S.join(T)$.
set<E>	$S - T$	returns $S.diff(T)$.
set<E>	S & T	returns $S.intersect(T)$.
set<E>	S % T	returns $S.symdiff(T)$.
set<E>&	S += T	assigns $S.join(T)$ to S and returns S.
set<E>&	S −= T	assigns $S.diff(T)$ to S and returns S.
set<E>&	S &= T	assigns $S.intersect(T)$ to S and returns S.

set<*E*>&	S %= T	assigns $S.symdiff(T)$ to S and returns S.
bool	$S \le T$	returns true if $S \subseteq T$, false otherwise.
bool	$S \ge T$	returns true if $T \subseteq S$, false otherwise.
bool	$S == T$	returns true if $S = T$, false otherwise.
bool	$S\ != T$	returns true if $S \neq T$, false otherwise.
bool	$S < T$	returns true if $S \subset T$, false otherwise.
bool	$S > T$	returns true if $T \subset S$, false otherwise.
bool	S.empty( )	returns true if S is empty, false otherwise.
int	S.size( )	returns the size of S.
void	S.clear( )	makes S the empty set.

Iteration

forall(x, S) { "the elements of S are successively assigned to x" }

4. Implementation

Sets are implemented by randomized search trees [2]. Operations insert, del, member take time $O(\log n)$, empty, size take time $O(1)$, and clear takes time $O(n)$, where n is the current size of the set.

5.10 Integer Sets (int_set)

1. Definition

An instance S of the data type *int_set* is a subset of a fixed interval $[a..b]$ of the integers, called the range of S.

#include < *LEDA/int_set.h* >

2. Creation

int_set S(*int* a, *int* b);

creates an instance S of type *int_set* for elements from $[a..b]$ and initializes it to the empty set.

int_set S(*int* n);

creates an instance S of type *int_set* for elements from $[0..n-1]$ and initializes it to the empty set.

3. Operations

void	S.insert(*int* x)	adds x to S. *Precondition*: $a \leq x \leq b$.
void	S.del(*int* x)	deletes x from S. *Precondition*: $a \leq x \leq b$.
bool	S.member(*int* x)	returns true if x in S, false otherwise. *Precondition*: $a \leq x \leq b$.
int	S.min( )	returns the minimal integer in the range of of S.
int	S.max( )	returns the maximal integer in the range of of S.
void	S.clear( )	makes S the empty set.
int_set&	S = $S1$	assignment.
int_set	S | $S1$	returns the union of S and $S1$.
int_set	S & $S1$	returns the intersection of S and $S1$.
int_set	$\sim S$	returns the complement of S.

4. Implementation

Integer sets are implemented by bit vectors. Operations insert, delete, member, empty, and size take constant time. clear, intersection, union and complement take time $O(b-a+1)$.

5.11 Dynamic Integer Sets (d_int_set)

1. Definition

An instance S of the data type *d_int_set* is a subset of the integers.

#include < *LEDA/d_int_set.h* >

2. Creation

d_int_set S;	creates an instance S of type *d_int_set* initializes it to the empty set.

3. Operations

void	S.insert(*int* x)	adds x to S. As the sets range is expanding dynamically during insertion for the range $[S.min(\ ), S.max(\ )]$ inserting the extrema early saves repeated reallocation time.
void	S.del(*int* x)	deletes x from S.
bool	S.member(*int* x)	returns true if x in S, false otherwise.
int	S.choose( )	returns a random element of S. *Precondition*: S is not empty.
bool	S.empty( )	returns true if S is empty, false otherwise.
int	S.size( )	returns the size of S.
void	S.clear( )	makes S the empty set.
d_int_set	S.join(*d_int_set* T)	returns $S \cup T$.
d_int_set	S.intersect(*d_int_set* T)	returns $S \cap T$.
d_int_set	S.diff(*d_int_set* T)	returns $S - T$.
d_int_set	S.symdiff(*d_int_set* T)	returns the symmectric difference of S and T.
d_int_set	$S + T$	returns the union $S.join(T)$.
d_int_set	$S - T$	returns the difference $S.diff(T)$.
d_int_set	S & T	returns the intersection of S and T.
d_int_set	$S \mid T$	returns the union $S.join(T)$.

d_int_set $S \,\%\, T$ — returns the symmetric difference $S.symdiff(T)$.

d_int_set& $S \mathrel{+}= T$ — assigns $S.join(T)$ to S and returns S.

d_int_set& $S \mathrel{-}= T$ — assigns $S.diff(T)$ to S and returns S.

d_int_set& $S \mathrel{\&}= T$ — assigns $S.intersect(T)$ to S and returns S.

d_int_set& $S \mathrel{|}= T$ — assigns $S.join(T)$ to S and returns S.

d_int_set& $S \,\%=\, T$ — assigns $S.symdiff(T)$ to S and returns S.

bool $S \mathrel{!}= T$ — returns true if $S \neq T$, false otherwise.

bool $S == T$ — returns true if $S = T$, false otherwise.

bool $S \geq T$ — returns true if $T \subseteq S$, false otherwise.

bool $S \leq T$ — returns true if $S \subseteq T$, false otherwise.

bool $S < T$ — returns true if $S \subset T$, false otherwise.

bool $S > T$ — returns true if $T \subset S$, false otherwise.

void S.get_element_list(*list*<*int*>& L)
fills L with all elements stored in the set in increasing order.

4. Implementation

Integer sets are implemented by bit vectors. Operations insert, delete, member, empty, and size take constant time. clear, intersection, union and complement take time $O(b - a + 1)$.

5.12 Partitions (partition)

1. Definition

An instance P of the data type *partition* consists of a finite set of items (*partition_item*) and a partition of this set into blocks.

#include < *LEDA/partition.h* >

2. Creation

partition P;	creates an instance P of type *partition* and initializes it to the empty partition.

3. Operations

partition_item	P.make_block( )	returns a new *partition_item* *it* and adds the block *it* to partition P.
partition_item	P.find(*partition_item* p)	returns a canonical item of the block that contains item p, i.e., iff $P.same\_block(p, q)$ then $P.find(p)$ and $P.find(q)$ return the same item. *Precondition*: p is an item in P.
int	P.size(*partition_item* p)	returns the size of the block containing p.
int	P.number_of_blocks( )	returns the number of blocks in P.
bool	P.same_block(*partition_item* p, *partition_item* q)	returns true if p and q belong to the same block of partition P. *Precondition*: p and q are items in P.
void	P.union_blocks(*partition_item* p, *partition_item* q)	unites the blocks of partition P containing items p and q. *Precondition*: p and q are items in P.
void	P.split(*list*<*partition_item*> L)	turns all items in L to singleton blocks. *Precondition*: L is a union of blocks.

4. Implementation

Partitions are implemented by the union find algorithm with weighted union and path compression (cf. [77]). Any sequence of n make_block and $m \geq n$ other operations (except for *split*) takes time $O(m\ \alpha(m,n))$. The cost of a split is proportional to the size of the blocks dismantled.

5. Example

Spanning Tree Algorithms (cf. section 9).

5.13 Parameterized Partitions (Partition)

1. Definition

An instance P of the data type *Partition*<*E*> consists of a finite set of items (*partition_item*) and a partition of this set into blocks. Each item has an associated information of type E.

#include < *LEDA/partition.h* >

2. Creation

Partition<*E*> P; creates an instance P of type *Partition*<*E*> and initializes it to the empty partition.

3. Operations

partition_item	P.make_block(*E* x)	returns a new *partition_item* *it*, adds the block *it* to partition P, and associates x with *it*.
partition_item	P.find(*partition_item* p)	returns a canonical item of the block that contains item p, i.e., iff $P.same\_block(p, q)$ then $P.find(p)$ and $P.find(q)$ return the same item. *Precondition*: p is an item in P.
int	P.size(*partition_item* p)	returns the size of the block containing p.
bool	P.same_block(*partition_item* p, *partition_item* q)	returns true if p and q belong to the same block of partition P. *Precondition*: p and q are items in P.
void	P.union_blocks(*partition_item* p, *partition_item* q)	unites the blocks of partition P containing items p and q. *Precondition*: p and q are items in P.
void	P.split(*list*<*partition_item*> L)	turns all items in L to singleton blocks. *Precondition*: L is a union of blocks
E	P.inf(*partition_item* *it*)	returns the information associated with *it*.
void	P.change_inf(*partition_item* *it*, *E* x)	changes the information associates with *it* to x.

5.14 Dynamic Trees (dynamic_trees)

1. Definition

An instance D of the data type *dynamic_trees* is a set of dynamically changing rooted trees. Each edge is directed towards the root and has a weight. Optionally, user defined information can be stored at the vertices and at the edges.

Remark Dynamic trees are very similar to the data type *tree_collection*. The main difference is that the former use edge weights instead of the node weights used by the latter.

#include < *LEDA/dynamic_trees.h* >

2. Creation

dynamic_trees D;	creates an empty instance D of type *dynamic_trees*.

3. Operations

vertex D.make(*void* $*\, x = nil$)	creates a tree containing a single node v which is returned. The additional user defined information x is stored at v.
*void** D.vertex_inf(*vertex* v)	returns the additional user defined information at v.
*void** D.edge_inf(*vertex* v)	returns the additional user defined information at (v,parent(v)) or *nil*, if v has no parent.
vertex D.parent(*vertex* v)	returns the parent of v in the tree or *nil*.
vertex D.root(*vertex* v)	returns the root of the tree containing v.
double D.cost(*vertex* v)	returns the cost of (v,parent(v)). *Precondition*: v is not a tree root.
vertex D.mincost(*vertex* v)	returns vertex w closest to root(v) s.t. (w,parent(w)) has minimal cost on the path $v \rightarrow$ root(v). *Precondition*: v is not a tree root.
void D.update(*vertex* v, *double* x)	adds x to each edge on the path $v \rightarrow$ root(v).
void D.link(*vertex* v, *vertex* w, *double* x, *void* $*\, e\_inf = nil$)	links the tree root v (prec.) to the vertex w in a different tree (prec.). The edge $e = (v, w)$ gets weight x, and the additional user defined information e_inf is stored at e.
double D.cut(*vertex* v)	deletes the edge (v,parent(v)) and returns its weight. *Precondition*: v is not a tree root.

void	D.evert(*vertex* v)	makes v the new root of its tree.
vertex	D.lca(*vertex* v, *vertex* w)	returns the lowest common ancestor of v and w or *nil*. *Precondition*: v and w are not *nil*.

4. Implementation

Dynamic Trees are implemented using binary trees with the randomized balancing scheme by Aragon and Seidel. Each operation takes $O(\log^2 n)$ amortized expected time except for make which takes constant time. n is the current number of nodes.

5.15 Dynamic Collections of Trees (tree_collection)

1. Definition

An instance D of the parameterized data type *tree_collection*<*I*> is a collection of vertex disjoint rooted trees, each of whose vertices has a double-valued cost and contains an information of type I, called the information type of D.

#include < *LEDA/tree_collection.h* >

2. Creation

tree_collection<*I*> D;

creates an instance D of type *tree_collection*<*I*>, initialized with the empty collection.

3. Operations

d_vertex D.maketree(*I* x) — adds a new tree to D containing a single vertex v with cost zero and information x, and returns v.

I D.inf(*d_vertex* v) — returns the information of vertex v.

d_vertex D.findroot(*d_vertex* v) — returns the root of the tree containing v.

d_vertex D.findcost(*d_vertex* v, *double&* x)

sets x to the minimum cost of a vertex on the tree path from v to findroot(v) and returns the last vertex (closest to the root) on this path of cost x.

d_vertex D.parent(*d_vertex* v) — returns the parent vertex of v.

void D.addcost(*d_vertex* v, *double* x)

adds double number x to the cost of every vertex on the tree path from v to findroot(v).

void D.link(*d_vertex* v, *d_vertex* x)

combines the trees containing vertices v and w by adding the edge (v, w). (We regard tree edges as directed from child to parent.)
Precondition: v and w are in different trees and v is a root.

void D.cut(*d_vertex* v) — divides the tree containing vertex v into two trees by deleting the edge out of v.
Precondition: v is not a tree root.

4. Implementation

Dynamic collections of trees are implemented by partitioning the trees into vertex disjoint paths and representing each path by a self-adjusting binary tree (see [77]). All operations take amortized time $O(\log n)$ where n is the number of maketree operations.

Chapter 6

Dictionaries

6.1 Dictionaries (dictionary)

1. Definition

An instance D of the parameterized data type *dictionary*<K, I> is a collection of items (*dic_item*). Every item in D contains a key from the linearly ordered data type K, called the key type of D, and an information from the data type I, called the information type of D. The number of items in D is called the size of D. A dictionary of size zero is called the empty dictionary. We use $\langle k, i\rangle$ to denote an item with key k and information i (i is said to be the information associated with key k). For each $k \in K$ there is at most one $i \in I$ with $\langle k, i\rangle \in D$.

#include < *LEDA/dictionary.h* >

2. Types

dictionary<K, I>:: *item* the item type.

dictionary<K, I>:: *key_type* the key type.

dictionary<K, I>:: *inf_type* the information type.

3. Creation

dictionary<K, I> D;

creates an instance D of type *dictionary*<K, I> based on the linear order defined by the global *compare* function and initializes it with the empty dictionary.

dictionary<K, I> $D(int\ (*cmp)(K,\ K\ ))$;

creates an instance D of type *dictionary*<K, I> based on the linear order defined by the compare function *cmp* and initializes it with the empty dictionary.

4. Operations

K	*D*.key(*dic_item it*)	returns the key of item *it*. *Precondition*: *it* is an item in *D*.
I	*D*.inf(*dic_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in *D*.
I&	*D*[*dic_item it*]	returns a reference to the information of item *it*. *Precondition*: *it* is an item in *D*.
dic_item	*D*.insert(*K k*, *I i*)	associates the information i with the key k. If there is an item $<k, j>$ in D then j is replaced by i, else a new item $\langle k, i\rangle$ is added to D. In both cases the item is returned.
dic_item	*D*.lookup(*K k*)	returns the item with key k (nil if no such item exists in D).
I	*D*.access(*K k*)	returns the information associated with key k. *Precondition*: there is an item with key k in D.
void	*D*.del(*K k*)	deletes the item with key k from D (null operation, if no such item exists).
void	*D*.del_item(*dic_item it*)	removes item *it* from *D*. *Precondition*: *it* is an item in *D*.
void	*D*.change_inf(*dic_item it*, *I i*)	makes i the information of item *it*. *Precondition*: *it* is an item in *D*.
void	*D*.clear( )	makes *D* the empty dictionary.
int	*D*.size( )	returns the size of *D*.
bool	*D*.empty( )	returns true if *D* is empty, false otherwise.

Iteration

STL compatible iterators are provided when compiled with `-DLEDA_STL_ITERATORS` (see *LEDAROOT/demo/stl/dic.c* for an example).

5. Implementation

Dictionaries are implemented by $(2, 4)$-trees. Operations insert, lookup, del_item, del take time $O(\log n)$, key, inf, empty, size, change_inf take time $O(1)$, and clear takes time $O(n)$. Here n is the current size of the dictionary. The space requirement is $O(n)$.

6. Example

We count the number of occurrences of each string in a sequence of strings.

```
#include <LEDA/dictionary.h>

main()
{ dictionary<string,int> D;
  string s;
  dic_item it;

  while (cin >> s)
  { it = D.lookup(s);
    if (it==nil) D.insert(s,1);
    else D.change_inf(it,D.inf(it)+1);
  }

  forall_items(it,D) cout << D.key(it) << " : " <<  D.inf(it) << endl;

}
```

6.2 Dictionaries with Implementation Parameter (_dictionary)

1. Definition

An instance of type *_dictionary*<$K, I, impl$> is a dictionary implemented by data type *impl*. *impl* must be one of the dictionary implementations listed in section 15.1.1 or a user defined data structure fulfilling the specification given in section 15.2.1. Note that depending on the actual implementation *impl* the key type K must either be linearly ordered or hashed.

#include < *LEDA/_dictionary.h* >

Example

Using a dictionary implemented by skiplists to count the number of occurrences of the elements in a sequence of strings.

```
#include <LEDA/_dictionary.h>
#include <LEDA/impl/skiplist.h>

main()
{
  _dictionary<string,int,skiplist> D;
  string s;
  dic_item it;

  while (cin >> s)
  { it = D.lookup(s);
    if (it==nil) D.insert(s,1);
    else D.change_inf(it,D.inf(it)+1);
  }

  forall_items(it,D)
    cout << D.key(it) << " : " <<  D.inf(it) << endl;

}
```

6.3 Sorted Sequences (sortseq)

1. Definition

An instance S of the parameterized data type *sortseq*<K, I> is a sequence of items (*seq_item*). Every item contains a key from a linearly ordered data type K, called the key type of S, and an information from a data type I, called the information type of S. The number of items in S is called the size of S. A sorted sequence of size zero is called empty. We use $\langle k, i \rangle$ to denote a *seq_item* with key k and information i (called the information associated with key k). For each k in K there is at most one item $\langle k, i \rangle$ in S and if item $\langle k1, i1 \rangle$ precedes item $\langle k2, i2 \rangle$ in S then $k1 < k2$.

Sorted sequences are a very powerful data type. They can do everything that dictionaries and priority queues can do. They also support many other operations, in particular *finger searches* and operations *conc*, *split*, *merge*, *reverse_items*, and *delete_subsequence*.

The key type K must be linearly ordered. The linear order on K may change over time subject to the condition that the order of the elements that are currently in the sorted sequence remains stable. More precisely, whenever an operation (except for *reverse_items*) is applied to a sorted sequence S, the keys of S must form an increasing sequence according to the currently valid linear order on K. For operation *reverse_items* this must hold after the execution of the operation. An application of sorted sequences where the linear order on the keys evolves over time is the plane sweep algorithm for line segment intersection. This algorithm sweeps an arrangement of segments by a vertical sweep line and keeps the intersected segments in a sorted sequence sorted according to the y-coordinates of their intersections with the sweep line. For intersecting segments this order depends on the position of the sweep line.

Sorted sequences support finger searches. A finger search takes an item *it* in a sorted sequence and a key k and searches for the key in the sorted sequence containing the item. The cost of a finger search is proportional to the logarithm of the distance of the key from the start of the search. A finger search does not need to know the sequence containing the item. We use IT to denote the sequence containing *it*. In a call *S.finger_search*(*it*, k) the types of S and IT must agree but S may or may not be the sequence containing *it*.

#include < *LEDA/sortseq.h* >

2. Types

sortseq<K, I>:: *item*	the item type *seq_item*.
sortseq<K, I>:: *key_type*	the key type K.
sortseq<K, I>:: *inf_type*	the information type I.

3. Creation

sortseq<K, I> S;

creates an instance S of type *sortseq*<K, I> based on the linear order defined by the global *compare* function and and initializes it to the empty sorted sequence.

sortseq<K, I> $S(int\ (*cmp)\ (K,\ K\ ))$;

creates an instance S of type *sortseq*<K, I> based on the linear order defined by the compare function *cmp* and initializes it with the empty sorted sequence.

4. Operations

K	S.key(*seq_item it*)	returns the key of item *it*.
I	S.inf(*seq_item it*)	returns the information of item *it*.
I&	S[*seq_item it*]	returns a reference to the information of item *it*. *Precondition*: *it* is an item in S.
seq_item	S.lookup(K k)	returns the item with key k (*nil* if there is no such item).
seq_item	S.finger_lookup(K k)	equivalent to $S.lookup(k)$
seq_item	S.finger_lookup_from_front(K k)	equivalent to $S.lookup(k)$
seq_item	S.finger_lookup_from_rear(K k)	equivalent to $S.lookup(k)$
seq_item	S.locate(K k)	returns the item $\langle k1, i\rangle$ in S such that $k1$ is minimal with $k1 \geq k$ (*nil* if no such item exists).
seq_item	S.finger_locate(K k)	equivalent to $S.locate(k)$
seq_item	S.finger_locate_from_front(K k)	equivalent to $S.locate(k)$
seq_item	S.finger_locate_from_rear(K k)	equivalent to $S.locate(k)$
seq_item	S.locate_succ(K k)	equivalent to $S.locate(k)$
seq_item	S.succ(K k)	equivalent to $S.locate(k)$
seq_item	S.finger_locate_succ(K k)	equivalent to $S.locate(k)$

seq_item S.finger_locate_succ_from_front(K k)
equivalent to $S.locate(k)$

seq_item S.finger_locate_succ_from_rear(K k)
equivalent to $S.locate(k)$

seq_item S.locate_pred(K k) returns the item $\langle k1, i\rangle$ in S such that $k1$ is maximal with $k1 \leq k$ (*nil* if no such item exists).

seq_item S.pred(K k) equivalent to $S.locate\_pred(k)$

seq_item S.finger_locate_pred(K k)
equivalent to $S.locate\_pred(k)$

seq_item S.finger_locate_pred_from_front(K k)
equivalent to $S.locate\_pred(k)$

seq_item S.finger_locate_pred_from_rear(K k)
equivalent to $S.locate\_pred(k)$

seq_item S.finger_lookup(*seq_item* it, K k)
equivalent to $IT.lookup(k)$ where IT is the sorted sequence containing it.
Precondition: S and IT must have the same type

seq_item S.finger_locate(*seq_item* it, K k)
equivalent to $IT.locate(k)$ where IT is the sorted sequence containing it.
Precondition: S and IT must have the same type.

seq_item S.finger_locate_succ(*seq_item* it, K k)
equivalent to $IT.locate\_succ(k)$ where IT is the sorted sequence containing it.
Precondition: S and IT must have the same type

seq_item S.finger_locate_pred(*seq_item* it, K k)
equivalent to $IT.locate\_pred(k)$ where IT is the sorted sequence containing it.
Precondition: S and IT must have the same type.

seq_item S.min_item( ) returns the item with minimal key (*nil* if S is empty).

seq_item S.max_item( ) returns the item with maximal key (*nil* if S is empty).

seq_item S.succ(*seq_item* it) returns the successor item of it in the sequence containing it (*nil* if there is no such item).

seq_item S.pred(*seq_item* x) returns the predecessor item of it in the sequence containing it (*nil* if there is no such item).

seq_item S.insert(K k, I i)

associates information i with key k: If there is an item $\langle k, j\rangle$ in S then j is replaced by i, else a new item $\langle k, i\rangle$ is added to S. In both cases the item is returned.

seq_item S.insert_at(*seq_item* it, K k, I i)

Like $IT.insert(k, i)$ where IT is the sequence containing item it.
Precondition: it is an item in IT with either $key(it)$ is maximal with $key(it) < k$ or $key(it)$ is minimal with $key(it) > k$. S and IT have the same type.

seq_item S.insert_at(*seq_item* it, K k, I i, *int* dir)

Like $IT.insert(k, i)$ where IT is the sequence containing item it.
Precondition: it is an item in IT with $key(it)$ is maximal with $key(it) < k$ if $dir = LEDA::before$ or $key(it)$ is minimal with $key(it) > k$ if $dir = LEDA::after$.
S and IT have the same type.

int S.size() — returns the size of S.

bool S.empty() — returns true if S is empty, false otherwise.

void S.clear() — makes S the empty sorted sequence.

void S.reverse_items(*seq_item* a, *seq_item* b)

the subsequence of IT from a to b is reversed, where IT is the sequence containing a and b.
Precondition: a appears before b in IT.

void S.flip_items(*seq_item* a, *seq_item* b)

equivalent to $S.reverse\_items(a, b)$.

void S.del(K k)

removes the item with key k from S (null operation if no such item exists).

void S.del_item(*seq_item* it)

removes the item it from the sequence containing it.

void S.change_inf(*seq_item* it, I i)

makes i the information of item it.

void S.split(*seq_item it*, *sortseq*<K, I>& *S1*, *sortseq*<K, I>& *S2*, *int dir* = *LEDA*:: *after*)

splits IT at item it, where IT is the sequence containing it, into sequences $S1$ and $S2$ and makes IT empty (if distinct from $S1$ and $S2$). More precisely, if $IT = x_1, \ldots, x_{k-1}, it, x_{k+1}, \ldots, x_n$ and $dir = LEDA::after$ then $S1 = x_1, \ldots, x_{k-1}, it$ and $S2 = x_{k+1}, \ldots, x_n$. If $dir = LEDA::before$ then $S2$ starts with it after the split.

void S.delete_subsequence(*seq_item a*, *seq_item b*, *sortseq*<K, I>& *S1*)

deletes the subsequence starting at a and ending at b from the sequence IT containing both and assigns the subsequence to $S1$.
Precondition: a and b belong to the same sequence IT, a is equal to or before b and IT and $S1$ have the same type.

sortseq<K, I>& S.conc(*sortseq*<K, I>& *S1*, *int dir* = *LEDA*:: *after*)

appends $S1$ at the front ($dir = LEDA::before$) or rear ($dir = LEDA::after$) end of S, makes $S1$ empty and returns S.
Precondition: $S.key(S.max\_item(\,)) < S1.key(S1.min\_item(\,))$ if $dir = LEDA::after$ and $S1.key(S1.max\_item(\,)) < S.key(S.min\_item(\,)$ if $dir = LEDA::before$.

void S.merge(*sortseq*<K, I>& *S1*)

merges the sequence $S1$ into sequence S and makes $S1$ empty.
Precondition: all keys are distinct.

void S.print(*ostream*& *out*, *string s*, *char c* = ' ')

prints s and all elements of S separated by c onto stream out.

void S.print(*string s*, *char c* = ' ')

equivalent to $S.print(cout, s, c)$.

bool $S == S1$

returns *true* if S agrees with $S1$ componentwise and *false* otherwise

sortseq<K, I>$*$ *sortseq*<K, I>:: my_sortseq(*seq_item it*)

returns a pointer to the *sortseq* containing it.
Precondition: The type of the *sortseq* containing it must be *sortseq*<K, I>.

5. Implementation

Sorted sequences are implemented by skiplists [71]. Let n denote the current size of the sequence. Operations *insert*, *locate*, *lookup* and *del* take time $O(\log n)$, operations *succ*, *pred*, *max*, *min_item*, *key*, *inf*, *insert_at* and *del_item* take time $O(1)$. *clear* takes time $O(n)$ and *reverse_items* $O(l)$, where l is the length of the reversed subsequence. *Finger_lookup*(x) and *finger_locate*(x) take time $O(\log min(d, n-d))$ if x is the d-th item in S. *Finger_lookup_from_front*(x) and *finger_locate_from_front*(x) take time $O(\log d)$ if x

is the d-th item in S. *Finger_lookup_from_rear*(x) and *finger_locate_from_rear*(x) take time $O(\log d)$ if x is the $n-d$-th item in S. *Finger_lookup*(it, x) and *finger_locate*(it, x) take time $O(\log min(d, n-d))$ where d is the number of items between it and the item containing x. Note that $min(d, n-d)$ is the smaller of the distances from it to x if sequences are viewed as circularly closed. *Split*, *delete_subsequence* and *conc* take time $O(\log min(n_1, n_2))$ where n_1 and n_2 are the sizes of the results of *split* and *delete_subsequence* and the arguments of *conc* respectively. *Merge* takes time $O(\log((n_1 + n_2)/n_1))$ where n_1 and n_2 are the sizes of the two arguments. The space requirement of sorted sequences is linear in the length of the sequence (about $25.5n$ Bytes for a sequence of size n plus the space for the keys and the informations.).

6. Example

We use a sorted sequence to list all elements in a sequence of strings lying lexicographically between two given search strings.

```
#include <LEDA/sortseq.h>
main(){
sortseq<string,int> S;
string s1,s2;
cout << "input a sequence of strings terminated by stop\n";
while (cin >> s1 && s1 != "stop") S.insert(s1, 0);
while ( true )
{ cout << "\ninput a pair of strings\n\n";
  cin >> s1 >> s2;
  cout << "all strings s with " <<
                           s1 <<" <= s <= " << s2 <<"\n";
  if ( s2 < s1 ) continue;
  seq_item last  = S.locate_pred(s2);
  seq_item first = S.locate(s1);
  if ( !first || !last || first == S.succ(last) ) continue;
  seq_item it = first;
  while ( true )
  { cout << "\n" << S.key(it);
    if ( it == last ) break;
    it = S.succ(it);
  }
}
}
```

Further examples can be found in section Sorted Sequences of [58].

6.4 Sorted Sequences with Implementation Parameter (_sortseq)

1. Definition

An instance of type *_sortseq*<$K, I,$ *impl*> is a sorted sequence implemented by data type *impl*. *impl* must be one of the sorted sequence implementations listed in section 15.1.1 or a user defined data structure fulfilling the specification given in section 15.2.3. Note that the key type K must be linearly ordered.

#include < *LEDA/_sortseq.h* >

Example

Using a sorted sequence implemented by skiplists to list all elements in a sequence of strings lying lexicographically between two given search strings.

```
#include <LEDA/_sortseq.h>
#include <LEDA/impl/skiplist.h>

main()
{
 _sortseq<string,int,skiplist> S;
 string s1,s2;

 while ( cin >> s1 &&  s1 != "stop" )  S.insert(s1,0);

 while ( cin >> s1 >> s2 )
 { seq_item start = S.locate(s1);
   seq_item stop  = S.locate(s2);
   for (seq_item it = start; it != stop; it = S.succ(it))
      cout << S.key(it) << endl;
  }

}
```

6.5 Dictionary Arrays (d_array)

1. Definition

An instance A of the parameterized data type *d_array*$<I, E>$ (dictionary array) is an injective mapping from the linearly ordered data type I, called the index type of A, to the set of variables of data type E, called the element type of A. We use $A(i)$ to denote the variable with index i and we use $dom(A)$ to denote the set of "used indices". This set is empty at the time of creation and is modified by array accesses. Each dictionary array has an associated default value *xdef*. The variable $A(i)$ has value *xdef* for all $i \notin dom(A)$.

Related data types are *h_arrays*, *maps*, and *dictionaries*.

#include < *LEDA/d_array.h* >

2. Types

d_array$<I, E>$:: *item* — the item type.

d_array$<I, E>$:: *index_type* — the index type.

d_array$<I, E>$:: *element_type* — the element type.

3. Creation

d_array$<I, E>$ A; — creates an injective function a from I to the set of unused variables of type E, sets *xdef* to the default value of type E (if E has no default value then *xdef* stays undefined) and $dom(A)$ to the empty set, and initializes A with a.

d_array$<I, E>$ $A(E\ x)$; — creates an injective function a from I to the set of unused variables of type E, sets *xdef* to x and $dom(A)$ to the empty set, and initializes A with a.

4. Operations

E&	$A[I\ i]$	returns the variable $A(i)$.
bool	A.defined($I\ i$)	returns true if $i \in dom(A)$ and false otherwise.
void	A.undefine($I\ i$)	removes i from $dom(A)$ and sets $A(i)$ to *xdef*.
void	A.clear( )	makes $dom(A)$ empty.
int	A.size( )	returns $\|dom(A)\|$.

Iteration

forall_defined(i, A) { "the elements from $dom(A)$ are successively assigned to i" }

forall(x, A) { "for all $i \in dom(A)$ the entries $A[i]$ are successively assigned to x" }

5. Implementation

Dictionary arrays are implemented by randomized search trees [2]. Access operations $A[i]$ take time $O(\log dom(A))$. The space requirement is $O(dom(A))$.

6. Example

Program 1:

We use a dictionary array to count the number of occurrences of the elements in a sequence of strings.

```
#include <LEDA/d_array.h>

main()
{
  d_array<string,int> N(0);
  string s;

  while (cin >> s) N[s]++;

  forall_defined(s,N) cout << s << "  " << N[s] << endl;

}
```

Program 2:

We use a *d_array*<*string*, *string*> to realize an english/german dictionary.

```
#include <LEDA/d_array.h>

main()
{
  d_array<string,string> dic;

  dic["hello"] = "hallo";
  dic["world"] = "Welt";
  dic["book"]  = "Buch";
  dic["key"]   = "Schluessel";
```

```
  string s;
  forall_defined(s,dic) cout << s << "  " << dic[s] << endl;

}
```

6.6 Dictionary Arrays with Implementation Parameter (_d_array)

1. Definition

An instance of type *_d_array*<$I, E, impl$> is a dictionary array implemented by data type *impl*. *impl* must be one of the dictionary implementations listed in section 15.1.1 or a user defined data structure fulfilling the specification given in section 15.2.1. Note that depending on the actual implementation *impl* the index type I must either be linearly ordered or hashed.

#include < *LEDA/_d_array.h* >

Example

Using a dictionary array implemented by hashing with chaining (*ch_hash*) to count the number of occurences of the elements in a sequence of strings.

```
#include <LEDA/_d_array.h>
#include <LEDA/impl/ch_hash.h>

//we first have to define a hash function for strings

int Hash(const string& x)
{ return (x.length() > 0) ? x[0] : 0; }

main()
{
  _d_array<string,int,ch_hash> N(0);
  string s;

  while (cin >> s) N[s]++;

  forall_defined(s,N)
    cout << s << "  " << N[s] << endl;
}
```

6.7 Hashing Arrays (h_array)

1. Definition

An instance A of the parameterized data type *h_array*<I, E> (hashing array) is an injective mapping from a hashed data type I (cf. section 1.3.2), called the index type of A, to the set of variables of arbitrary type E, called the element type of A. We use $A(i)$ to denote the variable indexed by i and we use $dom(A)$ to denote the set of "used indices". This set is empty at the time of creation and is modified by array accesses. Each hashing array has an associated default value *xdef*. The variable $A(i)$ has value *xdef* for all $i \notin dom(A)$.

Related data types are *d_arrays*, *maps*, and *dictionaries*.

#include < *LEDA/h_array.h* >

2. Creation

h_array<I, E> A; creates an injective function a from I to the set of unused variables of type E, sets *xdef* to the default value of type E (if E has no default value then *xdef* stays undefined) and $dom(A)$ to the empty set, and initializes A with a.

h_array<I, E> $A(E\ x)$;

creates an injective function a from I to the set of unused variables of type E, sets *xdef* to x and $dom(A)$ to the empty set, and initializes A with a.

h_array<I, E> $A(E\ x,\ int\ table\_sz)$;

as above, but uses an initial table size of *table_sz* instead of the default size 1.

3. Operations

E&	$A[I\ i]$	returns the variable $A(i)$.
bool	A.defined($I\ i$)	returns true if $i \in dom(A)$ and false otherwise.
void	A.undefine($I\ i$)	removes i from $dom(A)$ and sets $A(i)$ to *xdef*.
void	A.clear( )	makes $dom(A)$ empty.
int	A.size( )	returns $\|dom(A)\|$.

Iteration

forall_defined(i, A) { "the elements from $dom(A)$ are successively assigned to i" }

forall(x, A) { "for all $i \in dom(A)$ the entries $A[i]$ are successively assigned to x" }

4. Implementation

Hashing arrays are implemented by hashing with chaining. Access operations take expected time $O(1)$. In many cases, hashing arrays are more efficient than dictionary arrays (cf. 6.5).

6.8 Maps (map)

1. Definition

An instance M of the parameterized data type *map*<I, E> is an injective mapping from the data type I, called the index type of M, to the set of variables of data type E, called the element type of M. I must be a pointer, item, or handle type or the type int. We use $M(i)$ to denote the variable indexed by i. All variables are initialized to *xdef*, an element of E that is specified in the definition of M. A subset of I is designated as the domain of M. Elements are added to $dom(M)$ by the subscript operator; however, the domain may also contain indices for which the access operator was never executed.

Related data types are *d_arrays*, *h_arrays*, and *dictionaries*.

#include < *LEDA/map.h* >

2. Types

map<I, E>:: *item*	the item type.
map<I, E>:: *index_type*	the index type.
map<I, E>:: *element_type*	the element type.

3. Creation

map<I, E> M;	creates an injective function m from I to the set of unused variables of type E, sets *xdef* to the default value of type E (if E has no default value then *xdef* is set to an unspecified element of E), and initializes M with m.
map<I, E> $M(E\ x)$;	creates an injective function m from I to the set of unused variables of type E, sets *xdef* to x, and initializes M with m.
map<I, E> $M(E\ x$, *int table_sz*);	as above, but uses an initial table size of *table_sz* instead of the default size 1.

4. Operations

E&	$M[I\ i]$	returns the variable $M(i)$ and adds i to $dom(M)$. If M is a const-object then $M(i)$ is read-only and i is not added to $dom(M)$.
bool	M.defined($I\ i$)	returns true if $i \in dom(M)$.
void	M.clear( )	makes M empty.
void	M.clear($E\ x$)	makes M empty and sets *xdef* to x.

Iteration:

forall(x, M) { "the entries $M[i]$ with $i \in dom(M)$ are successively assigned to x" }

forall_defined(i, M) { "the indices in $dom(M)$ are successively assigned to i" }.
Precondition: I must be an integer or pointer type.

5. Implementation

Maps are implemented by hashing with chaining and table doubling. Access operations $M[i]$ take expected time $O(1)$.

6.9 Two-Dimensional Maps (map2)

1. Definition

An instance M of the parameterized data type *map2*<*I1*, *I2*, *E*> is an injective mapping from the pairs in $I1 \times I2$, called the index type of M, to the set of variables of data type E, called the element type of M. I must be a pointer, item, or handle type or the type int. We use $M(i, j)$ to denote the variable indexed by (i, j) and we use $dom(M)$ to denote the set of "used indices". This set is empty at the time of creation and is modified by map2 accesses.

Related data types are *map*, *d_arrays*, *h_arrays*, and *dictionaries*.

#include < *LEDA/map2.h* >

2. Creation

map2<*I1*, *I2*, *E*> M; creates an injective function m from $I1 \times I2$ to the set of unused variables of type E, sets *xdef* to the default value of type E (if E has no default value then *xdef* stays undefined) and $dom(M)$ to the empty set, and initializes M with m.

map2<*I1*, *I2*, *E*> $M(E\ x)$;

creates an injective function m from $I1 \times I2$ to the set of unused variables of type E, sets *xdef* to x and $dom(M)$ to the empty set, and initializes M with m.

3. Operations

E&	M(*I1* i, *I2* j)	returns the variable $M(i)$.
bool	M.defined(*I1* i, *I2* j)	returns true if $i \in dom(M)$ and false otherwise.
void	M.clear( )	clears M by making $dom(M)$ the empty set.

4. Implementation

Maps are implemented by hashing with chaining and table doubling. Access operations $M(i, j)$ take expected time $O(1)$.

6.10 Persistent Dictionaries (p_dictionary)

1. Definition

An instance D of the parameterized data type *p_dictionary*<K, I> is a set of items (type *p_dic_item*). Every item in D contains a key from the linearly ordered data type K, called the key type of D, and an information from data type I, called the information type of D. The number of items in D is called the size of D. A dictionary of size zero is called empty. We use $\langle k, i\rangle$ to denote an item with key k and information i (i is said to be the information associated with key k). For each $k \in K$ there is at most one item $\langle k, i\rangle \in D$.

The difference between dictionaries (cf. section 6.1) and persistent dictionaries lies in the fact that update operations performed on a persistent dictionary D do not change D but create and return a new dictionary D'. For example, D.del(k) returns the dictionary D' containing all items it of D with key(it) $\neq k$. Also, an assignment $D1 = D2$ does not assign a copy of $D2$ (with new items) to $D1$ but the value of $D2$ itself.

#include < *LEDA/p_dictionary.h* >

2. Creation

p_dictionary<K, I> D;

creates an instance D of type *p_dictionary*<K, I> and initializes D to an empty persistent dictionary.

3. Operations

K	D.key(*p_dic_item* it)	returns the key of item it. *Precondition*: $it \in D$.
I	D.inf(*p_dic_item* it)	returns the information of item it. *Precondition*: $it \in D$.
p_dic_item	D.locate(K k)	returns the item with key k (nil if no such item exists in D).
p_dictionary<K, I>	D.del(K k)	returns $\{x \in D \mid key(x) \neq k\}$.
p_dictionary<K, I>	D.del_item(*p_dic_item* it)	returns $\{x \in D \mid x \neq it\}$.
p_dictionary<K, I>	D.insert(K k, I i)	returns D.del(k) $\cup \{\langle k, i\rangle\}$.
p_dictionary<K, I>	D.change_inf(*p_dic_item* it, I i)	returns D.del_item(it) $\cup \{\langle k, i\rangle\}$, where $k = key(it)$. *Precondition*: $it \in D$.

int	D.size( )	returns the size of D.
bool	D.empty( )	returns true if D is empty, false otherwise.

4. Implementation

Persistent dictionaries are implemented by leaf oriented persistent red black trees. Operations insert, lookup, del_item, del take time $O(\log^2 n)$, key, inf, empty, size and change_inf take time $O(1)$. The space requirement is $O(1)$ for each update operation.

Chapter 7

Priority Queues

7.1 Priority Queues (p_queue)

1. Definition

An instance Q of the parameterized data type *p_queue*<P, I> is a collection of items (type *pq_item*). Every item contains a priority from a linearly ordered type P and an information from an arbitrary type I. P is called the priority type of Q and I is called the information type of Q. The number of items in Q is called the size of Q. If Q has size zero it is called the empty priority queue. We use $\langle p, i \rangle$ to denote a *pq_item* with priority p and information i.

#include < *LEDA/p_queue.h* >

2. Types

p_queue<P, I>:: *item*	the item type.
p_queue<P, I>:: *prio_type*	the priority type.
p_queue<P, I>:: *inf_type*	the information type.

3. Creation

p_queue<P, I> Q;

creates an instance Q of type *p_queue*<P, I> based on the linear order defined by the global compare function *compare*(*const* P&, *const* P&) and initializes it with the empty priority queue.

p_queue<P, I> Q(*int* (∗*cmp*)(P, P));

creates an instance Q of type *p_queue*<P, I> based on the linear order defined by the compare function *cmp* and initializes it with the empty priority queue. *Precondition*: *cmp* must define a linear order on P.

4. Operations

P	Q.prio(*pq_item it*)	returns the priority of item *it*. *Precondition*: *it* is an item in Q.
I	Q.inf(*pq_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in Q.
I&	Q[*pq_item it*]	returns a reference to the information of item *it*. *Precondition*: *it* is an item in Q.
pq_item	Q.insert(*P x*, *I i*)	adds a new item $<x, i>$ to Q and returns it.
pq_item	Q.find_min( )	returns an item with minimal priority (nil if Q is empty).
P	Q.del_min( )	removes the item $it = Q$.find_min() from Q and returns the priority of it. *Precondition*: Q is not empty.
void	Q.del_item(*pq_item it*)	removes the item *it* from Q. *Precondition*: *it* is an item in Q.
void	Q.change_inf(*pq_item it*, *I i*)	makes *i* the new information of item *it*. *Precondition*: *it* is an item in Q.
void	Q.decrease_p(*pq_item it*, *P x*)	makes *x* the new priority of item *it*. *Precondition*: *it* is an item in Q and *x* is not larger then *prio*(*it*).
int	Q.size( )	returns the size of Q.
bool	Q.empty( )	returns true, if Q is empty, false otherwise.
void	Q.clear( )	makes Q the empty priority queue.

5. Implementation

Priority queues are implemented by Fibonacci heaps [32]. Operations insert, del_item, del_min take time $O(\log n)$, find_min, decrease_p, prio, inf, empty take time $O(1)$ and clear takes time $O(n)$, where n is the size of Q. The space requirement is $O(n)$.

6. Example

Dijkstra's Algorithm (cf. section 9)

7.2 Priority Queues with Implementation Parameter (_p_queue)

1. Definition

An instance of type *_p_queue*<$P, I, impl$> is a priority queue implemented by data type *impl*. *impl* must be one of the priority queue implementations listed in section 15.1.2 or a user defined data structure fulfilling the specification given in section 15.2.2. Note that the priority type P must linearly ordered.

#include < *LEDA/_p_queue.h* >

7.3 Bounded Priority Queues (b_priority_queue)

1. Definition

An instance Q of the parameterized data type *b_priority_queue*<K> is a priority_queue (cf. section 7.1) whose information type is a fixed interval $[a..b]$ of integers.

#include < *LEDA/b_prio.h* >

2. Creation

b_priority_queue<K> Q(*int* a, *int* b);

creates an instance Q of type *b_priority_queue*<K> with information type $[a..b]$ and initializes it with the empty priority queue.

3. Operations

See section 7.1.

4. Implementation

Bounded priority queues are implemented by arrays of linear lists. Operations insert, find_min, del_item, decrease_inf, key, inf, and empty take time $O(1)$, del_min (= del_item for the minimal element) takes time $O(d)$, where d is the distance of the minimal element to the next bigger element in the queue ($= O(b-a)$ in the worst case). clear takes time $O(b-a+n)$ and the space requirement is $O(b-a+n)$, where n is the current size of the queue.

Chapter 8

Graphs and Related Data Types

8.1 Graphs (graph)

1. Definition

An instance G of the data type *graph* consists of a list V of nodes and a list E of edges (*node* and *edge* are item types). Distinct graphs have disjoint node and edge lists. The value of a variable of type node is either the node of some graph, or the special value *nil* (which is distinct from all nodes), or is undefined (before the first assignment to the variable). A corresponding statement is true for the variables of type edge.

A graph with empty node list is called *empty*. A pair of nodes $(v, w) \in V \times V$ is associated with every edge $e \in E$; v is called the *source* of e and w is called the *target* of e, and v and w are called *endpoints* of e. The edge e is said to be *incident* to its endpoints.

A graph is either *directed* or *undirected*. The difference between directed and undirected graphs is the way the edges incident to a node are stored and how the concept *adjacent* is defined.

In directed graphs two lists of edges are associated with every node v: $adj\_edges(v) = \{e \in E \mid v = source(e)\}$, i.e., the list of edges starting in v, and $in\_edges(v) = \{e \in E \mid v = target(e)\}$, i.e., the list of edges ending in v. The list $adj\_edges(v)$ is called the adjacency list of node v and the edges in $adj\_edges(v)$ are called the edges *adjacent* to node v. For directed graphs we often use $out\_edges(v)$ as a synonym for $adj\_edges(v)$.

In undirected graphs only the list $adj\_edges(v)$ is defined for every every node v. Here it contains all edges incident to v, i.e., $adj\_edges(v) = \{e \in E \mid v \in \{source(e), target(e)\}\}$. An undirected graph may not contain selfloops, i.e., it may not contain an edge whose source is equal to its target.

In a directed graph an edge is adjacent to its source and in an undirected graph it is adjacent to its source and target. In a directed graph a node w is adjacent to a node v if there is an edge $(v, w) \in E$; in an undirected graph w is adjacent to v if there is an edge (v, w) or (w, v) in the graph.

A directed graph can be made undirected and vice versa: G.make_undirected() makes the directed graph G undirected by appending for each node v the list *in_edges*(v) to the list *adj_edges*(v) (removing selfloops). Conversely, G.make_directed() makes the undirected graph G directed by splitting for each node v the list *adj_edges*(v) into the lists *out_edges*(v) and *in_edges*(v). Note that these two operations are not exactly inverse to each other. The data type *ugraph* (cf. section 8.3) can only represent undirected graphs.

Reversal Information, Maps and Faces

The reversal information of an edge e is accessed through *G.reversal*(e), it has type *edge* and may or may not be defined (= *nil*). Assume that *G.reversal*(e) is defined and let $e' =$ *G.reversal*(e). Then $e = (v, w)$ and $e' = (w, v)$ for some nodes v and w, *G.reversal*(e') is defined and $e =$ *G.reversal*(e'). In other words, *reversal* deserves its name.

We call a directed graph *bidirected* if for every edge of G the reversed edge also belongs to G and we call a bidirected graph a *map* if all edges have their reversal information defined. Maps are the data structure of choice for embedded graphs. For an edge e of a map G let *face_cycle_succ*(e) = *cyclic_adj_pred*(*reversal*(e)) and consider the sequence e, *face_cycle_succ*(e), *face_cycle_succ*(*face_cycle_succ*(e)), ... The first edge to repeat in this sequence is e (why?) and the set of edges appearing in this sequence is called the *face cycle* containing e. Each edge is contained in some face cycle and face cycles are pairwise disjoint. Let f be the number of face cycles, n be the number of nodes, m be the number of edges, and let c be the number of connected components. Then $g = (m/2 - n - f)/2 + c$ is called the *genus* of the map [79] (note that $m/2$ is the number of edges in the underlying undirected graph). The genus is zero if and only if the map is planar, i.e., there is an embedding of G into the plane such that for every node v the counter-clockwise ordering of the edges around v agrees with the cyclic ordering of v's adjacency list.

If a graph G is a map the faces of G can be constructed explicitly by *G.compute_faces*( ). Afterwards, the faces of G can be traversed by different iterators, e.g., *forall_faces*(f, G) iterates over all faces , *forall_adj_faces*(v) iterates over all faces adjacent to node v. By using face maps or arrays (data types *face_map* and *face_array*) additional information can be associated with the faces of a graph. Note that any update operation performed on G invalidates the list of faces. See the section on face operations for a complete list of available operations for faces.

#include < *LEDA/graph.h* >

2. Creation

graph G;	creates an object G of type *graph* and initializes it to the empty directed graph.

graph $G(int\ n\_slots,\ int\ e\_slots)$;

this constructor specifies the numbers of free data slots in the nodes and edges of G that can be used for storing the entries of node and edge arrays. See also the description of the *use_node_data*( ) and *use_edge_data*( ) operations in 8.7 and 8.8.

3. Operations

a) Access operations

int	G.outdeg(*node* v)	returns the number of edges adjacent to node v ($\|adj\_edges(v)\|$).
int	G.indeg(*node* v)	returns the number of edges ending at v ($\|in\_edges(v)\|$) if G is directed and zero if G is undirected).
int	G.degree(*node* v)	returns $outdeg(v) + indeg(v)$.
node	G.source(*edge* e)	returns the source node of edge e.
node	G.target(*edge* e)	returns the target node of edge e.
node	G.opposite(*node* v, *edge* e)	returns $target(e)$ if $v = source(e)$ and $source(e)$ otherwise.
int	G.number_of_nodes( )	returns the number of nodes in G.
int	G.number_of_edges( )	returns the number of edges in G.
list<*node*>	G.all_nodes( )	returns the list V of all nodes of G.
list<*edge*>	G.all_edges( )	returns the list E of all edges of G.
list<*edge*>	G.adj_edges(*node* v)	returns $adj\_edges(v)$.
list<*edge*>	G.out_edges(*node* v)	returns $adj\_edges(v)$ if G is directed and the empty list otherwise.
list<*edge*>	G.in_edges(*node* v)	returns $in\_edges(v)$ if G is directed and the empty list otherwise.
list<*node*>	G.adj_nodes(*node* v)	returns the list of all nodes adjacent to v.
node	G.first_node( )	returns the first node in V.
node	G.last_node( )	returns the last node in V.

node	G.choose_node( )	returns a random node of G (nil if G is empty).
node	G.succ_node(*node* v)	returns the successor of node v in V (nil if it does not exist).
node	G.pred_node(*node* v)	returns the predecessor of node v in V (nil if it does not exist).
edge	G.first_edge( )	returns the first edge in E.
edge	G.last_edge( )	returns the last edge in E.
edge	G.choose_edge( )	returns a random edge of G (nil if G is empty).
edge	G.succ_edge(*edge* e)	returns the successor of edge e in E (nil if it does not exist).
edge	G.pred_edge(*edge* e)	returns the predecessor of edge e in E (nil if it does not exist).
edge	G.first_adj_edge(*node* v)	returns the first edge in the adjacency list of v (nil if this list is empty).
edge	G.last_adj_edge(*node* v)	returns the last edge in the adjacency list of v (nil if this list is empty).
edge	G.adj_succ(*edge* e)	returns the successor of edge e in the adjacency list of node $source(e)$ (nil if it does not exist).
edge	G.adj_pred(*edge* e)	returns the predecessor of edge e in the adjacency list of node $source(e)$ (nil if it does not exist).
edge	G.cyclic_adj_succ(*edge* e)	returns the cyclic successor of edge e in the adjacency list of node $source(e)$.
edge	G.cyclic_adj_pred(*edge* e)	returns the cyclic predecessor of edge e in the adjacency list of node $source(e)$.
edge	G.first_in_edge(*node* v)	returns the first edge of $in\_edges(v)$ (nil if this list is empty).
edge	G.last_in_edge(*node* v)	returns the last edge of $in\_edges(v)$ (nil if this list is empty).
edge	G.in_succ(*edge* e)	returns the successor of edge e in $in\_edges(target(e))$ (nil if it does not exist).
edge	G.in_pred(*edge* e)	returns the predecessor of edge e in $in\_edges(target(e))$ (nil if it does not exist).
edge	G.cyclic_in_succ(*edge* e)	returns the cyclic successor of edge e in $in\_edges(target(e))$ (nil if it does not exist).

edge	G.cyclic_in_pred(*edge* e)	returns the cyclic predecessor of edge e in $in\_edges(target(e))$ (nil if it does not exist).
bool	G.is_directed( )	returns true iff G is directed.
bool	G.is_undirected( )	returns true iff G is undirected.
bool	G.empty( )	returns true iff G is empty.

b) Update operations

node	G.new_node( )	adds a new node to G and returns it.
node	G.new_node(*node* u, *int* dir)	adds a new node v to G and returns it. v is inserted before ($dir = LEDA{::}before$) or after ($dir = LEDA{::}after$) node u into the list of all nodes.
edge	G.new_edge(*node* v, *node* w)	adds a new edge (v, w) to G by appending it to $adj\_edges(v)$ and to $in\_edges(w)$ (if G is directed) or $adj\_edges(w)$ (if G is undirected), and returns it.
edge	G.new_edge(*edge* e, *node* w, *int* $dir = LEDA{::}after$)	adds a new edge $x = (source(e), w)$ to G. x is inserted before ($dir = LEDA{::}before$) or after ($dir = LEDA{::}\ after$) edge e into $adj\_edges(source(e))$ and appended to $in\_edges(w)$ (if G is directed) or $adj\_edges(w)$ (if G is undirected). Here $LEDA{::}before$ and $LEDA{::}after$ are predefined constants. The operation returns the new edge x. *Precondition*: $source(e) \neq w$ if G is undirected.
edge	G.new_edge(*node* v, *edge* e, *int* $dir = LEDA{::}after$)	adds a new edge $x = (v, target(e))$ to G. x is appended to $adj\_edges(v)$ and inserted before ($dir = LEDA{::}before$) or after ($dir = LEDA{::}after$) edge e into $in\_edges(target(e))$ (if G is directed) or $adj\_edges(target(e))$ (if G is undirected). The operation returns the new edge x. *Precondition*: $target(e) \neq v$ if G is undirected.

edge	G.new_edge(*edge e1*, *edge e2*, *int d1* = *LEDA*::*after*, *int d2* = *LEDA*::*after*)	adds a new edge $x = (source(e1), target(e2))$ to G. x is inserted before (if *d1* = *LEDA*:: *before*) or after (if *d1* = *LEDA*:: *after*) edge $e1$ into *adj_edges*(*source*($e1$)) and before (if *d2* = *LEDA*:: *before*) or after (if *d2* = *LEDA*:: *after*) edge $e2$ into *in_edges*(*target*($e2$)) (if G is directed) or *adj_edges*(*target*($e2$)) (if G is undirected). The operation returns the new edge x.
node	G.merge_nodes(*node v1*, *node v2*)	experimental.
node	G.merge_nodes(*edge e1*, *node v2*)	experimental.
node	G.split_edge(*edge e*, *edge*& *e1*, *edge*& *e2*)	experimental
void	G.hide_edge(*edge e*)	removes edge e temporarily from G until restored by *G.restore_edge*(e).
bool	G.is_hidden(*edge e*)	returns *true* if e is hidden and *false* otherwise.
list<*edge*>	G.hidden_edges( )	returns the list of all hidden edges of G.
void	G.restore_edge(*edge e*)	restores e by appending it to *adj_edges*(*source*(e)) and to *in_edges*(*target*(e)) (*adj_edges*(*target*(e)) if G is undirected). *Precondition*: e is hidden and neither *source*(e) nor *target*(e) is hidden.
void	G.restore_all_edges( )	restores all hidden edges.
void	G.hide_node(*node v*)	removes node v temporarily from G until restored by *G.restore_node*(v). All non-hidden edges in *adj_edges*(v) and *in_edges*(v) are hidden too.
void	G.hide_node(*node v*, *list*<*edge*>& *h_edges*)	as above, in addition, the list of leaving or entering edges which are hidden as a result of hiding v are appended to *h_edges*.
bool	G.is_hidden(*node v*)	returns *true* if v is hidden and *false* otherwise.
list<*node*>	G.hidden_nodes( )	returns the list of all hidden nodes of G.
void	G.restore_node(*node v*)	restores v by appending it to the list of all nodes. Note that no edge adjacent to v that was hidden by *G.hide_node*(v) is restored by this operation.

void	G.restore_all_nodes( )	restores all hidden nodes.
void	G.del_node(*node* v)	deletes v and all edges incident to v from G.
void	G.del_edge(*edge* e)	deletes the edge e from G.
void	G.del_nodes(*list*<*node*> L)	deletes all nodes in L from G.
void	G.del_edges(*list*<*edge*> L)	deletes all edges in L from G.
void	G.del_all_nodes( )	deletes all nodes from G.
void	G.del_all_edges( )	deletes all edges from G.
void	G.del_all_faces( )	deletes all faces from G.
void	G.move_edge(*edge* e, *node* v, *node* w)	moves edge e to source v and target w by appending it to $adj\_edges(v)$ and to $in\_edges(w)$ (if G is directed) or $adj\_edges(w)$) (if G is undirected).
void	G.move_edge(*edge* e, *edge* $e1$, *node* w, *int* $d = LEDA{::}after$)	moves edge e to source $source(e1)$ and target w by inserting it before (if $d = LEDA{::}\,before$) or after (if $d = LEDA{::}\,after$) edge $e1$ into $adj\_edges(source(e1))$ and by appending it to $in\_edges(w)$ (if G is directed) or $adj\_edges(w)$) (if G is undirected).
void	G.move_edge(*edge* e, *node* v, *edge* $e2$, *int* $d = LEDA{::}after$)	moves edge e to source v and target $target(e2)$ by appending it to $adj\_edges(v)$) and inserting it before (if $d = LEDA{::}before$) or after (if $d = LEDA{::}after$) edge $e2$ into $in\_edges(target(e2))$ (if G is directed) or $adj\_edges(target(e2))$ (if G is undirected).
void	G.move_edge(*edge* e, *edge* $e1$, *edge* $e2$, *int* $d1 = LEDA{::}after$, *int* $d2 = LEDA{::}after$)	moves edge e to source $source(e1)$ and target $target(e2)$ by inserting it before (if $d1 = LEDA{::}\,before$) or after (if $d1 = LEDA{::}\,after$) edge $e1$ into $adj\_edges(source(e1))$ and before (if $d2 = LEDA{::}before$) or after (if $d2 = LEDA{::}after$) edge $e2$ into $in\_edges(target(e2))$ (if G is directed) or $adj\_edges(target(e2))$ (if G is undirected).
edge	G.rev_edge(*edge* e)	reverses e ($move\_edge(e, target(e), source(e))$).
void	G.rev_all_edges( )	reverses all edges of G.

void G.sort_nodes(*int* (**cmp*)(*node*, *node*))

the nodes of G are sorted according to the ordering defined by the comparing function *cmp*. Subsequent executions of forall_nodes step through the nodes in this order. (cf. TOPSORT1 in section 9).

void G.sort_edges(*int* (**cmp*)(*edge*, *edge*))

the edges of G and all adjacency lists are sorted according to the ordering defined by the comparing function *cmp*. Subsequent executions of forall_edges step through the edges in this order. (cf. TOPSORT1 in section 9).

void G.sort_nodes(*node_array*<*T*> A)

the nodes of G are sorted according to the entries of node_array A (cf. section 8.7).
Precondition: T must be numerical.

void G.sort_edges(*edge_array*<*T*> A)

the edges of G are sorted according to the entries of edge_array A (cf. section 8.8).
Precondition: T must be numerical.

void G.bucket_sort_nodes(*int* l, *int* h, *int* (**ord*)(*node*))

sorts the nodes of G using *bucket sort*
Precondition: $l \leq ord(v) \leq h$ for all nodes v.

void G.bucket_sort_edges(*int* l, *int* h, *int* (**ord*)(*edge*))

sorts the edges of G using *bucket sort*
Precondition: $l \leq ord(e) \leq h$ for all edges e.

void G.bucket_sort_nodes(*int* (**ord*)(*node*))

same as $G.bucket\_sort\_nodes(l, h, ord$ with l (h) equal to the minimal (maximal) value of $ord(v)$.

void G.bucket_sort_edges(*int* (**ord*)(*edge*))

same as $G.bucket\_sort\_edges(l, h, ord$ with l (h) equal to the minimal (maximal) value of $ord(e)$.

void G.bucket_sort_nodes(*node_array*<*int*> A)

same as $G.bucket\_sort\_nodes(ord)$ with $ord(v) = A[v]$ for all nodes v of G.

void G.bucket_sort_edges(*edge_array*<*int*> A)

same as $G.bucket\_sort\_edges(ord)$ with $ord(e) = A[e]$ for all edges e of G.

void	G.set_node_position(*node* v, *node* p)	moves node v in the list V of all nodes such that p becomes the predecessor of v. If $p = nil$ then v is moved to the front of V.
void	G.set_edge_position(*edge* e, *edge* p)	moves edge e in the list E of all edges such that p becomes the predecessor of e. If $p = nil$ then e is moved to the front of E.
list<*edge*>	G.insert_reverse_edges( )	for every edge (v, w) in G the reverse edge (w, v) is inserted into G. Returns the list of all inserted edges.
void	G.make_undirected( )	makes G undirected by appending $in\_edges(v)$ to $adj\_edges(v)$ for all nodes v.
void	G.make_directed( )	makes G directed by splitting $adj\_edges(v)$ into $out\_edges(v)$ and $in\_edges(v)$.
void	G.clear( )	makes G the empty graph.
void	G.join(*graph*& H)	merges H into G by moving all objects (nodes,edges, and faces) from H to G. H is empty afterwards.

c) Reversal Edges and Maps

void	G.make_bidirected( )	makes G bidirected by inserting missing reversal edges.
void	G.make_bidirected(*list*<*edge*>& R)	makes G bidirected by inserting missing reversal edges. Appends all inserted edges to list R.
bool	G.is_bidirected( )	returns true if every edge has a reversal and false otherwise.
bool	G.make_map( )	sets the reversal information of a maximal number of edges of G. Returns *true* if G is bidirected and *false* otherwise.
void	G.make_map(*list*<*edge*>& R)	makes G bidirected by inserting missing reversal edges and then turns it into a map setting the reversals for all edges. Appends all inserted edges to list R.
bool	G.is_map( )	tests whether G is a map.
edge	G.reversal(*edge* e)	returns the reversal information of edge e (*nil* if not defined).

void	*G*.set_reversal(*edge e*, *edge r*)	makes r the reversal of e and vice versa. *Precondition*: $e = (v, w)$ and $r = (w, v)$ for some nodes v and w.
edge	*G*.face_cycle_succ(*edge e*)	returns the cyclic adjacency predecessor of $reversal(e)$. *Precondition*: $reversal(e)$ is defined.
edge	*G*.face_cycle_pred(*edge e*)	returns the reversal of the cyclic adjacency successor s of e. *Precondition*: $reversal(s)$ is defined.
edge	*G*.split_map_edge(*edge e*)	splits edge $e = (v, w)$ and its reversal $r = (w, v)$ into edges (v, u), (u, w), (w, u), and (u, v). Returns the edge (u, w).
edge	*G*.new_map_edge(*edge e1*, *edge e2*)	inserts a new edge $e = (source(e1), source(e2)$ after $e1$ into the adjacency list of $source(e1)$ and an edge r reversal to e after $e2$ into the adjacency list of $source(e2)$.
list<*edge*>	*G*.triangulate_map( )	triangulates the map G by inserting additional edges. The list of inserted edges is returned. *Precondition*: G must be connected. The algorithm ([43]) has running time $O(\|V\|+\|E\|)$.
void	*G*.dual_map(*graph*& *D*)	constructs the dual of G in D. The algorithm has linear running time. *Precondition*: G must be a map.

For backward compatibility

edge	*G*.reverse(*edge e*)	returns $reversal(e)$ (historical).
edge	*G*.succ_face_edge(*edge e*)	returns $face\_cycle\_succ(e)$ (historical).
edge	*G*.next_face_edge(*edge e*)	returns $face\_cycle\_succ(e)$ (historical).
edge	*G*.pred_face_edge(*edge e*)	returns $face\_cycle\_pred(e)$ (historical).

d) Faces and Planar Maps

void	*G*.compute_faces( )	constructs the list of face cycles of G. *Precondition*: G is a map.
face	*G*.face_of(*edge e*)	returns the face of G to the left of edge e.
face	*G*.adj_face(*edge e*)	returns $G.face\_of(e)$.

void	*G*.print_face(*face f*)	prints face f.
int	*G*.number_of_faces( )	returns the number of faces of G.
face	*G*.first_face( )	returns the first face of G. (nil if empty).
face	*G*.last_face( )	returns the last face of G.
face	*G*.choose_face( )	returns a random face of G (nil if G is empty).
face	*G*.succ_face(*face f*)	returns the successor of face f in the face list of G (nil if it does not exist).
face	*G*.pred_face(*face f*)	returns the predecessor of face f in the face list of G (nil if it does not exist).
list<*face*>	*G*.all_faces( )	returns the list of all faces of G.
list<*face*>	*G*.adj_faces(*node v*)	returns the list of all faces of G adjacent to node v in counter-clockwise order.
list<*node*>	*G*.adj_nodes(*face f*)	returns the list of all nodes of G adjacent to face f in counter-clockwise order.
list<*edge*>	*G*.adj_edges(*face*)	returns the list of all edges of G bounding face f in counter-clockwise order.
int	*G*.size(*face f*)	returns the number of edges bounding face f.
edge	*G*.first_face_edge(*face f*)	returns the first edge of face f in G.
edge	*G*.split_face(*edge e1*, *edge e2*)	inserts the edge $e = (source(e_1), source(e_2))$ and its reversal into G and returns e. *Precondition*: e_1 and e_2 are bounding the same face F. The operation splits F into two new faces.
face	*G*.join_faces(*edge e*)	deletes edge e and its reversal r and updates the list of faces accordingly. The function returns a face that is affected by the operations (see the LEDA book for details).

void G.make_planar_map( )

makes G a planar map by reordering the edges such that for every node v the ordering of the edges in the adjacency list of v corresponds to the counter-clockwise ordering of these edges around v for some planar embedding of G and constructs the list of faces.
Precondition: G is a planar bidirected graph (map).

list<*edge*> G.triangulate_planar_map( )

triangulates planar map G and recomputes its list of faces

e) Operations for undirected graphs

edge G.new_edge(*node v*, *edge e1*, *node w*, *edge e2*, *int d1* = *LEDA*:: *after*, *int d2* = *LEDA*:: *after*)

adds a new edge (v, w) to G by inserting it before (if *d1* = *LEDA*:: *before*) or after (if *d1* = *LEDA*:: *after*) edge $e1$ into $adj\_edges(v)$ and before (if *d2* = *LEDA*:: *before*) or after (if *d2* = *LEDA*:: *after*) edge $e2$ into $adj\_edges(w)$, and returns it.
Precondition: $e1$ is incident to v and $e2$ is incident to w and $v \neq w$.

edge G.new_edge(*node v*, *edge e*, *node w*, *int d* = *LEDA*:: *after*)

adds a new edge (v, w) to G by inserting it before (if d = *LEDA*:: *before*) or after (if d = *LEDA*:: *after*) edge e into $adj\_edges(v)$ and appending it to $adj\_edges(w)$, and returns it.
Precondition: e is incident to v and $v \neq w$.

edge G.new_edge(*node v*, *node w*, *edge e*, *int d* = *LEDA*:: *after*)

adds a new edge (v, w) to G by appending it to to $adj\_edges(v)$, and by inserting it before (if d = *LEDA*:: *before*) or after (if d = *LEDA*:: *after*) edge e into $adj\_edges(w)$, and returns it.
Precondition: e is incident to w and $v \neq w$.

edge G.adj_succ(*edge e*, *node v*)

returns the successor of edge e in the adjacency list of v.
Precondition: e is incident to v.

edge G.adj_pred(*edge e*, *node v*)

returns the predecessor of edge e in the adjacency list of v.
Precondition: e is incident to v.

edge G.cyclic_adj_succ(*edge* e, *node* v)

returns the cyclic successor of edge e in the adjacency list of v.
Precondition: e is incident to v.

edge G.cyclic_adj_pred(*edge* e, *node* v)

returns the cyclic predecessor of edge e in the adjacency list of v.
Precondition: e is incident to v.

f) I/O Operations

void G.write(*ostream&* O = *cout*)

writes G to the output stream O.

void G.write(*string* s) writes G to the file with name s.

int G.read(*istream&* I = *cin*)

reads a graph from the input stream I and assigns it to G.

int G.read(*string* s) reads a graph from the file with name s and assigns it to G. Returns 1 if file s does not exist, 2 if the edge and node parameter types of $*this$ and the graph in the file s do not match, 3 if file s does not contain a graph, and 0 otherwise.

bool G.write_gml(*ostream&* O = *cout*, *void* (**node_cb*)(*ostream&* , *const graph**, *const node*) = 0, *void* (**edge_cb*)(*ostream&* , *const graph**, *const edge*) = 0)

writes G to the output stream O in GML format ([42]). If *node_cb* is not equal to 0, it is called while writing a node v with output stream O, the graph and v as parameters. It can be used to write additional user defined node data. The output should conform with GML format (see manual page *gml_graph*). *edge_cb* is called while writing edges. If the operation fails, $false$ is returned.

bool G.write_gml(*string* s, *void* (**node_cb*)(*ostream&* , *const graph**, *const node*) = 0, *void* (**edge_cb*)(*ostream&* , *const graph**, *const edge*) = 0)

writes G to the file with name s in GML format. For a description of *node_cb* and *edge_cb*, see above. If the operation fails, $false$ is returned.

bool G.read_gml(*string* s) reads a graph in GML format from the file with name s and assigns it to G.

bool G.read_gml(*istream*& I = *cin*)
reads a graph in GML format from the input stream I and assigns it to G.

void G.print_node(*node* v, *ostream*& O = *cout*)
prints node v on the output stream O.

void G.print_edge(*edge* e, *ostream*& O = *cout*)
prints edge e on the output stream O. If G is directed e is represented by an arrow pointing from source to target. If G is undirected e is printed as an undirected line segment.

void G.print(*string* s, *ostream*& O = *cout*)
prints G with header line s on the output stream O.

void G.print(*ostream*& O = *cout*)
prints G on the output stream O.

g) Non-Member Functions

graph∗	graph_of(*node* v)	returns a pointer to the graph that v belongs to.
graph∗	graph_of(*edge* e)	returns a pointer to the graph that e belongs to.
graph∗	graph_of(*face* f)	returns a pointer to the graph that f belongs to.
node	source(*edge* e)	returns the source node of edge e.
node	target(*edge* e)	returns the target node of edge e.
face	face_of(*edge* e)	returns the face of edge e.

h) Iteration

All iteration macros listed in this section traverse the corresponding node and edge lists of the graph, i.e. they visit nodes and edges in the order in which they are stored in these lists.

forall_nodes(v, G)
{ “the nodes of G are successively assigned to v” }

forall_edges(e, G)
{ “the edges of G are successively assigned to e” }

forall_rev_nodes(v, G)
{ “the nodes of G are successively assigned to v in reverse order” }

forall_rev_edges(e, G)
{ "the edges of G are successively assigned to e in reverse order" }

forall_adj_edges(e, w)
{ "the edges adjacent to node w are successively assigned to e" }

forall_out_edges(e, w) a faster version of **forall_adj_edges** for directed graphs.

forall_in_edges(e, w)
{ "the edges of $in\_edges(w)$ are successively assigned to e" }

forall_inout_edges(e, w)
{ "the edges of $adj\_edges(w)$ and $in\_edges(w)$ are successively assigned to e" }

forall_adj_nodes(v, w)
{ "the nodes adjacent to node w are successively assigned to v" }

Faces

Before using any of the following face iterators the list of faces has to be computed by calling *G.compute_faces*( ). Note, that any update operation invalidates this list.

forall_faces(f, M)
{ "the faces of M are successively assigned to f" }

forall_face_edges(e, f)
{ "the edges of face f are successively assigned to e" }

forall_adj_faces(f, v)
{ "the faces adjacent to node v are successively assigned to f" }

4. Implementation

Graphs are implemented by doubly linked lists of nodes and edges. Most operations take constant time, except for all_nodes, all_edges, del_all_nodes, del_all_edges, make_map, make_planar_map, compute_faces, all_faces, make_map, clear, write, and read which take time $O(n + m)$, and adj_edges, adj_nodes, out_edges, in_edges, and adj_faces which take time $O(\mathit{output\ size})$ where n is the current number of nodes and m is the current number of edges. The space requirement is $O(n + m)$.

8.2 Parameterized Graphs (GRAPH)

1. Definition

A parameterized graph G is a graph whose nodes and edges contain additional (user defined) data. Every node contains an element of a data type *vtype*, called the node type of G and every edge contains an element of a data type *etype* called the edge type of G. We use $<v, w, y>$ to denote an edge (v, w) with information y and $<x>$ to denote a node with information x.

All operations defined for the basic graph type *graph* are also defined on instances of any parameterized graph type *GRAPH*<*vtype*, *etype*>. For parameterized graphs there are additional operations to access or update the information associated with its nodes and edges. Instances of a parameterized graph type can be used wherever an instance of the data type *graph* can be used, e.g., in assignments and as arguments to functions with formal parameters of type *graph*&. If a function f(*graph*& G) is called with an argument Q of type *GRAPH*<*vtype*, *etype*> then inside f only the basic graph structure of Q can be accessed. The node and edge entries are hidden. This allows the design of generic graph algorithms, i.e., algorithms accepting instances of any parametrized graph type as argument.

#include < *LEDA/graph.h* >

2. Creation

GRAPH<*vtype*, *etype*> G;

creates an instance G of type *GRAPH*<*vtype*, *etype*> and initializes it to the empty graph.

3. Operations

vtype G.inf(*node* v) — returns the information of node v.

const vtype& G[*node* v] — returns a reference to G.inf(v).

etype G.inf(*edge* e) — returns the information of edge e.

const etype& G[*edge* e] — returns a reference to G.inf(e).

node_array<*vtype*>& G.node_data( )

makes the information associated with the nodes of G available as a node array of type *node_array*<*vtype*>.

edge_array<*etype*>& G.edge_data( )

makes the information associated with the edges of G available as an edge array of type *edge_array*<*etype*>.

void G.assign(*node* v, *vtype* x)

makes x the information of node v.

void G.assign(*edge* e, *etype* x)

makes x the information of edge e.

node G.new_node(*vtype* x) adds a new node $<x>$ to G and returns it.

node G.new_node(*node* u, *vtype* x, *int* *dir*)

adds a new node $v = <x>$ to G and returns it. v is inserted before ($dir = LEDA::before$) or after ($dir = LEDA::after$) node u into the list of all nodes.

edge G.new_edge(*node* v, *node* w, *etype* x)

adds a new edge $<v, w, x>$ to G by appending it to $adj\_edges(v)$ and to $in\_edges(w)$ and returns it.

edge G.new_edge(*edge* e, *node* w, *etype* x, *int* $dir = LEDA::after$)

adds a new edge $<source(e), w, x>$ to G by inserting it after ($dir = LEDA::after$) or before ($dir = LEDA::before$) edge e into $adj\_edges(source(e))$ and appending it to $in\_edges(w)$. Returns the new edge.

edge G.new_edge(*node* v, *edge* e, *etype* x, *int* $dir = LEDA::after$)

adds a new edge $<v, target(e), x>$ to G by inserting it after ($dir = LEDA::after$) or before ($dir = LEDA::before$) edge e into $in\_edges(target(e))$ and appending it to $adj\_edges(v)$. Returns the new edge.

edge G.new_edge(*edge* $e1$, *edge* $e2$, *etype* x, *int* $d1 = LEDA::after$, *int* $d2 = LEDA::after$)

adds a new edge $x = (source(e1), target(e2), x)$ to G. x is inserted before (if $d1 = LEDA::before$) or after (if $d1 = LEDA::after$) edge $e1$ into $adj\_edges(source(e1))$ and before (if $d2 = LEDA::before$) or after (if $d2 = LEDA::after$) edge $e2$ into $in\_edges(target(e2))$ (if G is directed) or $adj\_edges(target(e2))$ (if G is undirected). The operation returns the new edge x.

edge G.new_edge(*node* v, *edge* $e1$, *node* w, *edge* $e2$, *etype* x, *int* $d1 = LEDA::after$, *int* $d2 = LEDA::after$)

adds a new edge (v, w, x) to G by inserting it before (if $d1 = LEDA::before$) or after (if $d1 = LEDA::after$) edge $e1$ into $adj\_edges(v)$ and before (if $d2 = LEDA::before$) or after (if $d2 = LEDA::after$) edge $e2$ into $adj\_edges(w)$, and returns it.
Precondition: G is undirected, $v \neq w$, $e1$ is incident to v, and $e2$ is incident to w.

edge	G.new_edge(*node* v, *edge* e, *node* w, *etype* x, *int* $d = LEDA::after$)	
		adds a new edge (v, w, x) to G by inserting it before (if $d = LEDA::before$) or after (if $d = LEDA::after$) edge e into $adj\_edges(v)$ and appending it to $adj\_edges(w)$, and returns it. *Precondition*: G is undirected, $v \neq w$, $e1$ is incident to v, and e is incident to v.
void	G.sort_nodes(*list*<*node*> *vl*)	
		makes *vl* the node list of G. *Precondition*: *vl* contains exactly the nodes of G.
void	G.sort_edges(*list*<*edge*> *el*)	
		makes *el* the edge list of G. *Precondition*: *el* contains exactly the edges of G.
void	G.sort_nodes( )	the nodes of G are sorted increasingly according to their contents. *Precondition*: *vtype* is linearly ordered.
void	G.sort_edges( )	the edges of G are sorted increasingly according to their contents. *Precondition*: *etype* is linearly ordered.
void	G.write(*string fname*)	writes G to the file with name $fname$. The output operators *operator* $\ll$ (*ostream*&, *const vtype*&) and *operator* $\ll$ (*ostream*&, *const etype*&)(cf. section 1.6) must be defined.
int	G.read(*string fname*)	reads G from the file with name $fname$. The input operators *operator* $\gg$ (*istream*&, *vtype*&) and *operator* $\gg$ (*istream*&, *etype*&) (cf. section 1.6) must be defined. Returns error code 1 if file $fname$ does not exist 2 if graph is not of type *GRAPH*<*vtype*, *etype*> 3 if file $fname$ does not contain a graph 0 if reading was successful.

4. Implementation

Parameterized graphs are derived from directed graphs. All additional operations for manipulating the node and edge entries take constant time.

8.3 Undirected Graphs (ugraph)

1. Definition

An instance U of the data type *ugraph* is an undirected graph as defined in section 8.1.

#include < *LEDA/ugraph.h* >

2. Creation

ugraph U; creates an instance U of type *ugraph* and initializes it to the empty undirected graph.

ugraph U(*graph* G); creates an instance U of type *ugraph* and initializes it with an undirected copy of G.

3. Operations

see section 8.1.

4. Implementation

see section 8.1.

8.4 Parameterized Ugraphs (UGRAPH)

1. Definition

A parameterized undirected graph G is an undirected graph whose nodes and contain additional (user defined) data (cf. 8.2). Every node contains an element of a data type *vtype*, called the node type of G and every edge contains an element of a data type *etype* called the edge type of G.

#include < *LEDA/ugraph.h* >

UGRAPH<*vtype*, *etype*> U;

creates an instance U of type *ugraph* and initializes it to the empty undirected graph.

2. Operations

see section 8.2.

3. Implementation

see section 8.2.

8.5 Planar Maps (planar_map)

1. Definition

An instance M of the data type *planar_map* is the combinatorial embedding of a planar graph, i.e., M is bidirected (for every edge (v, w) of M the reverse edge (w, v) is also in M) and there is a planar embedding of M such that for every node v the ordering of the edges in the adjacency list of v corresponds to the counter-clockwise ordering of these edges around v in the embedding.

#include < *LEDA/planar_map.h* >

2. Creation

planar_map M(*graph* G);

creates an instance M of type *planar_map* and initializes it to the planar map represented by the directed graph G.
Precondition: G represents a bidirected planar map, i.e. for every edge (v, w) in G the reverse edge (w, v) is also in G and there is a planar embedding of G such that for every node v the ordering of the edges in the adjacency list of v corresponds to the counter-clockwise ordering of these edges around v in the embedding.

3. Operations

edge M.new_edge(*edge e1*, *edge e2*)

inserts the edge $e = (source(e_1), source(e_2))$ and its reversal into M and returns e.
Precondition: e_1 and e_2 are bounding the same face F.
The operation splits F into two new faces.

face M.del_edge(*edge e*) — deletes the edge e and its reversal from M. The two faces adjacent to e are united to one new face which is returned.

edge M.split_edge(*edge e*) — splits edge $e = (v, w)$ and its reversal $r = (w, v)$ into edges (v, u), (u, w), (w, u), and (u, v). Returns the edge (u, w).

node M.new_node(*list*<*edge*> *el*)

splits the face bounded by the edges in el by inserting a new node u and connecting it to all source nodes of edges in el.
Precondition: all edges in el bound the same face.

node M.new_node(*face f*) — splits face f into triangles by inserting a new node u and connecting it to all nodes of f. Returns u.

list<*edge*> M.triangulate( ) triangulates all faces of M by inserting new edges. The list of inserted edges is returned.

4. Implementation

Planar maps are implemented by parameterized directed graphs. All operations take constant time, except for new_edge and del_edge which take time $O(f)$ where f is the number of edges in the created faces and triangulate and straight_line_embedding which take time $O(n)$ where n is the current size (number of edges) of the planar map.

8.6 Parameterized Planar Maps (PLANAR_MAP)

1. Definition

A parameterized planar map M is a planar map whose nodes, edges and faces contain additional (user defined) data. Every node contains an element of a data type *vtype*, called the node type of M,every edge contains an element of a data type *etype*, called the edge type of M, and every face contains an element of a data type $ftype$ called the face type of M. All operations of the data type *planar_map* are also defined for instances of any parameterized planar_map type. For parameterized planar maps there are additional operations to access or update the node and face entries.

#include < *LEDA/planar_map.h* >

2. Creation

PLANAR_MAP<*vtype*, *etype*, *ftype*>
M(*GRAPH*<*vtype*, *etype*> G);

creates an instance M of type *PLANAR_MAP*<*vtype*, *etype*, *ftype*> and initializes it to the planar map represented by the parameterized directed graph G. The node and edge entries of G are copied into the corresponding nodes and edges of M. Every face f of M is assigned the default value of type $ftype$.
Precondition: G represents a planar map.

3. Operations

vtype	M.inf(*node* v)	returns the information of node v.
etype	M.inf(*edge* e)	returns the information of node v.
ftype	M.inf(*face* f)	returns the information of face f.
vtype&	M[*node* v]	returns a reference to the information of node v.
etype&	M[*edge* e]	returns a reference to the information of edge e.
ftype&	M[*face* f]	returns a reference to the information of face f.
void	M.assign(*node* v, *vtype* x)	makes x the information of node v.
void	M.assign(*edge* e, *etype* x)	makes x the information of node v.
void	M.assign(*face* f, *ftype* x)	makes x the information of face f.

edge M.new_edge(*edge e1*, *edge e2*, *ftype y*)

inserts the edge $e = (source(e_1), source(e_2))$ and its reversal edge e' into M.
Precondition: e_1 and e_2 are bounding the same face F. The operation splits F into two new faces f, adjacent to edge e, and f', adjacent to edge e', with inf(f) = inf (F) and inf(f') = y.

edge M.split_edge(*edge e*, *vtype x*)

splits edge $e = (v, w)$ and its reversal $r = (w, v)$ into edges (v, u), (u, w), (w, u), and (u, v). Assigns information x to the created node u and returns the edge (u, w).

node M.new_node(*list<edge>& el*, *vtype x*)

splits the face bounded by the edges in el by inserting a new node u and connecting it to all source nodes of edges in el. Assigns information x to u and returns u.
Precondition: all edges in el bound the same face.

node M.new_node(*face f*, *vtype x*)

splits face f into triangles by inserting a new node u with information x and connecting it to all nodes of f. Returns u.

4. Implementation

Parameterized planar maps are derived from planar maps. All additional operations for manipulating the node and edge contents take constant time.

8.7 Node Arrays (node_array)

1. Definition

An instance A of the parameterized data type *node_array*<*E*> is a partial mapping from the node set of a graph G to the set of variables of type E, called the element type of the array. The domain I of A is called the index set of A and $A(v)$ is called the element at position v. A is said to be valid for all nodes in I.

#include < *LEDA/node_array.h* >

2. Creation

node_array<*E*> A;   creates an instance A of type *node_array*<*E*> with empty index set.

node_array<*E*> A(*graph* G);

creates an instance A of type *node_array*<*E*> and initializes the index set of A to the current node set of graph G.

node_array<*E*> A(*graph* G, E x);

creates an instance A of type *node_array*<*E*>, sets the index set of A to the current node set of graph G and initializes $A(v)$ with x for all nodes v of G.

node_array<*E*> A(*graph* G, *int* n, E x);

creates an instance A of type *node_array*<*E*> valid for up to n nodes of graph G and initializes $A(v)$ with x for all nodes v of G.
Precondition: $n \geq |V|$.
A is also valid for the next $n - |V|$ nodes added to G.

3. Operations

graph	A.get_graph( )	returns a reference to the graph of A.
$E\&$	A[*node* v]	returns the variable $A(v)$. *Precondition*: A must be valid for v.
void	A.init(*graph* G)	sets the index set I of A to the node set of G, i.e., makes A valid for all nodes of G.
void	A.init(*graph* G, E x)	makes A valid for all nodes of G and sets $A(v) = x$ for all nodes v of G.
void	A.init(*graph* G, *int* n, E x)	makes A valid for at most n nodes of G and sets $A(v) = x$ for all nodes v of G. *Precondition*: $n \geq |V|$. A is also valid for the next $n - |V|$ nodes added to G.

bool A.use_node_data(*graph* G, E x)

use free data slots in the nodes of G (if available) for storing the entries of A. The number of additional data slots in the nodes and edges of a graph can be specified in the *graph*::*graph*(*int n_slots*, *int e_slots*) constructor. The result is *true* if a free slot is available and *false* otherwise.

4. Implementation

Node arrays for a graph G are implemented by C++vectors and an internal numbering of the nodes and edges of G. The access operation takes constant time, *init* takes time $O(n)$, where n is the number of nodes in G. The space requirement is $O(n)$.

Remark: A node array is only valid for a bounded number of nodes of G. This number is either the number of nodes of G at the moment of creation of the array or it is explicitely set by the user. Dynamic node arrays can be realized by node maps (cf. section 8.10).

8.8 Edge Arrays (edge_array)

1. Definition

An instance A of the parameterized data type *edge_array*<*E*> is a partial mapping from the edge set of a graph G to the set of variables of type E, called the element type of the array. The domain I of A is called the index set of A and $A(e)$ is called the element at position e. A is said to be valid for all edges in I.

#include < *LEDA/edge_array.h* >

2. Creation

edge_array<*E*> A; creates an instance A of type *edge_array*<*E*> with empty index set.

edge_array<*E*> A(*graph* G);

creates an instance A of type *edge_array*<*E*> and initializes the index set of A to be the current edge set of graph G.

edge_array<*E*> A(*graph* G, E x);

creates an instance A of type *edge_array*<*E*>, sets the index set of A to the current edge set of graph G and initializes $A(v)$ with x for all edges v of G.

edge_array<*E*> A(*graph* G, *int* n, E x);

creates an instance A of type *edge_array*<*E*> valid for up to n edges of graph G and initializes $A(e)$ with x for all edges e of G.
Precondition: $n \geq |E|$.
A is also valid for the next $n - |E|$ edges added to G.

3. Operations

graph	A.get_graph( )	returns a reference to the graph of A.
E&	A[*edge* e]	returns the variable $A(e)$. *Precondition*: A must be valid for e.
void	A.init(*graph* G)	sets the index set I of A to the edge set of G, i.e., makes A valid for all edges of G.
void	A.init(*graph* G, E x)	makes A valid for all edges of G and sets $A(e) = x$ for all edges e of G.

void A.init(*graph* G, *int* n, E x)

makes A valid for at most n edges of G and sets $A(e) = x$ for all edges e of G.
Precondition: $n \geq |E|$.
A is also valid for the next $n - |E|$ edges added to G.

bool A.use_edge_data(*graph* G, E x)

use free data slots in the edges of G (if available) for storing the entries of A. The number of additional data slots in the nodes and edges of a graph can be specified in the *graph*::*graph*(*int n_slots*, *int e_slots*) constructor. The result is *true* if a free slot is available and *false* otherwise.

4. Implementation

Edge arrays for a graph G are implemented by C++vectors and an internal numbering of the nodes and edges of G. The access operation takes constant time, *init* takes time $O(n)$, where n is the number of edges in G. The space requirement is $O(n)$.

Remark: An edge array is only valid for a bounded number of edges of G. This number is either the number of edges of G at the moment of creation of the array or it is explicitely set by the user. Dynamic edge arrays can be realized by edge maps (cf. section 8.11).

8.9 Face Arrays (face_array)

1. Definition

An instance A of the parameterized data type *face_array*<E> is a partial mapping from the face set of a graph G to the set of variables of type E, called the element type of the array. The domain I of A is called the index set of A and $A(v)$ is called the element at position f. A is said to be valid for all faces in I.

#include < *LEDA/face_array.h* >

2. Creation

face_array<E> A; creates an instance A of type *face_array*<E> with empty index set.

face_array<E> A(*graph* G);

creates an instance A of type *face_array*<E> and initializes the index set of A to the current face set of graph G.

face_array<E> A(*graph* G, E x);

creates an instance A of type *face_array*<E>, sets the index set of A to the current face set of graph G and initializes $A(f)$ with x for all faces f of G.

face_array<E> A(*graph* G, *int* n, E x);

creates an instance A of type *face_array*<E> valid for up to n faces of graph G and initializes $A(f)$ with x for all faces f of G.
Precondition: $n \geq |V|$.
A is also valid for the next $n - |V|$ faces added to G.

3. Operations

graph	A.get_graph( )	returns a reference to the graph of A.
E&	A[*face* f]	returns the variable $A(f)$. *Precondition*: A must be valid for f.
void	A.init(*graph* G)	sets the index set I of A to the face set of G, i.e., makes A valid for all faces of G.
void	A.init(*graph* G, E x)	makes A valid for all faces of G and sets $A(f) = x$ for all faces f of G.

void A.init(*graph* G, *int* n, E x)

makes A valid for at most n faces of G and sets $A(f) = x$ for all faces f of G.
Precondition: $n \geq |V|$.
A is also valid for the next $n - |V|$ faces added to G.

bool A.use_face_data(*graph* G, E x)

use free data slots in the faces of G (if available) for storing the entries of A. The number of additional data slots in the nodes and edges of a graph can be specified in the *graph*::*graph*(*int n_slots*, *int e_slots*) constructor. The result is *true* if a free slot is available and *false* otherwise.

4. Implementation

Node arrays for a graph G are implemented by C++vectors and an internal numbering of the faces and edges of G. The access operation takes constant time, *init* takes time $O(n)$, where n is the number of faces in G. The space requirement is $O(n)$.

Remark: A face array is only valid for a bounded number of faces of G. This number is either the number of faces of G at the moment of creation of the array or it is explicitely set by the user. Dynamic face arrays can be realized by face maps (cf. section 8.10).

8.10 Node Maps (node_map)

1. Definition

An instance of the data type *node_map*<*E*> is a map for the nodes of a graph G, i.e., equivalent to *map*<*node*, *E*> (cf. 6.8). It can be used as a dynamic variant of the data type *node_array* (cf. 8.7). **New:** Since *node_map*<*E*> is derived from *node_array*<*E*> node maps can be passed (by reference) to functions with node array parameters. In particular, all LEDA graph algorithms expecting a *node_array*<*E*>& argument can be passed a *node_map*<*E*> instead.

#include < *LEDA/node_map.h* >

2. Creation

node_map<*E*> M;

introduces a variable M of type *node_map*<*E*> and initializes it to the map with empty domain.

node_map<*E*> M(*graph* G);

introduces a variable M of type *node_map*<*E*> and initializes it with a mapping m from the set of all nodes of G into the set of variables of type E. The variables in the range of m are initialized by a call of the default constructor of type E.

node_map<*E*> M(*graph* G, *E* x);

introduces a variable M of type *node_map*<*E*> and initializes it with a mapping m from the set of all nodes of G into the set of variables of type E. The variables in the range of m are initialized with a copy of x.

3. Operations

graph	M.get_graph( )	returns a reference to the graph of M.
void	M.init( )	makes M a node map with empty domain.
void	M.init(*graph* G)	makes M a mapping m from the set of all nodes of G into the set of variables of type E. The variables in the range of m are initialized by a call of the default constructor of type E.
void	M.init(*graph* G, *E* x)	makes M a mapping m from the set of all nodes of G into the set of variables of type E. The variables in the range of m are initialized with a copy of x.
E&	M[*node* v]	returns the variable $M(v)$.

4. Implementation

Node maps are implemented by an efficient hashing method based on the internal numbering of the nodes. An access operation takes expected time $O(1)$.

8.11 Edge Maps (edge_map)

1. Definition

An instance of the data type *edge_map*<*E*> is a map for the edges of a graph G, i.e., equivalent to *map*<*edge*, *E*> (cf. 6.8). It can be used as a dynamic variant of the data type *edge_array* (cf. 8.8). **New:** Since *edge_map*<*E*> is derived from *edge_array*<*E*> edge maps can be passed (by reference) to functions with edge array parameters. In particular, all LEDA graph algorithms expecting an *edge_array*<*E*>& argument can be passed an *edge_map*<*E*>& instead.

#include < *LEDA/edge_map.h* >

2. Creation

edge_map<*E*> M; introduces a variable M of type *edge_map*<*E*> and initializes it to the map with empty domain.

edge_map<*E*> M(*graph* G);

introduces a variable M of type *edge_map*<*E*> and initializes it with a mapping m from the set of all edges of G into the set of variables of type E. The variables in the range of m are initialized by a call of the default constructor of type E.

edge_map<*E*> M(*graph* G, *E* x);

introduces a variable M of type *edge_map*<*E*> and initializes it with a mapping m from the set of all edges of G into the set of variables of type E. The variables in the range of m are initialized with a copy of x.

3. Operations

graph	M.get_graph( )	returns a reference to the graph of M.
void	M.init( )	makes M a edge map with empty domain.
void	M.init(*graph* G)	makes M a mapping m from the set of all edges of G into the set of variables of type E. The variables in the range of m are initialized by a call of the default constructor of type E.
void	M.init(*graph* G, *E* x)	makes M a mapping m from the set of all edges of G into the set of variables of type E. The variables in the range of m are initialized with a copy of x.
E&	M[*edge* e]	returns the variable $M(v)$.

4. Implementation

Edge maps are implemented by an efficient hashing method based on the internal numbering of the edges. An access operation takes expected time $O(1)$.

8.12 Face Maps (face_map)

1. Definition

An instance of the data type *face_map*<*E*> is a map for the faces of a graph G, i.e., equivalent to *map*<*face*, *E*> (cf. 6.8). It can be used as a dynamic variant of the data type *face_array* (cf. 8.9). **New:** Since *face_map*<*E*> is derived from *face_array*<*E*> face maps can be passed (by reference) to functions with face array parameters. In particular, all LEDA graph algorithms expecting a *face_array*<*E*>& argument can be passed a *face_map*<*E*> instead.

#include < *LEDA/face_map.h* >

2. Creation

face_map<*E*> M; introduces a variable M of type *face_map*<*E*> and initializes it to the map with empty domain.

face_map<*E*> M(*graph* G);

introduces a variable M of type *face_map*<*E*> and initializes it with a mapping m from the set of all faces of G into the set of variables of type E. The variables in the range of m are initialized by a call of the default constructor of type E.

face_map<*E*> M(*graph* G, E x);

introduces a variable M of type *face_map*<*E*> and initializes it with a mapping m from the set of all faces of G into the set of variables of type E. The variables in the range of m are initialized with a copy of x.

3. Operations

graph	M.get_graph( )	returns a reference to the graph of M.
void	M.init( )	makes M a face map with empty domain.
void	M.init(*graph* G)	makes M a mapping m from the set of all faces of G into the set of variables of type E. The variables in the range of m are initialized by a call of the default constructor of type E.
void	M.init(*graph* G, E x)	makes M a mapping m from the set of all faces of G into the set of variables of type E. The variables in the range of m are initialized with a copy of x.
E&	M[*face* f]	returns the variable $M(v)$.

4. Implementation

Face maps are implemented by an efficient hashing method based on the internal numbering of the faces. An access operation takes expected time $O(1)$.

8.13 Two Dimensional Node Arrays (node_matrix)

1. Definition

An instance M of the parameterized data type *node_matrix*<*E*> is a partial mapping from the set of node pairs $V \times V$ of a graph to the set of variables of data type E, called the element type of M. The domain I of M is called the index set of M. M is said to be valid for all node pairs in I. A node matrix can also be viewed as a node array with element type *node_array*<*E*> (*node_array*<*node_array*<*E*> >).

#include < *LEDA/node_matrix.h* >

2. Creation

node_matrix<*E*> M; creates an instance M of type *node_matrix*<*E*> and initializes the index set of M to the empty set.

node_matrix<*E*> M(*graph G*);

creates an instance M of type *node_matrix*<*E*> and initializes the index set to be the set of all node pairs of graph G, i.e., M is made valid for all pairs in $V \times V$ where V is the set of nodes currently contained in G.

node_matrix<*E*> M(*graph G*, *E x*);

creates an instance M of type *node_matrix*<*E*> and initializes the index set of M to be the set of all node pairs of graph G, i.e., M is made valid for all pairs in $V \times V$ where V is the set of nodes currently contained in G. In addition, $M(v, w)$ is initialized with x for all nodes $v, w \in V$.

3. Operations

void M.init(*graph G*) sets the index set of M to $V \times V$, where V is the set of all nodes of G.

void M.init(*graph G*, *E x*)

sets the index set of M to $V \times V$, where V is the set of all nodes of G and initializes $M(v, w)$ to x for all $v, w \in V$.

const node_array<*E*>& M[*node v*] returns the node_array $M(v)$.

const E& M(*node v*, *node w*) returns the variable $M(v, w)$.
Precondition: M must be valid for v and w.

4. Implementation

Node matrices for a graph G are implemented by vectors of node arrays and an internal numbering of the nodes of G. The access operation takes constant time, the init operation takes time $O(n^2)$, where n is the number of nodes currently contained in G. The space requirement is $O(n^2)$. Note that a node matrix is only valid for the nodes contained in G at the moment of the matrix declaration or initialization (*init*). Access operations for later added nodes are not allowed.

8.14 Two-Dimensional Node Maps (node_map2)

1. Definition

An instance of the data type *node_map2*<*E*> is a map2 for the pairs of nodes of a graph G, i.e., equivalent to $map2<node, node, E>$ (cf. 6.9). It can be used as a dynamic variant of the data type *node_matrix* (cf. 8.13).

#include < *LEDA/node_map2.h* >

2. Creation

node_map2<*E*> M; introduces a variable M of type *node_map2*<*E*> and initializes it to the map2 with empty domain.

node_map2<*E*> M(*graph* G);

introduces a variable M of type *node_map2*<*E*> and initializes it with a mapping m from the set of all nodes of G into the set of variables of type E. The variables in the range of m are initialized by a call of the default constructor of type E.

node_map2<*E*> M(*graph* G, E x);

introduces a variable M of type *node_map2*<*E*> and initializes it with a mapping m from the set of all nodes of G into the set of variables of type E. The variables in the range of m are initialized with a copy of x.

3. Operations

void	M.init( )	makes M a node map2 with empty domain.
void	M.init(*graph* G)	makes M to a mapping m from the set of all nodes of G into the set of variables of type E. The variables in the range of m are initialized by a call of the default constructor of type E.
void	M.init(*graph* G, E x)	makes M to a mapping m from the set of all nodes of G into the set of variables of type E. The variables in the range of m are initialized with a copy of x.
$E\&$	M(*node* v, *node* w)	returns the variable $M(v, w)$.
bool	M.defined(*node* v, *node* w)	returns true if $(v, w) \in dom(m)$ and false otherwise.

4. Implementation

Node maps are implemented by an efficient hashing method based on the internal numbering of the nodes. An access operation takes expected time $O(1)$.

8.15 Sets of Nodes (node_set)

1. Definition

An instance S of the data type *node_set* is a subset of the nodes of a graph G. S is said to be valid for the nodes of G.

#include < *LEDA/node_set.h* >

2. Creation

node_set S(*graph* G);

creates an instance S of type *node_set* valid for all nodes currently contained in graph G and initializes it to the empty set.

3. Operations

void	S.insert(*node* x)	adds node x to S.
void	S.del(*node* x)	removes node x from S.
bool	S.member(*node* x)	returns true if x in S, false otherwise.
node	S.choose( )	returns a node of S.
int	S.size( )	returns the size of S.
bool	S.empty( )	returns true iff S is the empty set.
void	S.clear( )	makes S the empty set.

4. Implementation

A node set S for a graph G is implemented by a combination of a list L of nodes and a node array of list_items associating with each node its position in L. All operations take constant time, except for clear which takes time $O(S)$. The space requirement is $O(n)$, where n is the number of nodes of G.

8.16 Sets of Edges (edge_set)

1. Definition

An instance S of the data type *edge_set* is a subset of the edges of a graph G. S is said to be valid for the edges of G.

#include < *LEDA/edge_set.h* >

2. Creation

edge_set S(*graph* G);

creates an instance S of type *edge_set* valid for all edges currently in graph G and initializes it to the empty set.

3. Operations

void	S.insert(*edge* x)	adds edge x to S.
void	S.del(*edge* x)	removes edge x from S.
bool	S.member(*edge* x)	returns true if x in S, false otherwise.
edge	S.choose( )	returns an edge of S.
int	S.size( )	returns the size of S.
bool	S.empty( )	returns true iff S is the empty set.
void	S.clear( )	makes S the empty set.

4. Implementation

An edge set S for a graph G is implemented by a combination of a list L of edges and an edge array of list_items associating with each edge its position in L. All operations take constant time, except for clear which takes time $O(S)$. The space requirement is $O(n)$, where n is the number of edges of G.

8.17 Lists of Nodes (node_list)

1. Definition

An instance of the data type *node_list* is a doubly linked list of nodes. It is implemented more efficiently than the general list type *list*<*node*> (5.7). However, it can only be used with the restriction that every node is contained in at most one *node_list*.

#include < *LEDA/node_list.h* >

2. Creation

node_list L;	introduces a variable L of type *node_list* and initializes it with the empty list.

3. Operations

void	L.append(*node* v)	appends v to list L.
void	L.push(*node* v)	adds v at the front of L.
void	L.insert(*node* v, *node* w)	inserts v after w into L. *Precondition*: $w \in L$.
node	L.pop( )	deletes the first node from L and returns it. *Precondition*: L is not empty.
void	L.del(*node* v)	deletes v from L. *Precondition*: $v \in L$.
bool	L.member(*node* v)	returns true if $v \in L$ and false otherwise.
bool	L(*node* v)	returns true if $v \in L$ and false otherwise.
node	L.head( )	returns the first node in L (nil if L is empty).
node	L.tail( )	returns the last node in L (nil if L is empty).
node	L.succ(*node* v)	returns the successor of v in L. *Precondition*: $v \in L$.
node	L.pred(*node* v)	returns the predecessor of v in L. *Precondition*: $v \in L$.
node	L.cyclic_succ(*node* v)	returns the cyclic successor of v in L. *Precondition*: $v \in L$.
node	L.cyclic_pred(*node* v)	returns the cyclic predecessor of v in L. *Precondition*: $v \in L$.

bool	L.empty( )	returns *true* if L is empty and *false* otherwise.
void	L.clear( )	makes L the empty list.

forall(x, L) { "the elements of L are successively assigned to x" }

8.18 Node Partitions (node_partition)

1. Definition

An instance P of the data type *node_partition* is a partition of the nodes of a graph G.

#include < *LEDA/node_partition.h* >

2. Creation

node_partition P(*graph* G);

creates a *node_partition* P containing for every node v in G a block $\{v\}$.

3. Operations

int	P.same_block(*node* v, *node* w)	returns true if v and w belong to the same block of P, false otherwise.
void	P.union_blocks(*node* v, *node* w)	unites the blocks of P containing nodes v and w.
void	P.split(*list*<*node*> L)	makes all nodes in L to singleton blocks. *Precondition*: L is a union of blocks.
void	P.make_rep(*node* v)	makes v the canonical representative of the block containing v.
node	P.find(*node* v)	returns a canonical representative node of the block that contains node v.
int	P.size(*node* v)	returns the size of the block that contains node v.
int	P.number_of_blocks( )	returns the number of blocks of P.
node	P(*node* v)	returns P.find(v).

4. Implementation

A node partition for a graph G is implemented by a combination of a partition P and a node array of *partition_item* associating with each node in G a partition item in P. Initialization takes linear time, union_blocks takes time $O(1)$ (worst-case), and same_block and find take time $O(\alpha(n))$ (amortized). The cost of a split is proportional to the cost of the blocks dismantled. The space requirement is $O(n)$, where n is the number of nodes of G.

8.19 Node Priority Queues (node_pq)

1. Definition

An instance Q of the parameterized data type *node_pq*<P> is a partial function from the nodes of a graph G to a linearly ordered type P of priorities. The priority of a node is sometimes called the information of the node. For every graph G only one *node_pq*<P> may be used and every node of G may be contained in the queue at most once (cf. section 7.1 for general priority queues).

#include < *LEDA/node_pq.h* >

2. Creation

node_pq<P> Q(*graph* G);

creates an instance Q ot type *node_pq*<P> for the nodes of graph G with $dom(Q) = \emptyset$.

3. Operations

void	Q.insert(*node* v, P x)	adds the node v with priority x to Q. *Precondition*: $v \notin dom(Q)$.
P	Q.prio(*node* v)	returns the priority of node v.
bool	Q.member(*node* v)	returns true if v in Q, false otherwise.
void	Q.decrease_p(*node* v, P x)	makes x the new priority of node v. *Precondition*: $x \leq Q$.prio(v).
node	Q.find_min( )	returns a node with minimal priority (nil if Q is empty).
void	Q.del(*node* v)	removes the node v from Q.
node	Q.del_min( )	removes a node with minimal priority from Q and returns it (nil if Q is empty).
void	Q.clear( )	makes Q the empty node priority queue.
int	Q.size( )	returns $\|dom(Q)\|$.
int	Q.empty( )	returns true if Q is the empty node priority queue, false otherwise.
P	Q.inf(*node* v)	returns the priority of node v.

4. Implementation

Node priority queues are implemented by binary heaps and node arrays. Operations insert, del_node, del_min, decrease_p take time $O(\log m)$, find_min and empty take time $O(1)$ and clear takes time $O(m)$, where m is the size of Q. The space requirement is $O(n)$, where n is the number of nodes of G.

8.20 Bounded Node Priority Queues (b_node_pq)

1. Definition

An instance of the data type *b_node_pq*<*N*> is a priority queue of nodes with integer priorities with the restriction that the size of the minimal interval containing all priorities in the queue is bounded by N, the minimum priority is never decreasing, and every node is contained in at most one queue. When applied to the empty queue the del_min - operation returns a special default minimum node defined in the constructor of the queue.

#include < *LEDA/b_node_pq.h* >

2. Creation

b_node_pq<*N*> PQ; introduces a variable PQ of type *b_node_pq*<*N*> and initializes it with the empty queue with default minimum node *nil*.

b_node_pq<*N*> PQ(*node* v);

introduces a variable PQ of type *b_node_pq*<*N*> and initializes it with the empty queue with default minimum node v.

3. Operations

node	PQ.del_min( )	removes the node with minimal priority from PQ and returns it (the default minimum node if PQ is empty).
void	PQ.insert(*node* w, *int* p)	adds node w with priority p to PQ.
void	PQ.del(*node* w)	deletes node w from PQ.

4. Implementation

Bounded node priority queues are implemented by cyclic arrays of doubly linked node lists.

5. Example

Using a *b_node_pq* in Dijktra's shortest paths algorithm.

```
int dijkstra(const GRAPH<int,int>& g, node s, node t)
{ node_array<int> dist(g,MAXINT);
  b_node_pq<100> PQ(t);  // on empty queue del_min returns t
  dist[s] = 0;
```

```
  for (node v = s;  v != t; v = PQ.del_min() )
  { int dv = dist[v];
    edge e;
    forall_adj_edges(e,v)
    { node w = g.opposite(v,e);
      int d = dv + g.inf(e);
      if (d < dist[w])
      { if (dist[w] != MAXINT) PQ.del(w);
        dist[w] = d;
        PQ.insert(w,d);
       }
     }
   }
  return dist[t];
}
```

8.21 Graph Generators (graph_gen)

void complete_graph(*graph*& G, *int* n)

creates a complete graph G with n nodes.

void complete_ugraph(*graph*& G, *int* n)

creates a complete undirected graph G with n nodes.

void random_graph_noncompact(*graph*& G, *int* n, *int* m)

generates a random graph with n nodes and m edges. No attempt is made to store all edges in the same adjacency list consecutively. This function is only included for paedagocial reasons.

void random_graph(*graph*& G, *int* n, *int* m, *bool* *no_anti_parallel_edges*, *bool* *loopfree*, *bool* *no_parallel_edges*)

generates a random graph with n nodes and m edges. All edges in the same adjacency list are stored consecutively.
If *no_parallel_edges* is true then no parallel edges are generated, if *loopfree* is true then no self loops are generated, and if *no_anti_parallel_edges* is true then no anti parallel edges are generated.

void random_graph(*graph*& G, *int* n, *int* m)

same as $random\_graph(G, n, m, false, false, false)$.

void random_simple_graph(*graph*& G, *int* n, *int* m)

same as $random\_graph(G, n, m, false, false, true)$.

void random_simple_loopfree_graph(*graph*& G, *int* n, *int* m)

same as $random\_graph(G, n, m, false, true, true)$.

void random_simple_undirected_graph(*graph*& G, *int* n, *int* m)

same as $random\_graph(G, n, m, true, true, true)$.

void random_graph(*graph*& G, *int* n, *double* p)

generates a random graph with n nodes. Each edge of the complete graph with n nodes is included with probability p .

void test_graph(*graph*& G)

creates interactively a user defined graph G.

void complete_bigraph(*graph*& G, *int* a, *int* b, *list*<*node*>& A, *list*<*node*>& B)

creates a complete bipartite graph G with a nodes on side A and b nodes on side B. All edges are directed from A to B.

void random_bigraph(*graph*& G, *int* a, *int* b, *int* m, *list*<*node*>& A, *list*<*node*>& B, *int* $k = 1$)

creates a random bipartite graph G with a nodes on side A, b nodes on side B, and m edges. All edges are directed from A to B.
If $k > 1$ then A and B are divided into k groups of about equal size and the nodes in the i-th group of A have their edges to nodes in the $i-1$-th and $i+1$-th group in B. All indices are modulo k.

void test_bigraph(*graph*& G, *list*<*node*>& A, *list*<*node*>& B)

creates interactively a user defined bipartite graph G with sides A and B. All edges are directed from A to B.

void grid_graph(*graph*& G, *int* n)

creates a grid graph G with $n \times n$ nodes.

void grid_graph(*graph*& G, *node_array*<*double*>& *xcoord*, *node_array*<*double*>& *ycoord*, *int* n)

creates a grid graph G of size $n \times n$ embedded into the unit square. The embedding is given by $xcoord[v]$ and $ycoord[v]$ for every node v of G.

void cmdline_graph(*graph*& G, *int* *argc*, *char* $**argv$)

builds graph G as specified by the command line arguments:

prog		$\longrightarrow$	test_graph()
prog	n	$\longrightarrow$	complete_graph(n)
prog	n m	$\longrightarrow$	test_graph(n, m)
prog	*file*	$\longrightarrow$	G.read_graph(*file*)

Planar graphs: Combinatorial Constructions

A maximal planar map with n nodes, $n \geq 3$, has $3n-6$ uedges. It is constructed iteratively. For $n = 1$, the graph consists of a single isolated node, for $n = 2$, the graph consists of two nodes and one uedge, for $n = 3$ the graph consists of three nodes and three uedges. For $n > 3$, a random maximal planar map with $n-1$ nodes is constructed first and then an additional node is put into a random face.

The generator with the additional parameter m first generates a maximal planar map and then deletes all but m edges.

The generators with the word map replaced by graph, first generate a map and then delete one edge from each uedge.

void maximal_planar_map(*graph*& G, *int* n)

creates a maximal planar map G with n nodes.

void random_planar_map(*graph*& G, *int* n, *int* m)

creates a random planar map G with n nodes and m edges.

void maximal_planar_graph(*graph*& G, *int* n)

creates a maximal planar graph G with n nodes.

void random_planar_graph(*graph*& G, *int* n, *int* m)

creates a random planar graph G with n nodes and m edges.

Planar graphs: Geometric Constructions

We have two kinds of geometric constructions: triangulations of point sets and intersection graphs of line segments. The functions *triangulation_map* choose points in the unit square and compute a triangulation of them and the functions *random_planar_graph* construct the intersection graphs of segments.

The generators with the word map replaced by graph, first generate a map and then delete one edge from each uedge.

void triangulation_map(*graph*& G, *node_array*<*double*>& *xcoord*, *node_array*<*double*>& *ycoord*, *int* n)

chooses n random points in the unit square and returns their triangulation as a plane map in G. The coordinates of node v are returned as $xcoord[v]$ and $ycoord[v]$. The coordinates are random number of the form x/K where $K = 2^{20}$ and x is a random integer between 0 (inclusive) and K (exclusive).

void triangulation_map(*graph*& G, *list*<*node*>& *outer_face*, *node_array*<*double*>& *xcoord*, *node_array*<*double*>& *ycoord*, *int* n)

as above, in addition the list of nodes of the outer face (convex hull) is returned in *outer_face* in clockwise order.

void triangulation_map(*graph*& G, *int* n)

as above, but only the map is returned.

void random_planar_map(*graph*& G, *node_array*<*double*>& *xcoord*, *node_array*<*double*>& *ycoord*, *int* n, *int* m)

chooses n random points in the unit square and computes their triangulation as a plane map in G. It then keeps all but m uedges. The coordinates of node v are returned as $xcoord[v]$ and $ycoord[v]$.

void triangulation_graph(*graph*& *G*, *node_array*<*double*>& *xcoord*, *node_array*<*double*>& *ycoord*, *int* *n*)

calls *triangulation_map* and keeps only one of the edges comprising a uedge.

void triangulation_graph(*graph*& *G*, *list*<*node*>& *outer_face*, *node_array*<*double*>& *xcoord*, *node_array*<*double*>& *ycoord*, *int* *n*)

calls *triangulation_map* and keeps only one of the edges comprising a uedge.

void triangulation_graph(*graph*& *G*, *int* *n*)

calls *triangulation_map* and keeps only one of the edges comprising a uedge.

void random_planar_graph(*graph*& *G*, *node_array*<*double*>& *xcoord*, *node_array*<*double*>& *ycoord*, *int* *n*, *int* *m*)

calls *random_planar_map* and keeps only one of the edges comprising a uedge.

void triangulated_planar_graph(*graph*& *G*, *int* *n*)

old name for *triangulation_graph*.

void triangulated_planar_graph(*graph*& *G*, *node_array*<*double*>& *xcoord*, *node_array*<*double*>& *ycoord*, *int* *n*)

old name for *triangulation_graph*.

void triangulated_planar_graph(*graph*& *G*, *list*<*node*>& *outer_face*, *node_array*<*double*>& *xcoord*, *node_array*<*double*>& *ycoord*, *int* *n*)

old name for *triangulation_graph*.

void random_planar_graph(*graph*& *G*, *node_array*<*double*>& *xcoord*, *node_array*<*double*>& *ycoord*, *int* *n*)

creates a random planar graph G with n nodes embedded into the unit sqare. The embedding is given by $xcoord[v]$ and $ycoord[v]$ for every node v of G. The generator chooses n segments whose endpoints have random coordinates of the form x/K, where K is the smallest power of two greater or equal to n, and x is a random integer in 0 to $K-1$. It then constructs the arrangement defined by the segments and keeps the n nodes with the smallest x-coordinates. Finally, it adds edges to make the graph connected.

void random_planar_graph(*graph*& *G*, *int* *n*)

creates a random planar graph G with n nodes. Uses the preceding function.

Series-Parallel Graphs

void random_sp_graph(*graph*& G, *int* n, *int* m)

creates a random series-parallel graph G with n nodes and m edges.

8.22 Miscellaneous Graph Functions (graph_misc)

1. Operations

#include < *LEDA/graph_misc.h* >

void CopyGraph(*graph*& H, *graph* G)

constructs a copy H of graph G.

void CopyGraph(*GRAPH*<*node*, *edge*>& H, *graph* G)

constructs a copy H of graph G such that $H[v]$ is the node of G that corresponds to v and $H[e]$ is the edge of G that corresponds to e.

void CopyGraph(*GRAPH*<*node*, *edge*>& H, *graph* G, *list*<*node*> V, *list*<*edge*> E)

constructs a copy H of the subgraph (V, E) of G such that $H[v]$ is the node of G that corresponds to v and $H[e]$ is the edge of G that corresponds to e. *Precondition*: V is a subset of the nodes of G and E is a subset of $V \times V$.

void CopyGraph(*GRAPH*<*node*, *edge*>& H, *graph* G, *list*<*edge*> E)

constructs a copy H of the subgraph of G induced by the edges in E.

bool Is_Simple(*graph* G) returns true if G is simple, i.e., has no parallel edges, false otherwise.

bool Is_Simple(*graph* G, *list*<*edge*>& *el*)

as above, but returns in addition the list of all edges sorted lexicographically by source and target node, i.e, all parallel edges appear consecutively in *el*.

bool Is_Loopfree(*graph* G) returns true if G is loopfree, i.e., has no edge whose source is equal to its target.

bool Is_Simple_Loopfree(*graph* G)

returns true if G is simple and loopfree.

bool Is_Simple_Undirected(*graph* G)

returns true if G is simple, loopfree, and has no anti-parallel edges.

bool Is_Bidirected(*graph* G) returns true if every edge has a reversal and false otherwise.

bool Is_Bidirected(*graph G*, *edge_array<edge>& rev*)

computes for every edge $e = (v, w)$ in G its reversal $rev[e] = (w, v)$ in G (nil if not present). Returns true if every edge has a reversal and false otherwise.

bool Is_Map(*graph G*)

tests whether G is a map.

int Genus(*graph G*)

computes the genus of G.
Precondition: G must be a map.

bool Is_Plane_Map(*graph G*)

tests whether G is a plane map, i.e, whether G is a map of genus zero.

bool Is_Planar_Map(*graph G*)

old name for Is_Plane_Map

bool Is_Acyclic(*graph G*)

returns true if the directed G is acyclic and false otherwise.

bool Is_Acyclic(*graph G*, *list<edge>& L*)

as above; in addition, constructs a list of edges L whose deletion makes G acyclic.

bool Is_Connected(*graph G*)

returns true if the undirected graph underlying G is connected and false otherwise.

bool Is_Biconnected(*graph G*)

returns true if the undirected graph underlying G is biconnected and false otherwise.

bool Is_Biconnected(*graph G*, *node& s*)

as above, computes a split vertex s if the result is *false*.

bool Is_Triconnected(*graph G*)

returns true if the undirected graph underlying G is triconnected and false otherwise. The running time is $O(n(n + m)))$.

bool Is_Triconnected(*graph G*, *node& s1*, *node& s2*)

as above, computes a split pair $s1, s2$ if the result is *false*.

bool Is_Bipartite(*graph G*)

returns true if G is bipartite and false otherwise.

bool Is_Bipartite(*graph G*, *list<node>& A*, *list<node>& B*)

returns true if G is bipartite and false otherwise. If G is bipartite the two sides are returned in A and B, respectively. If G is not bipartite the node sequence of an odd-length circle is returned in A..

bool Is_Planar(*graph G*)

returns true if G is planar and false otherwise.

bool	Is_Series_Parallel(*graph* G)	returns true if G is series-parallel and false otherwise.
void	Make_Acyclic(*graph*& G)	makes G acyclic by removing all DFS back edges.
list<*edge*>	Make_Simple(*graph*& G)	makes G simple by removing all but one from each set of parallel edges. Returns the list of remaining edges with parallel edges in the original graph.
void	Make_Bidirected(*graph*& G, *list*<*edge*>& L)	makes G bidirected by inserting missing reversal edges. Appends all inserted edges to list L.
list<*edge*>	Make_Bidirected(*graph*& G)	makes G bidirected by inserting missing reversal edges. Returns the list of inserted edges.
void	Make_Connected(*graph*& G, *list*<*edge*>& L)	makes G connected; appends all inserted edges to list L.
list<*edge*>	Make_Connected(*graph*& G)	makes G connected; returns the list of inserted edges.
void	Make_Biconnected(*graph*& G, *list*<*edge*>& L)	makes G biconnected; appends all inserted edges to list L.
list<*edge*>	Make_Biconnected(*graph*& G)	makes G biconnected; returns the list of inserted edges.
list<*node*>	Delete_Loops(*graph*& G)	returns the list of nodes with self-loops and deletes all self-loops.

8.23 Markov Chains (markov_chain)

1. Definition

A Markov Chain is a graph G in which each edge has an associated non-negative integer weight $w[e]$. For every node (with at least one outgoing edge) the total weight of the outgoing edges must be positive A random walk in a Markov chain starts at some node s and then performs steps according to the following rule:

Initially, s is the current node. Suppose node v is the current node and that $e_0, \ldots, e_{d-1}$ are the edges out of v. If v has no outgoing edge no further step can be taken. Otherwise, the walk follows edge e_i with probability proportional to $w[e_i]$ for all i, $0 \leq i < d$. The target node of the chosen edge becomes the new current node.

#include < *LEDA/markov_chain.h* >

2. Creation

markov_chain M(*graph* G, *edge_array*<*int*> w, *node* s = *nil*);

creates a Markov chain for the graph G with edge weights w. The node s is taken as the start vertex (*G.first_node*( ) if s is nil).

3. Operations

void	M.step(*int* T = 1)	performs T steps of the Markov chain.
node	M.current_node( )	returns current vertex.
int	M.current_outdeg( )	returns the outdegree of the current vertex.
int	M.number_of_steps( )	returns number of steps performed.
int	M.number_of_visits(*node* v)	returns number of visits to node v.
double	M.rel_freq_of_visit(*node* v)	returns number of visits divided by the total number of steps.

8.24 Dynamic Markov Chains (dynamic_markov_chain)

1. Definition

A Markov Chain is a graph G in which each edge has an associated non-negative integer weight $w[e]$. For every node (with at least one outgoing edge) the total weight of the outgoing edges must be positive A random walk in a Markov chain starts at some node s and then performs steps according to the following rule:

Initially, s is the current node. Suppose node v is the current node and that $e_0, \ldots, e_{d-1}$ are the edges out of v. If v has no outgoing edge no further step can be taken. Otherwise, the walk follows edge e_i with probability proportional to $w[e_i]$ for all i, $0 \leq i < d$. The target node of the chosen edge becomes the new current node.

The include file is `<LEDA/markov_chain.h>`

#include < *LEDA/markov_chain.h* >

2. Creation

dynamic_markov_chain M(*graph* G, *edge_array*<*int*> w, *node* s = *nil*);

creates a Markov chain for the graph G with edge weights w. The node s is taken as the start vertex (G.*first_node*( ) if s is nil).

3. Operations

void	M.step(*int* T = 1)	performs T steps of the Markov chain.
node	M.current_node( )	returns current vertex.
int	M.current_outdeg( )	returns the outdegree of the current vertex.
int	M.number_of_steps( )	returns number of steps performed.
int	M.number_of_visits(*node* v)	returns number of visits to node v.
double	M.rel_freq_of_visit(*node* v)	returns number of visits divided by the total number of steps.
int	M.set_weight(*edge* e, *int* g)	changes the weight of edge e to g and returns the old weight of e

8.25 GML Parser for Graphs (gml_graph)

1. Definition

An instance *parser* of the data type *gml_graph* is a parser for graphs in GML format [42]. It is possible to extend the parser by user defined rules. This parser is used by the *read_gml* of class *graph*. The following is small example graph (a triangle) in GML format.

```
# This is a comment.
graph [         # Lists start with '['.
  directed 1    # This is a directed graph (0 for undirected).

  # The following is an object of type string.
  # It will be ignored unless you specify a rule for graph.text.
  text "This is a string object."

  node [ id 1 ] # This defines a node with id 1.
  node [ id 2 ]
  node [ id 3 ]

  edge [ # This defines an edge leading from node 1 to node 2.
    source 1
    target 2
  ]
  edge [
    source 2
    target 3
  ]
  edge [
    source 3
    target 1
  ]
] # Lists end with ']'.
```

An input in GML format is a list of GML objects. Each object consists of a key word and a value. A value may have one out of four possible types, an integer (type *gml_int*), a double (type *gml_double*), a string (type *gml_string*), or a list of GML objects (type *gml_list*). Since a value can be a list of objects, we get a tree structure on the input. We can describe a class C of objects being in the same list and having the same key word by the so-called path. The path is the list of key words leading to an object in the class C.

In principle, every data structure can be expressed in GML format. This parser specializes on graphs. A graph is represented by an object with key word *graph* and type *gml_list*. The nodes of the graph are objects with path *graph.node* and type *gml_list*. Each node has a unique identifier, which is represented by an object of type *gml_int* with path *graph.node.id*. An edge is an object of type *gml_list* with the path *graph.edge*. Each edge has a source and a target. These are objects of type *gml_int* with path *graph.edge.source* and *graph.edge.target*, respectively. The integer values of *source* and *target* refer to node identifiers. There are some global graph attributes, too. An object of type *gml_int* with path *graph.directed* determines whether the graph is undirected (value 0) or directed (every other integer). The type of node parameters and edge parameters in parameterized graphs (see manual page GRAPH) can be given by objects of type *gml_string* with path *graph.nodeType* and *graph.edgeType*, respectively. Parameters of nodes and edges are represented by objects of type *gml_string* with path *graph.node.parameter* and *graph.edge.parameter*, respectively.

No list has to be in a specific order, e.g., you can freely mix *node* and *edge* objects in the *graph* list. If there are several objects in a class where just one object is required like *graph.node.id*, only the last such object is taken into account.

Objects in classes with no predefined rules are simply ignored. This means that an application A might add specific objects to a graph description in GML format and this description is still readable for another application B which simply does not care about the objects which are specific for A.

This parser supports reading user defined objects by providing a mechanism for dealing with those objects by means of callback functions. You can specify a rule for, e.g., objects with path *graph.node.weight* and type *gml_double* like in the following code fragment.

```
...
bool get_node_weight(const gml_object* gobj, graph* G, node v)
{
  double w = gobj->get_double();
  do something with w, the graph and the corresponding node v
  return true; or false if the operation failed
}
...
main()
{
  char* filename;
  ...
  graph G;
  gml_graph parser(G);
  parser.add_node_rule(get_node_weight,gml_double,"weight");
  bool parsing_ok = parser.parse(filename);
  ...
}
```

You can add rules for the graph, for nodes, and for edges. The difference between them is the type. The type of node rules is as in the example above `bool (*gml_node_rule)(const gml_object*, graph*, node)`, the type for edge rules is `bool (*gml_edge_rule)(const gml_object*, graph*, edge)`, and the type for graph rules is `bool (*gml_graph_rule)(const gml_object*, graph*)`. A GML object is represented by an instance of class `gml_object`. You can get its value by using `double gml_object::get_double()`, `int gml_object::get_int()` or `char* gml_object::get_string()`. If one of your rules returns *false* during parsing, then parsing fails and the graph is cleared.

#include < *LEDA/gml_graph.h* >

2. Creation

gml_graph parser(*graph*& *G*);

creates an instance *parser* of type *gml_graph* and initializes it for *graph G*.

gml_graph parser(*graph*& *G*, *const char* ∗ *filename*);

creates an instance *parser* of type *gml_graph* and reads *graph G* from the file *filename*.

gml_graph parser(*graph*& *G*, *istream*& *ins*);

creates an instance *parser* of type *gml_graph* and reads *graph G* from the input stream *ins*.

3. Operations

3.1 Parsing

bool parser.parse(*const char* ∗ *filename*)

parses the input taken from the file *filename* using the current set of rules. The graph specified in the constructor is set up accordingly. This operations returns *false* and clears the graph, if syntax or parse errors occur. Otherwise *true* is returned.

bool parser.parse(*istream*& *ins*)

parses the input taken from the input stream *ins*.

bool parser.parse_string(*string s*)

parses the input taken from string *s*.

3.2 Path Manipulation

void parser.reset_path( ) resets the current path to the empty path.

void parser.append(*const char* ∗ *key*)

appends *key* to the current path.

void parser.goback( ) — removes the last key word from the current path. If the current path is empty this operation has no effect.

3.3 User Defined Rules

void parser.add_graph_rule(*gml_graph_rule f*, *gml_value_type t*, *char* ∗ *key* = 0)

adds graph rule f for value type t and path *graph.key* to *parser*, if key is specified. Otherwise, f is added for the current path.

void parser.add_node_rule(*gml_node_rule f*, *gml_value_type t*, *char* ∗ *key* = 0)

adds node rule f for path *graph.node.key* (or the current path, if no key is specified) and value type t to *parser*.

void parser.add_edge_rule(*gml_edge_rule f*, *gml_value_type t*, *char* ∗ *key* = 0)

adds edge rule f for path *graph.edge.key* (or the current path, if no key is specified) and value type t to *parser*.

void parser.add_new_graph_rule(*gml_graph_rule f*)

adds graph rule f to *parser*. During parsing f is called whenever an object o with path *graph* and type *gml_list* is encountered. f is called before objects in the list of o are parsed.

void parser.add_new_node_rule(*gml_node_rule f*)

adds node rule f for path *graph.node* and value type *gml_list* to *parser*. f is called before objects in the corresponding list are parsed.

void parser.add_new_edge_rule(*gml_edge_rule f*)

adds edge rule f for path *graph.edge* and value type *gml_list* to *parser*. f is called before objects in the corresponding list are parsed.

void parser.add_graph_done_rule(*gml_graph_rule f*)

adds graph rule f to *parser*. During parsing f is called whenever an object o with path *graph* and type *gml_list* is encountered. f is called after all objects in the list of o are parsed.

void parser.add_node_done_rule(*gml_node_rule f*)

adds node rule f to *parser* for path *graph.node* and value type *gml_list*. f is called after all objects in the corresponding list are parsed.

void parser.add_edge_done_rule(*gml_edge_rule f*)

adds edge rule f to *parser* for path *graph.edge* and value type *gml_list*. f is called after all objects in the corresponding list are parsed.

4. Implementation

The data type gml_graph is realized using lists and maps. It inherits from gml_parser which uses gml_object, gml_objecttree, and gml_pattern. gml_pattern uses dictionaries.

Chapter 9

Graph Algorithms

This chapter gives a summary of the graph algorithms contained in LEDA, basic graph algorithms for reachability problems, shortest path algorithms, matching algorithms, flow algorithms, ...

All graph algorithms are generic, i.e., they accept instances of any user defined parameterized graph type *GRAPH*<*vtype*, *etype*> as arguments.

All graph algorithms are available by including the header file <LEDA/graph_alg.h>. Alternatively, one may include a more specific header file.

An important subclass of graph algorithms are network algorithms. The input to most network algorithms is a graph whose edges or nodes are labeled with numbers, e.g., shortest path algorithms get edge costs, network flow algorithms get edge capacities, and min cost flow algorithms get edge capacities and edge costs. We use *NT* to denote the number type used for the edge and node labels.

Most network algorithms come in three kinds: A templated version in which *NT* is a template parameter, and reinstantiated and precompiled versions for the number types *int* (always) and *double* (except for a small number of functions). The function name of the templated version ends in _T. Thus MAX_FLOW_T is the name of the templated version of the max flow algorithm and MAX_FLOW is the name of the instantiated version.

In order to use the templated version a file <LEDA/templates/XXX.t> must be included, e.g., in order to use the templated version of the maxflow algorithm, one must include <LEDA/templates/max_flow.t>

Special care should be taken when using network algorithms with a number type *NT* that can incur rounding error, e.g., the type *double*. The functions perform correctly if the arithmetic is exact. This is the case if all numerical values in the input are integers (albeit stored as a number of type *NT*), if none of the intermediate results exceeds the maximal integer representable by the number type (2^{52} in the case of *doubles*), and if no round-off errors occur during the computation. We give more specific information on the arithmetic demand for each function below. If the arithmetic incurs rounding error, the computation may fail in two ways: give a wrong answer or run forever.

9.1 Basic Graph Algorithms (basic_graph_alg)

bool TOPSORT(*graph G*, *node_array*<*int*>& *ord*)

TOPSORT takes as argument a directed graph $G(V,E)$. It sorts G topologically (if G is acyclic) by computing for every node $v \in V$ an integer $ord[v]$ such that $1 \leq ord[v] \leq |V|$ and $ord[v] < ord[w]$ for all edges $(v,w) \in E$. TOPSORT returns true if G is acyclic and false otherwise. The algorithm ([44]) has running time $O(|V|+|E|)$.

list<*node*> DFS(*graph G*, *node s*, *node_array*<*bool*>& *reached*)

DFS takes as argument a directed graph $G(V,E)$, a node s of G and a node_array *reached* of boolean values. It performs a depth first search starting at s visiting all reachable nodes v with $reached[v]$ = false. For every visited node v $reached[v]$ is changed to true. DFS returns the list of all reached nodes. The algorithm ([76]) has running time $O(|V|+|E|)$.

list<*edge*> DFS_NUM(*graph G*, *node_array*<*int*>& *dfsnum*, *node_array*<*int*>& *compnum*)

DFS_NUM takes as argument a directed graph $G(V,E)$. It performs a depth first search of G numbering the nodes of G in two different ways. *dfsnum* is a numbering with respect to the calling time and *compnum* a numbering with respect to the completion time of the recursive calls. DFS_NUM returns a depth first search forest of G (list of tree edges). The algorithm ([76]) has running time $O(|V|+|E|)$.

list<*node*> BFS(*graph G*, *node s*, *node_array*<*int*>& *dist*)

BFS takes as argument a directed graph $G(V,E)$,a node s of G and a node array *dist* of integers. It performs a breadth first search starting at s visiting all nodes v with $dist[v] = -1$ reachable from s. The *dist* value of every visited node is replaced by its distance to s. BFS returns the list of all visited nodes. The algorithm ([52]) has running time $O(|V|+|E|)$.

list<*node*> BFS(*graph G*, *node s*, *node_array*<*int*>& *dist*, *node_array*<*edge*>& *pred*)

performs a bread first search as described above and computes for every node v the predecessor edge $pred[v]$ in the bfs shortest path tree.

int COMPONENTS(*graph G*, *node_array*<*int*>& *compnum*)

COMPONENTS takes a graph $G(V, E)$ as argument and computes the connected components of the underlying undirected graph, i.e., for every node $v \in V$ an integer $compnum[v]$ from $[0 \ldots c-1]$ where c is the number of connected components of G and v belongs to the i-th connected component iff $compnum[v] = i$. COMPONENTS returns c. The algorithm ([52]) has running time $O(|V| + |E|)$.

int STRONG_COMPONENTS(*graph G*, *node_array*<*int*>& *compnum*)

STRONG_COMPONENTS takes a directed graph $G(V, E)$ as argument and computes for every node $v \in V$ an integer $compnum[v]$ from $[0 \ldots c-1]$ where c is the number of strongly connected components of G and v belongs to the i-th strongly connected component iff $compnum[v] = i$. STRONG_COMPONENTS returns c. The algorithm ([52]) has running time $O(|V| + |E|)$.

int BICONNECTED_COMPONENTS(*graph G*, *edge_array*<*int*>& *compnum*)

BICONNECTED_COMPONENTS computes the biconnected components of the undirected version of G. A biconnected component of an undirected graph is a maximal biconnected subgraph and a biconnected graph is a graph which cannot be disconnected by removing one of its nodes. A graph having only one node is biconnected.
Let c be the number of biconnected component and let c' be the number of biconnected components containing at least one edge, $c - c'$ is the number of isolated nodes in G, where a node v is isolated if is not connected to a node different from v (it may be incident to self-loops). The function returns c and labels each edge of G (which is not a self-loop) by an integer in $[0 \ldots c' - 1]$. Two edges receive the same label iff they belong to the same biconnected component. The edge labels are returned in *compnum*. Be aware that self-loops receive no label since self-loops are ignored when interpreting a graph as an undirected graph.
The algorithm ([18]) has running time $O(|V| + |E|)$.

graph TRANSITIVE_CLOSURE(*graph G*)

TRANSITIVE_CLOSURE takes a directed graph $G(V, E)$ as argument and computes the transitive closure of $G(V, E)$. It returns a directed graph $G'(V', E')$ with $V' = V$ and $(v, w) \in E' \Leftrightarrow$ there is a path from v to w in G. The algorithm ([36]) has running time $O(|V| \cdot |E|)$.

9.2 Shortest Path Algorithms (shortest_path)

Let G be a graph, s a node in G, and c a cost function on the edges of G. Edge costs may be positive or negative. For a node v let $\mu(v)$ be the length of a shortest path from s to v (more precisely, the infimum of the lengths of all paths from s to v). If v is not reachable from s then $\mu(v) = +\infty$ and if v is reachable from s through a cycle of negative cost then $\mu(v) = -\infty$. Let V^+, V^f, and V^- be the set of nodes v with $\mu(v) = +\infty$, $-\infty < \mu(v) < +\infty$, and $\mu(v) = -\infty$, respectively.

The solution to a single source shortest path problem (G, s, c) is a pair $(dist, pred)$ where $dist$ is a *node_array*<*NT*> and $pred$ is a *node_array*<*edge*> with the following properties. Let $P = \{pred[v]\ ;\ v \in V \text{ and } pred[v] \neq nil\}$. A P-cycle is a cycle all of whose edges belong to P and a P-path is a path all of whose edges belong to P.

- $v \in V^+$ iff $v \neq s$ and $pred[v] = nil$ and $v \in V^f \cup V^-$ iff $v = s$ or $pred[v] \neq nil$.
- $s \in V^f$ if $pred[s] = nil$ and $s \in V^-$ otherwise.
- $v \in V^f$ if v is reachable from s by a P-path and $s \in V^f$. P restricted to V^f forms a shortest path tree and $dist[v] = \mu(s, v)$ for $v \in V^f$.
- All P-cycles have negative cost and $v \in V^-$ iff v lies on a P-cycle or is reachable from a P-cycle by a P-path.

Most functions in this section are template functions. The template parameter NT can be instantiated with any number type. In order to use the template version of the function the .t-file

```
#include <LEDA/templates/shortest_path.t>
```

must be included. The functions are pre-instantiated with *int* and *double*. The function names of the pre-instantiated versions are without the suffix `_T`.

Special care should be taken when using the functions with a number type NT that can incur rounding error, e.g., the type *double*. The functions perform correctly if all arithmetic performed is without rounding error. This is the case if all numerical values in the input are integers (albeit stored as a number of type NT) and if none of the intermediate results exceeds the maximal integer representable by the number type (2^{52} in the case of *doubles*). All intermediate results are sums and differences of input values, in particular, the algorithms do not use divisions and multiplications. All intermediate values are bounded by nC where n is the number of nodes and C is the maximal absolute value of any edge cost.

bool SHORTEST_PATH_T(*graph* G, *node* s, *edge_array*<*NT*> c, *node_array*<*NT*>& *dist*, *node_array*<*edge*>& *pred*)

SHORTEST_PATH solves the single source shortest path problem in the graph $G(V, E)$ with respect to the source node s and the cost-function given by the edge_array c.
The procedure returns false if there is a negative cycle in G that is reachable from s and returns true otherwise.
It runs in linear time on acyclic graphs, in time $O(m + n \log n)$ if all edge costs are non-negative, and runs in time $O(\min(D, n)m)$ otherwise. Here D is the maximal number of edges on any shortest path.

node_array<*int*> CHECK_SP_T(*graph* G, *node* s, *edge_array*<*NT*> c, *node_array*<*NT*> *dist*, *node_array*<*edge*> *pred*)

checks whether the pair $(dist, pred)$ is a correct solution to the shortest path problem (G, s, c) and returns a *node_array*<*int*> *label* with $label[v] < 0$ if v has distance $-\infty$ (-2 for nodes lying on a negative cycle and -1 for a node reachable from a negative cycle), $label[v] = 0$ if v has finite distance, and $label[v] > 0$ if v has distance $+\infty$. The program aborts if the check fails. The algorithm takes linear time.

void ACYCLIC_SHORTEST_PATH_T(*graph* G, *node* s, *edge_array*<*NT*> c, *node_array*<*NT*>& *dist*, *node_array*<*edge*>& *pred*)

solves the single source shortest path problem with respect to source s. The algorithm takes linear time.
Precondition: G must be acyclic.

void DIJKSTRA_T(*graph* G, *node* s, *edge_array*<*NT*> *cost*, *node_array*<*NT*>& *dist*, *node_array*<*edge*>& *pred*)

solves the shortest path problem in a graph with non-negative edges weights.
Precondition: The costs of all edges are non-negative.

void DIJKSTRA_T(*graph* G, *node* s, *edge_array*<*NT*> *cost*, *node_array*<*NT*>& *dist*)

as above, but *pred* is not computed.

NT DIJKSTRA_T(*graph* G, *node* s, *node* t, *edge_array*<*NT*> c, *node_array*<*edge*>& *pred*)

computes a shortest path from s to t and returns its lenght. The cost of all edges must be non-negative. The return value is unspecified if there is no path from s to t. The array *pred* records a shortest path from s to t in reverse order, i.e., $pred[t]$ is the last edge on the path. If there is no path from s to t or if $s = t$ then $pred[t] = nil$. The worst case running time is $O(m + n \log n)$, but frequently much better.

bool BELLMAN_FORD_B_T(*graph* G, *node* s, *edge_array*<*NT*> c,
node_array<*NT*>& *dist*, *node_array*<*edge*>& *pred*)

BELLMAN_FORD_B solves the single source shortest path problem in the graph $G(V, E)$ with respect to the source node s and the cost-function given by the edge_array c.
BELLMAN_FORD_B returns false if there is a negative cycle in G that is reachable from s and returns true otherwise. The algorithm ([9]) has running time $O(\min(D, n)m)$ where D is the maximal number of edges on any shortest path. The algorithm is only included for pedagogical purposes.

void BF_GEN(*GRAPH*<*int*, *int*>& G, *int* n, *int* m, *bool* *non_negative* = *true*)

generates a graph with at most n nodes and at most m edges. The edge costs are stored as edge data. The running time of BELLMAN_FORD_B on this graph is $\Omega(nm)$. The edge weights are non-negative if *non_negative* is true and are arbitrary otherwise.
Precondition: $m \geq 2n$ and $m \leq n^2/2$.

bool BELLMAN_FORD_T(*graph* G, *node* s, *edge_array*<*NT*> c, *node_array*<*NT*>& *dist*,
node_array<*edge*>& *pred*)

BELLMAN_FORD_T solves the single source shortest path problem in the graph $G(V, E)$ with respect to the source node s and the cost-function given by the edge_array c.
BELLMAN_FORD_T returns false if there is a negative cycle in G that is reachable from s and returns true otherwise. The algorithm ([9]) has running time $O(\min(D, n)m)$ where D is the maximal number of edges in any shortest path.
The algorithm is never significantly slower than BELLMAN_FORD_B and frequently much faster.

bool ALL_PAIRS_SHORTEST_PATHS_T(*graph*& G, *edge_array*<*NT*> c,
node_matrix<*NT*>& *DIST*)

returns *true* if G has no negative cycle and returns *false* otherwise. In the latter case all values returned in $DIST$ are unspecified. In the former case the following holds for all v and w: if $\mu(v, w) < \infty$ then $DIST(v, w) = \mu(v, w)$ and if $\mu(v, w) = \infty$ then the value of $DIST(v, w)$ is arbitrary. The procedure runs in time $O(nm + n^2 \log n)$.

rational MINIMUM_RATIO_CYCLE(*graph*& G, *edge_array*<*int*> c, *edge_array*<*int*> p, *list*<*edge*>& *C_star*)

Returns a minimum cost to profit ratio cycle *C_star* and the ratio of the cycle. For a cycle C let $c(C)$ be the sum of the c-values of the edges on the cycle and let $p(C)$ be the sum of the p-values of the edges on the cycle. The cost to profit ratio of the cycle is the quotient $c(C)/p(C)$. The cycle *C_start* realizes the minimum ratio for any cycle C. The procedure runs in time $O(nm \log(n \cdot C \cdot P))$ where C and P are the maximum cost and profit of any edge, respectively. The program returns zero if there is no cycle in G.

Precondition: There are no cycles of cost zero or less with respect to either c or p.

9.3 Maximum Flow (max_flow)

Let $G = (V, E)$ be a directed graph, let s and t be distinct vertices in G and let $cap : E \longrightarrow I\!R_{\geq 0}$ be a non-negative function on the edges of G. For an edge e, we call $cap(e)$ the *capacity* of e. An (s,t)-*flow* or simply *flow* is a function $f : E \longrightarrow I\!R_{\geq 0}$ satisfying the capacity constraints and the flow conservation constraints:

$$\begin{array}{lcl} (1) & 0 \leq f(e) \leq cap(e) & \text{for every edge } e \in E \\ (2) & \sum_{e; source(e)=v} f(e) = \sum_{e; target(e)=v} f(e) & \text{for every node } v \in V \backslash \{s,t\} \end{array}$$

The value of the flow in the net flow into t (equivalently, the net flow out of s). The net flow into t is the flow into t minus the flow out of t. A flow is maximum if its value is at least as large as the value of any other flow.

All max flow implementations are template functions. The template parameter NT can be instantiated with any number type. In order to use the template version of the function the files

```
#include <LEDA/graph_algs.h>
#include <LEDA/templates/max_flow.t>
```

must be included.

There are pre-instantiations for the number types *int* and *double*.The pre-instantiated versions have the same function names except for the suffix `_T`. In order to use them either

```
#include <LEDA/max_flow.h>
```

or

```
#include <LEDA/graph_alg.h>
```

has to be included (the latter file includes the former). The connection between template functions and pre-instantiated functions is discussed in detail in the section "Templates for Network Algorithms" of the LEDA book.

Special care should be taken when using the template functions with a number type NT that can incur rounding error, e.g., the type *double*. The section "Algorithms on Weighted Graphs and Arithmetic Demand" of the LEDA book contains a general discussion of this issue. The template functions are only guaranteed to perform correctly if all arithmetic performed is without rounding error. This is the case if all numerical values in the input are integers (albeit stored as a number of type NT) and if none of the intermediate results exceeds the maximal integer representable by the number type ($2^{53} - 1$ in the case of *doubles*). All intermediate results are sums and differences of input values, in particular, the algorithms do not use divisions and multiplications.

The algorithms have the following arithmetic demands. Let C be the maximal absolute value of any edge capacity. If all capacities are integral then all intermediate values are bounded by $d \cdot C$, where d is the out-degree of the source.

The pre-instantiations for number type *double* compute the optimal matching for a modified capacity function *cap1*, where for every edge e

$$cap1[e] = sign(cap[e]) \lfloor |cap[e]| \cdot S \rfloor / S$$

and S is the largest power of two such that $S < 2^{53}/(d \cdot C)$.

The value of the maximum flow for the modified capacity function and the value of the maximum flow for the original capacity function differ by at most $m \cdot d \cdot C \cdot 2^{-52}$.

The following functions are available:

NT MAX_FLOW_T(*graph G*, *node s*, *node t*, *edge_array*<*NT*> *cap*, *edge_array*<*NT*>& *f*)

> computes a maximum (s,t)-flow f in the network (G, s, t, cap) and returns the value of the flow.
> The implementation uses the preflow-push method of Goldberg and Tarjan [**?**] with the local and global relabeling heuristic and the gap heuristic. The highest level rule is used to select active nodes. The section on maximum flow of the LEDA book gives full information.

NT MAX_FLOW_T(*graph G*, *node s*, *node t*, *edge_array*<*NT*> *cap*, *edge_array*<*NT*>& *f*,
int& *num_pushes*, *int*& *num_edge_inspections*, *int*& *num_relabels*,
int& *num_global_relabels*, *int*& *num_gaps*, *float h*)

> as above;
> The additional parameter report some statistics of the execution (the number of pushes, the number of edge inspections, the number of relabels, the number of global relabels, and the number of nodes moved by the gap heuristic.
> The parameter h controls the global relabeling heuristic. The global relabeling heuristic is called every $h \cdot m$ edge inspections. The choice $h = 5$ seems reasonable.

bool CHECK_MAX_FLOW_T(*graph G*, *node s*, *node t*, *edge_array*<*NT*> *cap*,
edge_array<*NT*> *f*)

> checks whether f is a maximum flow in the network (G, s, t, cap). The functions returns false if this is not the case.

bool MAX_FLOW_SCALE_CAPS(*graph G*, *node s*, *edge_array*<*double*>& *cap*)

> replaces $cap[e]$ by $cap1[e]$ for every edge e, where $cap1[e]$ is as defined above. The function returns *false* if the scaling changed some capacity, and returns *true* otherwise.

The following functions are only available in template form. They allow to study the effect of different heuristics and of different selection rules on the preflow push method. The class SET must provide the following operations: construction, destruction, *del*, *insert*,

insert0, and *clear*; see the LEDA book for details. Three implementations are part of the distribution: *fifo_set*, *hl_set*, and *mfifo_set*.

NT MAX_FLOW_BASIC_T(*graph G*, *node s*, *node t*, *edge_array*<*NT*> *cap*, *edge_array*<*NT*>& *flow*, *SET*& *U*, *int*& *num_pushes*, *int*& *num_edge_inspections*, *int*& *num_relabels*)

The basic version of the preflow push algorithm: No heuristic is used.

NT MAX_FLOW_LH_T(*graph G*, *node s*, *node t*, *edge_array*<*NT*> *cap*, *edge_array*<*NT*>& *flow*, *SET*& *U*, *int*& *num_pushes*, *int*& *num_edge_inspections*, *int*& *num_relabels*)

The preflow push method with the distinction between low and high nodes.

NT MAX_FLOW_LRH_T(*graph G*, *node s*, *node t*, *edge_array*<*NT*> *cap*, *edge_array*<*NT*>& *flow*, *SET*& *U*, *int*& *num_pushes*, *int*& *num_edge_inspections*, *int*& *num_relabels*)

The preflow push method with the distinction between low and high nodes and the local relabeling heuristic.

NT MAX_FLOW_GRH_T(*graph G*, *node s*, *node t*, *edge_array*<*NT*> *cap*, *edge_array*<*NT*>& *flow*, *SET*& *U*, *int*& *num_pushes*, *int*& *num_edge_inspections*, *int*& *num_relabels*, *int*& *num_global_relabels*, *float h*)

The preflow push method with the distinction between low and high nodes, the local relabeling heuristic, the global relabeling heuristic, and the two-phase approach. The global relabeling heuristic is applied every $h \cdot m$ edge inspections.

NT MAX_FLOW_GAP_T(*graph G*, *node s*, *node t*, *edge_array*<*NT*> *cap*, *edge_array*<*NT*>& *flow*, *SET*& *U*, *int*& *num_pushes*, *int*& *num_edge_inspections*, *int*& *num_relabels*, *int*& *num_global_relabels*, *int*& *num_gaps*, *float h*)

The preflow push method with the distinction between low and high nodes, the local relabeling heuristic, the global relabeling heuristic, the two-phase approach, and the gap heuristic. The global relabeling heuristic is applied every $h \cdot m$ edge inspections.

The following generators can be used to generate max-flow problems with integer capacities. The generators return a *GRAPH*<*int*, *int*> *G* and nodes s and t. The edge capacities are available as edge data, i.e., the capacity of edge e is available as $G[e]$ and the edge array of capacities is passed as *G.edge_data*( ). For detailed information about the generators we refer to the LEDA book.

void max_flow_gen_rand(*GRAPH*<*int*, *int*>& *G*, *node*& *s*, *node*& *t*, *int n*, *int m*)

A random graph with n nodes, m edges, and random edge capacities in [2,11] for the edges out of s and in [1,10] for all other edges.

void max_flow_gen_CG1(*GRAPH*<*int*, *int*>& *G*, *node*& *s*, *node*& *t*, *int n*)

A generator suggested by Cherkassky and Goldberg.

void max_flow_gen_CG2(*GRAPH*<*int*, *int*>& *G*, *node*& *s*, *node*& *t*, *int n*)

Another generator suggested by Cherkassky and Goldberg.

void max_flow_gen_AMO(*GRAPH*<*int*, *int*>& *G*, *node*& *s*, *node*& *t*, *int n*)

A generator suggested by Ahuja, Magnanti, and Orlin.

9.4 Min Cost Flow Algorithms (min_cost_flow)

bool MIN_COST_FLOW(*graph*& *G*, *edge_array*<*int*> *lcap*, *edge_array*<*int*> *ucap*, *edge_array*<*int*> *cost*, *node_array*<*int*> *supply*, *edge_array*<*int*>& *flow*)

MIN_COST_FLOW takes as arguments a directed graph $G(V, E)$, an edge_array *lcap* (*ucap*) giving for each edge a lower (upper) capacity bound, an edge_array *cost* specifying for each edge an integer cost and a node_array *supply* defining for each node v a supply or demand (if $supply[v] < 0$). If a feasible flow (fulfilling the capacity and mass balance conditions) exists it computes such a $flow$ of minimal cost and returns *true*, otherwise *false* is returned. The algorithm is based on capacity scaling and successive shortest path computation (cf. [25] and [4]) and has running time $O(|E| \log U(|E| + |V| \log |V|))$.

bool MIN_COST_FLOW(*graph*& *G*, *edge_array*<*int*> *cap*, *edge_array*<*int*> *cost*, *node_array*<*int*> *supply*, *edge_array*<*int*>& *flow*)

This variant of MIN_COST_FLOW assumes that $lcap[e] = 0$ for every edge $e \in E$.

int MIN_COST_MAX_FLOW(*graph*& *G*, *node s*, *node t*, *edge_array*<*int*> *cap*, *edge_array*<*int*> *cost*, *edge_array*<*int*>& *flow*)

MIN_COST_MAX_FLOW takes as arguments a directed graph $G(V, E)$, a source node s, a sink node t, an edge_array *cap* giving for each edge in G a capacity, and an edge_array *cost* specifying for each edge an integer cost. It computes for every edge e in G a flow $flow[e]$ such that the total flow from s to t is maximal, the total cost of the flow is minimal, and $0 \leq flow[e] \leq cap[e]$ for all edges e. MIN_COST_MAX_FLOW returns the total flow from s to t.

9.5 Minimum Cut (min_cut)

A cut C in a network is a set S of nodes that is neither empty nor the entire set of nodes. The weight of a cut is the sum of the weights of the edges having exactly one endpoint in S.

int MIN_CUT(*graph G*, *edge_array*<*int*> *weight*, *list*<*node*>& *C*, *bool use_heuristic = true*)

MIN_CUT takes a graph G and an edge_array *weight* that gives for each edge a *non-negative* integer weight. The algorithm ([73]) computes a cut of minimum weight. A cut of minimum weight is returned in C and the value of the cut is the return value of the function. The running time is $O(nm + n^2 \log n)$. The function uses a heuristic to speed up its computation. *Precondition*: The edge weights are non-negative.

list<*node*> MIN_CUT(*graph G*, *edge_array*<*int*> *weight*)

as above, but the cut C is returned.

int CUT_VALUE(*graph G*, *edge_array*<*int*> *weight*, *list*<*node*> *C*)

returns the value of the cut C.

9.6 Maximum Cardinality Matchings in Bipartite Graphs (mcb_matching)

A *matching* in a graph G is a subset M of the edges of G such that no two share an endpoint. A node cover is a set of nodes NC such that every edge has at least one endpoint in NC. The maximum cardinality of a matching is at most the minimum cardinality of a node cover. In bipartite graphs, the two quantities are equal.

list<*edge*> MAX_CARD_BIPARTITE_MATCHING(*graph*& G)

returns a maximum cardinality matching.
Precondition: G must be bipartite.

list<*edge*> MAX_CARD_BIPARTITE_MATCHING(*graph*& G, *node_array*<*bool*>& NC)

returns a maximum cardinality matching and a minimum cardinality node cover NC. The node cover has the same cardinality as the matching and hence proves the optimality of the matching. *Precondition*: G must be bipartite.

bool CHECK_MCB(*graph* G, *list*<*edge*> M, *node_array*<*bool*> NC)

checks that M is a matching in G, i.e., that at most one edge in M is incident to any node of G, that NC is a node cover, i.e., for every edge of G at least one endpoint is in NC and that M and NC have the same cardinality. The function writes diagnostic output to cerr, if one of the conditions is violated.

list<*edge*> MAX_CARD_BIPARTITE_MATCHING(*graph*& G, *list*<*node*> A, *list*<*node*> B)

returns a maximum cardinality matching. The bipartition of G is given by A and B. All edges of G must be directed from A to B. *Precondition*: G must be bipartite.

list<*edge*> MAX_CARD_BIPARTITE_MATCHING(*graph*& G, *list*<*node*> A, *list*<*node*> B, *node_array*<*bool*>& NC)

returns a maximum cardinality matching. The bipartition of G is given by A and B. All edges of G must be directed from A to B. A minimal node cover is returned in NC. The node cover has the same cardinality as the matching and hence proves the maximality of the matching. *Precondition*: G must be bipartite.

We offer several implementations of bipartite matching algorithms. All of them require that the bipartition (A, B) is given and that all edges are directed from A to B and all of them return a maximum cardinality matching and a minimum cardinality node cover. The initial characters of the inventors are used to distinguish between the algorithms. The common interface is

```
list<edge> MAX_CARD_BIPARTITE_MATCHING_XX(graph& G,
                                          const list<node>& A,
                                          const list<node>& B,
                                          node_array<bool>& NC,
                                          bool use_heuristic = true);
```

where XX is to be replaced by either HK, ABMP, FF, or FFB. All algorithms can be asked to use a heuristic to find an initial matching. This is the default.

HK stands for the algorithm due to Hopcroft and Karp [40]. It has running time $O(\sqrt{n}m)$.

ABMP stands for algorithm due to Alt, Blum, Mehlhorn, and Paul [1]. The algorithm has running time $O(\sqrt{n}m)$. The algorithm consists of two major phases. In the first phase all augmenting paths of length less than $Lmax$ are found, and in the second phase the remaining augmenting paths are determined. The default value of $Lmax$ is $0.1\sqrt{n}$. $Lmax$ is an additional optional parameter of the procedure.

FF stands for the algorithm due to Ford and Fulkerson [30]. The algorithm has running time $O(nm)$ and FFB stands for a simple and slow version of FF. The algorithm FF has an additional optional parameter *use_bfs* of type *bool*. If set to true, breadth-first-search is used in the search for augmenting paths, and if set to false, depth-first-search is used.

Be aware that the algorithms _XX change the graph G. They leave the graph structure unchanged but reorder adjacency lists (and hence change the embedding). If this is undesirable you must restore the original order of the adjacency lists as follows.

```
edge_array<int> edge_number(G); int i = 0;
forall_nodes(v,G)
  forall_adj_edges(e,G) edge_number[e] = i++;
call matching algorithm;
G.sort_edges(edge_number);
```

9.7 Bipartite Weighted Matchings and Assignments (mwb_matching)

We give functions

- to compute maximum and minimum weighted matchings in bipartite graphs,

- to check the optimality of matchings, and
- to scale edge weights, so as to avoid round-off errors in computations with the number type *double*.

All functions for computing maximum or minimum weighted matchings provide a proof of optimality in the form of a potential function *pot*; see the chapter on bipartite weighted matchings of the LEDA book for a discussion of potential functions.

The functions in this section are template functions. The template parameter NT can be instantiated with any number type. In order to use the template version of the function the appropriate .t-file must be included.

```
#include <LEDA/templates/mwb_matching.t>
```

There are pre-instantiations for the number types *int* and *double*.The pre-instantiated versions have the same function names except for the suffix `_T`. In order to use them either

```
#include <LEDA/mwb_matching.h>
```

or

```
#include <LEDA/graph_alg.h>
```

has to be included (the latter file includes the former). The connection between template functions and pre-instantiated functions is discussed in detail in the section "Templates for Network Algorithms" of the LEDA book.

Special care should be taken when using the template functions with a number type NT that can incur rounding error, e.g., the type *double*. The section "Algorithms on Weighted Graphs and Arithmetic Demand" of the LEDA book contains a general discussion of this issue. The template functions are only guaranteed to perform correctly if all arithmetic performed is without rounding error. This is the case if all numerical values in the input are integers (albeit stored as a number of type NT) and if none of the intermediate results exceeds the maximal integer representable by the number type ($2^{53} - 1$ in the case of *doubles*). All intermediate results are sums and differences of input values, in particular, the algorithms do not use divisions and multiplications.

The algorithms have the following arithmetic demands. Let C be the maximal absolute value of any edge cost. If all weights are integral then all intermediate values are bounded by $3C$ in the case of maximum weight matchings and by $4nC$ in the case of the other matching algorithms. Let $f = 3$ in the former case and let $f = 4n$ in the latter case.

The pre-instantiations for number type *double* compute the optimal matching for a modified weight function *c1*, where for every edge e

$$c1[e] = sign(c[e]) \lfloor |c[e]| \cdot S \rfloor / S$$

and S is the largest power of two such that $S < 2^{53}/(f \cdot C)$.

The weight of the optimal matching for the modified weight function and the weight of the optimal matching for the original weight function differ by at most $n \cdot f \cdot C \cdot 2^{-52}$.

list<*edge*> MAX_WEIGHT_BIPARTITE_MATCHING_T(*graph*& G, *edge_array*<*NT*> c, *node_array*<*NT*>& *pot*)

computes a matching of maximal cost and a potential function *pot* that is tight with respect to M. The running time of the algorithm is $O(n \cdot (m + n\ log\ n))$. The argument *pot* is optional.
Precondition: G must be bipartite.

list<*edge*> MAX_WEIGHT_BIPARTITE_MATCHING_T(*graph*& G, *list*<*node*> A, *list*<*node*> B, *edge_array*<*NT*> c, *node_array*<*NT*>& *pot*)

As above. It is assumed that the partition (A, B) witnesses that G is bipartite and that all edges of G are directed from A to B. If A and B have different sizes then is is advisable that A is the smaller set; in general, this leads to smaller running time. The argument *pot* is optional.

bool CHECK_MWBM_T(*graph* G, *edge_array*<*NT*> c, *list*<*edge*> M, *node_array*<*NT*> *pot*)

checks that *pot* is a tight feasible potential function with respect to M and that M is a matching. Tightness of *pot* implies that M is a maximum weighted matching.

list<*edge*> MAX_WEIGHT_ASSIGNMENT_T(*graph*& G, *edge_array*<*NT*> c, *node_array*<*NT*>& *pot*)

computes a perfect matching of maximal cost and a potential function *pot* that is tight with respect to M. The running time of the algorithm is $O(n \cdot (m + n\ log\ n))$. If G contains no perfect matching the empty set of edges is returned. The argument *pot* is optional.
Precondition: G must be bipartite.

list<*edge*> MAX_WEIGHT_ASSIGNMENT_T(*graph*& G, *list*<*node*> A, *list*<*node*> B, *edge_array*<*NT*> c, *node_array*<*NT*>& *pot*)

As above. It is assumed that the partition (A, B) witnesses that G is bipartite and that all edges of G are directed from A to B. The argument *pot* is optional.

bool CHECK_MAX_WEIGHT_ASSIGNMENT_T(*graph G, edge_array*<*NT*> *c,*
list<*edge*> *M,*
node_array<*NT*> *pot*)

checks that *pot* is a tight feasible potential function with respect to M and that M is a perfect matching. Tightness of *pot* implies that M is a maximum cost assignment.

list<*edge*> MIN_WEIGHT_ASSIGNMENT_T(*graph& G, edge_array*<*NT*> *c,*
node_array<*NT*>& *pot*)

computes a perfect matching of minimal cost and a potential function *pot* that is tight with respect to M. The running time of the algorithm is $O(n \cdot (m + n\ log\ n))$. If G contains no perfect matching the empty set of edges is returned. The argument *pot* is optional.
Precondition: G must be bipartite.

list<*edge*> MIN_WEIGHT_ASSIGNMENT_T(*graph& G, list*<*node*> *A, list*<*node*> *B,*
edge_array<*NT*> *c, node_array*<*NT*>& *pot*)

As above. It is assumed that the partition (A, B) witnesses that G is bipartite and that all edges of G are directed from A to B. The argument *pot* is optional.

bool CHECK_MIN_WEIGHT_ASSIGNMENT_T(*graph G, edge_array*<*NT*> *c,*
list<*edge*> *M,*
node_array<*NT*> *pot*)

checks that *pot* is a tight feasible potential function with respect to M and that M is a perfect matching. Tightness of *pot* implies that M is a minimum cost assignment.

list<*edge*> MWMCB_MATCHING_T(*graph& G, list*<*node*> *A, list*<*node*> *B,*
edge_array<*NT*> *c, node_array*<*NT*>& *pot*)

Returns a maximum weight matching among the matching of maximum cardinality. The potential function *pot* is tight with respect to a modified cost function which increases the cost of every edge by $L = 1+2kC$ where C is the maximum absolute value of any weight and $k = \min(|A|, |B|)$. It is assumed that the partition (A, B) witnesses that G is bipartite and that all edges of G are directed from A to B. If A and B have different sizes, it is advisable that A is the smaller set; in general, this leads to smaller running time. The argument *pot* is optional.

bool MWBM_SCALE_WEIGHTS(*graph G, edge_array*<*double*>& *c*)

replaces $c[e]$ by $c1[e]$ for every edge e, where $c1[e]$ was defined above and $f = 3$. This scaling function is appropriate for the maximum weight matching algorithm. The function returns *false* if the scaling changed some weight, and returns *true* otherwise.

bool MWA_SCALE_WEIGHTS(*graph G*, *edge_array*<*double*>& *c*)

replaces $c[e]$ by $c1[e]$ for every edge e, where $c1[e]$ was defined above and $f = 4n$. This scaling function should be used for the algorithm that compute minimum of maximum weight assignments or maximum weighted matchings of maximum cardinality. The function returns *false* if the scaling changed some weight, and returns *true* otherwise.

9.8 Maximum Cardinality Matchings in General Graphs (mc_matching)

A *matching* in a graph G is a subset M of the edges of G such that no two share an endpoint.

An odd-set cover OSC of G is a labeling of the nodes of G with non-negative integers such that every edge of G (which is not a self-loop) is either incident to a node labeled 1 or connects two nodes labeled with the same i, $i \geq 2$.

Let n_i be the number of nodes labeled i and consider any matching N. For i, $i \geq 2$, let N_i be the edges in N that connect two nodes labeled i. Let N_1 be the remaining edges in N. Then $|N_i| \leq \lfloor n_i/2 \rfloor$ and $|N_1| \leq n_1$ and hence

$$|N| \leq n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor$$

for any matching N and any odd-set cover OSC.

It can be shown that for a maximum cardinality matching M there is always an odd-set cover OSC with

$$|M| = n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor,$$

thus proving the optimality of M. In such a cover all n_i with $i \geq 2$ are odd, hence the name.

list<*edge*> MAX_CARD_MATCHING(*graph G*, *node_array*<*int*>& *OSC*, *int heur* = 0)

computes a maximum cardinality matching M in G and returns it as a list of edges. The algorithm ([23], [34]) has running time $O(nm \cdot \alpha(n, m))$. With *heur* = 1 the algorithm uses a greedy heuristic to find an initial matching. This seems to have little effect on the running time of the algorithm.
An odd-set cover that proves the maximality of M is returned in OSC.

list<*edge*> MAX_CARD_MATCHING(*graph G*, *int heur* = 0)

as above, but no proof of optimality is returned.

bool CHECK_MAX_CARD_MATCHING(*graph* G, *list*<*edge*> M, *node_array*<*int*> OSC)

checks whether M is a maximum cardinality matching in G and OSC is a proof of optimality. Aborts if this is not the case.

9.9 Maximum Weight Matchings in General Graphs (max_weight_matching)

list<*edge*> MAX_WEIGHT_MATCHING(*graph* G, *edge_array*<*int*> *weights*, *bool checked* = *false*)

MAX_WEIGHT_MATCHING takes as arguments an undirected graph G and an edge_array giving for each edge an integer (real) weight. It computes a maximum weighted matching of G, i.e., a set of edges M such that the sum of weights of all edges in M is maximal and no two edges in M share an end point. MAX_WEIGHT_MATCHING returns M as a list of edges. The algorithm ([23], [33], [49]) has running time $O(|V|^3)$. If *checked* = *true* then internally we ensure the correctness of the output at the cost of some runtime checking.

9.10 Minimum Spanning Trees (min_span)

list<*edge*> SPANNING_TREE(*graph* G)

SPANNING_TREE takes as argument a graph $G(V, E)$. It computes a spanning tree T of the underlying undirected graph, SPANNING_TREE returns the list of edges of T. The algorithm ([52]) has running time $O(|V| + |E|)$.

list<*edge*> MIN_SPANNING_TREE(*graph* G, *edge_array*<*int*> *cost*)

MIN_SPANNING_TREE takes as argument an undirected graph $G(V, E)$ and an edge_array *cost* giving for each edge an integer cost. It computes a minimum spanning tree T of G, i.e., a spanning tree such that the sum of all edge costs is minimal. MIN_SPANNING_TREE returns the list of edges of T. The algorithm ([46]) has running time $O(|E| \log |V|)$.

list<*edge*> MIN_SPANNING_TREE(*graph* G, *int* $(*cmp)$(*edge*, *edge*))

Variant using a function *cmp* to compare edge costs.

9.11 Algorithms for Planar Graphs (plane_graph_alg)

node ST_NUMBERING(*graph G*, *node_array*<*int*>& *stnum*, *list*<*node*>& *stlist*, *edge e_st = nil*)

ST_NUMBERING computes an *st*-numbering of G. If *e_st* is nil then t is set to some arbitrary node of G. The node s is set to a neighbor of t and is returned. If *e_st* is not nil then s is set to the source of *e_st* and t is set to its target. The nodes of G are numbered such that s has number 1, t has number n, and every node v different from s and t has a smaller and a larger numbered neighbor. The ordered list of nodes is returned in *stlist*. If G has no nodes then *nil* is returned and if G has exactly one node then this node is returned and given number one.
Precondition: G is biconnected.

bool PLANAR(*graph*& , *bool embed = false*)

PLANAR takes as input a directed graph $G(V, E)$ and performs a planarity test for it. G must not contain selfloops. If the second argument *embed* has value *true* and G is a planar graph it is transformed into a planar map (a combinatorial embedding such that the edges in all adjacency lists are in clockwise ordering). PLANAR returns true if G is planar and false otherwise. The algorithm ([41]) has running time $O(|V| + |E|)$.

bool PLANAR(*graph*& *G*, *list*<*edge*>& *el*, *bool embed = false*)

PLANAR takes as input a directed graph $G(V, E)$ and performs a planarity test for G. PLANAR returns true if G is planar and false otherwise. If G is not planar a Kuratowsky-Subgraph is computed and returned in *el*.

bool CHECK_KURATOWSKI(*graph G*, *list*<*edge*> *el*)

returns *true* if all edges in *el* are edges of G and if the edges in *el* form a Kuratowski subgraph of G, returns *false* otherwise. Writes diagnostic output to *cerr*.

int KURATOWSKI(*graph*& G, *list*<*node*>& V, *list*<*edge*>& E, *node_array*<*int*>& *deg*)

KURATOWKI computes a Kuratowski subdivision K of G as follows. V is the list of all nodes and subdivision points of K. For all $v \in V$ the degree $deg[v]$ is equal to 2 for subdivision points, 4 for all other nodes if K is a K_5, and -3 (+3) for the nodes of the left (right) side if K is a $K_{3,3}$. E is the list of all edges in the Kuratowski subdivision.

list<*edge*> TRIANGULATE_PLANAR_MAP(*graph*& G)

TRIANGULATE_PLANAR_MAP takes a directed graph G representing a planar map. It triangulates the faces of G by inserting additional edges. The list of inserted edges is returned.
Precondition: G must be connected.
The algorithm ([43]) has running time $O(|V| + |E|)$.

void FIVE_COLOR(*graph*& G, *node_array*<*int*>& C)

colors the nodes of G using 5 colors, more precisely, computes for every node v a color $C[v] \in \{0, \ldots, 4\}$, such that $C[source(e)]! = C[target(e)]$ for every edge e. *Precondition*: G is planar. **Remark**: works also for many (sparse ?) non-planar graphs.

void INDEPENDENT_SET(*graph* G, *list*<*node*>& I)

determines an independent set of nodes I in G. Every node in I has degree at most 9. If G is planar and has no parallel edges then I contains at least $n/6$ nodes.

bool Is_CCW_Ordered(*graph* G, *node_array*<*int*> x, *node_array*<*int*> y)

checks whether the cyclic adjacency list of any node v agrees with the counter-clockwise ordering of the neighbors of v around v defined by their geometric positions.

bool SORT_EDGES(*graph*& G, *node_array*<*int*> x, *node_array*<*int*> y)

reorders all adjacency lists such the cyclic adjacency list of any node v agrees with the counter-clockwise order of v's neighbors around v defined by their geometric positions. The function returns true if G is a plane map after the call.

bool Is_CCW_Ordered(*graph G*, *edge_array*<*int*> *dx*, *edge_array*<*int*> *dy*)

checks whether the cyclic adjacency list of any node v agrees with the counter-clockwise ordering of the neighbors of v around v. The direction of edge e is given by the vector $(dx(e), dy(e))$.

bool SORT_EDGES(*graph& G*, *edge_array*<*int*> *dx*, *edge_array*<*int*> *dy*)

reorders all adjacency lists such the cyclic adjacency list of any node v agrees with the counter-clockwise order of v's neighbors around v. The direction of edge e is given by the vector $(dx(e), dy(e))$. The function returns true if G is a plane map after the call.

9.12 Graph Drawing Algorithms (graph_draw)

This section gives a summary of the graph drawing algorithms contained in LEDA. Before using them the header file <LEDA/graph_draw.h> has to be included.

int STRAIGHT_LINE_EMBED_MAP(*graph*& G, *node_array*<*int*>& *xcoord*, *node_array*<*int*>& *ycoord*)

STRAIGHT_LINE_EMBED_MAP takes as argument a graph G representing a planar map. It computes a straight line embedding of G by assigning non-negative integer coordinates (*xcoord* and *ycoord*) in the range $0..2(n-1)$ to the nodes. STRAIGHT_LINE_EMBED_MAP returns the maximal coordinate. The algorithm ([28]) has running time $O(|V|^2)$.

int STRAIGHT_LINE_EMBEDDING(*graph*& G, *node_array*<*int*>& *xc*, *node_array*<*int*>& *yc*)

STRAIGHT_LINE_EMBEDDING takes as argument a planar graph G and computes a straight line embedding of G by assigning non-negative integer coordinates (*xcoord* and *ycoord*) in the range $0..2(n-1)$ to the nodes. The algorithm returns the maximal coordinate and has running time $O(|V|^2)$.

bool VISIBILITY_REPRESENTATION(*graph*& G, *node_array*<*double*>& *x_pos*, *node_array*<*double*>& *y_pos*, *node_array*<*double*>& *x_rad*, *node_array*<*double*>& *y_rad*, *edge_array*<*double*>& *x_sanch*, *edge_array*<*double*>& *y_sanch*, *edge_array*<*double*>& *x_tanch*, *edge_array*<*double*>& *y_tanch*)

computes a visibility representation of the graph G, i.e., each node is represented by a horizontal segment (or box) and each edge is represented by a vertical segment.

bool TUTTE_EMBEDDING(*graph G*, *list*<*node*> *fixed_nodes*, *node_array*<*double*>& *xpos*, *node_array*<*double*>& *ypos*)

computes a convex drawing of the graph G if possible. The list *fixed_nodes* contains nodes with prescribed coordinates already given in *xpos* and *ypos*. The computed node positions of the other nodes are stored in *xpos* and *ypos*, too. If the operation is successful, true is returned.

void SPRING_EMBEDDING(*graph G*, *node_array*<*double*>& *xpos*, *node_array*<*double*>& *ypos*, *double xleft*, *double xright*, *double ybottom*, *double ytop*, *int iterations* = 250)

...

void SPRING_EMBEDDING(*graph G*, *list*<*node*> *fixed*, *node_array*<*double*>& *xpos*, *node_array*<*double*>& *ypos*, *double xleft*, *double xright*, *double ybottom*, *double ytop*, *int iterations* = 250)

...

int ORTHO_EMBEDDING(*graph G*, *edge_array*<*int*> *maxbends*, *node_array*<*int*>& *xcoord*, *node_array*<*int*>& *ycoord*, *edge_array*<*list*<*int*> >& *xbends*, *edge_array*<*list*<*int*> >& *ybends*)

Produces an orthogonal (Tamassia) embedding such that each edge e has at most *maxbends*[e] bends. Returns *true* if such an embedding exists and false otherwise.

int ORTHO_EMBEDDING(*graph G*, *node_array*<*int*>& *xpos*, *node_array*<*int*>& *ypos*, *edge_array*<*list*<*int*> >& *xbends*, *edge_array*<*list*<*int*> >& *ybends*)

orthogonal embedding with bend minimization (Tamassia).

bool ORTHO_DRAW(*graph G0*, *node_array*<*double*>& *xpos*, *node_array*<*double*>& *ypos*, *node_array*<*double*>& *xrad*, *node_array*<*double*>& *yrad*, *edge_array*<*list*<*double*> >& *xbends*, *edge_array*<*list*<*double*> >& *ybends*, *edge_array*<*double*>& *xsanch*, *edge_array*<*double*>& *ysanch*, *edge_array*<*double*>& *xtanch*, *edge_array*<*double*>& *ytanch*)

...

bool SP_EMBEDDING(*graph& G, node_array<double>& x_coord,*
node_array<double>& y_coord,
node_array<double>& x_radius,
node_array<double>& y_radius, edge_array<list<double> >& x_bends, edge_array<list<double> >& y_bends,
edge_array<double>& x_sanch,
edge_array<double>& y_sanch,
edge_array<double>& x_tanch,
edge_array<double>& y_tanch)
...

Chapter 10

Graphs and Iterators

10.1 Introduction

10.1.1 Iterators

Iterators are a powerful technique in object-oriented programming and one of the fundamental design patterns [35]. Roughly speaking, an iterator is a small, light-weight object, which is associated with a specific kind of linear sequence. An iterator can be used to access all items in a linear sequence step-by-step. In this section, different iterator classes are introduced for traversing the nodes and the edges of a graph, and for traversing all ingoing and/or outgoing edges of a single node.

Iterators are an alternative to the iteration macros introduced in sect. 8.1.3.(i). For example, consider the following iteration pattern:

```
node v;
forall_nodes (n, G) { ... }
```

Using the class *NodeIt* introduced in sect. 10.2, this iteration can be re-written as follows:

```
for (NodeIt it (G); it.valid(); ++it) { ... }
```

The crucial differences are:

- Iterators provide an intuitive means of movement through the topology of a graph.

- Iterators are not bound to a loop, which means that the user has finer control over the iteration process. For example, the continuation condition *it.valid()* in the above loop could be replaced by another condition to terminate the loop once a specific node has been found (and the loop may be re-started at the same position later on).
- The meaning of iteration may be modified seamlessly. For example, the filter iterators defined in sect. 10.9 restrict the iteration to a subset that is specified by an arbitrary logical condition (*predicate*). In other words, the nodes or edges that do not fulfill this predicate are filtered out automatically during iteration.
- The functionality of iteration may be extended seamlessly. For example, the observer iterators defined in sect. 10.11 can be used to record details of the iteration. A concrete example is given in sect. 10.11: an observer iterator can be initialized such that it records the number of iterations performed by the iterator.
- Iterator-based implementations of algorithms can be easily integrated into environments that are implemented according to the STL style [63], (this style has been adopted for the standard C++ library). For this purpose, sect. 10.12 define adapters, which convert graph iterators into STL iterators.

10.1.2 Handles and Iterators

Iterators can be used whenever the corresponding handle can be used. For example, node iterators can be used where a node is requested or edge iterators can be used where an edge is requested. For adjacency iterators, it is possible to use them whenever an edge is requested[1].

An example shows how iterators can be used as handles:

```
NodeIt it(G);
while (it.valid()) {
 cout << "current node " << index(it) << endl; }
```

10.1.3 STL Iterators

Those who are more used to STL may take advantage from the following iterator classes: `NodeIt_n`, `EdgeIt_e`, `AdjIt_n`, `AdjIt_e`, `OutAdjIt_n`, `OutAdjIt_e`, `InAdjIt_n`, `InAdjIt_e`. The purpose of each iterator is the same as in the corresponding standard iterator classes `NodeIt`, `EdgeIt` ... The difference is the interface, which is exactly that of the STL iterator wrapper classe (see sect. 10.12 for more information).

[1]Since the edge of an adjacency iterator changes while the fixed node remains fixed, we decided to focus on the edge.

An example shows why these classes are useful (remember the example from the beginning):

```
NodeIt_n base(G);
for(NodeIt_n::iterator it=base.begin();it!=base.end(); ++it) {
 cout << "current node " << index(*it) << endl; }
```

As in STL collections there are public type definitions in all STL style graph iterators. The advantage is that algorithms can be written that operate independingly of the underlying type (note: `NodeIt_n` and `NodeIt_n::iterator` are equal types).

10.1.4 Circulators

Circulators differ from Iterators in their semantics. Instead of becoming invalid at the end of a sequence, they perform cyclic iteration. This type of "none–ending–iterator" is heavily used in the CGAL .

10.1.5 Data Accessors

Data accessor is a design pattern[**?**] that decouples data access from underlying implementation. Here, the pattern is used to decouple data access in graph algorithms from how data is actually stored outside the algorithm.

Generally, an attributed graph consists of a (directed or undirected) graph and an arbitrary number of node and edge attributes. For example, the nodes of a graph are often assigned attributes such as names, flags, and coordinates, and likewise, the edges are assigned attributes such as lengths, costs, and capacities.

More formally, an *attribute* a of a set S has a certain type T and assigns a value of T to every element of S (in other words, a may be viewed as a function $a : S \to T$). An *attributed set* $A = (S, a_1, \ldots, a_m)$ consists of a set S and attributes $a_1, \ldots, a_m$. An attributed graph is a (directed or undirected) graph $G = (V, E)$ such that the node set V and the edge set E are attributed.

Basically, LEDA provides two features to define attributes for graphs:

- Classes *GRAPH* and *UGRAPH* (sects. 8.2 and 8.4) are templates with two arguments, *vtype* and *etype*, which are reserved for a node and an edge attribute, respectively. To attach several attributes to nodes and edges, *vtype* and *etype* must be instantiated by structs whose members are the attributes.

- A *node array* (sect. 8.7) or *node map* (sect. Node Maps) represents a node attribute, and analogously, *edge arrays* (sect. Edge Arrays) and *edge maps* (sect. 8.11), represent edge attributes. Several attributes can be attached to nodes and edges by instantiating several arrays or maps.

Data accessors provide a uniform interface to access attributes, and the concrete organization of the attributes is hidden behind this interface. Hence, if an implementation of an algorithm does not access attributes directly, but solely in terms of data accessors, it may be applied to any organization of the attributes (in contrast, the algorithms in sect. Graph Algorithms require an organization of all attributes as node and edge arrays).

Every data accessor class *DA* comes with a function template *get*:

```
T get(DA da, Iter it);
```

This function returns the value of the attribute managed by the data accessor *da* for the node or edge marked by the iterator *it*. Moreover, most data accessor classes also come with a function template *set*:

```
void set(DA da, Iter it, T value);
```

This function overwrites the value of the attribute managed by the data accessor *da* for the node or edge marked by the iterator *it* by *value*.

The data accessor classes that do not provide a function template *set* realize attributes in such a way that a function *set* does not make sense or is even impossible. The *constant accessor* in sect. 10.14 is a concrete example: it realizes an attribute that is constant over the whole attributed set and over the whole time of the program. Hence, it does not make sense to provide a function *set*. Moreover, since the constant accessor class organizes its attribute in a non-materialized fashion, an overwriting function *set* is even impossible.

Example: The following trivial algorithm may serve as an example to demonstrate the usage of data accessors and their interplay with various iterator types. The first, nested loop accesses all edges once. More specifically, the outer loop iterates over all nodes of the graph, and the inner loop iterates over all edges leaving the current node of the outer loop. Hence, for each edge, the value of the attribute managed by the data accessor *da* is overwritten by *t*. In the second loop, a linear edge iterator is used to check whether the first loop has set all values correctly.

```
template <class T, class DA>
void set_and_check (graph& G, DA da, T t) {
  for (NodeIt nit(G); nit.valid(); ++nit)
```

```
      for (OutAdjIt oait(nit); oait.valid(); ++oait)
        set (da, eit, t);
    for (EdgeIt eit(G); eit.valid(); ++eit)
      if (get(da,it) != t) cout << "Error!" << endl;
  }
```

To demonstrate the application of function *set_and_check*, we first consider the case that G is an object of the class *GRAPH* derived from *graph* (sect. 8.1), that the template argument *vtype* is instantiated by a struct type *attributes*, and that the `int`-member *my_attr* of *attributes* shall be processed by *set_and_check* with value 1. Then DA can be instantiated as a *node_member_da*:

```
    node_member_da<attributes,int> da (&attributes::my_attr);
    set_and_check (G, da, 1);
```

Now we consider the case that the attribute to be processed is stored in an *edge_array*$<$*int*$>$ named *my_attr_array*:

```
    node_array_da<int> da (my_attr_array);
    set_and_check (G, da, 1);
```

Hence, all differences between these two cases are factored out into a single declaration statement.

10.1.6 Graphiterator Algorithms

Several basic graph algorithms were re–implemented to use only graph iterators and data accessors. Moreover they share three design decisions:

1. **algorithms are instances** of classes
2. algorithm instances have the **ability to "advance"**
3. algorithm instances provide **access to their internal states**

An example for an algorithm that supports the first two decisions is:

```
class Algorithm {
  int state, endstate;
public:
  Algorithm(int max) : endstate(max), state(0) { }
  void next() { state++; }
  bool finished() { return state>=endstate; }
};
```

With this class `Algorithm` we can easily instantiate an algorithm object:

```
Algorithm alg(5);
while (!alg.finished()) alg.next();
```

This small piece of code creates an algorithm object and invokes "next()" until it has reached an end state.

An advantage of this design is that we can write basic algorithms, which can be used in a standardized way and if needed, inspection of internal states and variables can be provided without writing complex code. Additionally, it makes it possible to write persistent algorithms, if the member variables are persistent.

Actually, those algorithms are quite more flexible than ordinary written algorithm functions:

```
template<class Alg>
class OutputAlg {
  Alg alg;
public:
  OutputAlg(int m) : alg(m) {
    cout << "max state: " << m << endl; }
  void next() {
    cout << "old state: " << alg.state;
    alg.next();
    cout << " new state: " << alg.state << endl; }
  bool finished() { return alg.finished(); }
};
```

This wrapper algorithm can be used like this:

```
OutputAlg<Algorithm> alg(5);
while (!alg.finished()) alg.next();
```

In addition to the algorithm mentioned earlier this wrapper writes the internal states to the standard output.

This is as efficient as rewriting the "`Algorithm`"-class with an output mechanism, but provides more flexibility.

10.2 Node Iterators (NodeIt)

1. Definition

a variable *it* of class *NodeIt* is a linear node iterator that iterates over the node set of a graph; the current node of an iterator object is said to be "marked" by this object.

#include < *LEDA/graph_iterator.h* >

2. Creation

NodeIt it; introduces a variable *it* of this class associated with no graph.

NodeIt it(*leda_graph G*);

introduces a variable *it* of this class associated with G. The graph is initialized by G. The node is initialized by *G.first_node*( ).

NodeIt it(*leda_graph G*, *leda_node n*);

introduces a variable *it* of this class marked with n and associated with G.
Precondition: n is a node of G.

3. Operations

void *it*.init(*leda_graph G*) associates *it* with G and marks it with *G.first_node*( ).

void *it*.init(*leda_graph G*, *leda_node v*)

associates *it* with G and marks it with v.

void *it*.reset( ) resets *it* to *G.first_node*( ), where G is the associated graph.

void *it*.make_invalid( ) makes *it* invalid, i.e. *it.valid*( ) will be false afterwards and *it* marks no node.

void *it*.reset_end( ) resets *it* to *G.last_node*( ), where G is the associated graph.

void *it*.update(*leda_node n*)

it marks n afterwards.

void	*it*.insert( )	creates a new node and *it* marks it afterwards.
void	*it*.del( )	deletes the marked node, i.e. *it.valid*( ) returns false afterwards. Precondition: *it.valid*( ) returns true.
NodeIt&	*it* = *it2*	*it* is afterwards associated with the same graph and node as *it2*. This method returns a reference to *it*.
bool	*it* == *it2*	returns true if and only if *it* and *it2* are equal, i.e. if the marked nodes are equal.
leda_node	*it*.get_node( )	returns the marked node or nil if *it.valid*( ) returns false.
leda_graph	*it*.get_graph( )	returns the associated graph.
bool	*it*.valid( )	returns true if and only if end of sequence not yet passed, i.e. if there is a node in the node set that was not yet passed.
bool	*it*.eol( )	returns !*it.valid*( ) which is true if and only if there is no successor node left, i.e. if all nodes of the node set are passed (eol: end of list).
NodeIt&	++*it*	performs one step forward in the list of nodes of the associated graph. If there is no successor node, *it.eol*( ) will be true afterwards. This method returns a reference to *it*. Precondition: *it.valid*( ) returns true.
NodeIt&	−−*it*	performs one step backward in the list of nodes of the associated graph. If there is no predecessor node, *it.eol*( ) will be true afterwards. This method returns a reference to *it*. Precondition: *it.valid*( ) returns true.

4. Implementation

Creation of an iterator and all methods take constant time.

10.3 Edge Iterators (EdgeIt)

1. Definition

a variable *it* of class *EdgeIt* is a linear edge iterator that iterates over the edge set of a graph; the current edge of an iterator object is said to be “marked” by this object.

#include < *LEDA/graph_iterator.h* >

2. Creation

EdgeIt it;	introduces a variable *it* of this class associated with no graph.
EdgeIt it(*leda_graph G*);	introduces a variable *it* of this class associated with *G* and marked with *G.first_edge*( ).
EdgeIt it(*leda_graph G*, *leda_edge e*);	introduces a variable *it* of this class marked with *e* and associated with *G*. Precondition: *e* is an edge of *G*.

3. Operations

void	*it*.init(*leda_graph G*)	associates *it* with *G* and marks it with *G.first_edge*( ).
void	*it*.init(*leda_graph G*, *leda_edge e*)	associates *it* with *G* and marks it with *e*.
void	*it*.update(*leda_edge e*)	*it* marks *e* afterwards.
void	*it*.reset( )	resets *it* to *G.first_edge*( ) where *G* is the associated graph.
void	*it*.reset_end( )	resets *it* to *G.last_edge*( ) where *G* is the associated graph.
void	*it*.make_invalid( )	makes *it* invalid, i.e. *it.valid*( ) will be false afterwards and *it* marks no node.
void	*it*.insert(*leda_node v1*, *leda_node v2*)	creates a new edge from *v1* to *v2* and *it* marks it afterwards.
void	*it*.del( )	deletes the marked edge, i.e. *it.valid*( ) returns false afterwards. Precondition: *it.valid*( ) returns true.
EdgeIt&	*it* = *it2*	assigns *it2* to *it*. This method returns a reference to *it*.
bool	*it* == *it2*	returns true if and only if *it* and *it2* are equal, i.e. if the marked edges are equal.

bool	*it*.eol( )	returns !*it.valid*( ) which is true if and only if there is no successor edge left, i.e. if all edges leaving the marked node are passed (eol: end of list).
bool	*it*.valid( )	returns true if and only if end of sequence not yet passed, i.e. if there is an edge leaving the marked node that was not yet passed.
leda_edge	*it*.get_edge( )	returns the marked edge or nil if *it.valid*( ) returns false.
leda_graph	*it*.get_graph( )	returns the associated graph.
EdgeIt&	++*it*	performs one step forward in the list of edges of the associated graph. If there is no successor edge, *it.eol*( ) will be true afterwards. This method returns a reference to *it*. Precondition: *it.valid*( ) returns true.
EdgeIt&	−−*it*	performs one step backward in the list of edges of the associated graph. If there is no predecessor edge, *it.eol*( ) will be true afterwards. This method returns a reference to *it*. Precondition: *it.valid*( ) returns true.

4. Implementation

Creation of an iterator and all methods take constant time.

10.4 Face Iterators (FaceIt)

1. Definition

a variable *it* of class *FaceIt* is a linear face iterator that iterates over the face set of a graph; the current face of an iterator object is said to be “marked” by this object.

Precondition: Before using any face iterator the list of faces has to be computed by calling *G.compute_faces*( ). Note, that any update operation invalidates this list.

#include < *LEDA/graph_iterator.h* >

2. Creation

FaceIt it;	introduces a variable *it* of this class associated with no graph.

FaceIt *it*(*leda_graph* G);

introduces a variable *it* of this class associated with G. The graph is initialized by G. The face is initialized by *G.first_face*( ).

FaceIt *it*(*leda_graph* G, *leda_face* n);

introduces a variable *it* of this class marked with n and associated with G.
Precondition: n is a face of G.

3. Operations

void	*it*.init(*leda_graph* G)	associates *it* with G and marks it with *G.first_face*( ).
void	*it*.init(*leda_graph* G, *leda_face* v)	associates *it* with G and marks it with v.
void	*it*.reset( )	resets *it* to *G.first_face*( ), where G is the associated graph.
void	*it*.make_invalid( )	makes *it* invalid, i.e. *it.valid*( ) will be false afterwards and *it* marks no face.
void	*it*.reset_end( )	resets *it* to *G.last_face*( ), where G is the associated graph.
void	*it*.update(*leda_face* n)	*it* marks n afterwards.
FaceIt&	*it* = *it2*	*it* is afterwards associated with the same graph and face as *it2*. This method returns a reference to *it*.
bool	*it* == *it2*	returns true if and only if *it* and *it2* are equal, i.e. if the marked faces are equal.
leda_face	*it*.get_face( )	returns the marked face or nil if *it.valid*( ) returns false.
leda_graph	*it*.get_graph( )	returns the associated graph.
bool	*it*.valid( )	returns true if and only if end of sequence not yet passed, i.e. if there is a face in the face set that was not yet passed.
bool	*it*.eol( )	returns !*it.valid*( ) which is true if and only if there is no successor face left, i.e. if all faces of the face set are passed (eol: end of list).

FaceIt&	$++it$	performs one step forward in the list of faces of the associated graph. If there is no successor face, $it.eol(\,)$ will be true afterwards. This method returns a reference to it. Precondition: $it.valid(\,)$ returns true.
FaceIt&	$--it$	performs one step backward in the list of faces of the associated graph. If there is no predecessor face, $it.eol(\,)$ will be true afterwards. This method returns a reference to it. Precondition: $it.valid(\,)$ returns true.

4. Implementation

Creation of an iterator and all methods take constant time.

10.5 Adjacency Iterators for leaving edges (OutAdjIt)

1. Definition

a variable *it* of class *OutAdjIt* is an adjacency iterator that marks a node (which is fixed in contrast to linear node iterators) and iterates over the edges that leave this node.

There is a variant of the adjacency iterators, so–called circulators which are heavily used in the CGAL[2]. The names of the classes are `OutAdjCirc` and `InAdjCirc` and their interfaces are completely equal to the iterator versions while they internally use e.g. `cyclic_adj_succ()` instead of `adj_succ()`.

#include < *LEDA/graph_iterator.h* >

2. Creation

OutAdjIt *it*; introduces a variable *it* of this class associated with no graph.

OutAdjIt *it*(*leda_graph G*);

introduces a variable *it* of this class associated with G.
The node is initialized by $G.first\_node(\,)$ and the edge by $G.first\_adj\_edge(n)$ where n is the marked node.

OutAdjIt *it*(*leda_graph G*, *leda_node n*);

introduces a variable *it* of this class marked with n and associated with G. The marked edge is initialized by $G.first\_adj\_edge(n)$.
Precondition: n is a node of G.

[2]See the CGAL homepage at `http://www.cs.uu.nl/CGAL/`.

OutAdjIt $it(leda\_graph\ G,\ leda\_node\ \ n,\ leda\_edge\ e);$

introduces a variable *it* of this class marked with n and e and associated with G.
Precondition: n is a node and e an edge of G and $source(e) = n$.

3. Operations

void it.init($leda\_graph\ G$) — associates *it* with G and marks it with $n' = G.first\_node(\ )$ and $G.first\_adj\_edge(n')$.

void it.init($leda\_graph\ G,\ leda\_node\ n$)

associates *it* with G and marks it with n and $G.first\_adj\_edge(n)$.
Precondition: n is a node of G.

void it.init($leda\_graph\ G,\ leda\_node\ n,\ leda\_edge\ e$)

associates *it* with G and marks it with n and e.
Precondition: n is a node and e an edge of G and $source(e) = n$.

void it.update($leda\_edge\ \ e$)

it marks e afterwards.

void it.reset( ) — resets *it* to $G.first\_adj\_edge(n)$ where G and n are the marked node and associated graph.

void it.insert($OutAdjIt\ other$)

creates a new leaving edge from the marked node of *it* to the marked node of *other*. *it* is marked with the new edge afterwards. The marked node of *it* does not change.

void it.del( ) — deletes the marked leaving edge, i.e. $it.valid(\ )$ returns false afterwards.
Precondition: $it.valid(\ )$ returns true.

void it.reset_end( ) — resets *it* to $G.last\_adj\_edge(n)$ where G and n are the marked node and associated graph.

void it.make_invalid( ) — makes *it* invalid, i.e. $it.valid(\ )$ will be false afterwards and *it* marks no node.

void it.update($leda\_node\ \ n$)

it marks n and the first leaving edge of n afterwards.

void it.update($leda\_node\ \ n,\ leda\_edge\ \ e$)

it marks n and e afterwards.

OutAdjIt& $it\ =\ it2$ — assigns *it2* to *it*. This method returns a reference to *it*.

bool	*it* == *it2*	returns true if and only if *it* and *it2* are equal, i.e. if the marked nodes and edges are equal.
bool	*it*.has_node( )	returns true if and only if *it* marks a node.
bool	*it*.eol( )	returns !*it.valid*( ) which is true if and only if there is no successor edge left, i.e. if all edges of the edge set are passed (eol: end of list).
bool	*it*.valid( )	returns true if and only if end of sequence not yet passed, i.e. if there is an edge in the edge set that was not yet passed.
leda_edge	*it*.get_edge( )	returns the marked edge or nil if *it.valid*( ) returns false.
leda_node	*it*.get_node( )	returns the marked node or nil if *it.has_node*( ) returns false.
leda_graph	*it*.get_graph( )	returns the associated graph.
OutAdjIt	*it*.curr_adj( )	returns a new adjacency iterator that is associated with $n' = target(e)$ and $G.first\_adj\_edge(n')$ where G is the associated graph. Precondition: *it.valid*( ) returns true.
OutAdjIt&	++*it*	performs one step forward in the list of outgoing edges of the marked node. If there is no successor edge, *it.eol*( ) will be true afterwards. This method returns a reference to *it*. Precondition: *it.valid*( ) returns true.
OutAdjIt&	−−*it*	performs one step backward in the list of outgoing edges of the marked node. If there is no predecesssor edge, *it.eol*( ) will be true afterwards. This method returns a reference to *it*. Precondition: *it.valid*( ) returns true.

4. Implementation

Creation of an iterator and all methods take constant time.

10.6 Adjacency Iterators for incoming edges (InAdjIt)

1. Definition

a variable *it* of class *InAdjIt* is an adjacency iterator that marks a node (which is fixed in contrast to linear node iterators) and iterates over the incoming edges of this node.

#include < LEDA/graph_iterator.h >

2. Creation

InAdjIt it; introduces a variable *it* of this class associated with no graph.

InAdjIt it(*leda_graph G*);

introduces a variable *it* of this class associated with G. The node is initialized by *G.first_node*( ) and the edge by *G.first_in_edge*(n) where n is the marked node.

InAdjIt it(*leda_graph G*, *leda_node n*);

introduces a variable *it* of this class marked with n and associated with G. The marked edge is initialized by *G.first_in_edge*(n). Precondition: n is a node of G.

InAdjIt it(*leda_graph G*, *leda_node n*, *leda_edge e*);

introduces a variable *it* of this class marked with n and e and associated with G. Precondition: n is a node and e an edge of G and $target(e) = n$.

3. Operations

void *it*.init(*leda_graph G*) associates *it* with G and marks it with $n' = G.first\_node(\,)$ and $G.first\_adj\_edge(n')$.

void *it*.init(*leda_graph G*, *leda_node n*)

associates *it* with G and marks it with n and *G.first_adj_edge*(n). Precondition: n is a node of G.

void *it*.init(*leda_graph G*, *leda_node n*, *leda_edge e*)

associates *it* with G and marks it with n and e. Precondition: n is a node and e an edge of G and $target(e) = n$.

void *it*.update(*leda_edge e*)

it marks e afterwards.

void *it*.reset( ) resets *it* to *G.first_in_edge*(n) where G and n are the marked node and associated graph.

void	*it*.insert(*InAdjIt other*)	creates a new incoming edge from the marked node of *it* to the marked node of *other*. *it* is marked with the new edge afterwards. The marked node of *it* does not change.
void	*it*.del( )	deletes the marked incoming edge, i.e. *it.valid*( ) returns false afterwards. Precondition: *it.valid*( ) returns true.
void	*it*.reset_end( )	resets *it* to *G.last_in_edge*(n) where G and n are the marked node and associated graph.
void	*it*.make_invalid( )	makes *it* invalid, i.e. *it.valid*( ) will be false afterwards and *it* marks no node.
void	*it*.update(*leda_node n*)	*it* marks n and the first incoming edge of n afterwards.
void	*it*.update(*leda_node n*, *leda_edge e*)	*it* marks n and e afterwards.
InAdjIt&	*it* = *it2*	assigns *it2* to *it*. This method returns a reference this method returns a reference to *it*.
bool	*it* == *it2*	returns true if and only if *it* and *it2* are equal, i.e. if the marked nodes and edges are equal.
bool	*it*.has_node( )	returns true if and only if *it* marks a node.
bool	*it*.eol( )	returns !*it.valid*( ) which is true if and only if there is no successor edge left, i.e. if all edges of the edge set are passed (eol: end of list).
bool	*it*.valid( )	returns true if and only if end of sequence not yet passed, i.e. if there is an edge in the edge set that was not yet passed.
leda_edge	*it*.get_edge( )	returns the marked edge or nil if *it.valid*( ) returns false.
leda_node	*it*.get_node( )	returns the marked node or nil if *it.has_node*( ) returns false.
leda_graph	*it*.get_graph( )	returns the associated graph.
InAdjIt	*it*.curr_adj( )	returns a new adjacency iterator that is associated with $n' = source(e)$ and *G.first_in_edge*(n') where G is the associated graph. Precondition: *it.valid*( ) returns true.

InAdjIt&	$++it$	performs one step forward in the list of incoming edges of the marked node. If there is no successor edge, *it.eol*( ) will be true afterwards. This method returns a reference to *it*. Precondition: *it.valid*( ) returns true.
InAdjIt&	$--it$	performs one step backward in the list of incoming edges of the marked node. If there is no predecesssor edge, *it.eol*( ) will be true afterwards. This method returns a reference to *it*. Precondition: *it.valid*( ) returns true.

4. Implementation

Creation of an iterator and all methods take constant time.

10.7 Adjacency Iterators (AdjIt)

1. Definition

a variable *it* of class *AdjIt* is an adjacency iterator that marks a node (which is fixed in contrast to linear node iterators) and iterates over the edges that leave or enter this node. At first, all outgoing edges will be traversed.

Internally, this iterator creates two instances of `OutAdjIt` and `InAdjIt`. The iteration is a sequenced iteration over both iterators. Note that this only fits for directed graphs, for undirected graphs you should use `OutAdjIt` instead.

#include < *LEDA/graph_iterator.h* >

2. Creation

AdjIt *it*; introduces a variable *it* of this class associated with no graph.

AdjIt *it*(*leda_graph* G);

introduces a variable *it* of this class associated with G. The marked node is initialized by n = *G.first_node*( ) and the edge by *G.first_adj_edge*(n).

AdjIt *it*(*leda_graph* G, *leda_node* n);

introduces a variable *it* of this class marked with n and associated with G. The marked edge is initialized by by *G.first_adj_edge*(n).
Precondition: n is a node of G.

AdjIt $it(leda\_graph\ G,\ leda\_node\ \ n,\ leda\_edge\ e)$;

introduces a variable it of this class marked with n and e and associated with G.
Precondition: n is a node and e an edge of G and $source(e) = n$.

3. Operations

void	it.init($graphtype\ G$)	associates it with G and marks it with $n' = G.first\_node(\,)$ and $G.first\_adj\_edge(n')$.
void	it.init($graphtype\ G,\ nodetype\ n$)	associates it with G and marks it with n and $G.first\_adj\_edge(v)$. Precondition: n is a node of G.
void	it.init($graphtype\ G,\ nodetype\ n,\ edgetype\ e$)	associates it with G and marks it with n and e. Precondition: n is a node and e an edge of G and $source(e) = n$.
void	it.update($leda\_edge\ \ e$)	it marks e afterwards.
void	it.reset( )	resets it to $G.first\_adj\_edge(n)$ where G and n are the marked node and associated graph.
void	it.insert($AdjIt\ other$)	creates a new edge from the marked node of it to the marked node of $other$. it is marked with the new edge afterwards. The marked node of it does not change.
void	it.del( )	deletes the marked leaving edge, i.e. $it.valid(\,)$ returns false afterwards. Precondition: $it.valid(\,)$ returns true.
void	it.reset_end( )	resets it to $G.last\_adj\_edge(n)$ where G and n are the marked node and associated graph.
void	it.make_invalid( )	makes it invalid, i.e. $it.valid(\,)$ will be false afterwards and it marks no node.
void	it.update($leda\_node\ \ n$)	it marks n and the first leaving edge of n afterwards.
void	it.update($leda\_node\ \ n,\ leda\_edge\ \ e$)	it marks n and e afterwards.
AdjIt&	$it\ =\ it2$	assigns $it2$ to it. This method returns a reference to it.
bool	$it == it2$	returns true if and only if it and $it2$ are equal, i.e. if the marked nodes and edges are equal.

bool	*it*.has_node( )	returns true if and only if *it* marks a node.
bool	*it*.eol( )	returns !*it.valid*( ) which is true if and only if there is no successor edge left, i.e. if all edges of the edge set are passed (eol: end of list).
bool	*it*.valid( )	returns true if and only if end of sequence not yet passed, i.e. if there is an edge in the edge set that was not yet passed.
leda_edge	*it*.get_edge( )	returns the marked edge or nil if *it.valid*( ) returns false.
leda_node	*it*.get_node( )	returns the marked node or nil if *it.has_node*( ) returns false.
leda_graph	*it*.get_graph( )	returns the associated graph.
AdjIt	*it*.curr_adj( )	If the currently associated edge leaves the marked node, this method returns a new adjacency iterator that is associated with $n' = target(e)$ and $G.first\_adj\_edge(n')$ where G is the associated graph. Otherwise it returns a new adjacency iterator that is associated with $n' = source(e)$ and $G.first\_in\_edge(n')$ where G is the associated graph. Precondition: *it.valid*( ) returns true.
AdjIt&	++*it*	performs one step forward in the list of incident edges of the marked node. If the formerly marked edge was a leaving edge and there is no successor edge, *it* is associated to $G.first\_in\_edge(n)$ where G and n are the associated graph and node. If the formerly marked edge was an incoming edge and there is no successor edge, *it.eol*( ) will be true afterwards. This method returns a reference to *it*. Precondition: *it.valid*( ) returns true.
AdjIt&	−−*it*	performs one step backward in the list of incident edges of the marked node. If the formerly marked edge was an incoming edge and there is no predecessor edge, *it* is associated to $G.last\_adj\_edge(n)$ where G and n are the associated graph and node. If the formerly marked edge was a leaving edge and there is no successor edge, *it.eol*( ) will be true afterwards. This method returns a reference to *it*. Precondition: *it.valid*( ) returns true.

4. Implementation

Creation of an iterator and all methods take constant time.

10.8 Face Circulators (FaceCirc)

1. Definition

a variable *fc* of class *FaceCirc* is a face circulator that circulates through the set of edges of a face as long as the graph is embedded combinatorically correct, i.e. the graph has to be bidirected and a map (see 8.1).

#include < *LEDA/graph_iterator.h* >

2. Creation

FaceCirc *fc*; introduces a variable *fc* of this class associated with no graph.

FaceCirc *fc*(*leda_graph* *G*);

introduces a variable *fc* of this class associated with *G*. The edge is initialized to `nil`.

FaceCirc *fc*(*leda_graph* *G*, *leda_edge* *e*);

introduces a variable *fc* of this class marked with *e* and associated with *G*.
Precondition: *e* is an edge of *G*.

3. Operations

void	*fc*.init(*leda_graph* *G*)	associates *fc* with *G*.
void	*fc*.init(*leda_graph* *G*, *leda_edge* *e*)	associates *fc* with *G* and marks it with *e*. Precondition: *e* is an edge of *G*.
void	*fc*.update(*leda_edge* *e*)	*fc* marks *e* afterwards.
void	*fc*.make_invalid( )	makes *fc* invalid, i.e. *fc.valid*( ) will be false afterwards and *fc* marks no edge.
FaceCirc&	*fc* = *fc2*	assigns *fc2* to *fc*. This method returns a reference to *fc*.
bool	*fc* == *fc2*	returns true if and only if *fc* and *fc2* are equal, i.e. if the marked edges are equal.
bool	*fc*.has_edge( )	returns true if and only if *fc* marks an edge.
bool	*fc*.eol( )	returns !*fc.valid*( ).

bool	*fc*.valid( )	returns true if and only if the circulator is marked with an edge.
leda_edge	*fc*.get_edge( )	returns the marked edge or nil if *fc.valid*( ) returns false.
leda_graph	*fc*.get_graph( )	returns the associated graph.
FaceCirc&	++*fc*	redirects the circulator to the cyclic adjacency predecessor of *reversal*(e), where e is the marked edge. This method returns a reference to *fc*. Precondition: *fc.valid*( ) returns true.
FaceCirc&	−−*fc*	redirects the circulator to the cyclic adjacency successor of e, where e is the marked edge. This method returns a reference to *fc*. Precondition: *fc.valid*( ) returns true.

4. Implementation

Creation of a circulator and all methods take constant time.

10.9 Filter Node Iterator (FilterNodeIt)

1. Definition

An instance *it* of class *FilterNodeIt*< *Predicate*, *Iter* > encapsulates an object of type `Iter` and creates a restricted view on the set of nodes over which this internal iterator iterates. More specifically, all nodes that do not fulfill the predicate defined by `Predicate` are filtered out during this traversal.

Class `FilterEdgeIt` and `FilterAdjIt` are defined analogously, i.e. can be used for edge iterators or adjacency iterators, respectively.

Precondition: The template parameter `Iter` must be a node iterator, e.g. `NodeIt` or `FilterNodeIt<pred,NodeIt>`. `Predicate` must be a class which provides a method *operator*( ) according to the following signature: `bool operator() (Iter)`.

#include < *LEDA/graph_iterator.h* >

2. Creation

FilterNodeIt< *Predicate*, *Iter* > *it*;

introduces a variable *it* of this class, not bound to a predicate or iterator.

FilterNodeIt< *Predicate*, *Iter* > *it*(*Predicate pred*, *Iter base_it*);

introduces a variable *it* of this class bound to *pred* and *base_it*.

3. Operations

void *it*.init(*Predicate pred*, *Iter base_it*)

initializes *it*, which is bound to *pred* and *base_it* afterwards.
Precondition: *it* is not yet bound to a predicate or iterator.

4. Implementation

Constant overhead.

5. Example

Suppose each node has an own colour and we only want to see those with a specific colour, for example red (we use the LEDA colours). At first the data structures:

```
GRAPH<color,double> G;
NodeIt it(G);
```

We would have to write something like this:

```
while(it.valid()) {
  if (G[it.get_node()]==red) do_something(it);
  ++it;
}
```

With the filter wrapper class we can add the test if the node is red to the behaviour of the iterator.

```
struct RedPred {
  bool operator() (const NodeIt& it) const {
  return G[it.get_node()]==red; }
} redpred;
FilterNodeIt<RedPred,NodeIt> red_it(redpred,it);
```

This simplifies the loop to the following:

```
while(red_it.valid()) {
  do_something(red_it);
  ++red_it; }
```

All ingredients of the comparison are hard-wired in struct `RedPred`: the type of the compared values (`color`), the comparison value (red) and the binary comparison (equality). The following class `CompPred` renders these three choices flexible.

10.10 Comparison Predicate (CompPred)

1. Definition

An instance *cp* of class $CompPred$<$Iter, DA, Comp$> is a predicate comparator that produces boolean values with the given compare function and the attribute associated with an iterator.

#include < *LEDA/graph_iterator.h* >

2. Creation

$CompPred$<$Iter, DA, Comp$> $cp(DA\ da,\ Comp\ comp,\ DA::value\_type\ val)$;

introduces a variable *cp* of this class and associates it to the given data accessor *da*, compare function *comp* and value *val*.

Precondition: `Comp` is a pointer-to-function type which takes two values of type *_typename DA*:: *value_type* and produces a boolean return value. `Comp` might also be a class with member function *bool operator*( )(*_typename DA*:: *value_type*, *_typename DA*:: *value_type*).

3. Example

In the following example, a node iterator for red nodes will be created. At first the basic part (see sect. 10.13 for explanation of the data accessor `node_array_da`):

```
graph G;
NodeIt it(G);
node_array<color> na_colour(G,black);
node_array_da<color> da_colour(na_colour);
assign_some_color_to_each_node();
```

Now follows the definition of a “red iterator” (`Equal<T>` yields true, if the given two values are equal):

```
template<class T>
class Equal {
  public:
  bool operator() (T t1, T t2) const {
  return t1==t2; }
};

typedef CompPred<NodeIt,node_array_da<color>,Equal<color> > Predicate;
Predicate PredColour(da_colour,Equal<color>(),red);
FilterNodeIt<Predicate,NodeIt> red_it(PredColour,it);
```

This simplifies the loop to the following:

```
while(red_it.valid()) {
  do_something(red_it);
++red_it; }
```

`Equal<T>` is a class that compares two items of the template parameter `T` by means of a method `bool operator()(T,T);`. There are some classes available for this purpose: `Equal<T>`, `Unequal<T>`, `LessThan<T>`, `LessEqual<T>`, `GreaterThan<T>` and `GreaterEqual<T>` with obvious semantics, where `T` is the type of the values. Predicates of the STL can be used as well since they have the same interface.

10.11 Observer Node Iterator (ObserverNodeIt)

1. Definition

An instance *it* of class *ObserverNodeIt*<*Obs*, *Iter*> is an observer iterator. Any method call of iterators will be "observed" by an internal object of class *Obs*.

Class `ObserverEdgeIt` and `ObserverAdjIt` are defined analogously, i.e. can be used for edge iterators or adjacency iterators, respectively.

Precondition: The template parameter `Iter` must be a node iterator.

#include < *LEDA/graph_iterator.h* >

2. Creation

ObserverNodeIt<*Obs*, *Iter*> *it*;

introduces a variable *it* of this class, not bound to an observer or iterator.

ObserverNodeIt<Obs, Iter> it(Obs& obs, Iter base_it);

introduces a variable *it* of this class bound to the observer *obs* and *base_it*.
Precondition: `Obs` must have methods `observe_constructor()`, `observe_forward()`, `observe_update()`. These three methods may have arbitrary return types (incl. void).

3. Operations

void *it*.init(*Obs obs*, *Iter base_it*)

initializes *it*, which is bound to *obs* and *base_it* afterwards.
Precondition: *it* is not bound to an observer or iterator.

Obs& *it*.get_observer( ) returns a reference to the observer to which it is bound.

4. Example

First two simple observer classes. The first one is a dummy class, which ignores all notifications. The second one merely counts the number of calls to `operator++` for all iterators that share the same observer object through copy construction or assignment (of course, a real implementation should apply some kind of reference counting or other garbage collection).

In this example, the counter variable *_count* of class `SimpleCountObserver` will be initialized with the counter variable *_count* of class `DummyObserver`, i.e. the variable is created only once.

```
template <class Iter>
class DummyObserver {
  int* _count;
public:
  DummyObserver() : _count(new int(0)) { }
  void notify_constructor(const Iter& ) { }
  void notify_forward(const Iter& ) { }
  void notify_update(const Iter& ) { }
  int  counter() const { return *_count; }
  int* counter_ptr() const { return _count; }
  bool operator==(const DummyObserver& D) const {
    return _count==D._count; }
};

template <class Iter, class Observer>
class SimpleCountObserver {
```

```
  int*  _count;
public:
  SimpleCountObserver() : _count(new int(0)) { }
  SimpleCountObserver(Observer& obs) :
    _count(obs.counter_ptr()) { }
  void notify_constructor(const Iter& ) { }
  void notify_forward(const Iter& ) { ++(*_count); }
  void notify_update(const Iter& ) { }
  int  counter() const {  return *_count; }
  int* counter_ptr() const { return _count; }
  bool operator==(const SimpleCountObserver& S) const {
    return _count==S._count; }
};
```

Next an exemplary application, which counts the number of calls to `operator++` of all adjacency iterator objects inside `dummy_algorithm`. Here the dummy observer class is used only as a "Trojan horse," which carries the pointer to the counter without affecting the code of the algorithm.

```
template<class Iter>
bool break_condition (const Iter&) { ... }

template<class ONodeIt, class OAdjIt>
void  dummy_algorithm(ONodeIt& it, OAdjIt& it2) {
  while (it.valid()) {
    for (it2.update(it); it2.valid() && !break_condition(it2); ++it2)
    ++it;
  }
}

int write_count(graph& G) {
  typedef DummyObserver<NodeIt>                  DummyObs;
  typedef SimpleCountObserver<AdjIt,DummyObs>    CountObs;
  typedef ObserverNodeIt<DummyObs,NodeIt>        ONodeIt;
  typedef ObserverAdjIt<CountObs,AdjIt>          OAdjIt;

  DummyObs observer;
  ONodeIt   it(observer,NodeIt(G));
  CountObs  observer2(observer);
  OAdjIt    it2(observer2,AdjIt(G));
  dummy_algorithm(it,it2);
  return it2.get_observer().counter();
}
```

10.12 STL Iterator Wrapper (STLNodeIt)

1. Definition

An instance *it* of class *STLNodeIt*< *DataAccessor*, *Iter* > is a STL iterator wrapper for node iterators (e.g. `NodeIt`, `FilterNodeIt<pred,NodeIt>`). It adds all type tags and methods that are necessary for STL conformance; see the standard draft working paper for details. The type tag `value_type` is equal to `__typename DataAccessor::value_type` and the return value of `operator*`.

Class `STLEdgeIt` and `STLAdjIt` are defined analogously, i.e. can be used for edge iterators or adjacency iterators, respectively.

Precondition: The template parameter `Iter` must be a node iterator. *DataAccessor* must be a data accessor.

Note: There are specialized versions of STL wrapper iterator classes for each kind of iterator that return different LEDA graph objects.

class name	`operator*()` returns
`NodeIt_n`	`node`
`EdgeIt_e`	`edge`
`AdjIt_n`	`node`
`AdjIt_e`	`edge`
`OutAdjIt_n`	`node`
`OutAdjIt_e`	`edge`
`InAdjIt_n`	`node`
`InAdjIt_e`	`edge`

#include < *LEDA/graph_iterator.h* >

2. Creation

STLNodeIt< *DataAccessor*, *Iter* > *it*(*DataAccessor da*, *Iter base_it*);

introduces a variable *it* of this class bound to *da* and *base_it*.

3. Operations

STLNodeIt<*DataAccessor*, *Iter*>& *it* = *DataAccessor* :: *value_type i*

assigns the value i, i.e. $set(DA, it, i)$ will be invoked where DA is the associated data accessor and it the associated iterator.

bool $it == STLNodeIt\texttt{<}DataAccessor, Iter\texttt{>}\ it2$

returns `true` if the associated values of it and $it2$ are equal, i.e. $get(DA, cit) == get(DA, cit2)$ is true where cit is the associated iterator of it and $cit2$ is the associated iterator of $it2$ and DA is the associated data accessor.

bool $it\ != STLNodeIt\texttt{<}DataAccessor, Iter\texttt{>}\ it2$

returns `false` if the associated value equals the one of the given iterator.

$STLNodeIt\texttt{<}DataAccessor, Iter\texttt{>}\&$ it.begin( )

resets the iterator to the beginning of the sequence.

$STLNodeIt\texttt{<}DataAccessor, Iter\texttt{>}\&$ it.last( )

resets the iterator to the ending of the sequence.

$STLNodeIt\texttt{<}DataAccessor, Iter\texttt{>}\&$ it.end( )

makes the iterators invalid, i.e. past-the-end-value.

$DataAccessor::value\_type\&\ *it$

returns a reference to the associated value, which originally comes from data accessor da. If the associated iterator it is not valid, a dummy value reference is returned and should not be used.
Precondition: $access(DA, it)$ returns a non constant reference to the data associated to it in DA. This functions is defined for all implemented data accessors (e.g. `node_array_da`, `edge_array_da`).

10.13 Node Array Data Accessor (node_array_da)

1. Definition

An instance da of class $node\_array\_da\texttt{<}T\texttt{>}$ is instantiated with a LEDA `node_array<T>`.

The data in the node array can be accessed by the functions $get(da, it)$ and $set(da, it, value)$ that take as parameters an instance of $node\_array\_da\texttt{<}T\texttt{>}$ and an iterator, see below.

$node\_array\_da\texttt{<}T\texttt{>}$`::value_type` is a type and equals `T`.

For $node\_map\texttt{<}T\texttt{>}$ there is the variant $node\_map\_da\texttt{<}T\texttt{>}$ which is defined completely analogous to $node\_array\_da\texttt{<}T\texttt{>}$. Classes $edge\_array\_da\texttt{<}T\texttt{>}$ and $edge\_map\_da\texttt{<}T\texttt{>}$ are defined analogously, as well.

#include < LEDA/graph_iterator.h >

2. Creation

node_array_da<*T*> *da*;

introduces a variable *da* of this class that is not bound.

node_array_da<*T*> *da*(*leda_node_array*<*T*>& *na*);

introduces a variable *da* of this class bound to *na*.

3. Operations

T get(*node_array_da*<*T*> *da*, *Iter it*)

returns the associated value of *it* for this accessor.

void set(*node_array_da*<*T*>& *da*, *Iter it*, *T val*)

sets the associated value of *it* for this accessor to the given value.

4. Implementation

Constant Overhead.

5. Example

We count the number of 'red nodes' in a parameterized graph G.

```
int count_red(graph G, node_array<color> COL) {
  node_array_da<color> Color(COL);
  int counter=0;
  NodeIt it(G);
  while (it.valid()) {
    if (get(Color,it)==red) counter++;
    it++; }
  return counter;
}
```

Suppose we want to make this 'algorithm' flexible in the representation of colors. Then we could write this version:

```
template<class DA>
int count_red_t(graph G, DA Color) {
  int counter=0;
  NodeIt it(G);
```

```
    while (it.valid()) {
      if (get(Color,it)==red) counter++;
      it++; }
    return counter;
  }
```

With the templatized version it is easily to customize it to match the interface of the version:

```
  int count_red(graph G, node_array<color> COL) {
    node_array_da<color> Color(COL);
    return count_red_t(G,Color); }
```

10.14 Constant Accessors (constant_da)

1. Definition

An instance *ca* of class *constant_da*<*T*> is bound to a specific value of type `T`, and the function $get(ca, it)$ simply returns this value for each iterator.

#include < *LEDA/graph_iterator.h* >

2. Creation

constant_da<*T*> *ca*(*T t*);

introduces a variable *ca* of this class bound to the given value *t*.

3. Operations

T get(*constant_da*<*T*> *ca*, *Iter it*)

returns the value to which *ca* is bound.

4. Example

With the template function of sect. 10.13 we can write a function that counts the number of nodes in a graph:

```
  int count_all(graph G) {
    constant_da<color> Color(red);
    return count_red_t(G,Color); }
```

10.15 Node Member Accessors (node_member_da)

1. Definition

An instance *da* of class *node_member_da*<*Str*, *T*> manages the access to a node parameter that is organized as a member of a struct type, which is the first template argument of a parameterized graph `GRAPH<Str,?>`. The parameter is of type `T` and the struct of type `Str`.

Classes *edge_member_da*<*Str*, *T*> is defined completely analogously.

#include < *LEDA/graph_iterator.h* >

2. Creation

node_member_da<*Str*, *T*> *da*;

introduces a variable *da* of this class that is not bound.

node_member_da<*Str*, *T*> *da*(*Ptr ptr*);

introduces a variable *da* of this class, which is bound to *ptr*.

3. Operations

T get(*node_member_da*<*Str*, *T*> *ma*, *Iter it*)

returns the associated value of *it* for this accessor.

void set(*node_member_da*<*Str*, *T*>& *ma*, *Iter it*, *T val*)

sets the associated value of *it* for this accessor to the given value.

4. Implementation

Constant Overhead.

The instance *da* accesses its parameter through a pointer to member of type `Ptr`, which is defined for example by `typedef T Str::*Ptr`.

5. Example

We have a parameterized graph G where the node information type is the following struct type `Str`:

```
struct Str {
  int x;
  color col; };
```

We want to count the number of red nodes. Since we have the template function of sect. 10.13 we can easily use it to do the computation:

```
int count_red(GRAPH<Str,double> G) {
  node_member_da<Str,color> Color(&Str::col);
  return count_red_t(G,Color); }
```

10.16 Node Attribute Accessors (node_attribute_da)

1. Definition

An instance *da* of class *node_attribute_da*<*T*> manages the access to a node parameter with type `T` of a parameterized graph `GRAPH<T,?>`.

Classes *edge_attribute_da*<*T*> is defined completely analogously.

#include < *LEDA/graph_iterator.h* >

2. Creation

node_attribute_da<*T*> *da*;

introduces a variable *da* of this class.

3. Operations

T get(*node_attribute_da*<*T*> *ma*, *Iter it*)

returns the associated value of *it* for this accessor.

void set(*node_attribute_da*<*T*>& *ma*, *Iter it*, *T val*)

sets the associated value of *it* for this accessor to the given value.

4. Implementation

Constant Overhead.

5. Example

Given a parameterized graph G with nodes associated with colours, we want to count the number of red nodes. Since we have the template function of sect. 10.13 we can easily use it to do the computation:

```
int count_red(GRAPH<color,double> G) {
  node_attribute_da<color> Color;
  return count_red_t(G,Color); }
```

10.17 Breadth First Search (flexible) (GIT_BFS)

1. Definition

An instance *algorithm* of class *GIT_BFS*< *OutAdjIt*, *Queuetype*, *Mark* > is an implementation of an algorithm that traverses a graph in a breadth first order. The queue used for the search must be provided by the caller and contains the source(s) of the search.

- If the queue is only modified by appending the iterator representing the source node onto the queue, a normal breadth first search beginning at the node of the graph is performed.
- It is possible to initialize the queue with several iterators that represent different roots of breadth first trees.
- By modifying the queue while running the algorithm the behaviour of the algorithm can be changed.
- After the algorithm performed a breadth first search, one may append another iterator onto the queue to restart the algorithm.

Iterator version: There is an iterator version of this algorithm: `BFS_It`. Usage is similar to that of node iterators without the ability to go backward in the sequence.

#include < *LEDA/graph_iterator.h* >

2. Creation

GIT_BFS< *OutAdjIt*, *Queuetype*, *Mark* >
algorithm(*Queuetype q*, *Mark*& *ma*);

creates an instance *algorithm* of this class bound to the Queue q and data accessor ma.

Preconditions:

- `Queuetype` is a queue parameterized with items of type `OutAdjIt`.
- q contains the sources of the traversal (for each source node an adjacency iterator referring to it) and

- *ma* is a data accessor that provides read and write access to a boolean value for each node (accessed through iterators). This value is assumed to be freely usable by *algorithm*.

GIT_BFS< *OutAdjIt*, *Queuetype*, *Mark* >
algorithm(*Queuetype q*, *Mark*& *ma*, *OutAdjIt ai*);

creates an instance *algorithm* of this class bound to the queue *q*, data accessor *ma* and the adjacency iterator *ai* representing the source node of the breadth first traversal.

3. Operations

void	*algorithm*.next( )	Performs one iteration of the core loop of the algorithm.
OutAdjIt	*algorithm*.current( )	returns the "current" iterator.
void	*algorithm*.finish_algo( )	executes the algorithm until *finished*( ) is true, i.e. exactly if the Queue is empty.
bool	*algorithm*.finished( )	returns `true` if the internal Queue is empty.
Queuetype&	*algorithm*.get_queue( )	gives direct access to internal Queue.

4. Example

This example shows how to implement an algorithmic iterator for breadth first search:

```
class BFS_It {
  AdjIt               _source;
  node_array<da>      _handler;
  node_array_da<bool> _mark;
  queue<AdjIt>        _q;
  GIT_BFS<AdjIt,queue<AdjIt>,node_array_da<bool> >  _search;
public:
  BFS_It(graph& G) :
   _source(AdjIt(G)), _handler(G,false),
   _mark(_handler), _search(_q,_mark)
   {
    _search.get_queue().clear();
    _search.get_queue().append(_source);
   }
  bool valid() const { return !_search.finished();  }
```

```
  node get_node() const { return _search.current().get_node(); }
  BFS_It& operator++() {
   _search.next(); return *this; }
};
```

With this iterator you can easily iterate through a graph in breadth first fashion :

```
  graph G;
  BFS_It it(G);
  while (it.valid()) {
    // do something reasonable with 'it.get_node()'
    ++it;
  }
```

5. Implementation

Each operation requires constant time. Therefore, a normal breadth-first search needs $\mathcal{O}(m+n)$ time.

10.18 Depth First Search (flexible) (GIT_DFS)

1. Definition

An instance *algorithm* of class *GIT_DFS*< *OutAdjIt*, *Stacktype*, *Mark* > is an implementation of an algorithm that traverses a graph in a depth first order. The stack used for the search must be provided by the caller and contains the source(s) of the search.

- If the stack is only modified by pushing the iterator representing the source node onto the stack, a normal depth first search beginning at the node of the graph is performed.
- It is possible to initialize the stack with several iterators that represent different roots of depth first trees.
- By modifying the stack while running the algorithm the behaviour of the algorithm can be changed.
- After the algorithm performed a depth first search, one may push another iterator onto the stack to restart the algorithm.

A next step may return a state which describes the last action. There are the following three possibilities:

1. `dfs_shrink`: an adjacency iterator was popped from the stack, i.e. the treewalk returns in root-direction
2. `dfs_leaf`: same as `dfs_shrink`, but a leaf occured
3. `dfs_grow_depth`: a new adjacency iterator was appended to the stack because it was detected as not seen before, i.e. the treewalk goes in depth-direction
4. `dfs_grow_breadth`: the former current adjacency iterator was replaced by the successor iterator, i.e. the treewalk goes in breadth-direction

Iterator version: There is an iterator version of this algorithm: `DFS_It`. Usage is similar to that of node iterators without the ability to go backward in the sequence.

#include < *LEDA/graph_iterator.h* >

2. Creation

GIT_DFS< *OutAdjIt*, *Stacktype*, *Mark* >
algorithm(*Stacktype st*, *Mark*& *ma*);

creates an instance *algorithm* of this class bound to the stack *st* and data accessor *ma*.

Preconditions:

- `Stacktype` is a stack parameterized with items of type `OutAdjIt`.
- *st* contains the sources of the traversal (for each source node an adjacency iterator referring to it) and
- *ma* is a data accessor that provides read and write access to a boolean value for each node (accessed through iterators). This value is assumed to be freely usable by *algorithm*.

GIT_DFS< *OutAdjIt*, *Stacktype*, *Mark* >
algorithm(*Stacktype st*, *Mark*& *ma*, *OutAdjIt ai*);

creates an instance *algorithm* of this class bound to the stack *st*, data accessor *ma* and the adjacency iterator *ai* representing the source node of the depth first traversal.

3. Operations

void *algorithm*.next_unseen( )

Performs one iteration of the core loop of the algorithm for one unseen node of the graph.

dfs_return	*algorithm*.next( )	Performs one iteration of the core loop of the algorithm.
OutAdjIt	*algorithm*.current( )	returns the "current" iterator.
void	*algorithm*.finish_algo( )	executes the algorithm until *finished*( ) is true, i.e. exactly if the stack is empty.
bool	*algorithm*.finished( )	returns `true` if the internal stack is empty.
void	*algorithm*.init(*OutAdjIt* s)	initializes the internal stack with s.
Stacktype&	*algorithm*.get_stack( )	gives direct access to internal stack.

4. Implementation

Each operation requires constant time. Therefore, a normal depth-first search needs $\mathcal{O}(m+n)$ time.

10.19 Topological Sort (flexible) (GIT_TOPOSORT)

1. Definition

An instance *algorithm* of class *GIT_TOPOSORT*< *OutAdjIt*, *Indeg*, *Queuetype* > is an implementation of an algorithm that iterates over all nodes in some topological order, if the underlying graph is acyclic. An object of this class maintains an *internal queue*, which contains all nodes (in form of adjacency iterators where the current node is equal to the fixed node) that are not yet passed, but all its predecessors have been passed.

Iterator version: There is an iterator version of this algorithm: `TOPO_It`. Usage is similar to that of node iterators without the ability to go backward in the sequence and only a graph is allowed at creation time. Additionally there is `TOPO_rev_It` which traverses the graph in reversed topological order.

#include < *LEDA/graph_iterator.h* >

2. Creation

GIT_TOPOSORT< *OutAdjIt*, *Indeg*, *Queuetype* >
algorithm(*Indeg*& *indegree*);

creates an instance *algorithm* of this class bound to *indeg*. The internal queue of adjacency iterators is empty.

Preconditions:

- `Indeg` is a data accessor that must provide both read and write access
- *indegree* stores for every node that corresponds to any iterator the number of incoming edges (has to be to computed before)
- `Queuetype` is a queue parameterized with elements of type `OutAdjIt`

The underlying graph need not be acyclic. Whether or not it is acyclic can be tested after execution of the algorithm (function `cycle_found()`).

3. Operations

void *algorithm*.next( ) Performs one iteration of the core loop of the algorithm. More specifically, the first element of `get_queue()` is removed from the queue, and every immediate successor n of this node for which currently holds `get(indeg,n)==0` is inserted in `get_queue()`.

void *algorithm*.finish_algo( )
executes the algorithm until *finished*( ) is true, i.e. exactly if the queue is empty.

bool *algorithm*.finished( ) returns `true` if the internal queue is empty.

OutAdjIt *algorithm*.current( ) returns the current adjacency iterator.

Queuetype& *algorithm*.get_queue( )
gives direct access to internal queue.

bool *algorithm*.cycle_found( )
returns `true` if a cycle was found.

void *algorithm*.reset_acyclic( )
resets the internal flag that a cycle was found.

4. Implementation

The asymptotic complexity is $\mathcal{O}(m + n)$, where m is the number of edges and n the number of nodes.

5. Example

This algorithm performs a normal topological sort if the queue is initialized by the set of all nodes with indegree zero:

Definition of *algorithm*, where *indeg* is a data accessor that provides full data access to the number of incoming edges for each node:

```
GIT_TOPOSORT<OutAdjIt,Indeg,Queuetype<Nodehandle> > algorithm(indeg);
```

Initialization of `get_queue()` with all nodes of type `OutAdjIt::nodetype` that have zero indegree, i.e. `get(indeg,it)==indeg.value_null`.

```
while ( !algorithm.finished() ) {
  // do something reasonable with algo.current()
  algo.next();
}
```

The source code of function `toposort_count()` is implemented according to this pattern and may serve as a concrete example.

10.20 Strongly Connected Components (flexible) (GIT_SCC)

1. Definition

An instance *algorithm* of class *GIT_SCC*< *Out*, *In*, *It*, *OutSt*, *InSt*, *NSt*, *Mark* > is an implementation of an algorithm that computes the strongly connected components.

Iterator version: There is an iterator version of this algorithm: `SCC_It`. Usage is similar to that of node iterators without the ability to go backward in the sequence and only a graph is allowed at creation time. Method `compnumb()` returns the component number of the current node.

#include < *LEDA/graph_iterator.h* >

2. Creation

GIT_SCC< *Out*, *In*, *It*, *OutSt*, *InSt*, *NSt*, *Mark* >
algorithm(*OutSt ost*, *InSt ist*, *Mark ma*, *Out oai*, *It it*, *In iai*);

creates an instance *algorithm* of this class bound to the stack *st* and data accessor *ma*.

Preconditions:

- `Out` is an adjacency iterator that iterates over the outgoing edges of a fixed vertex
- `In` is an adjacency iterator that iterates over the incoming edges of a fixed vertex
- `OutSt` is stack parameterized with items of type `Out`

- `InSt` is stack parameterized with items of type `In`
- `Mark` is a data accessor that has access to a boolean value that is associated with each node of the graph

3. Operations

int *algorithm*.state( ) returns the internal state.

- `NEXT_OUT` =first phase,
- `NEXT_ORDER` =order phase,
- `NEXT_IN`=second phase,
- `NEXT_DONE` =algorithm finished

void *algorithm*.finish_algo( )
executes the algorithm until *finished*( ) is true.

bool *algorithm*.finished( ) returns `true` if the algorithm is finished.

InSt& *algorithm*.get_in_stack( )
gives direct access to the internal stack of incoming adjacency iterators.

In *algorithm*.in_current( )
returns the current iterator of the internal stack of incoming adjacency iterators.

OutSt& *algorithm*.get_out_stack( )
gives direct access to the internal stack of outgoing adjacency iterators.

Out *algorithm*.out_current( )
returns the current iterator of the internal stack of outgoing adjacency iterators.

itnodetype *algorithm*.current_node( )
returns the current node.

int *algorithm*.compnumb( )
returns the component number of the fixed node of the current iterator if current state is `NEXT_IN`.

int *algorithm*.next( ) Performs one iteration of the core loop of the algorithm.

4. Implementation

Each operation requires constant time. The algorithm has running time $\mathcal{O}(|V| + |E|)$.

10.21 Dijkstra(flexible) (GIT_DIJKSTRA)

1. Definition

An instance *algorithm* of this class is an implementation of Dijkstra that can be flexibly initialized, stopped after each iteration of the core loop, and continued, time and again.

Iterator version: There is an iterator version of this algorithm: `DIJKSTRA_It`. Usage is more complex and is documented in the graphiterator leda extension package.

#include < *LEDA/graph_iterator.h* >

2. Creation

GIT_DIJKSTRA< *OutAdjIt*, *Length*, *Distance*, *PriorityQueue*, *QueueItem* >
algorithm(*Length l*, *Distance& d*, *QueueItem qi*);

creates an instance *algorithm* of this class.

The length and distance data accessors are initialized by the parameter list. The set of sources is empty. `Length` is a read only data accessor that gives access to the length of edges and `Distance` is a read/write data accessor that stores the distance of the nodes. `PriorityQueue` is a Queue parameterized with element of type `OutAdjIt` and `QueueItem` is a data accessor gives access to elements of type `PriorityQueue::pq_item`.

Precondition: All edge lengths are initialized by values that are large enough to be taken as infinity.

Remark: This precondition is not necessary for the algorithm to have a defined behavior. In fact, it may even make sense to break this precondition deliberately. For example, if the distances have been computed before and shall only be updated after inserting new edges, it makes perfect sense to start the algorithm with these distances.

For a completely new computation, the node distances of all nodes are initialized to infinity(i.e. `distance.value_max`).

3. Operations

PriorityQueue& *algorithm*.get_queue( )

gives direct access to internal priority queue.

void *algorithm*.init(*OutAdjIt s*)

s is added to the set of sources.

bool	*algorithm*.finished( )	is true iff the algorithm is finished, i.e. the priority queue is empty.
OutAdjIt	*algorithm*.current( )	returns the current adjacency iterator.
OutAdjIt	*algorithm*.curr_adj( )	returns the an adjacency iterator that is currently adjacent to *current*( ).
bool	*algorithm*.is_pred( )	returns true if the current iterator satisfies the dijkstra condition. Can be used to compute the predecessors.
void	*algorithm*.next( )	performs one iteration of the core loop of the algorithm.
void	*algorithm*.finish_algo( )	executes the algorithm until *finished*( ) is true, i.e. exactly if the priority queue is empty.

4. Example

Class `GIT_DIJKSTRA` may be used in a deeper layer in a hierarchy of classes and functions. For example, you may write a function which computes shortes path distances with given iterators and data accessors:

```
template<class OutAdjIt, class Length, class Distance,
         class PriorityQueue, class QueueItem>
void GIT_dijkstra_core(OutAdjIt s, Length& length, Distance& distance,
  PriorityQueue& pq, QueueItem& qi) {
  GIT_DIJKSTRA<OutAdjIt,Length,Distance,PriorityQueue,QueueItem>
    internal_dijk(length,distance,qi);
  internal_dijk.get_queue()=pq;
  set(distance,s,distance.value_null);
  if (s.valid()) {
    internal_dijk.init(s);
    internal_dijk.finish_algo();
  }
}
```

In another layer, you would instantiate these iterators and data acessors for a graph and invoke this function.

5. Implementation

The asymptotic complexity is $\mathcal{O}(m + n \cdot T(n))$, where $T(n)$ is the(possibly amortized) complexity of a single queue update.

For the priority queues described in Chapter 7.1, it is $T(n) = \mathcal{O}(\log n)$.

Chapter 11

Basic Data Types for Two-Dimensional Geometry

LEDA provides a collection of simple data types for computational geometry, such as points, vectors, directions, hyperplanes, segments, rays, lines, affine transformations, circles, polygons, and operations connecting these types.

The computational geometry kernel has evolved over time. The first kernel (types *point*, *line*, ...) was restricted to two-dimensional geometry and used floating point arithmetic as the underlying arithmetic. We found it very difficult to implement reliable geometric algorithms based on this kernel. See the chapter on computational geometry of [58] for some examples of the danger of floating point arithmetic in geometric computations. Starting with version 3.2 we therefore also provided a kernel based on exact rational arithmetic (types *rat_point*, *rat_segment* ...). This kernel is still restricted to two dimensions. The further development of computational geometry will occur in close cooperation with the CGAL effort [27, 69].

All two-dimensional object types defined in this section support the following operations:

Equality and Identity Tests

bool	identical(*object* p, *object* q)	Test for identity.
bool	$p == q$	Test for equality.
bool	$p != q$	Test for inequality.

I/O Operators

ostream&	*ostream&* O << *object* x	writes the object x to output stream O.
istream&	*istream&* I >> *object&* x	reads an object from input stream I into variable x.

11.1 Points (point)

1. Definition

An instance of the data type *point* is a point in the two-dimensional plane $\mathbb{R}^2$. We use (x, y) to denote a point with first (or x-) coordinate x and second (or y-) coordinate y.

#include < *LEDA/point.h* >

2. Types

point:: *coord_type*	the coordinate type (*double*).
point:: *point_type*	the point type (*point*).

3. Creation

point *p*;	introduces a variable p of type *point* initialized to the point $(0, 0)$.
point *p*(*double* *x*, *double* *y*);	introduces a variable p of type *point* initialized to the point (x, y).
point *p*(*vector* *v*);	introduces a variable p of type *point* initialized to the point $(v[0], v[1])$. *Precondition*: : $v.dim(\,) \;=\; 2$.
point *p*(*point* *p*, *int* *prec*);	introduces a variable p of type *point* initialized to the point with coordinates $(\lfloor P * x\rfloor/P, \lfloor P * x\rfloor/P)$, where $p = (x, y)$ and $P = 2^{prec}$. If *prec* is non-positive, the new point has coordinates x and y.

4. Operations

double	*p*.xcoord( )	returns the first coordinate of p.
double	*p*.ycoord( )	returns the second coordinate of p.
vector	*p*.to_vector( )	returns the vector $\vec{xy}$.
double	*p*.sqr_dist(*point* *q*)	returns the square of the Euclidean distance between p and q.
int	*p*.cmp_dist(*point* *q*, *point* *r*)	returns $compare(p.sqr\_dist(q), p.sqr\_dist(r))$.
double	*p*.xdist(*point* *q*)	returns the horizontal distance between p and q.

double	p.ydist(*point* q)	returns the vertical distance between p and q.
double	p.distance(*point* q)	returns the Euclidean distance between p and q.
double	p.distance( )	returns the Euclidean distance between p and $(0,0)$.
double	p.angle(*point* q, *point* r)	returns the angle between $\vec{pq}$ and $\vec{pr}$.
point	p.translate_by_angle(*double* $alpha$, *double* d)	returns p translated in direction $alpha$ by distance d. The direction is given by its angle with a right oriented horizontal ray.
point	p.translate(*double* dx, *double* dy)	returns p translated by vector (dx, dy).
point	p.translate(*vector* v)	returns p+v, i.e., p translated by vector v. *Precondition*: v.dim() = 2.
point	p + *vector* v	returns p translated by vector v.
point	p − *vector* v	returns p translated by vector $-v$.
point	p.rotate(*point* q, *double* a)	returns p rotated about q by angle a.
point	p.rotate(*double* a)	returns p.rotate(*point*$(0,0)$, a).
point	p.rotate90(*point* q)	returns p rotated about q by an angle of 90 degrees.
point	p.rotate90( )	returns p.rotate90(*point*$(0,0)$).
point	p.reflect(*point* q, *point* r)	returns p reflected across the straight line passing through q and r.
point	p.reflect(*point* q)	returns p reflected across point q.
vector	$p - q$	returns the difference vector of the coordinates.
ostream&	*ostream&* $O \ll p$	writes p to output stream O.
istream&	*istream&* $I \gg$ *point&* p	reads the coordinates of p (two *double* numbers) from input stream I.

Non-Member Functions

int cmp_distances(*point p1*, *point p2*, *point p3*, *point p4*)

compares the distances $(p1, p2)$ and $(p3, p4)$. Returns $+1$ (-1) if distance $(p1, p2)$ is larger (smaller) than distance $(p3, p4)$, otherwise 0.

point center(*point a*, *point b*)

returns the center of a and b, i.e. $a + \vec{ab}/2$.

point midpoint(*point a*, *point b*)

returns the center of a and b.

int orientation(*point a*, *point b*, *point c*)

computes the orientation of points a, b, and c as the sign of the determinant

$$\begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix}$$

i.e., it returns $+1$ if point c lies left of the directed line through a and b, 0 if a,b, and c are collinear, and -1 otherwise.

int cmp_signed_dist(*point a*, *point b*, *point c*, *point d*)

compares (signed) distances of c and d to the straight line passing through a and b (directed from a to b). Returns $+1$ (-1)if c has larger (smaller) distance than d and 0 if distances are equal.

double area(*point a*, *point b*, *point c*)

computes the signed area of the triangle determined by a,b,c, positive if $orientation(a, b, c) > 0$ and negative otherwise.

bool collinear(*point a*, *point b*, *point c*)

returns *true* if points a, b, c are collinear, i.e., $orientation(a, b, c) = 0$, and *false* otherwise.

bool right_turn(*point a*, *point b*, *point c*)

returns *true* if points a, b, c form a righ turn, i.e., $orientation(a, b, c) > 0$, and *false* otherwise.

bool left_turn(*point a*, *point b*, *point c*)

returns *true* if points a, b, c form a left turn, i.e., $orientation(a, b, c) < 0$, and *false* otherwise.

int side_of_halfspace(*point a*, *point b*, *point c*)

returns the sign of the scalar product $(b-a)\cdot(c-a)$. If $b \neq a$ this amounts to: Let h be the open halfspace orthogonal to the vector $b-a$, containing b, and having a in its boundary. Returns $+1$ if c is contained in h, returns 0 is c lies on the the boundary of h, and returns -1 is c is contained in the interior of the complement of h.

int side_of_circle(*point a*, *point b*, *point c*, *point d*)

returns $+1$ if point d lies left of the directed circle through points a, b, and c, 0 if a,b,c,and d are cocircular, and -1 otherwise.

bool incircle(*point a*, *point b*, *point c*, *point d*)

returns *true* if point d lies in the interior of the circle through points a, b, and c, and *false* otherwise.

bool outcircle(*point a*, *point b*, *point c*, *point d*)

returns *true* if point d lies outside of the circle through points a, b, and c, and *false* otherwise.

bool cocircular(*point a*, *point b*, *point c*, *point d*)

returns *true* if points a, b, c, and d are corcircular.

bool affinely_independent(*array*<*point*> *A*)

decides whether the points in A are affinely independent.

bool contained_in_simplex(*array*<*point*> *A*, *point p*)

determines whether p is contained in the simplex spanned by the points in A. A may consists of up to 3 points.
Precondition: The points in A are affinely independent.

bool contained_in_affine_hull(*array*<*point*> *A*, *point p*)

determines whether p is contained in the affine hull of the points in A.

11.2 Segments (segment)

1. Definition

An instance s of the data type *segment* is a directed straight line segment in the two-dimensional plane, i.e., a straight line segment $[p, q]$ connecting two points $p, q \in \mathbb{R}^2$. p is called the *source* or start point and q is called the *target* or end point of s. The length of s is the Euclidean distance between p and q. If $p = q$ s is called empty. We use $line(s)$ to denote a straight line containing s. The angle between a right oriented horizontal ray and s is called the direction of s.

#include < *LEDA/segment.h* >

2. Types

segment:: *coord_type* — the coordinate type (*double*).

segment:: *point_type* — the point type (*point*).

3. Creation

segment s(*point* p, *point* q);

introduces a variable s of type *segment*. s is initialized to the segment $[p, q]/$

segment s(*point* p, *vector* v);

introduces a variable s of type *segment*. s is initialized to the segment $[p, p + v]$.
Precondition: $v.dim() = 2$.

segment s(*double* *x1*, *double* *y1*, *double* *x2*, *double* *y2*);

introduces a variable s of type *segment*. s is initialized to the segment $[(x_1, y_1), (x_2, y_2)]$.

segment s(*point* p, *double* *alpha*, *double* *length*);

introduces a variable s of type *segment*. s is initialized to the segment with start point p, direction *alpha*, and length *length*.

segment s; — introduces a variable s of type *segment*. s is initialized to the empty segment.

segment s(*segment* s, *int* *prec*);

introduces a variable s of type *segment*. s is initialized to a copy of s.

4. Operations

point	s.start( )	returns the source point of segment s.
point	s.end( )	returns the target point of segment s.
double	s.xcoord1( )	returns the x-coordinate of s.source().
double	s.xcoord2( )	returns the x-coordinate of s.target().
double	s.ycoord1( )	returns the y-coordinate of s.source().
double	s.ycoord2( )	returns the y-coordinate of s.target().
double	s.dx( )	returns the $xcoord2 - xcoord1$.
double	s.dy( )	returns the $ycoord2 - ycoord1$.
double	s.slope( )	returns the slope of s. *Precondition*: s is not vertical.
double	s.sqr_length( )	returns the square of the length of s.
double	s.length( )	returns the length of s.
vector	s.to_vector( )	returns the vector $s.target(\ ) - s.source(\ )$.
double	s.direction( )	returns the direction of s as an angle in the intervall $[0, 2\pi)$.
double	s.angle( )	returns s.direction().
double	s.angle(*segment* t)	returns the angle between s and t, i.e., t.direction() - s.direction().
bool	s.is_trivial( )	returns true if s is trivial.
bool	s.is_vertical( )	returns true iff s is vertical.
bool	s.is_horizontal( )	returns true iff s is horizontal.
double	s.x_proj(*double* y)	returns p.xcoord(), where $p \in line(s)$ with p.ycoord() $= y$. *Precondition*: s is not horizontal.
double	s.y_proj(*double* x)	returns p.ycoord(), where $p \in line(s)$ with p.xcoord() $= x$. *Precondition*: s is not vertical.
double	s.y_abs( )	returns the y-abscissa of $line(s)$, i.e., s.y_proj(0). *Precondition*: s is not vertical.
bool	s.contains(*point* p)	decides whether s contains p.

bool s.intersection(*segment t*)
decides whether s and t intersect in one point.

bool s.intersection(*segment t*, *point*& p)
if s and t intersect in a single point this point is assigned to p and the result is true, otherwise the result is false.

bool s.intersection_of_lines(*segment t*, *point*& p)
if $line(s)$ and $line(t)$ intersect in a single point this point is assigned to p and the result is true, otherwise the result is false.

segment s.translate_by_angle(*double alpha*, *double d*)
returns s translated in direction $alpha$ by distance d.

segment s.translate(*double dx*, *double dy*)
returns s translated by vector (dx, dy).

segment s.translate(*vector v*)
returns $s + v$, i.e., s translated by vector v.
Precondition: v.dim() = 2.

segment s + *vector v* returns s translated by vector v.

segment s − *vector v* returns s translated by vector $-v$.

segment s.perpendicular(*point p*)
returns the segment perpendicular to s with source p and target on $line(s)$.

double s.distance(*point p*) returns the Euclidean distance between p and s.

double s.sqr_dist(*point p*) returns the squared Euclidean distance between p and s.

double s.distance( ) returns the Euclidean distance between $(0, 0)$ and s.

segment s.rotate(*point q*, *double a*)
returns s rotated about point q by angle a.

segment s.rotate(*double alpha*)
returns s.rotate(s.source(), $alpha$).

segment s.rotate90(*point q*) returns s rotated about q by an angle of 90 degrees.

segment s.rotate90( ) returns s.rotate90(s.source()).

segment s.reflect(*point p*, *point q*)
returns s reflected across the straight line passing through p and q.

segment s.reflect(*point* p) returns s reflected across point p.

segment s.reverse( ) returns s reversed.

ostream& *ostream&* $O \ll s$ writes s to output stream O.

istream& *istream&* $I \gg$ *segment&* s

reads the coordinates of s (four *double* numbers) from input stream I.

Non-Member Functions

int orientation(*segment* s, *point* p)

computes orientation($s.source()$, $s.target()$, p).

int cmp_slopes(*segment s1*, *segment s2*)

returns compare(slope(s_1), slope(s_2)).

int cmp_segments_at_xcoord(*segment s1*, *segment s2*, *point* p)

compares points $l_1 \cap v$ and $l_2 \cap v$ where l_i is the line underlying segment s_i and v is the vertical straight line passing through point p.

bool parallel(*segment s1*, *segment s2*)

returns true if $s1$ and $s2$ are parallel and false otherwise.

11.3 Straight Rays (ray)

1. Definition

An instance r of the data type *ray* is a directed straight ray in the two-dimensional plane. The angle between a right oriented horizontal ray and r is called the direction of r.

#include < *LEDA/ray.h* >

2. Types

ray :: *coord_type*	the coordinate type (*double*).
ray :: *point_type*	the point type (*point*).

3. Creation

ray r(*point* p, *point* q);	introduces a variable r of type *ray*. r is initialized to the ray starting at point p and passing through point q.
ray r(*segment* s);	introduces a variable r of type *ray*. r is initialized to $ray(s.source(), s.target())$.
ray r(*point* p, *vector* v);	introduces a variable r of type *ray*. r is initialized to $ray(p, p+v)$.
ray r(*point* p, *double* *alpha*);	introduces a variable r of type *ray*. r is initialized to the ray starting at point p with direction *alpha*.
ray r;	introduces a variable r of type *ray*. r is initialized to the ray starting at the origin with direction 0.
ray r(*ray* r, *int* *prec*);	introduces a variable r of type *ray*. r is initialized to a copy of r. The second argument is for compatibility with *rat_ray*.

4. Operations

point	r.source( )	returns the source of r.
point	r.point1( )	returns the source of r.
point	r.point2( )	returns a point on r different from $r.source(\,)$.
double	r.direction( )	returns the direction of r.
double	r.angle(*ray* s)	returns the angle between r and s, i.e., s.direction() $-$ r.direction().

bool	r.is_vertical( )	returns true iff r is vertical.
bool	r.is_horizontal( )	returns true iff r is horizontal.
double	r.slope( )	returns the slope of r. *Precondition*: r is not vertical.
bool	r.intersection(*ray* s, *point*& *inter*)	if r and s intersect in a single point this point is assigned to *inter* and the result is *true*, otherwise the result is *false*.
bool	r.intersection(*segment* s, *point*& *inter*)	if r and s intersect in a single point this point is assigned to *inter* and the result is *true*, otherwise the result is *false*.
ray	r.translate_by_angle(*double* a, *double* d)	returns r translated in direction a by distance d.
ray	r.translate(*double* dx, *double* dy)	returns r translated by vector (dx, dy).
ray	r.translate(*vector* v)	returns r translated by vector v *Precondition*: v.dim() = 2.
ray	r + *vector* v	returns r translated by vector v.
ray	r − *vector* v	returns r translated by vector $-v$.
ray	r.rotate(*point* q, *double* a)	returns r rotated about point q by angle a.
ray	r.rotate(*double* a)	returns r.rotate(*point*$(0, 0)$, a).
ray	r.rotate90(*point* q)	returns r rotated about q by angle of 90 degrees.
ray	r.reflect(*point* p, *point* q)	returns r reflected across the straight line passing through p and q.
ray	r.reflect(*point* p)	returns r reflected across point p.
ray	r.reverse( )	returns r reversed.
bool	r.contains(*point*)	decides whether r contains p.
bool	r.contains(*segment*)	decides whether r contains s.

Non-Member Functions

int orientation(*ray r*, *point p*)

computes orientation(a, b, p), where $a \neq b$ and a and b appear in this order on ray r.

int cmp_slopes(*ray r1*, *ray r2*)

returns compare(slope(r_1), slope(r_2)).

11.4 Straight Lines (line)

1. Definition

An instance l of the data type *line* is a directed straight line in the two-dimensional plane. The angle between a right oriented horizontal line and l is called the direction of l.

#include < *LEDA/line.h* >

2. Types

line:: *coord_type*	the coordinate type (*double*).
line:: *point_type*	the point type (*point*).

3. Creation

line l(*point* p, *point* q);	introduces a variable l of type *line*. l is initialized to the line passing through points p and q directed form p to q.
line l(*segment* s);	introduces a variable l of type *line*. l is initialized to the line supporting segment s.
line l(*ray* r);	introduces a variable l of type *line*. l is initialized to the line supporting ray r.
line l(*point* p, *vector* v);	introduces a variable l of type *line*. l is initialized to the line passing through points p and $p + v$.
line l(*point* p, *double* *alpha*);	introduces a variable l of type *line*. l is initialized to the line passing through point p with direction *alpha*.
line l;	introduces a variable l of type *line*. l is initialized to the line passing through the origin with direction 0.

4. Operations

point	l.point1( )	returns a point on l.
point	l.point2( )	returns a second point on l.
segment	l.seg( )	returns a segment on l.
double	l.angle(*line* g)	returns the angle between l and g, i.e., g.direction() $-$ l.direction().
double	l.direction( )	returns the direction of l.

double	l.angle( )	returns l.direction().
bool	l.is_vertical( )	returns true iff l is vertical.
bool	l.is_horizontal( )	returns true iff l is horizontal.
double	l.sqr_dist(*point q*)	returns the square of the distance between l and q.
double	l.distance(*point q*)	returns the distance between l and q.
double	l.slope( )	returns the slope of l. *Precondition*: l is not vertical.
double	l.y_proj(*double x*)	returns p.ycoord(), where $p \in l$ with p.xcoord() $= x$. *Precondition*: l is not vertical.
double	l.x_proj(*double y*)	returns p.xcoord(), where $p \in l$ with p.ycoord() $= y$. *Precondition*: l is not horizontal.
double	l.y_abs( )	returns the y-abscissa of l (l.y_proj(0)). *Precondition*: l is not vertical.
bool	l.intersection(*line g*, *point*& *p*)	if l and g intersect in a single point this point is assigned to p and the result is true, otherwise the result is false.
bool	l.intersection(*segment s*, *point*& *inter*)	if l and s intersect in a single point this point is assigned to p and the result is true, otherwise the result is false.
line	l.translate_by_angle(*double a*, *double d*)	returns l translated in direction a by distance d.
line	l.translate(*double dx*, *double dy*)	returns l translated by vector (dx, dy).
line	l.translate(*vector v*)	returns l translated by vector v. *Precondition*: v.dim() $= 2$.
line	l + *vector v*	returns l translated by vector v.
line	l − *vector v*	returns l translated by vector $-v$.
line	l.rotate(*point q*, *double a*)	returns l rotated about point q by angle a.
line	l.rotate(*double a*)	returns l.rotate(*point*$(0,0)$, a).
line	l.rotate90(*point q*)	returns l rotated about q by angle of 90 degrees.

line	l.reflect(*point* p, *point* q)	returns l reflected across the straight line passing through p and q.
line	l.reverse( )	returns l reversed.
segment	l.perpendicular(*point* p)	returns the segment perpendicular to l with source p. and target on l.
point	l.dual( )	returns the point dual to l. /precond l is not vertical.
bool	l.contains(*point* p)	returns true if p lies on l.
bool	l.clip(*point* p, *point* q, *segment*& s)	clips l at the rectangle R defined by p and q. Returns true if the intersection of R and l is non-empty and returns false otherwise. If the intersection is non-empty the intersection is assigned to s; It is guaranteed that the source node of s is no larger than its target node.

Non-Member Functions

int	orientation(*line* l, *point* p)	computes orientation(a, b, p), where $a \neq b$ and a and b appear in this order on line l.
int	cmp_slopes(*line* *l1*, *line* *l2*)	returns compare(slope(l_1), slope(l_2)).

11.5 Circles (circle)

1. Definition

An instance C of the data type *circle* is an oriented circle in the plane passing through three points p_1, p_2, p_3. The orientation of C is equal to the orientation of the three defining points, i.e. $orientation(p_1, p_2, p_3)$. If $|\{p_1, p_2, p_3\}| = 1$ C is the empty circle with center p_1. If p_1, p_2, p_3 are collinear C is a straight line passing through p_1, p_2 and p_3 in this order and the center of C is undefined.

#include < *LEDA/circle.h* >

2. Types

circle:: *coord_type*	the coordinate type (*double*).
circle:: *point_type*	the point type (*point*).

3. Creation

circle C(*point* a, *point* b, *point* c);

introduces a variable C of type *circle*. C is initialized to the oriented circle through points a, b, and c.

circle C(*point* a, *point* b); introduces a variable C of type *circle*. C is initialized to the counter-clockwise oriented circle with center a passing through b.

circle C(*point* a); introduces a variable C of type *circle*. C is initialized to the trivial circle with center a.

circle C; introduces a variable C of type *circle*. C is initialized to the trivial circle with center $(0, 0)$.

circle C(*point* c, *double* r);

introduces a variable C of type *circle*. C is initialized to the circle with center c and radius r with positive (i.e. counter-clockwise) orientation.

circle C(*double* x, *double* y, *double* r);

introduces a variable C of type *circle*. C is initialized to the circle with center (x, y) and radius r with positive (i.e. counter-clockwise) orientation.

circle C(*circle* c, *int* *prec*);

introduces a variable C of type *circle*. C is initialized to a copy of c. The second argument is for compatability with *rat_circle*.

4. Operations

point	C.center( )	returns the center of C. *Precondition*: The orientation of C is not 0.
double	C.radius( )	returns the radius of C. *Precondition*: The orientation of C is not 0.
double	C.sqr_radius( )	returns the squared radius of C. *Precondition*: The orientation of C is not 0.
point	C.point1( )	returns p_1.
point	C.point2( )	returns p_2.
point	C.point3( )	returns p_3.
point	C.point_on_circle(*double alpha*, *double* $= 0$)	returns a point p on C with angle of *alpha*. The second argument is for compatability with *rat_circle*.
bool	C.is_degenerate( )	returns true if the defining points are collinear.
bool	C.is_trivial( )	returns true if C has radius zero.
int	C.orientation( )	returns the orientation of C.
int	C.side_of(*point* p)	returns -1, $+1$, or 0 if p lies right of, left of, or on C respectively.
bool	C.inside(*point* p)	returns true if p lies inside of C, false otherwise.
bool	C.outside(*point* p)	returns !C.inside(p).
bool	C.contains(*point* p)	returns true if p lies on C, false otherwise.
circle	C.translate_by_angle(*double* a, *double* d)	returns C translated in direction a by distance d.
circle	C.translate(*double* dx, *double* dy)	returns C translated by vector (dx, dy).
circle	C.translate(*vector* v)	returns C translated by vector v.
circle	C + *vector* v	returns C translated by vector v.
circle	C − *vector* v	returns C translated by vector $-v$.

circle	C.rotate(*point* q, *double* a)	returns C rotated about point q by angle a.
circle	C.rotate(*double* a)	returns C rotated about the origin by angle a.
circle	C.rotate90(*point* q)	returns C rotated about q by an angle of 90 degrees.
circle	C.reflect(*point* p, *point* q)	returns C reflected across the straight line passing through p and q.
circle	C.reflect(*point* p)	returns C reflected across point p.
circle	C.reverse( )	returns C reversed.
list<*point*>	C.intersection(*circle* D)	returns $C \cap D$ as a list of points.
list<*point*>	C.intersection(*line* l)	returns $C \cap l$ as a list of (zero, one, or two) points sorted along l.
list<*point*>	C.intersection(*segment* s)	returns $C \cap s$ as a list of (zero, one, or two) points sorted along s.
segment	C.left_tangent(*point* p)	returns the line segment starting in p tangent to C and left of segment $[p, C.center()]$.
segment	C.right_tangent(*point* p)	returns the line segment starting in p tangent to C and right of segment $[p, C.center()]$.
double	C.distance(*point* p)	returns the distance between C and p.
double	C.sqr_dist(*point* p)	returns the squared distance between C and p.
double	C.distance(*line* l)	returns the distance between C and l.
double	C.distance(*circle* D)	returns the distance between C and D.

11.6 Polygons (POLYGON)

1. Definition

There are two instantiations of POLYGON. The instantiation with the rational kernel is called *rat_polygon* and requires inclusion of rat_polygon.h and the instantiation with the floating point kernel is called *polygon* and requires inclusion of polygon.h.

An instance P of the data type $POLYGON$ is a cyclic list of points (equivalently segments) in the plane. A polygon is called *simple* if all nodes of the graph induced by its segments have degree two and it is called weakly simple, if its segments are disjoint except for common endpoints and if the chain does not cross itself. See the LEDA book for more details.

A weakly simple polygon splits the plane into an unbounded region and one or more bounded regions. For a simple polygon there is just one bounded region. When a weakly simple polygon P is traversed either the bounded region is consistently to the left of P or the unbounded region is consistently to the left of P. We say that P is positively oriented in the former case and negatively oriented in the latter case. We use P to also denote the region to the left of P and call this region the positive side of P.

The number of vertices is called the size of P. A polygon with empty vertex sequence is called empty.

There are two instantiations of $POLYGONS$, namely *rat_polygon* and *polygon*. Only the former guarantee correct results. All operations for *rat_polygon* are also available for *polygon*. There is a small number of operations that are only available for *polygon*.

#include < *LEDA/generic/POLYGON.h* >

2. Types

$POLYGON$ :: *coord_type*	the coordinate type (*rational*).
$POLYGON$ :: *point_type*	the point type (*rat_point*).
$POLYGON$ :: *float_type*	the corresponding floatin-point type (*polygon*).

3. Creation

$POLYGON$ P;	introduces a variable P of type $POLYGON$. P is initialized to the empty polygon.

POLYGON P(*list*<*POINT*> *pl*, *CHECK_TYPE check* = *POLYGON*::*SIMPLE*, *RESPECT_TYPE respect_orientation* = *POLYGON*::*RESPECT_ORIENTATION*);

introduces a variable P of type *POLYGON*. P is initialized to the polygon with vertex sequence *pl*. If *respect_orientation* is DISREGARD_ORIENTATION, the positive orientation is chosen.
Precondition: If *check* is SIMPLE, *pl* must define a simple polygon, and if *check* is WEAKLY_SIMPLE, *pl* must define a weakly simple polygon. If no test is to performed, the second argument has to be set to NO_CHECK. The constants NO_CHECK, SIMPLE, and WEAKLY_SIMPLE are part of a local enumeration type CHECK_TYPE.

POLYGON P(*polygon* Q, *int prec* = *rat_point*::*default_precision*);

introduces a variable P of type *POLYGON*. P is initialized to a rational approximation of the (floating point) polygon Q of coordinates with denominator at most *prec*. If *prec* is zero, the implementation chooses *prec* large enough such that there is no loss of precision in the conversion.

4. Operations

polygon	P.to_float( )	returns a floating point approximation of P.
bool	P.is_simple( )	tests whether P is simple or not.
bool	P.is_weakly_simple( )	tests whether P is weakly simple or not.
list<*POINT*>	P.vertices( )	returns the sequence of vertices of P in counterclockwise ordering.
list<*SEGMENT*>	P.segments( )	returns the sequence of bounding segments of P in counterclockwise ordering.
list<*POINT*>	P.intersection(*SEGMENT* s)	returns the proper crossings between P and s as a list of points.
list<*POINT*>	P.intersection(*LINE* l)	returns the proper crossings between P and l as a list of points.
POLYGON	P.intersect_halfplane(*LINE* l)	returns the intersection of P with the halfspace on the positive side of l.
int	P.size( )	returns the size of P.

bool	*P*.empty( )	returns true if *P* is empty, false otherwise.
POLYGON	*P*.translate(*RAT_TYPE dx*, *RAT_TYPE dy*)	returns *P* translated by vector (dx, dy).
POLYGON	*P*.translate(*INT_TYPE dx*, *INT_TYPE dy*, *INT_TYPE dw*)	returns *P* translated by vector $(dx/dw, dy/dw)$.
POLYGON	*P*.translate(*VECTOR v*)	returns *P* translated by vector *v*.
POLYGON	$P +$ *VECTOR v*	returns *P* translated by vector *v*.
POLYGON	$P -$ *VECTOR v*	returns *P* translated by vector $-v$.
POLYGON	*P*.rotate90(*POINT q*)	returns *P* rotated by 90 degrees about *q*.
POLYGON	*P*.reflect(*POINT p*, *POINT q*)	returns *P* reflected across the straight line passing through *p* and *q*.
POLYGON	*P*.reflect(*POINT p*)	returns *P* reflected across point *p*.
RAT_TYPE	*P*.sqr_dist(*POINT p*)	returns the square of the minimal Euclidean distance between a segment in *P* and *p*. Returns zero if *P* is empty.
list<*POLYGON*>	*P*.simple_parts( )	returns the simple parts of *P* as a list of simple polygons. WHAT TO DO WITH IT.
POLYGON	*P*.complement( )	returns the complement of *P*.

The functions in the following group are only available for *polygons*. They have no counterpart for *rat_polygons*.

polygon	*P*.translate_by_angle(*double alpha*, *double d*)	returns *P* translated in direction *alpha* by distance *d*.
polygon	*P*.rotate(*point p*, *double alpha*)	returns *P* rotated by α degrees about *p*.
polygon	*P*.rotate(*double alpha*)	returns *P* rotated by α degrees about the origin.
double	*P*.distance(*point p*)	returns the Euclidean distance between *P* and *p*.

All functions below assume that P is weakly simple.

int	P.side_of(*POINT p*)	returns +1 if p lies to the left of P, 0 if p lies on P, and -1 if p lies to the right of P.
region_kind	P.region_of(*POINT p*)	returns BOUNDED_REGION if p lies in the bounded region of P, returns ON_REGION if p lies on P, and returns UNBOUNDED_REGION if p lies in the unbounded region.
bool	P.inside(*POINT p*)	returns true if p lies to the left of P, i.e., *side_of*$(p) == +1$.
bool	P.on_boundary(*POINT p*)	returns true if p lies on P, i.e., *side_of*$(p) == 0$.
bool	P.outside(*POINT p*)	returns true if p lies to the right of P, i.e., *side_of*$(p) == -1$.
bool	P.contains(*POINT p*)	returns true if p lies to the left of or on P.
RAT_TYPE	P.area()	returns the signed area of the bounded region of P. The sign of the area is positive if the bounded region is the positive side of P.
int	P.orientation()	returns the orientation of P.

Iterations Macros

forall_vertices(v, P) { "the vertices of P are successively assigned to rat_point v" }

forall_segments(s, P) { "the edges of P are successively assigned to rat_segment s" }

Non-Member Functions

POLYGON	reg_n_gon(*int n*, *CIRCLE C*, *double epsilon*)	generates a (nearly) regular n-gon whose vertices lie on the circle C. The i-th point is generated by $C.point\_of\_circle(2\pi i/n, epsilon)$. With the rational kernel the vertices of the n-gon are guaranteed to lie on the circle, with the floating point kernel they are only guaranteed to lie near C.

POLYGON n_gon(*int n*, *CIRCLE C*, *double epsilon*)

generates a (nearly) regular n-gon whose vertices lie near the circle C. For the flaoting point kernel the function is equivalent to the function above. For the rational kernel the function first generates a n-gon with floating point arithmetic and then converts the resulting *polygon* to a *rat_polygon*.

POLYGON hilbert(*int n*, *RAT_TYPE x1*, *RAT_TYPE y1*, *RAT_TYPE x2*, *RAT_TYPE y2*)

generates the Hilbert polygon of order n within the rectangle with boundary $(x1, y1)$ and $(x2, y2)$.
Precondition: $x1 < x2$ and $y1 < y2$.

11.7 Generalized Polygons (GEN_POLYGON)

1. Definition

There are two instantiations of GEN_POLYGON. The instantiation with the rational kernel is called *rat_gen_polygon* and requires inclusion of rat_gen_polygon.h and the instantiation with the floating point kernel is called *gen_polygon* and requires inclusion of gen_polygon.h.

An instance P of the data type *GEN_POLYGON* is a regular polygonal region in the plane. A regular region is an open set that is equal to the interior of its closure. A region is polygonal if its boundary consists of a finite number of line segments.

The boundary of a *GEN_POLYGON* consists of zero or more weakly simple closed polygonal chains. There are two regions whose boundary is empty, namely the empty region and the full region. The full region encompasses the entire plane. We call a region non-trivial if its boundary is non-empty. The boundary cycles P_1, P_2, ..., P_k of a *GEN_POLYGON* are ordered such that no P_j is nested in a P_i with $i < j$.

There are two instantiations of *GEN_POLYGONS*, namely *rat_gen_polygon* and *gen_polygon*. Only the former guarantee correct results. All operations for *rat_gen_polygon* are also available for *gen_polygon*. There is a small number of operations that are only available for *gen_polygon*.

A detailed discussion of polygons and generalized polygons can be found in the LEDA book.

The local enumeration type KIND consists of elements EMPTY, FULL, and NON_TRIVIAL.

#include < *LEDA/generic/GEN_POLYGON.h* >

2. Creation

GEN_POLYGON P(*KIND* k = *GEN_POLYGON_REP*::*EMPTY*);

introduces a variable P of type *GEN_POLYGON*. P is initialized to the empty polygon if k is EMPTY and to the full polygon if k is FULL.

GEN_POLYGON *P*(*POLYGON p*, *CHECK_TYPE check* = *WEAKLY_SIMPLE*, *RESPECT_TYPE respect_orientation* = *RESPECT_ORIENTATION*);

introduces a variable *P* of type *GEN_POLYGON*. *P* is initialized to the polygonal region with boundary *p*. If *respect_orientation* is DISREGARD_ORIENTATION, the orientation is chosen such *P* is bounded.
Precondition: *p* must be a weakly simple polygon. If *check* is set appropriately this is checked.

GEN_POLYGON *P*(*list*<*POINT*> *pl*, *CHECK_TYPE check* = *GEN_POLYGON*::*SIMPLE*, *RESPECT_TYPE respect_orientation* = *RESPECT_ORIENTATION*);

introduces a variable *P* of type *GEN_POLYGON*. *P* is initialized to the polygon with vertex sequence *pl*. If *respect_orientation* is DISREGARD_ORIENTATION, the orientation is chosen such that *P* is bounded.
Precondition: If *check* is SIMPLE, *pl* must define a simple polygon, and if *check* is WEAKLY_SIMPLE, *pl* must define a weakly simple polygon. If no test is to performed, the second argument has to be set to NO_CHECK. The three constants NO_CHECK, SIMPLE, and WEAKLY_SIMPLE are part of a local enumeration type CHECK_TYPE.

GEN_POLYGON *P*(*list*<*POLYGON*> *PL*, *CHECK_TYPE check* = *CHECK_REP*);

introduces a variable *P* of type *GEN_POLYGON*. *P* is initialized to the polygon with boundary representation *PL*.
Precondition: *PL* must be a boundary representation. This conditions is checked if *check* is set to CHECK_REP.

GEN_POLYGON *P*(*gen_polygon Q*, *int prec* = *rat_point*::*default_precision*);

introduces a variable *P* of type *GEN_POLYGON*. *P* is initialized to a rational approximation of the (floating point) polygon *Q* of coordinates with denominator at most *prec*. If *prec* is zero, the implementation chooses *prec* large enough such that there is no loss of precision in the conversion

3. Operations

bool	*P*.empty( )	returns true if *P* is empty, false otherwise.
bool	*P*.full( )	returns true if *P* is the entire plane, false otherwise.
bool	*P*.trivial( )	returns true if *P* is either empty or full, false otherwise.

KIND	P.kind( )	returns the kind of P.
gen_polygon	P.to_float( )	returns a floating point approximation of P.
bool	P.is_simple( )	returns true if the polygonal region is simple, i.e., if the graph defined by the segments in the boundary of P has only vertices of degree two.
bool	P.check_representation( )	tests whether the representation of P is OK. This test is partial.
void	P.canonical_rep( )	NOT IMPLEMENTED YET.
list<*POINT*>	P.vertices( )	returns the concatenated vertex lists of all polygons in the boundary representation of P.
list<*SEGMENT*>	P.edges( )	returns the concatenated edge lists of all polygons in the boundary representation of P.
list<*POLYGON*>	P.polygons( )	returns the lists of all polygons in the boundary representation of P.
list<*POINT*>	P.intersection(*SEGMENT* s)	returns the list of all proper intersections between s and the boundary of P.
list<*POINT*>	P.intersection(*LINE* l)	returns the list of all proper intersections between l and the boundary of P.
int	P.size( )	returns the number of segments in the boundary of P.
GEN_POLYGON	P.translate(*RAT_TYPE* dx, *RAT_TYPE* dy)	returns P translated by vector (dx, dy).
GEN_POLYGON	P.translate(*INT_TYPE* dx, *INT_TYPE* dy, *INT_TYPE* dw)	returns P translated by vector $(dx/dw, dy/dw)$.
GEN_POLYGON	P.translate(*VECTOR* v)	returns P translated by vector v.
GEN_POLYGON	P + *VECTOR* v	returns P translated by vector v.
GEN_POLYGON	P − *VECTOR* v	returns P translated by vector $-v$.
GEN_POLYGON	P.rotate90(*POINT* q)	returns P rotated by 90 degrees about q.

GEN_POLYGON	P.reflect(*POINT* p, *POINT* q)	returns P reflected across the straight line passing through p and q.
GEN_POLYGON	P.reflect(*POINT* p)	returns P reflected across point p.
RAT_TYPE	P.sqr_dist(*POINT* p)	returns the square of the minimal Euclidean distance between a segment in the boundary of P and p. Returns zero is P is trivial.
( )	returns the simple parts of P as a list of simple polygons. WAS MACH ICH DAMIT.	
GEN_POLYGON	P.complement( )	returns the complement of P.
int	P.side_of(*POINT* p)	returns $+1$ if p lies to the left of P, 0 if p lies on P, and -1 if p lies to the right of P.
region_kind	P.region_of(*POINT* p)	returns BOUNDED_REGION if p lies in the bounded region of P, returns ON_REGION if p lies on P, and returns UNBOUNDED_REGION if p lies in the unbounded region. The bounded region of the full polygon is the entire plane.
bool	P.inside(*POINT* p)	returns true if p lies to the left of P, i.e., $side\_of(p) == +1$.
bool	P.on_boundary(*POINT* p)	returns true if p lies on P, i.e., $side\_of(p) == 0$.
bool	P.outside(*POINT* p)	returns true if p lies to the right of P, i.e., $side\_of(p) == -1$.
bool	P.contains(*POINT* p)	returns true if p lies to the left of or on P.
RAT_TYPE	P.area( )	returns the signed area of the bounded region of P. The sign of the area is positive if the bounded region is the positive side of P. *Precondition*: P is not the full polygon.

All binary boolean operations are regularized, i.e., the result R of the standard boolean operation is replaced by the interior of the closure of R. We use reg X to denote the regularization of a set X.

GEN_POLYGON	P.unite(*GEN_POLYGON* Q)	returns $\text{reg}(P \cup Q)$.

GEN_POLYGON	*P*.intersection(*GEN_POLYGON Q*)	
		returns $\text{reg}(P \cap Q)$.
GEN_POLYGON	*P*.diff(*GEN_POLYGON Q*)	
		returns $\text{reg}(P \setminus Q)$.
GEN_POLYGON	*P*.sym_diff(*GEN_POLYGON Q*)	
		returns $\text{reg}((P \cup Q) - (P \cap Q))$.

The following functions are only available for *gen_polygons*. They have no counterpart for *rat_gen_polygons*.

gen_polygon	*P*.translate_by_angle(*double alpha*, *double d*)	
		returns P translated in direction *alpha* by distance d.
gen_polygon	*P*.rotate(*point p*, *double alpha*)	
		return P rotated by α degrees about p.
gen_polygon	*P*.rotate(*double alpha*)	return P rotated by α degrees about the origin.
double	*P*.distance(*point p*)	returns the Euclidean distance between P and p.

Iterations Macros

forall_polygons(p, P) { "the boundary polygons of P are successively assigned to POLYGON p" }

11.8 Rational Points (rat_point)

1. Definition

An instance of data type *rat_point* is a point with rational coordinates in the two-dimensional plane. A point with cartesian coordinates (a, b) is represented by homogeneous coordinates (x, y, w) of arbitrary length integers (see 4.1) such that $a = x/w$ and $b = y/w$ and $w > 0$.

#include < *LEDA/rat_point.h* >

2. Types

rat_point:: *coord_type* — the coordinate type (*rational*).

rat_point:: *point_type* — the point type (*rat_point*).

rat_point:: *float_type* — the corresponding floating-point type (*point*).

3. Creation

rat_point *p*; — introduces a variable p of type *rat_point* initialized to the point $(0, 0)$.

rat_point *p*(*rational a*, *rational b*);

introduces a variable p of type *rat_point* initialized to the point (a, b).

rat_point *p*(*int a*, *int b*); — introduces a variable p of type *rat_point* initialized to the point (a, b).

rat_point *p*(*integer x*, *integer y*, *integer w*);

introduces a variable p of type *rat_point* initialized to the point with homogeneous coordinates (x, y, w) if $w > 0$ and to point $(-x, -y, -w)$ if $w < 0$.
Precondition: $w \neq 0$.

rat_point *p*(*rat_vector v*); — introduces a variable p of type *rat_point* initialized to the point $(v[0], v[1])$.
Precondition: : *v.dim*( ) = 2.

rat_point *p*(*point p*, *int prec* = *rat_point*:: *default_precision*);

introduces a variable p of type *rat_point* initialized to the point with homogeneous coordinates $(\lfloor P*x \rfloor, \lfloor P*x \rfloor, P)$, where $p = (x, y)$ and $P = 2^{prec}$. If *prec* is non-positive, the conversion is without loss of precision, i.e., *prec* is chosen as a sufficiently large power of two such that $P * x$ and $P * y$ are *integers*.

4. Operations

point	p.to_float( )	returns a floating point approximation of p.
rat_vector	p.to_vector( )	returns the vector extending from the origin to p.
integer	p.X( )	returns the first homogeneous coordinate of p.
integer	p.Y( )	returns the second homogeneous coordinate of p.
integer	p.W( )	returns the third homogeneous coordinate of p.
double	p.XD( )	returns a floating point approximation of $p.X(\ )$.
double	p.YD( )	returns a floating point approximation of $p.Y(\ )$.
double	p.WD( )	returns a floating point approximation of $p.W(\ )$.
rational	p.xcoord( )	returns the x-coordinate of p.
rational	p.ycoord( )	returns the y-coordinate of p.
double	p.xcoordD( )	returns a floating point approximation of $p.xcoord(\ )$.
double	p.ycoordD( )	returns a floating point approximation of $p.ycoord(\ )$.
rat_point	p.rotate90(*rat_point* q)	returns p rotated by 90 degrees about q.
rat_point	p.rotate90( )	returns p rotated by 90 degrees about the origin.
rat_point	p.reflect(*rat_point* p, *rat_point* q)	returns p reflected across the straight line passing through p and q. *Precondition*: $p \neq q$.
rat_point	p.reflect(*rat_point* q)	returns p reflected across point q.
rat_point	p.translate(*rational* dx, *rational* dy)	returns p translated by vector (dx, dy).
rat_point	p.translate(*integer* dx, *integer* dy, *integer* dw)	returns p translated by vector $(dx/dw, dy/dw)$.
rat_point	p.translate(*rat_vector* v)	returns $p + v$, i.e., p translated by vector v. *Precondition*: v.dim() $= 2$.

rat_point	$p + $ *rat_vector* v	returns p translated by vector v.
rat_point	$p - $ *rat_vector* v	returns p translated by vector $-v$.

rational p.sqr_dist(*rat_point* q)

returns the squared distance between p and q.

int p.cmp_dist(*rat_point* q, *rat_point* r)

returns $compare(p.sqr\_dist(q), p.sqr\_dist(r))$.

rational p.xdist(*rat_point* q)

returns the horizontal distance between p and q.

rational p.ydist(*rat_point* q)

returns the vertical distance between p and q.

rat_vector	$p - q$	returns the difference vector of the coordinates.
ostream&	*ostream&* $O \ll p$	writes the homogeneous coordinates (x, y, w) of p to output stream O.

istream& *istream&* $I \gg$ *rat_point&* p

reads the homogeneous coordinates (x, y, w) of p from input stream I.

Non-Member Functions

int orientation(*rat_point* a, *rat_point* b, *rat_point* c)

computes the orientation of points a, b, c as the sign of the determinant

$$\begin{vmatrix} a_x & a_y & a_w \\ b_x & b_y & b_w \\ c_x & c_y & c_w \end{vmatrix}$$

i.e., it returns $+1$ if point c lies left of the directed line through a and b, 0 if a,b, and c are collinear, and -1 otherwise.

int cmp_signed_dist(*rat_point* a, *rat_point* b, *rat_point* c, *rat_point* d)

compares (signed) distances of c and d to the straight line passing through a and b (directed from a to b). Returns $+1$ (-1)if c has larger (smaller) distance than d and 0 if distances are equal.

int cmp_distances(*rat_point* $p1$, *rat_point* $p2$, *rat_point* $p3$, *rat_point* $p4$)

compares the distances $(p1, p2)$ and $(p3, p4)$. Returns $+1$ (-1) if distance $(p1, p2)$ is larger (smaller) than distance $(p3, p4)$, otherwise 0.

rat_point midpoint(*rat_point a*, *rat_point b*)

returns the midpoint of a and b.

rational area(*rat_point a*, *rat_point b*, *rat_point c*)

computes the signed area of the triangle determined by a,b,c, positive if $orientation(a, b, c) > 0$ and negative otherwise.

bool collinear(*rat_point a*, *rat_point b*, *rat_point c*)

returns true if points a, b, c are collinear, i.e., $orientation(a, b, c) = 0$, and false otherwise.

bool right_turn(*rat_point a*, *rat_point b*, *rat_point c*)

returns true if points a, b, c form a righ turn, i.e., $orientation(a, b, c) < 0$, and false otherwise.

bool left_turn(*rat_point a*, *rat_point b*, *rat_point c*)

returns true if points a, b, c form a left turn, i.e., $orientation(a, b, c) > 0$, and false otherwise.

int side_of_halfspace(*rat_point a*, *rat_point b*, *rat_point c*)

returns the sign of the scalar product $(b-a)\cdot(c-a)$. If $b \neq a$ this amounts to: Let h be the open halfspace orthogonal to the vector $b-a$, containing b, and having a in its boundary. Returns $+1$ if c is contained in h, returns 0 is c lies on the the boundary of h, and returns -1 is c is contained in the interior of the complement of h.

int side_of_circle(*rat_point a*, *rat_point b*, *rat_point c*, *rat_point d*)

returns $+1$ if point d lies left of the directed circle through points a, b, and c, 0 if a,b,c,and d are cocircular, and -1 otherwise.

bool incircle(*rat_point a*, *rat_point b*, *rat_point c*, *rat_point d*)

returns $true$ if point d lies in the interior of the circle through points a, b, and c, and $false$ otherwise.

bool outcircle(*rat_point a*, *rat_point b*, *rat_point c*, *rat_point d*)

returns $true$ if point d lies outside of the circle through points a, b, and c, and $false$ otherwise.

bool on_circle(*rat_point a*, *rat_point b*, *rat_point c*, *rat_point d*)

returns $true$ if points a, b, c, and d are corcircular.

bool cocircular(*rat_point a*, *rat_point b*, *rat_point c*, *rat_point d*)

returns $true$ if points a, b, c, and d are corcircular.

bool affinely_independent(*array*<*rat_point*> *A*)

decides whether the points in A are affinely independent.

bool contained_in_simplex(*array*<*rat_point*> *A*, *rat_point* *p*)

determines whether p is contained in the simplex spanned by the points in A. A may consists of up to 3 points.
Precondition: The points in A are affinely independent.

bool contained_in_affine_hull(*array*<*rat_point*> *A*, *rat_point* *p*)

determines whether p is contained in the affine hull of the points in A.

11.9 Rational Segments (rat_segment)

1. Definition

An instance s of the data type *rat_segment* is a directed straight line segment in the two-dimensional plane, i.e., a line segment $[p, q]$ connecting two rational points p and q (cf. 11.8). p is called the *source* or start point and q is called the *target* or end point of s. A segment is called *trivial* if its source is equal to its target.

#include < *LEDA/rat_segment.h* >

2. Types

rat_segment :: *coord_type*	the coordinate type (*rational*).
rat_segment :: *point_type*	the point type (*rat_point*).
rat_segment :: *float_type*	the corresponding floatin-point type (*segment*).

3. Creation

rat_segment s;

introduces a variable s of type *rat_segment*. s is initialized to the empty segment.

rat_segment s(*rat_point* p, *rat_point* q);

introduces a variable s of type *rat_segment*. s is initialized to the segment $[p, q]$.

rat_segment s(*rat_point* p, *rat_vector* v);

introduces a variable s of type *rat_segment*. s is initialized to the segment $[p, p + v]$.
Precondition: $v.dim() = 2$.

rat_segment s(*rational x1*, *rational y1*, *rational x2*, *rational y2*);

introduces a variable s of type *rat_segment*. s is initialized to the segment $[(x1, y1), (x2, y2)]$.

rat_segment s(*integer x1*, *integer y1*, *integer w1*, *integer x2*, *integer y2*, *integer w2*);

introduces a variable s of type *rat_segment*. s is initialized to the segment $[(x1, y1, w1), (x2, y2, w2)]$.

rat_segment s(*integer x1*, *integer y1*, *integer x2*, *integer y2*);

introduces a variable s of type *rat_segment*. s is initialized to the segment $[(x1, y1), (x2, y2)]$.

rat_segment s(*segment s*, *int prec* = *rat_point*:: *default_precision*);

introduces a variable s of type *rat_segment*. s is initialized to the segment obtained by approximating the two defining points of s.

4. Operations

segment	s.to_float( )	returns a floating point approximation of s.
rat_point	s.start( )	returns the source point of s.
rat_point	s.end( )	returns the target point of s.
rat_segment	s.reversal( )	returns the segment (*target*( ), *source*( )).
rational	s.xcoord1( )	returns the x-coordinate of the source point of s.
rational	s.xcoord2( )	returns the x-coordinate of the target point of s.
rational	s.ycoord1( )	returns the y-coordinate of the source point of s.
rational	s.ycoord2( )	returns the y-coordinate of the target point of s.
double	s.xcoord1D( )	returns a double precision approximation of *s.xcoord1*( ).
double	s.xcoord2D( )	returns a double precision approximation of *s.xcoord2*( ).
double	s.ycoord1D( )	returns a double precision approximation of *s.ycoord1*( ).
double	s.ycoord2D( )	returns a double precision approximation of *s.ycoord2*( ).
integer	s.X1( )	returns the first homogeneous coordinate of the source point of s.
integer	s.X2( )	returns the first homogeneous coordinate of the target point of s.
integer	s.Y1( )	returns the second homogeneous coordinate of the source point of s.
integer	s.Y2( )	returns the second homogeneous coordinate of the target point of s.
integer	s.W1( )	returns the third homogeneous coordinate of the source point of s.
integer	s.W2( )	returns the third homogeneous coordinate of the target point of s.
double	s.XD1( )	returns a floating point approximation of *s.X1*( ).

double *s*.XD2( ) returns a floating point approximation of *s.X2*( ).

double *s*.YD1( ) returns a floating point approximation of *s.Y1*( ).

double *s*.YD2( ) returns a floating point approximation of *s.Y2*( ).

double *s*.WD1( ) returns a floating point approximation of *s.W1*( ).

double *s*.WD2( ) returns a floating point approximation of *s.W2*( ).

integer *s*.dx( ) returns the normalized x-difference $X2 \cdot W1 - X1 \cdot W2$ of s.

integer *s*.dy( ) returns the normalized y-difference $Y2 \cdot W1 - Y1 \cdot W2$ of s.

double *s*.dxD( ) returns a floating point approximation of *s.dx*( ).

double *s*.dyD( ) returns a floating point approximation of *s.dy*( ).

bool *s*.is_trivial( ) returns true if s is trivial.

bool *s*.is_vertical( ) returns true if s is vertical.
Precondition: s is non-trivial.

bool *s*.is_horizontal( ) returns true if s is horizontal.
Precondition: s is non-trivial.

int *s*.cmp_slope(*rat_segment s1*)
compares the slopes of s and s_1.
Precondition: s and s_1 are non-trivial.

bool *s*.contains(*rat_point p*)
decides whether s contains p.

bool *s*.intersection(*rat_segment t*)
decides whether s and t intersect.

bool *s*.intersection(*rat_segment t*, *rat_point*& *p*)
decides whether s and t intersect. If so, some point of intersection is assigned to p.

bool *s*.intersection(*rat_segment t*, *rat_segment*& *inter*)
decides whether s and t intersect. If so, the segment formed by the points of intersection is assigned to *inter*.

bool *s*.intersection_of_lines(*rat_segment t*, *rat_point*& *p*)
decides if the lines supporting s and t intersect in a single point. If so, the point of intersection is assigned to p.
Precondition: s and t are nontrivial.

rat_segment	*s*.translate(*rational dx*, *rational dy*)	
		returns s translated by vector (dx, dy).
rat_segment	*s*.translate(*integer dx*, *integer dy*, *integer dw*)	
		returns s translated by vector $(dx/dw, dy/dw)$.
rat_segment	*s*.translate(*rat_vector v*)	
		returns $s + v$, i.e., s translated by vector v. *Precondition*: v.dim() = 2.
rat_segment	s + *rat_vector v*	returns s translated by vector v.
rat_segment	s − *rat_vector v*	returns s translated by vector $-v$.
rat_segment	*s*.rotate90(*rat_point q*)	
		returns s rotated by an angle of 90 degrees about point q.
rat_segment	*s*.rotate90( )	returns *s.rotate90*(*s.source*( )).
rat_segment	*s*.reflect(*rat_point p*, *rat_point q*)	
		returns s reflected across the straight line passing through p and q.
rat_segment	*s*.reflect(*rat_point p*)	
		returns s reflected across point p.
rat_segment	*s*.reverse( )	returns s reversed.
rat_segment	*s*.perpendicular(*rat_point p*)	
		returns the segment perpendicular to s with source p and target on *line*(s). *Precondition*: s is nontrivial.
rational	*s*.sqr_length( )	returns the square of the length of s.
rational	*s*.sqr_dist(*rat_point p*)	
		returns the squared Euclidean distance between p and s.
rational	*s*.sqr_dist( )	returns the squared distance between s and the origin.
rat_vector	*s*.to_vector( )	returns the vector *s.target*( ) − *s.source*( ).
int	$s == t$	returns true if s and t are equal as oriented segments
int	equal_as_sets(*rat_segment s*, *rat_segment t*)	
		returns true if s and t are equal as unoriented segments

ostream& *ostream& O* $\ll$ *s* writes `[s.source() === s.target()]` to output stream O.

istream& *istream& I* $\gg$ *rat_segment& s*

reads `{[} p {===} q {]}` from input stream I.

Non-Member Functions

int orientation(*rat_segment s*, *rat_point p*)

computes orientation(a, b, p), where $a \neq b$ and a and b appear in this order on segment s.

int cmp_slopes(*rat_segment s1*, *rat_segment s2*)

returns compare(slope(s_1), slope(s_2)).

int cmp_segments_at_xcoord(*rat_segment s1*, *rat_segment s2*, *rat_point p*)

compares points $l_1 \cap v$ and $l_2 \cap v$ where l_i is the line underlying segment s_i and v is the vertical straight line passing through point p.

11.10 Rational Rays (rat_ray)

1. Definition

An instance r of the data type *rat_ray* is a directed straight ray defined by two points with rational coordinates in the two-dimensional plane.

#include < *LEDA/rat_ray.h* >

2. Types

rat_ray :: *coord_type*	the coordinate type (*rational*).
rat_ray :: *point_type*	the point type (*rat_point*).
rat_ray :: *float_type*	the corresponding floatin-point type (*ray*).

3. Creation

rat_ray r(*rat_point* p, *rat_point* q);

introduces a variable r of type *rat_ray*. r is initialized to the ray starting at point p and passing through point q.
Precondition: $p \neq q$.

rat_ray r(*rat_segment* s); introduces a variable r of type *rat_ray*. r is initialized to the ($rat\_ray(s.source(), s.target())$).
Precondition: s is nontrivial.

rat_ray r; introduces a variable r of type *rat_ray*.

rat_ray r(*ray* r, *int prec* = *rat_point* :: *default_precision*);

introduces a variable r of type *rat_ray*. r is initialized to the ray obtained by approximating the two defining points of r.

4. Operations

ray	r.to_float( )	returns a floating point approximation of r.
rat_point	r.source( )	returns the source of r.
rat_point	r.point1( )	returns the source of r.
rat_point	r.point2( )	returns a point on r different from $r.source(\,)$.
bool	r.is_vertical( )	returns true iff r is vertical.
bool	r.is_horizontal( )	returns true iff r is horizontal.

bool r.intersection(*rat_ray* s, *rat_point*& *inter*)
returns true if r and s intersect. If so, a point of intersection is returned in *inter*.

bool r.intersection(*rat_segment* s, *rat_point*& *inter*)
returns true if r and s intersect. If so, a point of intersection is returned in *inter*.

rat_ray r.translate(*rational* dx, *rational* dy)
returns r translated by vector (dx, dy).

rat_ray r.translate(*integer* dx, *integer* dy, *integer* dw)
returns r translated by vector $(dx/dw, dy/dw)$.

rat_ray r.translate(*rat_vector* v) returns $r + v$, i.e., r translated by vector v.
Precondition: v.dim() $= 2$.

rat_ray r + *rat_vector* v returns r translated by vector v.

rat_ray r − *rat_vector* v returns r translated by vector $-v$.

rat_ray r.rotate90(*rat_point* q) returns r rotated about q by an angle of 90 degrees.

rat_ray r.reflect(*rat_point* p, *rat_point* q)
returns r reflected across the straight line passing through p and q.
Precondition: $p \neq q$.

rat_ray r.reflect(*rat_point* p) returns r reflected across point p.

rat_ray r.reverse( ) returns r reversed.

bool r.contains(*rat_point* p) decides whether r contains p.

bool r.contains(*rat_segment* s)
decides whether r contains s.

Non-Member Functions

int orientation(*rat_ray* r, *rat_point* p)
computes orientation(a, b, p), where $a \neq b$ and a and b appear in this order on ray r.

int cmp_slopes(*rat_ray* *r1*, *rat_ray* *r2*)
returns compare(slope(r_1), slope(r_2)).

11.11 Straight Rational Lines (rat_line)

1. Definition

An instance l of the data type *rat_line* is a directed straight line in the two-dimensional plane.

#include < *LEDA/rat_line.h* >

2. Types

rat_line :: *coord_type*	the coordinate type (*rational*).
rat_line :: *point_type*	the point type (*rat_point*).
rat_line :: *float_type*	the corresponding floatin-point type (*line*).

3. Creation

rat_line l(*rat_point* p, *rat_point* q);

introduces a variable l of type *rat_line*. l is initialized to the line passing through points p and q directed form p to q.
Precondition: $p \neq q$.

rat_line l(*rat_segment* s); introduces a variable l of type *rat_line*. l is initialized to the line supporting segment s.
Precondition: s is nontrivial.

rat_line l(*rat_point* p, *rat_vector* v);

introduces a variable l of type *rat_line*. l is initialized to the line passing through points p and $p + v$.
Precondition: v is a nonzero vector.

rat_line l(*rat_ray* r); introduces a variable l of type *rat_line*. l is initialized to the line supporting ray r.

rat_line l; introduces a variable l of type *rat_line*.

rat_line l(*line* l, *int prec* = *rat_point* :: *default_precision*);

introduces a variable l of type *rat_line*. l is initialized to the line obtained by approximating the two defining points of l.

4. Operations

line	l.to_float( )	returns a floating point approximation of l.
rat_point	l.point1( )	returns a point on l.

rat_point	l.point2( )	returns a second point on l.
rat_segment	l.seg( )	returns a segment on l.
bool	l.is_vertical( )	decides whether l is vertical.
bool	l.is_horizontal( )	decides whether l is horizontal.
bool	l.intersection(*rat_line* g, *rat_point*& *inter*)	returns true if l and g intersect. In case of intersection a common point is returned in *inter*.
bool	l.intersection(*rat_segment* s, *rat_point*& *inter*)	returns true if l and s intersect. In case of intersection a common point is returned in *inter*.
rat_line	l.translate(*rational* dx, *rational* dy)	returns l translated by vector (dx, dy).
rat_line	l.translate(*integer* dx, *integer* dy, *integer* dw)	returns l translated by vector $(dx/dw, dy/dw)$.
rat_line	l.translate(*rat_vector* v)	returns l translated by vector v. *Precondition*: v.dim() = 2.
rat_line	l + *rat_vector* v	returns l translated by vector v.
rat_line	l − *rat_vector* v	returns l translated by vector $-v$.
rat_line	l.rotate90(*rat_point* q)	returns l rotated about q by an angle of 90 degrees.
rat_line	l.reflect(*rat_point* p, *rat_point* q)	returns l reflected across the straight line passing through p and q.
rat_line	l.reverse( )	returns l reversed.
rational	l.sqr_dist(*rat_point* q)	returns the square of the distance between l and q.
rat_segment	l.perpendicular(*rat_point* p)	returns the segment perpendicular to l with source p and target on l.
rat_point	l.dual( )	returns the point dual to l. /precond l is not vertical.
int	l.side_of(*rat_point* p)	computes orientation(a, b, p), where $a \neq b$ and a and b appear in this order on line l.

bool	*l*.contains(*rat_point p*)	returns true if p lies on l.
bool	*l*.clip(*rat_point p*, *rat_point q*, *rat_segment& s*)	
		clips l at the rectangle R defined by p and q. Returns true if the intersection of R and l is non-empty and returns false otherwise. If the intersection is non-empty the intersection is assigned to s; It is guaranteed that the source node of s is no larger than its target node.
bool	$l == g$	returns true if the l and g are equal as oriented lines.
bool	equal_as_sets(*rat_line l*, *rat_line g*)	
		returns true if the l and g are equal as unoriented lines.

Non-Member Functions

int	orientation(*rat_line l*, *rat_point p*)	
		computes orientation(a, b, p), where $a \neq b$ and a and b appear in this order on line l.
int	cmp_slopes(*rat_line l1*, *rat_line l2*)	
		returns compare(slope(l_1), slope(l_2)).
rat_line	p_bisector(*rat_point p*, *rat_point q*)	
		returns the perpendicular bisector of p and q. The bisector has p on its left. *Precondition*: $p \neq q$.

11.12 Rational Circles (rat_circle)

1. Definition

An instance C of data type *rat_circle* is an oriented circle in the plane. A circle is defined by three points p_1, p_2, p_3 with rational coordinates (*rat_points*). The orientation of C is equal to the orientation of the three defining points, i.e., $orientation(p_1, p_2, p_3)$. Positive orientation corresponds to counterclockwise orientation and negative orientation corresponds to clockwise orientation.

Some triples of points are unsuitable for defining a circle. A triple is *admissable* if $|\{p_1, p_2, p_3\}| \neq 2$. Assume now that p_1, p_2, p_3 are admissable. If $|\{p_1, p_2, p_3\}| = 1$ they define the circle with center p_1 and radius zero. If p_1, p_2, and p_3 are collinear C is a straight line passing through p_1, p_2 and p_3 in this order and the center of C is undefined. If p_1, p_2, and p_3 are not collinear, C is the circle passing through them.

#include < *LEDA/rat_circle.h* >

2. Types

rat_circle :: *coord_type* — the coordinate type (*rational*).

rat_circle :: *point_type* — the point type (*rat_point*).

rat_circle :: *float_type* — the corresponding floatin-point type (*circle*).

3. Creation

rat_circle C(*rat_point* a, *rat_point* b, *rat_point* c);

introduces a variable C of type *rat_circle*. C is initialized to the circle through points a, b, and c.
Precondition: a, b, and c are admissable.

rat_circle C(*rat_point* a, *rat_point* b);

introduces a variable C of type *circle*. C is initialized to the counter-clockwise oriented circle with center a passing through b.

rat_circle C(*rat_point* a); introduces a variable C of type *circle*. C is initialized to the trivial circle with center a.

rat_circle C; introduces a variable C of type *rat_circle*. C is initialized to the trivial circle centered at $(0, 0)$.

rat_circle C(*circle* c, *int* *prec* = *rat_point* :: *default_precision*);

introduces a variable C of type *rat_circle*. C is initialized to the circle obtained by approximating threee defining points of c.

4. Operations

circle	C.to_float( )	returns a floating point approximation of C.
int	C.orientation( )	returns the orientation of C.
rat_point	C.center( )	returns the center of C. *Precondition*: C has a center, i.e., it not a line.
rat_point	C.point1( )	returns p_1.
rat_point	C.point2( )	returns p_2.
rat_point	C.point3( )	returns p_3.
rational	C.sqr_radius( )	returns the square of the radius of C.
rat_point	C.point_on_circle(*double alpha*, *double epsilon*)	returns a point p on C such that the angle of p differs from alpha by at most *epsilon*.
bool	C.is_degenerate( )	returns true if the defining points are collinear.
bool	C.is_trivial( )	returns true if C has radius zero.
bool	C.is_line( )	returns true if C is a line.
int	C.side_of(*rat_point* p)	returns -1, $+1$, or 0 if p lies right of, left of, or on C respectively.
bool	C.inside(*rat_point* p)	returns true if p lies inside C, false otherwise.
bool	C.outside(*rat_point* p)	returns true if p lies outside C.
bool	C.contains(*rat_point* p)	returns true if p lies on C, false otherwise.
rat_circle	C.translate(*rational dx*, *rational dy*)	returns C translated by vector (dx, dy).
rat_circle	C.translate(*integer dx*, *integer dy*, *integer dw*)	returns C translated by vector $(dx/dw, dy/dw)$.
rat_circle	C.translate(*rat_vector* v)	returns C translated by vector v.
rat_circle	C + *rat_vector* v	returns C translated by vector v.
rat_circle	C − *rat_vector* v	returns C translated by vector $-v$.

rat_circle	C.rotate90(*rat_point* q)	returns C rotated about q by an angle of 90 degrees.
rat_circle	C.reflect(*rat_point* p, *rat_point* q)	returns C reflected across the straight line passing through p and q.
rat_circle	C.reflect(*rat_point* p)	returns C reflected across point p.
rat_circle	C.reverse( )	returns C reversed.
bool	$C == D$	returns true if C and D are equal as oriented circles
bool	C.equal_as_sets(*rat_circle* $C1$, *rat_circle* $C2$)	returns true if $C1$ and $C2$ are equal as unoriented circles
ostream&	*ostream& out* $\ll$ c	writes the three defining points
istream&	*istream& in* $\gg$ *rat_circle&* c	reads three points and assigns the circle defined by them to c.

11.13 Geometry Algorithms (geo_alg)

All functions listed in this section work for geometric objects based on both floating-point and exact (rational) arithmetic. In particular, *point* can be replace by *rat_point*, *segment* by *rat_segment*, and *circle* by *rat_circle*.

The floating point versions are faster but unreliable. They may produce incorrect results, abort, or run forever. Only the rational versions will produce correct results for all inputs.

The include-file for the rational version is rat_geo_alg.h, the include-file for the floating point version is float_geo_alg.h, and geo_alg.h includes both versions. Including both versions increases compile time. An alternative name for geo_alg.h is plane_alg.h.

- **Convex Hulls**

list<point> CONVEX_HULL(*list<point> L*)

CONVEX_HULL takes as argument a list of points and returns the polygon representing the convex hull of L. The cyclic order of the vertices in the result list corresponds to clockwise order of the vertices on the hull. The algorithm calls our current favorite of the algorithms below.

polygon CONVEX_HULL_POLY(*list<point> L*)

as above, but returns the convex hull of L as a polygon.

list<point> UPPER_CONVEX_HULL(*list<point> L*)

returns the upper convex hull of L.

list<point> LOWER_CONVEX_HULL(*list<point> L*)

returns the lower convex hull of L.

list<point> CONVEX_HULL_S(*list<point> L*)

as above, but the algorithm is based on the sweep paradigm. Running time is $O(n \log n)$ in the worst and in the best case.

list<point> CONVEX_HULL_IC(*list<point> L*)

as above, but the algorithm is based on incremental construction. The running time is $O(n^2)$ worst case and is $O(n \log n)$ expected case. The expectation is computed as the average over all permutations of L. The running time is linear in the best case.

list<point> CONVEX_HULL_RIC(*list<point> L*)

as above. The algorithm permutes L randomly and then calls the preceding function.

double WIDTH(*list*<*point*> L, *line*& $l1$, *line*& $l2$)

returns the square of the minimum width of a stripe covering all points in L and the two boundaries of the stripe.
Precondition: L is non-empty

• **Triangulations**

edge TRIANGULATE_POINTS(*list*<*point*> L, *GRAPH*<*point*, *int*>& T)

computes a triangulation (planar map) T of the points in L and returns an edge of the outer face (convex hull).

void DELAUNAY_TRIANG(*list*<*point*> L, *GRAPH*<*point*, *int*>& DT)

computes the delaunay triangulation DT of the points in L.

void DELAUNAY_DIAGRAM(*list*<*point*> L, *GRAPH*<*point*, *int*>& DD)

computes the delaunay diagram DD of the points in L.

void F_DELAUNAY_TRIANG(*list*<*point*> L, *GRAPH*<*point*, *int*>& FDT)

computes the furthest point delaunay triangulation FDT of the points in L.

void F_DELAUNAY_DIAGRAM(*list*<*point*> L, *GRAPH*<*point*, *int*>& FDD)

computes the furthest point delaunay diagram FDD of the points in L.

• **Constraint Triangulations**

edge TRIANGULATE_SEGMENTS(*list*<*segment*> L, *GRAPH*<*point*, *segment*>& G)

computes a constrained triangulation (planar map) T of the segments in L (trivial segments representing points are allowed). The function returns an edge of the outer face (convex hull).

edge DELAUNAY_TRIANG(*list*<*segment*> L, *GRAPH*<*point*, *segment*>& G)

computes a constrained Delaunay triangulation T of the segments in L. The function returns an edge of the outer face (convex hull).

edge TRIANGULATE_PLANE_MAP(*GRAPH*<*point*, *segment*>& G)

computes a constrained triangulation T of the plane map (counterclockwise straight-line embedded Graph) G. The function returns an edge of the outer face (convex hull). *Precondition*: G is simple.

edge DELAUNAY_TRIANG(*GRAPH*<*point*, *segment*>& G)

computes a constrained Delaunay triangulation T of the plane map G. The function returns an edge of the outer face (convex hull). *Precondition*: G is simple.

void TRIANGULATE_POLYGON(*list*<*point*> L, *GRAPH*<*point*, *segment*>& G)

triangulates the interior of the simple polygon with vertex sequence L. *Precondition*: The polygon spanned by the points in L is weakly simple.

edge TRIANGULATE_POLYGON(*list*<*point*> L, *GRAPH*<*point*, *segment*>& G, *list*<*edge*>& *inner_edges*, *list*<*edge*>& *outer_edges*)

triangulates the interior and exterior of the simple polygon with vertex sequence L and stores all edges of the inner (outer) triangulation in *inner_edges* (*outer_edges*). The function returns an edge of the outer face (convex hull) of the polygon. *Precondition*: The polygon spanned by the points of L is weakly simple.

- **Euclidean Spanning Trees**

void MIN_SPANNING_TREE(*list*<*point*> L, *GRAPH*<*point*, *int*>& T)

computes the Euclidian minimum spanning tree T of the points in L.

- **Triangulation Checker**

bool Is_Convex_Subdivision(*GRAPH*<*point*, *int*> G)

returns true if G is a convex planar subdivision.

bool Is_Triangulation(*GRAPH*<*point*, *int*> G)

returns true if G is convex planar subdivision in which every bounded face is a triangle or if all nodes of G lie on a common line.

bool Is_Delaunay_Triangulation(*GRAPH*<*point*, *int*> G, *delaunay_voronoi_kind kind*)

checks whether G is a nearest (*kind* = *NEAREST*) or furthest (*kind* = *FURTHEST*) site Delaunay triangulation of its vertex set. G is a Delaunay triangulation iff it is a triangulation and all triangles have the Delaunay property. A triangle has the Delaunay property if no vertex of an adjacent triangle is contained in the interior (*kind* = *NEAREST*) or exterior (*kind* = *FURTHEST*) of the triangle.

bool Is_Delaunay_Diagram(*GRAPH*<*point*, *int*> G, *delaunay_voronoi_kind kind*)

checks whether G is a nearest (*kind* = *NEAREST*) or furthest (*kind* = *FURTHEST*) site Delaunay diagram of its vertex set. G is a Delaunay diagram if it is a convex subdivision, if the vertices of any bounded face are co-circular, and if every triangulation of G is a Delaunay triangulation.

- **Voronoi Diagrams**

void VORONOI(*list*<*point*> L, *GRAPH*<*circle*, *point*>& *VD*)

VORONOI takes as input a list of points (sites) L. It computes a directed graph VD representing the planar subdivision defined by the Voronoi diagram of L. For each node v of *VD* $G[v]$ is the corresponding Voronoi vertex (*point*) and for each edge e $G[e]$ is the site (*point*) whose Voronoi region is bounded by e. The algorithm has running time $O(n^2)$ in the worst case and $O(n \log n)$ with high probability, where n is the number of sites.

void F_VORONOI(*list*<*point*> *L*, *GRAPH*<*circle*, *point*>& *FVD*)

computes the farthest point Voronoi Diagram *FVD* of the points in *L*.

circle LARGEST_EMPTY_CIRCLE(*list*<*point*> *L*)

computes a largest circle whose center lies inside the convex hull of *L* that contains no point of *L* in its interior. Returns the trivial circle if *L* is empty.

circle SMALLEST_ENCLOSING_CIRCLE(*list*<*point*> *L*)

computes a smallest circle containing all points of *L* in its interior.

void ALL_EMPTY_CIRCLES(*list*<*point*> *L*, *list*<*circle*>& *CL*)

computes the list *CL* of all empty circles passing through three or more points of *L*.

void ALL_ENCLOSING_CIRCLES(*list*<*point*> *L*, *list*<*circle*>& *CL*)

computes the list *CL* of all enclosing circles passing through three or more points of *L*.

An annulus is either the region between two concentric circles or the region between two parallel lines.

bool MIN_AREA_ANNULUS(*list*<*point*> *L*, *point*& *center*, *point*& *ipoint*, *point*& *opoint*, *line*& *l1*)

computes the minimum area annulus containing the points of *L*. The function returns false if all points in *L* are collinear and returns true otherwise. In the former case a line passing through the points in *L* is returned in *l1*, and in the latter case the annulus is returned by its *center* and a point on the inner and the outer circle, respectively.

bool MIN_WIDTH_ANNULUS(*list*<*point*> *L*, *point*& *center*, *point*& *ipoint*, *point*& *opoint*, *line*& *l1*, *line*& *l2*)

computes the minimum width annulus containing the points of *L*. The function returns false if the minimum width annulus is a stripe and returns true otherwise. In the former case the boundaries of the stripes are returned in *l1* and *l2* and in the latter case the annulus is returned by its *center* and a point on the inner and the outer circle, respectively.

void CRUST(*list*<*point*> *L0*, *GRAPH*<*point*, *int*>& *G*)

takes a list *L0* of points and traces to guess the curve(s) from which *L0* are sampled. The algorithm is due to Amenta, Bern, and Eppstein [**?**]. The algorithm is guaranteed to succeed if *L0* is a sufficiently dense sample from a smooth closed curve.

bool Is_Voronoi_Diagram(*GRAPH*<*circle*, *point*> *G*, *delaunay_voronoi_kind kind*)

checks whether G represents a nearest ($kind = NEAREST$) or furthest ($kind = FURTHEST$) site Voronoi diagram.

Voronoi diagrams of point sites are represented as planar maps as follows: There is a vertex for each vertex of the Voronoi diagram and, in addition, a vertex "at infinity" for each ray of the Voronoi diagram. Vertices at infinity have degree one. The edges of the graph correspond to the edges of the Voronoi diagram. Figure **??** shows some examples and the chapter on Voronoi diagrams of the LEDA-book [58] contains more details. Each edge is labeled with the site (class *POINT*) owning the region to its left and each vertex is labeled with a triple of points (= the three defining points of a *CIRCLE*). For a "finite" vertex the three points are any three sites associated with regions incident to the vertex (and hence the center of the circle is the position of the vertex in the plane) and for a vertex at infinity the three points are collinear and the first point and the third point of the triple are the sites whose regions are incident to the vertex at infinity. Let a and c be the first and third point of the triple respectively; a and c encode the geometric position of the vertex at infinity as follows: the vertex lies on the perpendicular bisector of a and c and to the left of the segment ac.

- **Line segment intersection**

void SEGMENT_INTERSECTION(*list*<*segment*> *S*, *GRAPH*<*point*, *segment*>& *G*, *bool embed* = *false*)

takes a list of segments S as input and computes the planar graph G induced by the set of straight line segments in S. The nodes of G are all endpoints and all proper intersection points of segments in S. The edges of G are the maximal relatively open subsegments of segments in S that contain no node of G. The edges are directed as the corresponding segments. If the flag *embed* is true, the corresponding planar map is computed. Note that for each edge e $G[e]$ is the input segment that contains e (see the LEDA book for details)

void SWEEP_SEGMENTS(*list*<*segment*> *S*, *GRAPH*<*point*, *segment*>& *G*, *bool embed* = *false*, *bool use_optimization* = *true*)

as above.

The algorithm ([10]) runs in time $O((n+s)\log n) + m)$, where n is the number of segments, s is the number of vertices of the graph G, and m is the number of edges of G. If S contains no overlapping segments then $m = O(n+s)$. If *embed* is true the running time increases by $O(m \log m)$. If *use_optimization* is true an optimization described in the LEDA book is used.

void MULMULEY_SEGMENTS(*list*<*segment*> S, $GRAPH$<*point*, *segment*>& G, *bool embed* = *false*)

as above.
There is one additional output convention. If G is an undirected graph, the undirected planar map corresponding to $G(s)$ is computed. The computation follows the incremental algorithm of Mulmuley ([62]) whose expected running time is $O(M + s + n \log n)$, where n is the number of segments, s is the number of vertices of the graph G, and m is the number of edges.

void SEGMENT_INTERSECTION(*list*<*segment*> S, *void* (**report*)(*segment*, *segment*))

takes a list of segments S as input and executes for every pair (s_1, s_2) of intersecting segments $report(s_1, s_2)$. The algorithm ([6]) has running time $O(nlog^2n + k)$, where n is the number of segments and k is the number intersecting pairs of segments.

void SEGMENT_INTERSECTION(*list*<*segment*> S, *list*<*point*>& P)

takes a list of segments S as input, computes the set of (proper) intersection points between all segments in S and stores this set in P. The algorithm ([10]) has running time $O((|P| + |S|) \log |S|)$.

- **Closest Pairs**

double CLOSEST_PAIR(*list*<*point*>& L, *point*& *r1*, *point*& *r2*)

CLOSEST_PAIR takes as input a list of points L. It computes a pair of points $r1, r2 \in L$ with minimal Euclidean distance and returns the squared distance between $r1$ and $r2$. The algorithm ([70]) has running time $O(n \log n)$ where n is the number of input points.

- **Miscellaneous Functions**

void Bounding_Box(*list*<*point*> L, *point*& *pl*, *point*& *pb*, *point*& *pr*, *point*& *pt*)

computes four points pl, pb, pr, pt from L such that $(xleft, ybot, xright, ytop)$ with $xleft = pl.xcoord(\,)$, $ybot = pb.ycoord(\,)$, $xright = pr.xcoord(\,)$ and $ytop = pt.ycoord(\,)$ is the smallest iso-oriented rectangle containing all points of L. *Precondition*: L is not empty.

bool Is_Simple_Polygon(*list*<*point*> L)

takes as input a list of points L and returns $true$ if L is the vertex sequence of a simple polygon and false otherwise. The algorithms has running time $O(n \log n)$, where n is the number of points in L.

- **Properties of Geometric Graphs**

We give procedures to check properties of geometric graphs. We give procedures to verify properties of *geometric graphs*. A geometric graph is a straight-line embedded map. Every node is mapped to a point in the plane and every dart is mapped to the line segment connecting its endpoints.

We use *geo_graph* as a template parameter for geometric graphs. Any instantiation of *geo_graph* must provide a function

VECTOR edge_vector(*const geo_graph*& G, *const edge*& e)

that returns a vector from the source to the target of e. In order to use any of these template functions the file /LEDA/templates/geo_check.t must be included.

bool Is_CCW_Ordered(*geo_graph* G)

returns true if for all nodes v the neighbors of v are in increasing counter-clockwise order around v.

bool Is_CCW_Weakly_Ordered(*geo_graph* G)

returns true if for all nodes v the neighbors of v are in non-decreasing counter-clockwise order around v.

bool Is_CCW_Ordered_Plane_Map(*geo_graph* G)

Equivalent to *Is_Plane_Map*(G) and *Is_CCW_Ordered*(G).

bool Is_CCW_Weakly_Ordered_Plane_Map(*geo_graph* G)

Equivalent to *Is_Plane_Map*(G) and *Is_CCW_Weakly_Ordered*(G).

void SORT_EDGES(*geo_graph*& G)

Reorders the edges of G such that for every node v the edges in $A(v)$ are in non-decreasing order by angle.

bool Is_CCW_Convex_Face_Cycle(*geo_graph* G, *edge* e)

returns true if the face cycle of G containing e defines a counter-clockwise convex polygon, i.e, if the face cycle forms a cyclically increasing sequence of edges according to the compare-by-angles ordering.

bool Is_CCW_Weakly_Convex_Face_Cycle(*geo_graph* G, *edge* e)

returns true if the face cycle of G containing e defines a counter-clockwise weakly convex polygon, i.e, if the face cycle forms a cyclically non-decreasing sequence of edges according to the compare-by-angles ordering.

bool Is_CW_Convex_Face_Cycle(*geo_graph* G, *edge* e)

returns true if the face cycle of G containing e defines a clockwise convex polygon, i.e, if the face cycle forms a cyclically decreasing sequence of edges according to the compare-by-angles ordering.

bool Is_CW_Weakly_Convex_Face_Cycle(*geo_graph* G, *edge* e)

returns true if the face cycle of G containing e defines a clockwise weakly convex polygon, i.e, if the face cycle forms a cyclically non-increasing sequence of edges according to the compare-by-angles ordering.

11.14 Transformation (TRANSFORM)

1. Definition

There are two instantiations of *TRANSFORM*. The instantiation with the rational kernel is called *rat_transform* and the instantiation with the floating point kernel is called *transform*.

An instance T of type *TRANSFORM* is an affine transformation of two-dimensional space. It is given by a 3×3 matrix T with $T_{2,0} = T_{2,1} = 0$ and $T_{2,2} \neq 0$ and maps the point p with homogeneous coordinate vector (p_x, p_y, p_w) to the point $T \cdot p$.

A matrix of the form

$$\begin{pmatrix} w & 0 & x \\ 0 & w & y \\ 0 & 0 & w \end{pmatrix}$$

realizes an translation by the vector $(x/w, y/w)$ and a matrix of the form

$$\begin{pmatrix} a & -b & 0 \\ b & a & 0 \\ 0 & 0 & w \end{pmatrix}$$

where $a^2 + b^2 = w^2$ realizes a rotation by the angle α about the origin, where $\cos\alpha = a/w$ and $\sin\alpha = b/w$. Rotations are in counter-clockwise direction.

There are two instantiations: *transform* for the floating point kernel and *rat_transform* for the rational kernel.

#include < *LEDA/generic/TRANSFORM.h* >

2. Creation

TRANSFORM T; creates a variable introduces a variable T of type *TRANSFORM*. T is initialized with the identity transformation.

TRANSFORM T(*const INT_MATRIX* t);

introduces a variable T of type *TRANSFORM*. T is initialized with the matrix t.
Precondition: t is a 3×3 matrix with $t_{2,0} = t2, 1 = 0$ and $t_{2,2} \neq 0$.

3. Operations

INT_MATRIX	T.T_matrix( )	returns the transformation matrix
void	T.simplify( )	The operation has no effect for *transform*. For *rat_transform* let g be the ggT of all matrix entries. Cancels out g.

RAT_TYPE *T*.norm( ) returns the norm of the transformation

TRANSFORM *T*(*TRANSFORM T1*)
returns the transformation $T \circ T1$.

POINT *T*(*POINT p*) returns $T(p)$.

VECTOR *T*(*VECTOR v*) returns $T(v)$.

SEGMENT *T*(*SEGMENT s*) returns $T(s)$.

LINE *T*(*LINE l*) returns $T(l)$.

RAY *T*(*RAY r*) returns $T(r)$.

CIRCLE *T*(*CIRCLE C*) returns $T(C)$.

POLYGON *T*(*POLYGON P*) returns $T(P)$.

GEN_POLYGON *T*(*GEN_POLYGON P*)
returns $T(P)$.

Non-member Functions

In any of the function below a point can be specified to the origin by replacing it by an anononymous object of type POINT, e.g., *rotation90*(*POINT*( )) will generate a rotation about the origin.

TRANSFORM translation(*INT_TYPE dx*, *INT_TYPE dy*, *INT_TYPE dw*)
returns the translation by the vector $(dx/dw, dy/dw)$.

TRANSFORM translation(*RAT_TYPE dx*, *RAT_TYPE dy*)
returns the translation by the vector (dx, dy).

TRANSFORM translation(*VECTOR v*)
returns the translation by the vector v.

TRANSFORM rotation(*POINT q*, *double alpha*, *double eps*)
returns the rotation about q by an angle $alpha \pm eps$.

TRANSFORM rotation90(*POINT q*)
returns the rotation about q by an angle of 90 degrees.

TRANSFORM reflection(*POINT q*, *POINT r*)
returns the reflection across the straight line passing through q and r.

TRANSFORM reflection(*POINT* q)

returns the reflection across point q.

11.15 Point Generators (point generators)

All generators are available for *point*, *rat_point*, *d3_point*, and *d3_rat_point*. We use *POINT* to stand for any of these classes. The corresponding header files are called random_point.h, random_rat_point.h, random_d3_point.h, and random_d3_rat_point.h, respectively. These header files are included in the corresponding kernel header files, e.g., random_rat_point.h is part of rat_kernel.h.

We use the following naming conventions: square, circle, segment, and disk refer to two-dimensional objects (even in 3d) and cube, ball, and sphere refer to full-dimensional objects, i.e, in 2d cube and square, ball and disk, and circle and sphere are synonymous.

void random_point_in_square(*POINT*& *p*, *int maxc*)

returns a point whose x and y-coordinates are random integers in $[-maxc\,..\,maxc]$. The z-coordinate is zero.

void random_points_in_square(*int n*, *int maxc*, *list*<*POINT*>& *L*)

returns a list L of n points ... .

void random_point_in_unit_square(*POINT*& *p*, *int* $D = (1 \ll 30) - 1$)

returns a point whose coordinates are random rationals of the form i/D where i is a random integer in the range $[0\,..\,D]$. The default value of D is $2^{30} - 1$.

void random_points_in_unit_square(*int n*, *int D*, *list*<*POINT*>& *L*)

returns a list L of n points ... .

void random_points_in_unit_square(*int n*, *list*<*POINT*>& *L*)

returns a list L of n points ... . The default value of D is used.

void random_point_in_cube(*POINT*& *p*, *int maxc*)

returns a point whose coordinates are random integers in $[-maxc\,..\,maxc]$. In 2d this function is equivalent to *random_point_in_square*.

void random_points_in_cube(*int n*, *int maxc*, *list*<*POINT*>& *L*)

returns a list L of n points ... .

void random_point_in_unit_cube(*POINT*& *p*, *int* $D = (1 \ll 30) - 1$)

returns a point whose coordinates are random rationals of the form i/D where i is a random integer in the range $[0\,..\,D]$. The default value of D is $2^{30} - 1$.

void random_points_in_unit_cube(*int n*, *int D*, *list*<*POINT*>& *L*)

returns a list L of n points ... .

void random_points_in_unit_cube(*int* n, *list*<*POINT*>& L)

as above, but the default value of D is used.

void random_point_in_disc(*POINT*& p, *int* R)

returns a random point with integer x and y-coordinates in the disc with radius R centered at the origin. The z-coordinate is zero.
Precondition: $R \leq 2^{30}$.

void random_points_in_disc(*int* n, *int* R, *list*<*POINT*>& L)

returns a list L of n points ... .

void random_point_in_unit_disc(*POINT*& p, *int* $D = (1 \ll 30) - 1$)

returns a point in the unit disc whose coordinates are quotients with denominator D. The default value of D is $2^{30} - 1$.

void random_points_in_unit_disc(*int* n, *int* D, *list*<*POINT*>& L)

returns a list L of n points ... .

void random_points_in_unit_disc(*int* n, *list*<*POINT*>& L)

returns a list L of n points ... . The default value of D is used.

void random_point_in_ball(*POINT*& p, *int* R)

returns a random point with integer coordinates in the ball with radius R centered at the origin. In 2d this function is equivalent to *random_point_in_disc*.
Precondition: $R \leq 2^{30}$.

void random_points_in_ball(*int* n, *int* R, *list*<*POINT*>& L)

returns a list L of n points ... .

void random_point_in_unit_ball(*POINT*& p, *int* $D = (1 \ll 30) - 1$)

returns a point in the unit ball whose coordinates are quotients with denominator D. The default value of D is $2^{30} - 1$.

void random_points_in_unit_ball(*int* n, *int* D, *list*<*POINT*>& L)

returns a list L of n points ... .

void random_points_in_unit_ball(*int* n, *list*<*POINT*>& L)

returns a list L of n points ... . The default value of D is used.

void random_point_near_circle(*POINT*& p, *int* R)

returns a random point with integer coordinates that lies close to the circle with radius R centered at the origin.

void random_points_near_circle(*int* n, *int* R, *list*<*POINT*>& L)

returns a list L of n points ... .

void random_point_near_unit_circle(*POINT*& p, *int* $D = (1 \ll 30) - 1$)

returns a point close to the unit circle whose coordinates are quotients with denominator D. The default value of D is $2^{30} - 1$.

void random_points_near_unit_circle(*int* n, *int* D, *list*<*POINT*>& L)

returns a list L of n points ... .

void random_points_near_unit_circle(*int* n, *list*<*POINT*>& L)

returns a list L of n points ... . The default value of D is used.

void random_point_near_sphere(*POINT*& p, *int* R)

returns a point with integer coordinates close to the sphere with radius R centered at the origin.

void random_points_near_sphere(*int* n, *int* R, *list*<*POINT*>& L)

returns a list L of n points ... .

void random_point_near_unit_sphere(*POINT*& p, *int* $D = (1 \ll 30) - 1$)

returns a point close to the unit sphere whose coordinates are quotients with denominator D. In 2d this function is equivalent to *point_near_unit_circle*.

void random_points_near_unit_sphere(*int* n, *int* D, *list*<*POINT*>& L)

returns a list L of n points ... .

void random_points_near_unit_sphere(*int* n, *list*<*POINT*>& L)

returns a list L of n points ... . The default value of D is used.

Wit the rational kernel the functions *_on_circle* are guaranteed to produce points that lie precisely on the specified circle. With the floating point kernel the functions are equivalent to the *_near_circle* functions.

void random_point_on_circle(*POINT*& p, *int* R, *int* $C = 1000000$)

returns a random point with integer coordinates that lies on the circle with radius R centered at the origin. The point is chosen from a set of at least C candidates.

void random_points_on_circle(*int* n, *int* R, *list*<*POINT*>& L, *int* $C = 1000000$)

returns a list L of n points ... .

void random_point_on_unit_circle(*POINT*& p, *int* $C = 1000000$)

returns a point on the unit circle. The point is chosen from a set of at least C candidates.

void random_points_on_unit_circle(*int* n, *list*<*POINT*>& L, *int* $C = 1000000$)

returns a list L of n points … .

void random_point_on_sphere(*POINT*& p, *int* R)

same as *random_point_near_sphere*.

void random_points_on_sphere(*int* n, *int* R, *list*<*POINT*>& L)

returns a list L of n points … .

void random_point_on_unit_sphere(*POINT*& p, *int* $D = (1 \ll 30) - 1$)

same as *random_point_near_unit_sphere*.

void random_points_on_unit_sphere(*int* n, *int* D, *list*<*POINT*>& L)

returns a list L of n points … .

void random_points_on_unit_sphere(*int* n, *list*<*POINT*>& L)

returns a list L of n points … . The default value of D is used.

void random_point_on_segment(*POINT*& p, *SEGMENT* s)

generates a random point on s.

void random_points_on_segment(*SEGMENT* s, *int* n, *list*<*POINT*>& L)

generates a list L of n points … .

void points_on_segment(*SEGMENT* s, *int* n, *list*<*POINT*>& L)

generates a list L of n equally spaced points on s.

void random_point_on_paraboloid(*POINT*& p, *int* $maxc$)

returns a point (x, y, z) with x and y random integers in the range $[-maxc \,..\, maxc]$, and $z = 0.004 * (x * x + y * y) - 1.25 * maxc$. The function does not make sense in 2d.

void random_points_on_paraboloid(*int* n, *int* $maxc$, *list*<*POINT*>& L)

returns a list L of n points … .

void lattice_points(*int* n, *int* $maxc$, *list*<*POINT*>& L)

returns a list L of approximately n points. The points have integer coordinates $id/maxc$ for an appropriately chosen d and $-maxc/d \leq i \leq maxc/d$.

void random_points_on_diagonal(*int* n, *int* $maxc$, *list*<*POINT*>& L)

generates n points on the diagonal whose coordinates are random integer in the range from $-maxc$ to $maxc$.

Chapter 12

Advanced Data Types for Two-Dimensional Geometry

12.1 Two-Dimensional Dictionaries (d2_dictionary)

1. Definition

An instance D of the parameterized data type *d2_dictionary*<*K1*, *K2*, *I*> is a collection of items (*dic2_item*). Every item in D contains a key from the linearly ordered data type $K1$, a key from the linearly ordered data type $K2$, and an information from data type I. $K1$ and $K2$ are called the key types of D, and I is called the information type of D. The number of items in D is called the size of D. A two-dimensional dictionary of size zero is said to be empty. We use $<k_1, k_2, i>$ to denote the item with first key k_1, second key k_2, and information i. For each pair $(k_1, k_2) \in K1 \times K2$ there is at most one item $<k_1, k_2, i> \in D$. Additionally to the normal dictionary operations, the data type *d2_dictionary* supports rectangular range queries on $K1 \times K2$.

#include < *LEDA/d2_dictionary.h* >

2. Creation

d2_dictionary<*K1*, *K2*, *I*> D;

creates an instance D of type *d2_dictionary*<*K1*, *K2*, *I*> and initializes D to the empty dictionary.

3. Operations

K1	*D*.key1(*dic2_item it*)	returns the first key of item *it*. *Precondition*: *it* is an item in D.
K2	*D*.key2(*dic2_item it*)	returns the second key of item *it*. *Precondition*: *it* is an item in D.

I	*D*.inf(*dic2_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in *D*.
I&	*D*[*dic2_item it*]	returns a reference to the information of item *it*. *Precondition*: *it* is an item in *D*.
dic2_item	*D*.min_key1( )	returns the item with minimal first key.
dic2_item	*D*.min_key2( )	returns the item with minimal second key.
dic2_item	*D*.max_key1( )	returns the item with maximal first key.
dic2_item	*D*.max_key2( )	returns the item with maximal second key.
dic2_item	*D*.insert(*K1 x*, *K2 y*, *I i*)	associates the information i with the keys x and y. If there is an item $<x, y, j>$ in D then j is replaced by i, else a new item $<x, y, i>$ is added to D. In both cases the item is returned.
dic2_item	*D*.lookup(*K1 x*, *K2 y*)	returns the item with keys x and y (nil if no such item exists in D).
list<*dic2_item*>	*D*.range_search(*K1 x0*, *K1 x1*, *K2 y0*, *K2 y1*)	returns the list of all items $<k_1, k_2, i>$ in D with $x_0 \leq k_1 \leq x_1$ and $y_0 \leq k_2 \leq y_1$.
list<*dic2_item*>	*D*.all_items( )	returns the list of all items of *D*.
void	*D*.del(*K1 x*, *K2 y*)	deletes the item with keys x and y from *D*.
void	*D*.del_item(*dic2_item it*)	removes item *it* from *D*. *Precondition*: *it* is an item in *D*.
void	*D*.change_inf(*dic2_item it*, *I i*)	makes i the information of item *it*. *Precondition*: *it* is an item in *D*.
void	*D*.clear( )	makes *D* the empty d2_dictionary.
bool	*D*.empty( )	returns true if *D* is empty, false otherwise.
int	*D*.size( )	returns the size of *D*.

4. Implementation

Two-dimensional dictionaries are implemented by dynamic two-dimensional range trees [80, 51] based on BB[α] trees. Operations insert, lookup, del_item, del take time $O(\log^2 n)$, range_search takes time $O(k + \log^2 n)$, where k is the size of the returned list, key, inf, empty, size, change_inf take time $O(1)$, and clear takes time $O(n \log n)$. Here n is the current size of the dictionary. The space requirement is $O(n \log n)$.

12.2 Point Sets and Delaunay Triangulations (POINT_SET)

1. Definition

There are two instantiations of POINT_SET. The instantiation with the rational kernel is called *rat_point_set* and requires inclusion of rat_point_set.h and the instantiation with the floating point kernel is called *point_set* and requires inclusion of point_set.h.

An instance T of data type *POINT_SET* is a planar embedded bidirected graph (map) representing the *Delaunay Triangulation* of its vertex set. The position of a vertex v is given by $T.pos(v)$ and we use $S = \{T.pos(v) \mid v \in T\}$ to denote the underlying point set. Each face of T (except for the outer face) is a triangle whose circumscribing circle does not contain any point of S in its interior. For every edge e, the sequence

$$e, T.face\_cycle\_succ(e), T.face\_cycle\_succ(T.face\_cycle\_succ(e)), \ldots$$

traces the boundary of the face to the left of e. The edges of the outer face of T form the convex hull of S; the trace of the convex hull is clockwise. The subgraph obtained from T by removing all diagonals of co-circular quadrilaterals is called the *Delaunay Diagram* of S.

POINT_SET provides all *constant* graph operations, e.g., $T.reversal(e)$ returns the reversal of edge e, $T.all\_edges(\,)$ returns the list of all edges of T, and $forall\_edges(e, T)$ iterates over all edges of T. In addition, *POINT_SET* provides operations for inserting and deleting points, point location, nearest neighbor searches, and navigation in both the triangulation and the diagram.

*POINT_SET*s are essentially objects of type *GRAPH*<*POINT*, *int*>, where the node information is the position of the node and the edge information is irrelevant. For a graph G of type *GRAPH*<*POINT*, *int*> the function $Is\_Delaunay(G)$ tests whether G is a Delaunay triangulation of its vertices.

The data type *POINT_SET* is illustrated by the *point_set_demo* in the LEDA demo directory.

Be aware that the nearest neighbor queries for a point (not for a node) and the range search queries for circles, triangles, and rectangles are non-const operations and modify the underlying graph. The set of nodes and edges is not changed; however, it is not guaranteed that the underlying Delaunay triangulation is unchanged.

#include < LEDA/generic/POINT_SET.h >

2. Creation

POINT_SET T;   creates an empty *POINT_SET* T.

POINT_SET T(*list*<*POINT*> S);

creates a *POINT_SET* T of the points in S. If S contains multiple occurrences of points only the last occurrence of each point is retained.

POINT_SET T(*GRAPH*<*POINT*, *int*> G);

initializes T with a copy of G.
Precondition: *Is_Delaunay*(G) is true.

3. Operations

void	T.init(*list*<*POINT*> L)	makes T a *POINT_SET* for the points in S.
POINT	T.pos(*node* v)	returns the position of node v.
POINT	T.pos_source(*edge* e)	returns the position of $source(e)$.
POINT	T.pos_target(*edge* e)	returns the position of $target(e)$.
SEGMENT	T.seg(*edge* e)	returns the line segment corresponding to edge e (*SEGMENT*($T.pos\_source(e), T.pos\_target(e)$)).
LINE	T.supporting_line(*edge* e)	returns the supporting line of edge e (*LINE*($T.pos\_source(e), T.pos\_target(e)$)).
int	T.orientation(*edge* e, *POINT* p)	returns $orientation(T.seg(e), p)$.
int	T.dim()	returns -1 if S is empty, returns 0 if S consists of only one point, returns 1 if S consists of at least two points and all points in S are collinear, and returns 2 otherwise.
list<*POINT*>	T.points()	returns S.
edge	T.get_hull_dart()	returns a dart of the outer face of T (i.e., a dart of the convex hull).
edge	T.get_hull_edge()	as above.
bool	T.is_hull_dart(*edge* e)	returns true if e is a dart of the convex hull of T, i.e., a dart on the face cycle of the outer face.
bool	T.is_hull_edge(*edge* e)	as above.

bool	T.is_diagram_dart(*edge* e)	returns true if e is a dart of the Delaunay diagram, i.e., either a dart on the convex hull or a dart where the incident triangles have distinct circumcircles.
bool	T.is_diagram_edge(*edge* e)	as above.
edge	T.d_face_cycle_succ(*edge* e)	returns the face cycle successor of e in the Delaunay diagram of T. *Precondition*: e belongs to the Delaunay diagram.
edge	T.d_face_cycle_pred(*edge* e)	returns the face cycle predecessor of e in the Delaunay diagram of T. *Precondition*: e belongs to the Delaunay diagram.
bool	T.empty()	decides whether T is empty.
void	T.clear()	makes T empty.
edge	T.locate(*POINT* p)	returns an edge e of T that contains p or that borders the face that contains p. In the former case, a hull dart is returned if p lies on the boundary of the convex hull. In the latter case we have $T.orientation(e,p) > 0$ except if all points of T are collinear and p lies on the induced line. In this case $target(e)$ is visible from p. The function returns *nil* if T has no edge.
node	T.lookup(*POINT* p)	if T contains a node v with $pos(v) = p$ the result is v otherwise the result is *nil*.
node	T.insert(*POINT* p)	inserts point p into T and returns the corresponding node. More precisely, if there is already a node v in T positioned at p (i.e., $pos(v)$ is equal to p) then $pos(v)$ is changed to p (i.e., $pos(v)$ is made identical to p) and if there is no such node then a new node v with $pos(v) = p$ is added to T. In either case, v is returned.
void	T.del(*node* v)	removes the node v, i.e., makes T a Delaunay triangulation for $S \setminus \{pos(v)\}$.
void	T.del(*POINT* p)	removes the node p, i.e., makes T a Delaunay triangulation for $S \setminus p$.

node T.nearest_neighbor(*POINT* p)

computes a node v of T that is closest to p, i.e., $dist(p, pos(v)) = \min\{\ dist(p, pos(u)) \mid u \in T\ \}$. This is a non-const operation.

node T.nearest_neighbor(*node* w)

computes a node v of T that is closest to $p = T[w]$, i.e., $dist(p, pos(v)) = \min\{\ dist(p, pos(u)) \mid u \in T\ \}$.

list<*node*> T.nearest_neighbors(*POINT* p, *int* k)

returns the k nearest neighbors of p, i.e., a list of the $\min(k, |S|)$ nodes of T closest to p. The list is ordered by distance from p. This is a non-const operation.

list<*node*> T.nearest_neighbors(*node* w, *int* k)

returns the k nearest neighbors of $p = T[w]$, i.e., a list of the $\min(k, |S|)$ nodes of T closest to p. The list is ordered by distance from p.

list<*node*> T.range_search(*CIRCLE* C)

returns the list of all nodes contained in the closure of disk C.
Precondition: C must be a proper circle (not a straight line). This is a non-const operation.

list<*node*> T.range_search(*node* v, *POINT* p)

returns the list of all nodes contained in the closure of disk C with center $pos[v]$ and having p in its boundary.

list<*node*> T.range_search(*POINT* a, *POINT* b, *POINT* c)

returns the list of all nodes contained in the closure of the triangle (a, b, c).
Precondition: a, b, and c must not be collinear. This is a non-const operation.

list<*node*> T.range_search(*POINT* a, *POINT* b)

returns the list of all nodes contained in the closure of the rectangle with diagonal (a, b). This is a non-const operation.

list<*edge*> T.minimum_spanning_tree( )

returns the list of edges of T that comprise a minimum spanning tree of S.

void T.compute_voronoi(*GRAPH*<*CIRCLE*, *POINT*>& V)

computes the corresponding Voronoi diagram V. Each node of VD is labeled with its defining circle. Each edge is labeled with the site lying in the face to its left.

Drawing Routines

The functions in this section were designed to support the drawing of Delaunay triangulations and Voronoi diagrams.

void T.draw_nodes(*void* (**draw_node*)(*POINT*))

calls *draw_node*(*pos*(v)) for every node v of T.

void T.draw_edge(*edge* e, *void* (**draw_diagram_edge*)(*POINT*, *POINT*),
void (**draw_triang_edge*) (*POINT*, *POINT*),
void (**draw_hull_dart*) (*POINT*, *POINT*))

calls *draw_diagram_edge*(*pos_source*(e), *pos_target*(e) if e is a diagram dart, *draw_hull_dart*(*pos_source*(e), *pos_target*(e) if e is a hull dart, and *draw_triang_edge*(*pos_source*(e), *pos_target*(e) if e is a non-diagram edge.

void T.draw_edges(*void* (**draw_diagram_edge*)(*POINT*, *POINT*),
void (**draw_triang_edge*) (*POINT*, *POINT*),
void (**draw_hull_dart*) (*POINT*, *POINT*))

calls the corresponding function for all edges of T.

void T.draw_edges(*list*<*edge*> L, *void* (**draw_edge*)(*POINT*, *POINT*))

calls *draw_edge*(*pos_source*(e), *pos_target*(e) for every edge $e \in L$.

void T.draw_voro_edges(*void* (**draw_edge*)(*POINT*, *POINT*),
void (**draw_ray*) (*POINT*, *POINT*))

calls *draw_edge* and *draw_ray* for the edges of the Voronoi diagram.

void T.draw_hull(*void* (**draw_poly*)(*list*<*POINT*>))

calls *draw_poly* with the list of vertices of the convex hull.

void T.draw_voro(*GRAPH*<*CIRCLE*, *POINT*>, *void* (**draw_node*)(*POINT*),
void (**draw_edge*)(*POINT*, *POINT*), *void* (**draw_ray*) (*POINT*, *POINT*))

calls ...

4. Implementation

The main ingredients for the implementation are Delaunay flipping, segment walking, and plane sweep.

The constructor *POINT_SET*(*list*<*POINT*> S) first constructs a triangulation of S by sweeping and then makes the triangulation Delaunay by a sequence of Delaunay flips.

Locate walks through the triangulation along the segment from some fixed point of T to the query point. *Insert* first locates the point, then updates the triangulation locally, and finally performs flips to reestablish the Delaunay property. *Delete* deletes the node, retriangulates the resulting face, and then performs flips. Nearest neighbor searching, circular range queries, and triangular range queries insert the query point into the triangulation, then perform an appropriate graph search on the triangulation, and finally remove the query point.

All algorithms show good expected behavior.

For details we refer the reader to the LEDA implementation report "Point Sets and Dynamic Delaunay Triangulations".

12.3 Sets of Intervals (interval_set)

1. Definition

An instance S of the parameterized data type *interval_set*<*I*> is a collection of items (*is_item*). Every item in S contains a closed interval of the *double* numbers as key and an information from data type I, called the information type of S. The number of items in S is called the size of S. An interval set of size zero is said to be empty. We use $<x, y, i>$ to denote the item with interval $[x, y]$ and information i; x (y) is called the left (right) boundary of the item. For each interval $[x, y]$ there is at most one item $<x, y, i> \in S$.

#include < *LEDA/interval_set.h* >

2. Creation

interval_set<*I*> S; — creates an instance S of type *interval_set*<*I*> and initializes S to the empty set.

3. Operations

double	S.left(*is_item it*)	returns the left boundary of item *it*. *Precondition*: *it* is an item in S.
double	S.right(*is_item it*)	returns the right boundary of item *it*. *Precondition*: *it* is an item in S.
I	S.inf(*is_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in S.
is_item	S.insert(*double x*, *double y*, *I i*)	associates the information i with interval $[x, y]$. If there is an item $<x, y, j>$ in S then j is replaced by i, else a new item $<x, y, i>$ is added to S. In both cases the item is returned.
is_item	S.lookup(*double x*, *double y*)	returns the item with interval $[x, y]$ (nil if no such item exists in S).
list<*is_item*>	S.intersection(*double a*, *double b*)	returns all items $<x, y, i> \in S$ with $[x, y] \cap [a, b] \neq \emptyset$.
void	S.del(*double x*, *double y*)	deletes the item with interval $[x, y]$ from S.
void	S.del_item(*is_item it*)	removes item *it* from S. *Precondition*: *it* is an item in S.

void	S.change_inf(*is_item it*, *I i*)	makes i the information of item *it*. *Precondition*: *it* is an item in S.
void	S.clear( )	makes S the empty interval_set.
bool	S.empty( )	returns true iff S is empty.
int	S.size( )	returns the size of S.

4. Implementation

Interval sets are implemented by two-dimensional range trees [80, 51]. Operations insert, lookup, del_item and del take time $O(\log^2 n)$, intersection takes time $O(k + \log^2 n)$, where k is the size of the returned list. Operations left, right, inf, empty, and size take time $O(1)$, and clear $O(n \log n)$. Here n is always the current size of the interval set. The space requirement is $O(n \log n)$.

12.4 Sets of Parallel Segments (segment_set)

1. Definition

An instance S of the parameterized data type *segment_set*<*I*> is a collection of items (*seg_item*). Every item in S contains as key a line segment with a fixed direction α (see data type segment) and an information from data type I, called the information type of S. α is called the orientation of S. We use $<s,i>$ to denote the item with segment s and information i. For each segment s there is at most one item $<s,i> \in S$.

#include < LEDA/segment_set.h >

2. Creation

segment_set<*I*> S(*double a*);

creates an empty instance S of type *segment_set*<*I*> with orientation a.

segment_set<*I*> S; creates an empty instance S of type *segment_set*<*I*> with orientation zero, i.e., horizontal segments.

3. Operations

segment	S.key(*seg_item it*)	returns the segment of item *it*. *Precondition*: *it* is an item in S.
I	S.inf(*seg_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in S.
seg_item	S.insert(*segment s*, *I i*)	associates the information i with segment s. If there is an item $<s,j>$ in S then j is replaced by i, else a new item $<s,i>$ is added to S. In both cases the item is returned.
seg_item	S.lookup(*segment s*)	returns the item with segment s (nil if no such item exists in S).
list<*seg_item*>	S.intersection(*segment q*)	returns all items $<s,i> \in S$ with $s \cap q \neq \emptyset$. *Precondition*: q is orthogonal to the segments in S.
list<*seg_item*>	S.intersection(*line l*)	returns all items $<s,i> \in S$ with $s \cap l \neq \emptyset$. *Precondition*: l is orthogonal to the segments in S.
void	S.del(*segment s*)	deletes the item with segment s from S.
void	S.del_item(*seg_item it*)	removes item *it* from S. *Precondition*: *it* is an item in S.

void	S.change_inf(*seg_item* it, I i)	makes i the information of item it. *Precondition*: it is an item in S.
void	S.clear( )	makes S the empty segment_set.
bool	S.empty( )	returns true iff S is empty.
int	S.size( )	returns the size of S.

4. Implementation

Segment sets are implemented by dynamic segment trees based on BB[α] trees ([80, 51]) trees. Operations key, inf, change_inf, empty, and size take time $O(1)$, insert, lookup, del, and del_item take time $O(\log^2 n)$ and an intersection operation takes time $O(k + \log^2 n)$, where k is the size of the returned list. Here n is the current size of the set. The space requirement is $O(n \log n)$.

12.5 Planar Subdivisions (subdivision)

1. Definition

An instance S of the parameterized data type *subdivision*<*I*> is a subdivision of the two-dimensional plane, i.e., an embedded planar graph with straight line edges (see also sections 8.5 and 8.6). With each node v of S is associated a point, called the position of v and with each face of S is associated an information from data type I, called the information type of S.

#include < *LEDA/subdivision.h* >

2. Creation

subdivision<*I*> S(*GRAPH*<*point*, *I*>& G);

creates an instance S of type *subdivision*<*I*> and initializes it to the subdivision represented by the parameterized directed graph G. The node entries of G (of type point) define the positions of the corresponding nodes of S. Every face f of S is assigned the information of one of its bounding edges in G.
Precondition: G represents a planar subdivision, i.e., a straight line embedded planar map.

3. Operations

point	S.position(*node* v)	returns the position of node v.
I	S.inf(*face* f)	returns the information of face f.
face	S.locate_point(*point* p)	returns the face containing point p.

4. Implementation

Planar subdivisions are implemented by parameterized planar maps and an additional data structure for point location based on persistent search trees [22]. Operations position and inf take constant time, a locate_point operation takes time $O(\log^2 n)$. Here n is the number of nodes. The space requirement and the initialization time is $O(n^2)$.

Chapter 13

Basic Data Types for Three-Dimensional Geometry

13.1 Points in 3D-Space (d3_point)

1. Definition

An instance of the data type *d3_point* is a point in the three-dimensional space $\mathbb{R}^3$. We use (x, y, z) to denote a point with first (or x-) coordinate x, second (or y-) coordinate y, and third (or z-) coordinate z.

#include < *LEDA/d3_point.h* >

2. Creation

d3_point p; — introduces a variable p of type *d3_point* initialized to the point $(0, 0, 0)$.

d3_point p(*double* x, *double* y, *double* z);

introduces a variable p of type *d3_point* initialized to the point (x, y, z).

d3_point p(*vector* v);

introduces a variable p of type *d3_point* initialized to the point $(v[0], v[1], v[2])$.
Precondition: $v.dim() = 3$.

3. Operations

double	*p*.xcoord( )	returns the first coordinate of p.
double	*p*.ycoord( )	returns the second coordinate of p.
double	*p*.zcoord( )	returns the third coordinate of p.
vector	*p*.to_vector( )	returns the vector $x\vec{y}z$.
point	*p*.project_xy( )	returns p projected into the xy-plane.
point	*p*.project_yz( )	returns p projected into the yz-plane.
point	*p*.project_xz( )	returns p projected into the xz-plane.
double	*p*.sqr_dist(*d3_point* q)	returns the square of the Euclidean distance between p and q.
double	*p*.xdist(*d3_point* q)	returns the x-distance between p and q.
double	*p*.ydist(*d3_point* q)	returns the y-distance between p and q.
double	*p*.zdist(*d3_point* q)	returns the z-distance between p and q.

double	p.distance(*d3_point* q)	
		returns the Euclidean distance between p and q.
d3_point	p.translate(*double* dx, *double* dy, *double* dz)	
		returns p translated by vector (dx, dy, dz).
d3_point	p.translate(*vector* v)	returns p+v, i.e., p translated by vector v. *Precondition*: v.dim() = 3.
d3_point	p + *vector* v	returns p translated by vector v.
d3_point	p − *vector* v	returns p translated by vector $-v$.
d3_point	p.reflect(*d3_point* q, *d3_point* r, *d3_point* s)	
		returns p reflected across the plane passing through q, r and s.
d3_point	p.reflect(*d3_point* q)	returns p reflected across point q.
vector	$p - q$	returns the difference vector of the coordinates.
ostream&	*ostream&* O ≪ p	writes p to output stream O.
istream&	*istream&* I ≫ *d3_point&* p	
		reads the coordinates of p (three *double* numbers) from input stream I.

Non-Member Functions

int	cmp_distances(*d3_point* $p1$, *d3_point* $p2$, *d3_point* $p3$, *d3_point* $p4$)	
		compares the distances $(p1, p2)$ and $(p3, p4)$. Returns +1 (−1) if distance $(p1, p2)$ is larger (smaller) than distance $(p3, p4)$, otherwise 0.
d3_point	center(*d3_point* a, *d3_point* b)	
		returns the center of a and b, i.e. $a + \vec{ab}/2$.
d3_point	midpoint(*d3_point* a, *d3_point* b)	
		returns the center of a and b.
int	orientation(*d3_point* a, *d3_point* b, *d3_point* c, *d3_point* d)	
		computes the orientation of points a, b, c, and d as the sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 & 1 \\ a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \end{vmatrix}$$

double volume(*d3_point a*, *d3_point b*, *d3_point c*, *d3_point d*)

computes the signed volume of the simplex determined by a,b, c, and d, positive if $orientation(a, b, c, d) > 0$ and negative otherwise.

bool collinear(*d3_point a*, *d3_point b*, *d3_point c*)

returns true if points a, b, c are collinear and false otherwise.

bool coplanar(*d3_point a*, *d3_point b*, *d3_point c*, *d3_point d*)

returns true if points a, b, c, d are coplanar and false otherwise.

int side_of_sphere(*d3_point a*, *d3_point b*, *d3_point c*, *d3_point d*, *d3_point x*)

returns $+1$ (-1) if point x lies on the positive (negative) side of the oriented sphere through points a, b, c, and d, and 0 if x is contained in this sphere.

int region_of_sphere(*d3_point a*, *d3_point b*, *d3_point c*, *d3_point d*, *d3_point x*)

determines whether the point x lies inside $(= +1)$, on $(= 0)$, or outside $(= -1)$ the sphere through points a, b, c, d, (equivalent to $orientation(a, b, c, d) * side\_of\_sphere(a, b, c, d, x)$)
Precondition: $orientation(A) \neq 0$

bool contained_in_simplex(*d3_point a*, *d3_point b*, *d3_point c*, *d3_point d*, *d3_point x*)

determines whether x is contained in the simplex spanned by the points a, b, c, d.
Precondition: a, b, c, d are affinely independent.

bool contained_in_simplex(*array*<*d3_point*> *A*, *d3_point x*)

determines whether x is contained in the simplex spanned by the points in A.
Precondition: A must have size ≤ 4 and the points in A must be affinely independent.

bool contained_in_affine_hull(*list*<*d3_point*> *L*, *d3_point x*)

determines whether x is contained in the affine hull of the points in L.

bool contained_in_affine_hull(*array*<*d3_point*> *A*, *d3_point x*)

determines whether x is contained in the affine hull of the points in A.

int affine_rank(*array*<*d3_point*> *L*)

computes the affine rank of the points in L.

int affine_rank(*array*<*d3_point*> *A*)

computes the affine rank of the points in A.

bool affinely_independent(*list*<*d3_point*> *L*)

decides whether the points in A are affinely independent.

bool affinely_independent(*array*<*d3_point*> *A*)

decides whether the points in A are affinely independent.

bool inside_sphere(*d3_point a*, *d3_point b*, *d3_point c*, *d3_point d*, *d3_point e*)

returns $true$ if point e lies in the interior of the sphere through points a, b, c, and d, and $false$ otherwise.

bool outside_sphere(*d3_point a*, *d3_point b*, *d3_point c*, *d3_point d*, *d3_point e*)

returns $true$ if point e lies in the exterior of the sphere through points a, b, c, and d, and $false$ otherwise.

bool on_sphere(*d3_point a*, *d3_point b*, *d3_point c*, *d3_point d*, *d3_point e*)

returns $true$ if a, b, c, d, and e lie on a common sphere.

d3_point point_on_positive_side(*d3_point a*, *d3_point b*, *d3_point c*)

returns a point d with $orientation(a, b, c, d) > 0$.

13.2 Straight Rays in 3D-Space (d3_ray)

1. Definition

An instance r of the data type *d3_ray* is a directed straight ray in three-dimensional space.

#include < *LEDA/d3_ray.h* >

2. Creation

d3_ray r(*d3_point p1*, *d3_point p2*);

introduces a variable r of type *d3_ray*. r is initialized to the ray starting at point *p1* and going through *p2*.

d3_ray r(*d3_segment s*);

introduces a variable r of type *d3_ray*. r is initialized to *ray*(*s.source*( ), *s.target*( )) .

3. Operations

d3_point	r.source( )	returns the source of r.
d3_point	r.point1( )	returns the source of r.
d3_point	r.point2( )	returns a point on r different from the source.
d3_segment	r.seg( )	returns a segment on r.
bool	r.contains(*d3_point p*)	returns true if p lies on r.
bool	r.contains(*d3_segment s*)	returns true if s lies on r.
bool	r.intersection(*d3_segment s*, *d3_point& inter*)	if s and r intersect in a single point, true is returned and the point of intersection is assigned to inter. Otherwise false is returned.
bool	r.intersection(*d3_ray r*, *d3_point& inter*)	if r and r intersect in a single point, true is returned and the point of intersection is assigned to inter. Otherwise false is returned.
bool	r.project_xy(*ray& m*)	if the projection of r into the xy plane is not a point, the function returns true and assignes the projection to m. Otherwise false is returned.

bool	r.project_xz(*ray*& m)	if the projection of r into the xz plane is not a point, the function returns true and assignes the projection to m. Otherwise false is returned.
bool	r.project_yz(*ray*& m)	if the projection of r into the yz plane is not a point, the function returns true and assignes the projection to m. Otherwise false is returned.
bool	r.project(*d3_point* p, *d3_point* q, *d3_point* v, *d3_ray*& m)	if the projection of r into the plane through (p, q, v) is not a point, the function returns true and assignes the projection to m. Otherwise false is returned.
d3_ray	r.reverse( )	returns a ray starting at r.source() with direction -r.to_vector().
d3_ray	r.translate(*vector* v)	returns r translated by vector v. Precond. : $v.dim( ) = 3$.
d3_ray	r.translate(*double* dx, *double* dy, *double* dz)	returns r translated by vector (dx, dy, dz).
d3_ray	r + *vector* v	returns r translated by vector v.
d3_ray	r − *vector* v	returns r translated by vector $-v$.
d3_ray	r.reflect(*d3_point* p, *d3_point* q, *d3_point* v)	returns r reflected across the plane through (p, q, v).
d3_ray	r.reflect(*d3_point* p)	returns r reflected across p.
vector	r.to_vector( )	returns *point2*( ) − *point1*( ).

13.3 Segments in 3D-Space (d3_segment)

1. Definition

An instance s of the data type *d3_segment* is a directed straight line segment in three-dimensional space, i.e., a straight line segment $[p, q]$ connecting two points $p, q \in \mathbb{R}^3$. p is called the *source* or start point and q is called the *target* or end point of s. The length of s is the Euclidean distance between p and q. A segment is called *trivial* if its source is equal to its target. If s is not trivial, we use $line(s)$ to denote the straight line containing s.

#include < *LEDA/d3_segment.h* >

2. Creation

d3_segment s(*d3_point p1*, *d3_point p2*);

introduces a variable s of type *d3_segment*. s is initialized to the segment from *p1* to *p2*.

d3_segment s; introduces a variable s of type *d3_segment*. s is initialized to the segment from $(0, 0, 0)$ to $(1, 0, 0)$.

3. Operations

bool	s.contains(*d3_point p*)	decides whether s contains p.
d3_point	s.source( )	returns the source point of segment s.
d3_point	s.target( )	returns the target point of segment s.
double	s.xcoord1( )	returns the x-coordinate of s.source().
double	s.xcoord2( )	returns the x-coordinate of s.target().
double	s.ycoord1( )	returns the y-coordinate of s.source().
double	s.ycoord2( )	returns the y-coordinate of s.target().
double	s.zcoord1( )	returns the z-coordinate of s.source().
double	s.zcoord2( )	returns the z-coordinate of s.target().
double	s.dx( )	returns *xcoord2*( ) − *xcoord1*( ).
double	s.dy( )	returns *ycoord2*( ) − *ycoord1*( ).
double	s.dz( )	returns *zcoord2*( ) − *zcoord1*( ).

segment	s.project_xy( )	returns the projection into the xy plane.
segment	s.project_xz( )	returns the projection into the xz plane.
segment	s.project_yz( )	returns the projection into the yz plane.
d3_segment	s.project(*d3_point* p, *d3_point* q, *d3_point* v)	returns s projected into the plane through (p, q, v).
d3_segment	s.reflect(*d3_point* p, *d3_point* q, *d3_point* v)	returns s reflected across the plane through (p, q, v).
d3_segment	s.reflect(*d3_point* p)	returns s reflected across point p.
d3_segment	s.reverse( )	returns s reversed.
vector	s.to_vector( )	returns $s.target(\,) - s.source(\,)$.
bool	s.intersection(*d3_segment* t)	decides, whether s and t intersect in a single point.
bool	s.intersection(*d3_segment* t, *d3_point*& p)	decides, whether s and t intersect in a single point. If they intersect in a single point, the point is assigned to p and the result is true, otherwise the result is false
bool	s.intersection_of_lines(*d3_segment* t, *d3_point*& p)	If $line(s)$ and $line(t)$ intersect in a single point this point is assigned to p and the result is true, otherwise the result is false.
bool	s.is_trivial( )	returns true if s is trivial.
double	s.sqr_length( )	returns the square of the length of s.
double	s.length( )	returns the length of s.
d3_segment	s.translate(*vector* v)	returns s translated by vector v. *Precond.* : $v.dim(\,) = 3$.
d3_segment	s.translate(*double* dx, *double* dy, *double* dz)	returns s translated by vector (dx, dy, dz).
d3_segment	s + *vector* v	returns s translated by vector v.
d3_segment	s − *vector* v	returns s translated by vector $-v$.

13.4 Straight Lines in 3D-Space (d3_line)

1. Definition

An instance l of the data type *d3_line* is a directed straight line in three-dimensional space.

#include < *LEDA/d3_line.h* >

2. Creation

d3_line l(*d3_point p1*, *d3_point p2*);

introduces a variable l of type *d3_line*. l is initialized to the line through points *p1*, *p2*.
Precondition : *p1* != *p2*.

d3_line l(*d3_segment s*);

introduces a variable l of type *d3_line*. l is initialized to the line supporting segment s.
Precondition : *s is not trivial*.

d3_line l; introduces a variable l of type *d3_line*. l is initialized to the line through points $(0, 0, 0)$ and $(1, 0, 0)$.

3. Operations

bool	l.contains(*d3_point p*)	returns true if p lies on l.
d3_point	l.point1( )	returns a point on l.
d3_point	l.point2( )	returns a second point on l.
d3_segment	l.seg( )	returns a non-trivial segment on l with the same direction.
bool	l.project_xy(*line*& m)	if the projection of l into the xy plane is not a point, the function returns true and assignes the projection to m. Otherwise false is returned.
bool	l.project_xz(*line*& m)	if the projection of l into the xz plane is not a point, the function returns true and assignes the projection to m. Otherwise false is returned.
bool	l.project_yz(*line*& m)	if the projection of l into the yz plane is not a point, the function returns true and assignes the projection to m. Otherwise false is returned.

bool	l.project(*d3_point* p, *d3_point* q, *d3_point* v, *d3_line*& m)	
		if the projection of l into the plane through (p, q, v) is not a point, the function returns true and assignes the projection to m. Otherwise false is returned.
d3_line	l.translate(*double* dx, *double* dy, *double* dz)	
		returns l translated by vector (dx, dy, dz).
d3_line	l.translate(*vector* v)	returns l translated by v. *Precond.* : $v.dim() = 3$.
d3_line	l + *vector* v	returns l translated by vector v.
d3_line	l − *vector* v	returns l translated by vector $-v$.
d3_line	l.reflect(*d3_point* p, *d3_point* q, *d3_point* v)	
		returns l reflected across the plane through (p, q, v).
d3_line	l.reflect(*d3_point* p)	returns l reflected across point p.
d3_line	l.reverse( )	returns l reversed.
vector	l.to_vector( )	returns *point2*( ) − *point1*( ).
bool	l.intersection(*d3_segment* s)	
		decides, whether l and s intersect in a single point.
bool	l.intersection(*d3_segment* s, *d3_point*& p)	
		decides, whether l and s intersect in a single point. If so, the point of intersection is assigned to p.
bool	l.intersection(*d3_line* m)	
		decides, whether l and m intersect.
bool	l.intersection(*d3_line* m, *d3_point*& p)	
		decides, whether l and m intersect in a single point. If so, the point of intersection is assigned to p.
double	l.sqr_dist(*d3_point* p)	returns the square of the distance between l and p.
double	l.distance(*d3_point* p)	returns the distance between l and p.

13.5 Planes (d3_plane)

1. Definition

An instance P of the data type *d3_plane* is an oriented plane in the three-dimensional space $\mathbb{R}^3$. It can be defined by a tripel (a,b,c) of non-collinear points or a single point a and a normal vector v.

#include < *LEDA/d3_plane.h* >

2. Creation

d3_plane *p*;

introduces a variable p of type *d3_plane* initialized to the xy-plane.

d3_plane *p*(*d3_point a*, *d3_point b*, *d3_point c*);

introduces a variable p of type *d3_plane* initialized to the plane through (a, b, c).
Precondition: a, b, and c are not collinear.

d3_plane *p*(*d3_point a*, *vector v*);

introduces a variable p of type *d3_plane* initialized to the plane that contains a with normal vector v.
Precondition: $v.dim() = 3$ and $v.length() > 0$.

d3_plane *p*(*d3_point a*, *d3_point b*);

introduces a variable p of type *d3_plane* initialized to the plane that contains a with normal vector $b - a$.

3. Operations

d3_point	*p*.point1( )	returns the first point of p.
d3_point	*p*.point2( )	returns the second point of p.
d3_point	*p*.point3( )	returns the third point of p.
vector	*p*.normal( )	returns a normal vector of p.
double	*p*.sqr_dist(*d3_point q*)	returns the square of the Euclidean distance between p and q.
double	*p*.distance(*d3_point q*)	returns the Euclidean distance between p and q.
vector	*p*.normal_project(*d3_point q*)	returns the vector pointing from q to its projection on p along the normal direction.

int p.intersection(*const d3_point p1*, *const d3_point p2*, *d3_point& q*)

if the line l through *p1* and *p2* intersects p in a single point this point is assigned to q and the result is 1, if l and p do not intersect the result is 0, and if l is contained in p the result is 2.

int p.intersection(*d3_plane Q*, *d3_point& i1*, *d3_point& i2*)

if p and plane Q intersect in a line L then $(i1, i2)$ are assigned two different points on L and the result is 1, if p and Q do not intersect the result is 0, and if $p = Q$ the result is 2.

d3_plane p.translate(*double dx*, *double dy*, *double dz*)

returns p translated by vector (dx, dy, dz).

d3_plane p.translate(*vector v*) returns p+v, i.e., p translated by vector v.
Precondition: v.dim() = 3.

d3_plane p + *vector v* returns p translated by vector v.

d3_plane p.reflect(*d3_plane Q*) returns p reflected across plane Q.

d3_plane p.reflect(*d3_point q*) returns p reflected across point q.

d3_point p.reflect_point(*d3_point q*)

returns q reflected across plane p.

int p.side_of(*d3_point q*) computes the side of p on which q lies.

bool p.contains(*d3_point q*)

returns true if point q lies on plane p, i.e., $(p.side\_of(q) == 0)$, and false otherwise.

bool p.parallel(*d3_plane Q*)

returns true if planes p and Q are parallel and false otherwise.

ostream& *ostream& O* $\ll$ p writes p to output stream O.

istream& *istream& I* $\gg$ *d3_plane& p*

reads the coordinates of p (six *double* numbers) from input stream I.

Non-Member Functions

int orientation(*d3_plane p*, *d3_point q*)

computes the orientation of $p.sideof(q)$.

13.6 Spheres in 3D-Space (d3_sphere)

1. Definition

An instance of the data type *d3_sphere* is an oriented sphere in 3d space. The sphere is defined by four points *p1*, *p2*, *p3*, *p4* (*d3_points*).

#*include* < *LEDA/d3_sphere.h* >

2. Creation

d3_sphere S(*d3_point p1*, *d3_point p2*, *d3_point p3*, *d3_point p4*);

introduces a variable S of type *d3_sphere*. S is initialized to the sphere through points *p1*, *p2*, *p3*, *p4*.

3. Operations

bool	S.contains(*d3_point p*)	returns true, if p is on the sphere, false otherwise.
bool	S.inside(*d3_point p*)	returns true, if p is inside the sphere, false otherwise.
bool	S.outside(*d3_point p*)	returns true, if p is outside the sphere, false otherwise.
d3_point	S.point1( )	returns *p1*.
d3_point	S.point2( )	returns *p2*.
d3_point	S.point3( )	returns *p3*.
d3_point	S.point4( )	returns *p4*.
bool	S.is_degenerate( )	returns true, if the 4 defining points are coplanar.
d3_point	S.center( )	returns the center of the sphere.
double	S.sqr_radius( )	returns the square of the radius.
double	S.radius( )	returns the radius.
double	S.surface( )	returns the size of the surface.
double	S.volume( )	returns the volume of the sphere.
d3_sphere	S.translate(*vector v*)	returns S translated by vector v.
d3_sphere	S.translate(*double dx*, *double dy*, *double dz*)	returns S translated by vector (dx, dy, dz).

13.7 Simplices in 3d space (d3_simplex)

1. Definition

An instance of the data type *d3_simplex* is a simplex in 3d space. The simplex is defined by four points $p1, p2, p3, p4$ (*d3_points*). We call the simplex degenerate, if the four defining points are coplanar.

#include < *LEDA/d3_simplex.h* >

2. Creation

d3_simplex S(*d3_point* a, *d3_point* b, *d3_point* c, *d3_point* d);
creates the simplex (a, b, c, d).

d3_simplex S; creates the simplex $((0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1))$.

3. Operations

d3_point	S.point1( )	returns $p1$.
d3_point	S.point2( )	returns $p2$.
d3_point	S.point3( )	returns $p3$.
d3_point	S.point4( )	returns $p4$.
int	S.index(*d3_point* p)	returns 1 if $p == p1$, 2 if $p == p2$, 3 if $p == p3$, 4 if $p == p4$, and 0 otherwise.
bool	S.is_degenerate( )	returns true if S is degenerate and false otherwise.
d3_sphere	S.circumscribing_sphere( )	returns a *d3_sphere* through $(p1, p2, p3, p4)$ (precondition: the *d3_simplex* is not degenerate).
bool	S.in_simplex(*d3_point* p)	returns true, if p is contained in the simplex.
bool	S.insphere(*d3_point* p)	returns true, if p lies in the interior of the sphere through $p1, p2, p3, p4$.
double	S.vol( )	returns the signed volume of the simplex.
d3_simplex	S.reflect(*d3_point* p, *d3_point* q, *d3_point* v)	returns S reflected across the plane through (p, q, v).

d3_simplex	S.reflect(*d3_point p*)	returns S reflected across point p.
d3_simplex	S.translate(*vector v*)	returns S translated by vector v. *Precond.*: $v.dim() = 3$.
d3_simplex	S.translate(*double dx*, *double dy*, *double dz*)	returns S translated by vector (dx, dy, dz).
d3_simplex	S + *vector v*	returns S translated by vector v.
d3_simplex	S − *vector v*	returns S translated by vector $-v$.

13.8 Rational Points in 3D-Space (d3_rat_point)

1. Definition

An instance of data type *d3_rat_point* is a point with rational coordinates in the three-dimensional space. A point with cartesian coordinates (a, b, c) is represented by homogeneous coordinates (x, y, z, w) of arbitrary length integers (see 4.1) such that $a = x/w$, $b = y/w$, $c = z/w$ and $w > 0$.

#include < *LEDA/d3_rat_point.h* >

2. Creation

d3_rat_point p; introduces a variable p of type *d3_rat_point* initialized to the point $(0, 0, 0)$.

d3_rat_point p(*rational* a, *rational* b, *rational* c);

introduces a variable p of type *d3_rat_point* initialized to the point (a, b, c).

d3_rat_point p(*integer* a, *integer* b, *integer* c);

introduces a variable p of type *d3_rat_point* initialized to the point (a, b, c).

d3_rat_point p(*integer* x, *integer* y, *integer* z, *integer* w);

introduces a variable p of type *d3_rat_point* initialized to the point with homogeneous coordinates (x, y, z, w) if $w > 0$ and to point $(-x, -y, -z, -w)$ if $w < 0$.
Precondition: $w \neq 0$.

d3_rat_point p(*rat_vector* v);

introduces a variable p of type *d3_rat_point* initialized to the point $(v[0], v[1], v[2])$.
Precondition: : $v.dim(\,) = 3$.

3. Operations

d3_point	p.to_float( )	returns a floating point approximation of p.
rat_vector	p.to_vector( )	returns the vector extending from the origin to p.
integer	p.X( )	returns the first homogeneous coordinate of p.
integer	p.Y( )	returns the second homogeneous coordinate of p.
integer	p.Z( )	returns the third homogeneous coordinate of p.

integer	p.W( )	returns the fourth homogeneous coordinate of p.
double	p.XD( )	returns a floating point approximation of $p.X(\,)$.
double	p.YD( )	returns a floating point approximation of $p.Y(\,)$.
double	p.ZD( )	returns a floating point approximation of $p.Z(\,)$.
double	p.WD( )	returns a floating point approximation of $p.W(\,)$.
rational	p.xcoord( )	returns the x-coordinate of p.
rational	p.ycoord( )	returns the y-coordinate of p.
rational	p.zcoord( )	returns the z-coordinate of p.
rational	p[*int* i]	returns the ith cartesian coordinate of p *Precondition*: $0 \leq i \leq 2$.
double	p.xcoordD( )	returns a floating point approximation of $p.xcoord(\,)$.
double	p.ycoordD( )	returns a floating point approximation of $p.ycoord(\,)$.
double	p.zcoordD( )	returns a floating point approximation of $p.zcoord(\,)$.
integer	p.hcoord(*int* i)	returns the ith homogeneous coordinate of p. *Precondition*: $0 \leq i \leq 3$.
rat_point	p.project_xy( )	returns p projected into the xy-plane.
rat_point	p.project_yz( )	returns p projected into the yz-plane.
rat_point	p.project_xz( )	returns p projected into the xz-plane.
d3_rat_point	p.reflect(*d3_rat_point* p, *d3_rat_point* q, *d3_rat_point* r)	returns p reflected across the plane passing through p, q and r. *Precondition*: p, q and r are not collinear.
d3_rat_point	p.reflect(*d3_rat_point* q)	returns p reflected across point q.
d3_rat_point	p.translate(*rational* dx, *rational* dy, *rational* dz)	returns p translated by vector (dx, dy, dz).
d3_rat_point	p.translate(*integer* dx, *integer* dy, *integer* dz, *integer* dw)	returns p translated by vector $(dx/dw, dy/dw, dz/dw)$.

d3_rat_point p.translate(*rat_vector* v)

returns $p + v$, i.e., p translated by vector v
Precondition: v.dim() = 3.

d3_rat_point p + *rat_vector* v — returns p translated by vector v
Precondition: v.dim() = 3.

d3_rat_point p − *rat_vector* v — returns p translated by vector $-v$
Precondition: v.dim() = 3.

rational p.sqr_dist(*d3_rat_point* q)

returns the squared distance between p and q.

rational p.xdist(*d3_rat_point* q)

returns the x-distance between p and q.

rational p.ydist(*d3_rat_point* q)

returns the y-distance between p and q.

rational p.zdist(*d3_rat_point* q)

returns the z-distance between p and q.

rat_vector $p - q$ — returns the difference vector of the coordinates.

ostream& *ostream&* $O \ll p$ — writes the homogeneous coordinates (x, y, z, w) of p to output stream O.

istream& *istream&* $I \gg$ *d3_rat_point&* p

reads the homogeneous coordinates (x, y, z, w) of p from input stream I.

Non-Member Functions

int orientation(*d3_rat_point* a, *d3_rat_point* b, *d3_rat_point* c, *d3_rat_point* d)

computes the orientation of points a, b, c and d as the sign of the determinant

$$\begin{vmatrix} a_w & b_w & c_w & d_w \\ a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \end{vmatrix}$$

i.e., it returns +1 if point d lies left of the directed plane through a, b, c, 0 if a,b, c and d are coplanar, and -1 otherwise.

int cmp_distances(*d3_rat_point p1*, *d3_rat_point p2*, *d3_rat_point p3*, *d3_rat_point p4*)

compares the distances (*p1*, *p2*) and (*p3*, *p4*). Returns +1 (−1) if distance (*p1*, *p2*) is larger (smaller) than distance (*p3*, *p4*), otherwise 0.

d3_rat_point midpoint(*d3_rat_point a*, *d3_rat_point b*)

returns the midpoint of a and b.

rational volume(*d3_rat_point a*, *d3_rat_point b*, *d3_rat_point c*, *d3_rat_point d*)

computes the signed volume of the simplex determined by a,b,c, and d, positive if $orientation(a, b, c, d) > 0$ and negative otherwise.

bool collinear(*d3_rat_point a*, *d3_rat_point b*, *d3_rat_point c*)

returns true if points a, b, c are collinear, and false otherwise.

bool coplanar(*d3_rat_point a*, *d3_rat_point b*, *d3_rat_point c*, *d3_rat_point d*)

returns true if points a, b, c, d are coplanar and false otherwise.

int side_of_sphere(*d3_rat_point a*, *d3_rat_point b*, *d3_rat_point c*, *d3_rat_point d*, *d3_rat_point e*)

returns +1 (−1) if point e lies on the positive (negative) side of the oriented sphere through points a, b, c, and d, and 0 if e is contained in this sphere.

int region_of_sphere(*d3_rat_point a*, *d3_rat_point b*, *d3_rat_point c*, *d3_rat_point d*, *d3_rat_point x*)

determines whether the point x lies inside (= +1), on (= 0), or outside (= −1) the sphere through points a, b, c, d, (equivalent to $orientation(a, b, c, d) * side\_of\_sphere(a, b, c, d, x)$)
Precondition: $orientation(A) \neq 0$

bool contained_in_simplex(*d3_rat_point a*, *d3_rat_point b*, *d3_rat_point c*, *d3_rat_point d*, *d3_rat_point x*)

determines whether x is contained in the simplex spanned by the points a, b, c, d.
Precondition: a, b, c, d are affinely independent.

bool contained_in_simplex(*array<d3_rat_point> A*, *d3_rat_point x*)

determines whether x is contained in the simplex spanned by the points in A.
Precondition: A must have size ≤ 4 and the points in A must be affinely independent.

bool contained_in_affine_hull(*list*<*d3_rat_point*> *L*, *d3_rat_point* *x*)
determines whether x is contained in the affine hull of the points in L.

bool contained_in_affine_hull(*array*<*d3_rat_point*> *A*, *d3_rat_point* *x*)
determines whether x is contained in the affine hull of the points in A.

int affine_rank(*array*<*d3_rat_point*> *L*)
computes the affine rank of the points in L.

int affine_rank(*array*<*d3_rat_point*> *A*)
computes the affine rank of the points in A.

bool affinely_independent(*list*<*d3_rat_point*> *L*)
decides whether the points in A are affinely independent.

bool affinely_independent(*array*<*d3_rat_point*> *A*)
decides whether the points in A are affinely independent.

bool inside_sphere(*d3_rat_point* *a*, *d3_rat_point* *b*, *d3_rat_point* *c*, *d3_rat_point* *d*, *d3_rat_point* *e*)
returns $true$ if point e lies in the interior of the sphere through points a, b, c, and d, and $false$ otherwise.

bool outside_sphere(*d3_rat_point* *a*, *d3_rat_point* *b*, *d3_rat_point* *c*, *d3_rat_point* *d*, *d3_rat_point* *e*)
returns $true$ if point e lies in the exterior of the sphere through points a, b, c, and d, and $false$ otherwise.

bool on_sphere(*d3_rat_point* *a*, *d3_rat_point* *b*, *d3_rat_point* *c*, *d3_rat_point* *d*, *d3_rat_point* *e*)
returns $true$ if points a, b, c, d, and e lie on a common sphere.

d3_rat_point point_on_positive_side(*d3_rat_point* *a*, *d3_rat_point* *b*, *d3_rat_point* *c*)
returns a point d with $orientation(a, b, c, d) > 0$.

Point Generators

d3_rat_point random_d3_rat_point_in_cube(*int* *maxc*)
returns a point whose coordinates are random integers in $[-maxc .. maxc]$.

void random_d3_rat_points_in_cube(*int n*, *int maxc*, *list*<*d3_rat_point*>& *L*)

returns a list L of n points ... .

d3_rat_point random_d3_rat_point_in_square(*int maxc*)

returns a point whose x and y-coordinates are random integers in $[-maxc \,..\, maxc]$. The z-coordinate is zero. In 2d, this function is equivalent to *random_rat_point_in_cube*.

void random_d3_rat_points_in_square(*int n*, *int maxc*, *list*<*d3_rat_point*>& *L*)

returns a list L of n points ... .

d3_rat_point random_d3_rat_point_in_unit_cube(*int D* = 16383)

returns a point whose coordinates are random rationals of the form i/D where i is a random integer in the range $[0 \,..\, D]$. The default value of D is $2^{14} - 1$.

void random_d3_rat_points_in_unit_cube(*int n*, *int D*, *list*<*d3_rat_point*>& *L*)

returns a list L of n points ... .

void random_d3_rat_points_in_unit_cube(*int n*, *list*<*d3_rat_point*>& *L*)

as above, but the default value of D is used.

d3_rat_point random_d3_rat_point_in_ball(*int R*)

returns a random point with integer coordinates in the ball with radius R centered at the origin.
Precondition: $R \leq 2^{14}$.

void random_d3_rat_points_in_ball(*int n*, *int R*, *list*<*d3_rat_point*>& *L*)

returns a list L of n points ... .

d3_rat_point random_d3_rat_point_in_unit_ball(*int D* = 16383)

returns a point in the unit ball whose coordinates are random rationals of the form i/D where i is a random integer in the range $[0 \,..\, D]$. The default value of D is $2^{14} - 1$.

void random_d3_rat_points_in_unit_ball(*int n*, *int D*, *list*<*d3_rat_point*>& *L*)

returns a list L of n points ... .

void random_d3_rat_points_in_unit_ball(*int n*, *list*<*d3_rat_point*>& *L*)

returns a list L of n points ... . The default value of D is used.

d3_rat_point random_d3_rat_point_in_disc(*int* R)

returns a random point with integer x and y-coordinates in the disc with radius R centered at the origin. The z-coordinate is zero. In 2d this is the same as the function *random_rat_point_in_ball*.
Precondition: $R \leq 2^{14}$.

void random_d3_rat_points_in_disc(*int* n, *int* R, *list*<*d3_rat_point*>& L)

returns a list L of n points ... .

d3_rat_point random_d3_rat_point_on_circle(*int* R)

returns a random point with integer coordinates that lies close to the circle with radius R centered at the origin.

void random_d3_rat_points_on_circle(*int* m, *int* R, *list*<*d3_rat_point*>& L)

returns a list L of n points ... .

d3_rat_point random_d3_rat_point_on_unit_circle(*int* D = 16383)

returns a point close to the unit circle whose coordinates are random rationals of the form i/D where i is a random integer in the range $[0..D]$. The default value of D is $2^{14} - 1$.

void random_d3_rat_points_on_unit_circle(*int* m, *int* D, *list*<*d3_rat_point*>& L)

returns a list L of n points ... .

void random_d3_rat_points_on_unit_circle(*int* m, *list*<*d3_rat_point*>& L)

returns a list L of n points ... . The default value of D is used.

d3_rat_point random_d3_rat_point_on_sphere(*int* R)

returns a point with integer coordinates close to the sphere with radius R centered at the origin.

void random_d3_rat_points_on_sphere(*int* m, *int* R, *list*<*d3_rat_point*>& L)

returns a list L of n points ... .

d3_rat_point random_d3_rat_point_on_unit_sphere(*int* D = 16383)

returns a point close to the unit sphere whose coordinates are random rationals of the form i/D where i is a random integer in the range $[0..D]$. The default value of D is $2^{14} - 1$. In 2d this function is equivalent to *point_on_unit_circle*.

void random_d3_rat_points_on_unit_sphere(*int* m, *int* D, *list*<*d3_rat_point*>& L)

returns a list L of n points ... .

void random_d3_rat_points_on_unit_sphere(*int m*, *list*<*d3_rat_point*>& *L*)

returns a list L of n points ... . The default value of D is used.

d3_rat_point random_d3_rat_point_on_paraboloid(*int maxc*)

returns a point (x, y, z) with x and y random integers in the range $[-maxc \mathinner{..} maxc]$, and $z = 0.004 * (x * x + y * y) - 1.25 * maxc$. The function does not make sense in 2d.

void random_d3_rat_points_on_paraboloid(*int n*, *int maxc*, *list*<*d3_rat_point*>& *L*)

returns a list L of n points ... .

void lattice_d3_rat_points(*int n*, *int maxc*, *list*<*d3_rat_point*>& *L*)

returns a list L of approximately n points. The points have integer coordinates $id/maxc$ for an appropriately chosen d and $-maxc/d \leq i \leq maxc/d$.

void random_d3_rat_points_on_segment(*int n*, *int maxc*, *list*<*d3_rat_point*>& *L*)

generates n points on the diagonal whose coordinates are random integer in the range from $-maxc$ to $maxc$.

13.9 Straight Rational Rays in 3D-Space (d3_rat_ray)

1. Definition

An instance r of the data type *d3_rat_ray* is a directed straight ray defined by two points with rational coordinates in three-dimensional space.

#include < *LEDA/d3_rat_ray.h* >

2. Creation

d3_rat_ray r(*d3_rat_point p1*, *d3_rat_point p2*);

introduces a variable r of type *d3_rat_ray*. r is initialized to the ray starting at point *p1* and going through *p2*.

d3_rat_ray r(*d3_rat_segment s*);

introduces a variable r of type *d3_rat_ray*. r is initialized to *ray*(*s.source*( ), *s.target*( )) .

3. Operations

d3_rat_point	r.source( )	returns the source of r.
d3_rat_point	r.point1( )	returns the source of r.
d3_rat_point	r.point2( )	returns a point on r different from the source.
d3_rat_segment	r.seg( )	returns a segment on r.
bool	r.contains(*d3_rat_point p*)	returns true if p lies on r.
bool	r.contains(*d3_rat_segment s*)	returns true if s lies on r.
bool	r.intersection(*d3_rat_segment s*, *d3_rat_point& inter*)	if s and r intersect in a single point, true is returned and the point of intersection is assigned to inter. Otherwise false is returned.
bool	r.intersection(*d3_rat_ray r*, *d3_rat_point& inter*)	if r and r intersect in a single point, true is returned and the point of intersection is assigned to inter. Otherwise false is returned.

bool r.project_xy(*rat_ray*& m)

if the projection of r into the xy plane is not a point, the function returns true and assignes the projection to m. Otherwise false is returned.

bool r.project_xz(*rat_ray*& m)

if the projection of r into the xz plane is not a point, the function returns true and assignes the projection to m. Otherwise false is returned.

bool r.project_yz(*rat_ray*& m)

if the projection of r into the yz plane is not a point, the function returns true and assignes the projection to m. Otherwise false is returned.

bool r.project(*d3_rat_point* p, *d3_rat_point* q, *d3_rat_point* v, *d3_rat_ray*& m)

if the projection of r into the plane through (p, q, v) is not a point, the function returns true and assignes the projection to m. Otherwise false is returned.

d3_rat_ray r.reverse( ) returns a *rat_ray* starting at r.source() with direction -r.to_vector() .

d3_rat_ray r.translate(*rat_vector* v)

returns r translated by vector v. Precond. : $v.dim(\,)$ $= 3$.

d3_rat_ray r.translate(*rational* dx, *rational* dy, *rational* dz)

returns r translated by vector (dx, dy, dz).

d3_rat_ray r + *rat_vector* v returns r translated by vector v.

d3_rat_ray r − *rat_vector* v returns r translated by vector $-v$.

d3_rat_ray r.reflect(*d3_rat_point* p, *d3_rat_point* q, *d3_rat_point* v)

returns r reflected across the plane through (p, q, v).

d3_rat_ray r.reflect(*d3_rat_point* p)

returns r reflected across point p.

rat_vector r.to_vector( ) returns *point2*( ) − *point1*( ).

13.10 Rational Lines in 3D-Space (d3_rat_line)

1. Definition

An instance l of the data type *d3_rat_line* is a directed straight line in three-dimensional space.

#include < *LEDA/d3_rat_line.h* >

2. Creation

d3_rat_line l(*d3_rat_point* $p1$, *d3_rat_point* $p2$);

introduces a variable l of type *d3_rat_line*. l is initialized to the line through points $p1$, $p2$.

d3_rat_line l(*d3_rat_segment* s);

introduces a variable l of type *d3_rat_line*. l is initialized to the line supporting segment s.

d3_rat_line l; introduces a variable l of type *d3_rat_line*. l is initialized to the line through points $(0, 0, 0, 1)$ and $(1, 0, 0, 1)$.

3. Operations

d3_line	l.to_float( )	returns a floating point approximation of l.
bool	l.contains(*d3_rat_point* p)	returns true if p lies on l.
d3_rat_point	l.point1( )	returns a point on l.
d3_rat_point	l.point2( )	returns a second point on l.
d3_rat_segment	l.seg( )	returns a segment on l.
bool	l.project_xy(*rat_line*& m)	if the projection of l into the xy plane is not a point, the function returns true and assignes the projection to m. Otherwise false is returned.
bool	l.project_xz(*rat_line*& m)	if the projection of l into the xz plane is not a point, the function returns true and assignes the projection to m. Otherwise false is returned.

bool l.project_yz(*rat_line*& m)

if the projection of l into the yz plane is not a point, the function returns true and assignes the projection to m. Otherwise false is returned.

bool l.project(*d3_rat_point* p, *d3_rat_point* q, *d3_rat_point* v, *d3_rat_line*& m)

if the projection of l into the plane through (p, q, v) is not a point, the function returns true and assignes the projection to m. Otherwise false is returned.

d3_rat_line l.translate(*integer* dx, *integer* dy, *integer* dz, *integer* dw)

returns l translated by vector $(dx/dw, dy/dw, dz/dw)$.

d3_rat_line l.translate(*rat_vector* v)

returns l translated by v.
Precond. : $v.dim() = 3$.

d3_rat_line l + *rat_vector* v — returns l translated by vector v.

d3_rat_line l − *rat_vector* v — returns l translated by vector $-v$.

d3_rat_line l.reflect(*d3_rat_point* p, *d3_rat_point* q, *d3_rat_point* v)

returns l reflected across the plane through (p, q, v).

d3_rat_line l.reflect(*d3_rat_point* p)

returns l reflected across point p.

d3_rat_line l.reverse() — returns l reversed.

rat_vector l.to_vector() — returns *point2*( ) − *point1*( ).

bool l.intersection(*d3_rat_segment* s)

decides, whether l and s intersect in a single point.

bool l.intersection(*d3_rat_segment* s, *d3_rat_point*& p)

decides, whether l and s intersect in a single point. If so, the point of intersection is assigned to p.

bool l.intersection(*d3_rat_line* m)

decides, whether l and m intersect in a single point.

bool l.intersection(*d3_rat_line* m, *d3_rat_point*& p)

decides, whether l and m intersect in a single point. If so, the point of intersection is assigned to p.

rational l.sqr_dist(*d3_rat_point* p)

returns the square of the distance between l and p.

13.11 Rational Segments in 3D-Space (d3_rat_segment)

1. Definition

An instance s of the data type *d3_rat_segment* is a directed straight line segment in three-dimensional space, i.e., a line segment connecting two rational points $p, q \in \mathbb{R}^3$. p is called the *source* or start point and q is called the *target* or end point of s. A segment is called *trivial* if its source is equal to its target. If s is not trivial, we use $line(s)$ to denote the straight line containing s.

#include < *LEDA/d3_rat_segment.h* >

2. Creation

d3_rat_segment s(*d3_rat_point p1*, *d3_rat_point p2*);

introduces a variable S of type *d3_rat_segment*. S is initialized to the segment through points *p1*, *p2*.

d3_rat_segment s; introduces a variable S of type *d3_rat_segment*. S is initialized to the segment through points $(0,0,0,1)$ and $(1,0,0,1)$.

3. Operations

d3_segment	*s*.to_float( )	returns a floating point approximation of s.
bool	*s*.contains(*d3_rat_point p*)	decides whether s contains p.
d3_rat_point	*s*.source( )	returns the source point of segment s.
d3_rat_point	*s*.target( )	returns the target point of segment s.
rational	*s*.xcoord1( )	returns the x-coordinate of s.source().
rational	*s*.xcoord2( )	returns the x-coordinate of s.target().
rational	*s*.ycoord1( )	returns the y-coordinate of s.source().
rational	*s*.ycoord2( )	returns the y-coordinate of s.target().
rational	*s*.zcoord1( )	returns the z-coordinate of s.source().
rational	*s*.zcoord2( )	returns the z-coordinate of s.target().
rational	*s*.dx( )	returns *xcoord2*( ) − *xcoord1*( ).

rational	s.dy( )	returns $ycoord2(\,) - ycoord1(\,)$.
rational	s.dz( )	returns $zcoord2(\,) - zcoord1(\,)$.
rat_segment	s.project_xy( )	returns the projection into the xy plane.
rat_segment	s.project_xz( )	returns the projection into the xz plane.
rat_segment	s.project_yz( )	returns the projection into the yz plane.

d3_rat_segment s.project(*d3_rat_point* p, *d3_rat_point* q, *d3_rat_point* v)
returns s projected into the plane through (p, q, v).

d3_rat_segment s.reflect(*d3_rat_point* p, *d3_rat_point* q, *d3_rat_point* v)
returns s reflected across the plane through (p, q, v).

d3_rat_segment s.reflect(*d3_rat_point* p)
returns s reflected across point p.

d3_rat_segment s.reverse( ) returns s reversed.

rat_vector s.to_vector( ) returns $S.target(\,) - S.source(\,)$.

bool s.intersection(*d3_rat_segment* t)
decides, whether s and t intersect in a single point.

bool s.intersection(*d3_rat_segment* t, *d3_rat_point*& p)
decides, whether s and t intersect. If they intersect in a single point, the point is assigned to p

bool s.intersection_of_lines(*d3_rat_segment* t, *d3_rat_point*& p)
If $line(s)$ and $line(t)$ intersect in a single point this point is assigned to p and the result is true, otherwise the result is false.

bool s.is_trivial( ) returns true if s is trivial.

rational s.sqr_length( ) returns the square of the length of s.

d3_rat_segment s.translate(*rat_vector* v)
returns s translated by vector v.
Precond.: $v.dim(\,) = 3$.

d3_rat_segment s.translate(*rational* dx, *rational* dy, *rational* dz)
returns s translated by vector (dx, dy, dz).

d3_rat_segment s.translate(*integer* dx, *integer* dy, *integer* dz, *integer* dw)
returns s translated by vector $(dx/dw, dy/dw, dz/w)$.

d3_rat_segment s + *rat_vector* v returns s translated by vector v.

d3_rat_segment s − *rat_vector* v returns s translated by vector $-v$.

13.12 Rational Planes (d3_rat_plane)

1. Definition

An instance P of the data type *d3_rat_plane* is an oriented rational plane in the three-dimensional space $\mathbb{R}^3$. It can be defined by a tripel (a,b,c) of non-collinear rational points or a single rational point a and a normal vector v.

#include < *LEDA/d3_rat_plane.h* >

2. Creation

d3_rat_plane p; introduces a variable p of type *d3_rat_plane* initialized to the trivial plane.

d3_rat_plane p(*d3_rat_point* a, *d3_rat_point* b, *d3_rat_point* c);

introduces a variable p of type *d3_rat_plane* initialized to the plane through (a, b, c).
Precondition: a, b, and c are not collinear.

d3_rat_plane p(*d3_rat_point* a, *rat_vector* v);

introduces a variable p of type *d3_rat_plane* initialized to the plane that contains a with normal vector v.
Precondition: $v.dim() = 3$ and $v.length() > 0$.

d3_rat_plane p(*d3_rat_point* a, *d3_rat_point* b);

introduces a variable p of type *d3_rat_plane* initialized to the plane that contains a with normal vector $b - a$.

3. Operations

d3_rat_point	p.point1( )	returns the first point of p.
d3_rat_point	p.point2( )	returns the second point of p.
d3_rat_point	p.point3( )	returns the third point of p.
rat_vector	p.normal( )	returns a normal vector of p.
d3_plane	p.to_float( )	returns a floating point approximation of p.
rational	p.sqr_dist(*d3_rat_point* q)	returns the square of the Euclidean distance between p and q.

rat_vector p.normal_project(*d3_rat_point q*)

returns the vector pointing from q to its projection on p along the normal direction.

int p.intersection(*const d3_rat_point p1*, *const d3_rat_point p2*, *d3_rat_point& q*)

if the line l through *p1* and *p2* intersects p in a single point this point is assigned to q and the result is 1, if l and p do not intersect the result is 0, and if l is contained in p the result is 2.

int p.intersection(*d3_rat_plane Q*, *d3_rat_point& i1*, *d3_rat_point& i2*)

if p and plane Q intersect in a line L then $(i1, i2)$ are assigned two different points on L and the result is 1, if p and Q do not intersect the result is 0, and if $p = Q$ the result is 2.

d3_rat_plane p.translate(*rational dx*, *rational dy*, *rational dz*)

returns p translated by vector (dx, dy, dz).

d3_rat_plane p.translate(*integer dx*, *integer dy*, *integer dz*, *integer dw*)

returns p translated by vector $(dx/dw, dy/dw, dz/dw)$.

d3_rat_plane p.translate(*rat_vector v*)

returns p+v, i.e., p translated by vector v.
Precondition: v.dim() = 3.

d3_rat_plane p + *rat_vector v* returns p translated by vector v.

d3_rat_plane p.reflect(*d3_rat_plane Q*)

returns p reflected across plane Q.

d3_rat_plane p.reflect(*d3_rat_point q*)

returns p reflected across point q.

d3_rat_point p.reflect_point(*d3_rat_point q*)

returns q reflected across plane p.

int p.side_of(*d3_rat_point q*)

computes the side of p on which q lies.

bool p.contains(*d3_rat_point q*)

returns true if point q lies on plane p, i.e., $(p.side\_of(q) == 0)$, and false otherwise .

bool p.parallel(*d3_rat_plane Q*)

returns true if planes p and Q are parallel, and false otherwise.

ostream&	*ostream&* $O \ll p$	writes p to output stream O.
istream&	*istream&* $I \gg$ *d3_rat_plane&* p	reads p from input stream I.

Non-Member Functions

int	orientation(*d3_rat_plane* p, *d3_rat_point* q)	computes the orientation of $p.sideof(q)$.

13.13 Rational Spheres (d3_rat_sphere)

1. Definition

An instance of the data type *d3_rat_sphere* is an oriented sphere in 3d space. The sphere is defined by four points *p1*, *p2*, *p3*, *p4* with rational coordinates (*d3_rat_points*).

#include < *LEDA/d3_rat_sphere.h* >

2. Creation

d3_rat_sphere S(*d3_rat_point p1*, *d3_rat_point p2*, *d3_rat_point p3*, *d3_rat_point p4*);

introduces a variable S of type *d3_rat_sphere*. S is initialized to the sphere through points *p1*, *p2*, *p3*, *p4*.

3. Operations

d3_sphere	S.to_float()	returns a floating point approximation of S.
bool	S.contains(*d3_rat_point p*)	returns true, if p is on the sphere, false otherwise.
bool	S.inside(*d3_rat_point p*)	returns true, if p is inside the sphere, false otherwise.
bool	S.outside(*d3_rat_point p*)	returns true, if p is outside the sphere, false otherwise.
d3_rat_point	S.point1()	returns *p1*.
d3_rat_point	S.point2()	returns *p2*.
d3_rat_point	S.point3()	returns *p3*.
d3_rat_point	S.point4()	returns *p4*.
bool	S.is_degenerate()	returns true, if the 4 defining points are coplanar.
d3_rat_point	S.center()	returns the center of the sphere.
rational	S.sqr_radius()	returns the square of the radius.
d3_rat_sphere	S.translate(*rat_vector v*)	translates the sphere by vector v and returns a new *d3_rat_sphere*.

d3_rat_sphere S.translate(*rational r1*, *rational r2*, *rational r3*)

translates the sphere by vector (r1,r2,r3) and returns a new *d3_rat_sphere*.

13.14 Rational Simplices (d3_rat_simplex)

1. Definition

An instance of the data type *d3_rat_simplex* is a simplex in 3d space. The simplex is defined by four points *p1*, *p2*, *p3*, *p4* with rational coordinates (*d3_rat_points*). We call the simplex degenerate, if the four defining points are coplanar.

#include < *LEDA/d3_rat_simplex.h* >

2. Creation

d3_rat_simplex S(*d3_rat_point* a, *d3_rat_point* b, *d3_rat_point* c, *d3_rat_point* d);
creates the simplex (a, b, c, d).

d3_rat_simplex S; creates the simplex $((0,0,0),(1,0,0),(0,1,0),(0,0,1)$).

3. Operations

d3_simplex	S.to_d3_simplex( )	returns a floating point approximation of S.
d3_rat_point	S.point1( )	returns *p1*.
d3_rat_point	S.point2( )	returns *p2*.
d3_rat_point	S.point3( )	returns *p3*.
d3_rat_point	S.point4( )	returns *p4*.
int	S.index(*d3_rat_point* p)	returns 1 if $p ==$ *p1*, 2 if $p ==$ *p2*, 3 if $p ==$ *p3*, 4 if $p ==$ *p4*, 0 otherwise.
bool	S.is_degenerate( )	returns true if S is degenerate and false otherwise.
d3_rat_sphere	S.circumscribing_sphere( )	returns a *d3_rat_sphere* through (*p1*, *p2*, *p3*, *p4*) (precondition: the *d3_rat_simplex* is not degenerate).
bool	S.in_simplex(*d3_rat_point* p)	returns true, if p is contained in the simplex.
bool	S.insphere(*d3_rat_point* p)	returns true, if p lies in the interior of the sphere through *p1*, *p2*, *p3*, *p4*.
rational	S.vol( )	returns the signed volume of the simplex.

d3_rat_simplex S.reflect(*d3_rat_point* p, *d3_rat_point* q, *d3_rat_point* v)

returns S reflected across the plane through (p, q, v).

d3_rat_simplex S.reflect(*d3_rat_point* p)

returns S reflected across point p.

d3_rat_simplex S.translate(*rat_vector* v)

returns S translated by vector v.
Precond. : $v.dim() = 3$.

d3_rat_simplex S.translate(*rational* dx, *rational* dy, *rational* dz)

returns S translated by vector (dx, dy, dz).

d3_rat_simplex S.translate(*integer* dx, *integer* dy, *integer* dz, *integer* dw)

returns S translated by vector $(dx/dw, dy/dw, dz/w)$.

d3_rat_simplex S + *rat_vector* v — returns S translated by vector v.

d3_rat_simplex S − *rat_vector* v — returns S translated by vector $-v$.

13.15 3D Convex Hull Algorithms (d3_hull)

void CONVEX_HULL(*list<d3_rat_point> L*, *GRAPH<d3_rat_point, int>& H*)

CONVEX_HULL takes as argument a list of points and returns the (planar embedded) surface graph H of the convex hull of L. The algorithm is based on an incremental space sweep. The running time is $O(n^2)$ in the worst case and $O(n \log n)$ for most inputs.

bool CHECK_HULL(*GRAPH<d3_rat_point, int> H*)

a checker for convex hulls.

void CONVEX_HULL(*list<d3_point> L*, *GRAPH<d3_point, int>& H*)

a floating point version of *CONVEX_HULL*.

bool CHECK_HULL(*GRAPH<d3_point, int> H*)

a checker for floating-point convex hulls.

Chapter 14

Graphics

This section describes the data types *color*, *window*, *panel*, and *menu*.

14.1 Colors (color)

1. Definition

The data type *color* is the type of all colors available for drawing operations in windows (cf. 14.2). Each color is defined by a triple of integers (r, g, b) with $0 \leq r, g, b \leq 255$, the so-called *rgb-value* of the color. The number of available colors is restricted and depends on the underlying hardware. Colors can be created from rgb-values, from names in a color data base (X11), or from the 16 integer constants (enumeration in <LEDA/incl/impl/x_window.h>) *black*, *white*, *red*, *green*, *blue*, *yellow*, *violet*, *orange*; *cyan*, *brown*, *pink*, *green*2, *blue*2, *grey*1, *grey*2, *grey*3.

#include < *LEDA/color.h* >

2. Creation

color col; creates a color with rgb-value $(0, 0, 0)$ (i.e. black).

color col(*int r*, *int g*, *int b*);

creates a color with rgb-value (r, g, b).

color col(*const char* ∗ *name*);

creates a color and initializes it with the rgb-value of color *name* from the X11 color data base ($(0, 0, 0)$ if *name* does not exist).

color col(*int i*); creates a color and initializes it with one of the 16 predefined colors. Here i is one of the 16 integer constants *black*, *white*, *red*, *green*, *blue*, *yellow*, *violet*, *orange*, *cyan*, *brown*, *pink*, *green*2, *blue*2, *grey*1, *grey*2, or *grey*3.

3. Operations

void	*col*.set_rgb(*int* r, *int* g, *int* b)	sets the rgb-value of *col* to (r, g, b).
void	*col*.get_rgb(*int*& r, *int*& g, *int*& b)	assigns the rgb-value of *col* to (r, g, b).
void	*col*.set_red(*int* r)	changes the r-value of *col* to r.
void	*col*.set_green(*int* g)	changes the g-value of *col* to g.
void	*col*.set_blue(*int* b)	changes the b-value of *col* to b.

14.2 Windows (window)

1. Definition

The data type *window* provides an interface for graphical input and output of basic two-dimensional geometric objects. Application programs using this data type have to be linked with *libW.a* and (on UNIX systems) with the X11 base library *libX11.a* (cf. section 1.6):

CC *prog.c* -lW -lP -lG -lL -lX11 -lm

An instance W of type *window* is an iso-oriented rectangular window in the two-dimensional plane. The default representation of W on the screen is a square of maximal possible edge length positioned in the upper right corner of the display.

In general, a window consists of two rectangular sections, a *panel section* in the upper part and a *drawing section* in the rest of the window. The panel section contains *panel items* such as sliders, choice fields, string items and buttons. They have to be created before the window is opened by special panel operations described in section 14.2.

The drawing section can be used for the output of geometric objects such as points, lines, segments, arrows, circles, polygons, graphs, ... and for the input of all these objects using the mouse input device. All drawing and input operations in the drawing section use a coordinate system that is defined by three parameters of type *double*: $xmin$, the minimal x-coordinate, $xmax$, the maximal x-coordinate, and $ymin$, the minimal y-coordinate. The two parameters $xmin$ and $xmax$ define the scaling factor $scaling$ as $w/(xmax - xmin)$, where w is the width of the drawing section in pixels. The maximal y-coordinate $ymax$ of the drawing section is equal to $ymin + h \cdot scaling$ and depends on the actual shape of the window on the screen. Here, h is the height of the drawing section in pixels.

A list of all window parameters:

1. The *foreground color* parameter (default *black*) defines the default color to be used in all drawing operations. There are 18 predefined colors (enumeration in <LEDA/incl/impl/x_window.h>): $black$, $white$, red, $green$, $blue$, $yellow$, $violet$, $orange$, $cyan$, $brown$, $pink$, $green2$, $blue2$, $grey1$, $grey2$, $grey3$ $ivory$, and *invisible*. Note that all drawing operations have an optional color argument that can be used to override the default foreground color. On monochrome systems all colors different from $white$ are displayed as $black$. The color *invisible* can be used for invisible (transparent) objects.
2. The *background color* parameter (default $white$) defines the default background color (e.g. used by W.clear()).
3. The *text font* parameter defines the name of the font to be used in all text drawing operations.

4. Minimal and maximal coordinates of the drawing area *xmin* (default 0), *xmax* (default 100), *ymin* (default 0).

5. The *grid width* parameter (default 0) defines the width of the grid that is used in the drawing area. A grid width of 0 indicates that no grid is to be used.

6. The *frame label* parameter defines a string to be displayed in the frame of the window.

7. The *show coordinates* flag (default *true*) determines whether the current coordinates of the mouse cursor in the drawing section are displayed in the upper right corner.

8. The *flush output* flag (default *true*) determines whether the graphics output stream is flushed after each draw action.

9. The *line width* parameter (default value 1 pixel) defines the width of all kinds of lines (segments, arrows, edges, circles, polygons).

10. The *line style* parameter defines the style of lines. Possible line styles are *solid* (default), *dashed*, and *dotted*.

11. The *point style* parameter defines the style points are drawn by the *draw_point* operation. Possible point styles are *pixel_point*, *cross_point* (default), *plus_point*, *circle_point*, *disc_point*, *rect_point*, and *box_point*.

12. The *node width* parameter (default value 8 pixels) defines the diameter of nodes created by the draw_node and draw_filled_node operations.

13. The *text mode* parameter defines how text is inserted into the window. Possible values are *transparent* (default) and *opaque*.

14. The *drawing mode* parameter defines the logical operation that is used for setting pixels in all drawing operations. Possible values are *src_mode* (default) and *xor_mode*. In *src_mode* pixels are set to the respective color value, in *xor_mode* the value is bitwise added to the current pixel value.

15. The *redraw function* parameter is a pointer to a function of type void (∗F)(window∗). It is called with a pointer to the corresponding window as argument to redraw (parts of) the window whenever a redrawing is necessary, e.g., if the shape of the window is changed or previously hidden parts of it become visible.

16. The *window delete handler* parameter is a pointer to a function of type void (∗F)(window∗). It is called with a pointer to the corresponding window as argument when the window is to be closed by the window manager (e.g. by pressing the ×-button on Windows-NT systems). The default window delete handler closes the window and terminates the program.

17. The *buttons per line* parameter (default ∞) defines the maximal number of buttons in one line of the panel section.

18. The *precision* parameter (default 16) defines the precision that is used for representing window coordinates, more precisely, all x and y coordinates generated by window input operations are doubles whose mantissa are truncated after $precision - 1$ bits after the binary point.

#include < *LEDA/window.h* >

2. Creation

window W; creates a squared window with maximal possible edge length (minimum of width and height of the display).

window W(*const char* ∗ *label*);
creates a maximal squared window with frame label *label*.

window W(*float w*, *float h*);
creates a window W of physical size w pixels × h pixels .

window W(*float w*, *float h*, *const char* ∗ *label*);
creates a window W of physical size w pixels × h pixels and frame label *label*.

All four variants initialize the coordinates of W to $xmin = 0$, $xmax = 100$ and $ymin = 0$. The *init* operation (see below) can later be used to change the window coordinates and scaling. Please note, that a window is not displayed before the function *display* is called for it.

3. Operations

3.1 Initialization

void W.init(*double x_0*, *double x_1*, *double y_0*)
sets $xmin$ to x_0, $xmax$ to x_1, and $ymin$ to y_0, the scaling factor $scaling$ to $w/(xmax - xmin)$, and $ymax$ to $ymin + h \cdot scaling$. Here w and h are the width and height of the drawing section in pixels.

double W.set_grid_dist(*double d*)
sets the grid distance of W to d.

grid_style W.set_grid_style(*grid_style s*)
sets the grid style of W to s.

int W.set_grid_mode(*int d*) sets the grid distance of W to d pixels.

int	W.set_precision(*int prec*)	sets the precision of W to *prec*.
void	W.init(*double x_0*, *double x_1*, *double y_0*, *int d*, *bool erase = true*)	same as $W.\mathrm{init}(x_0, x_1, y_0)$ followed by W.set_grid_mode(d). If the optional flag *erase* is set to *false* the window will not be erased.
void	W.display( )	opens W and displays it with its right upper corner in the upper right corner of the screen. Note that W.display() has to be called before all drawing operations and that all operations adding panel items to W (cf. 14.2) have to be called before the first call of W.display().
void	W.display(*int x*, *int y*)	opens W and displays it with its left upper corner at position (x, y). Special values for x and y are *window :: min*, *window :: center*, and *window :: max* for positioning W at the minimal or maximal x or y coordinate or centering it in the x or y dimension.
void	W.display(*window& w*, *int x*, *int y*)	opens W and displays it with its left upper corner at position (x, y) relative to the upper left corner of window W_0.

W.open... can be used as a synonym for *W.display...* Note, that the *open* operation for panels (cf. 14.3) is defined slightly different.

void	W.close( )	closes W by removing it from the display.
void	W.clear( )	clears W using the current background color or pixmap, i.e., if W has a background pixmap defined it is tiled with P such that the upper left corner is the tiling origin. Otherwise, it is filled with background color of W.
void	W.clear(*double x_0*, *double y_0*, *double x_1*, *double y_1*)	only clears the rectangular area $(x0, y0, x1, y1)$ of window W using the current background color or pixmap.
void	W.clear(*color c*)	clears W with color c and sets the background color of W to c.

void W.clear(*double xorig*, *double yorig*)
clears W. If a background pixmap is defined the point $(xorig, yorig)$ is used as the origin of tiling.

void W.redraw( ) repaints the drawing area if W has a redraw function.

3.2 Setting parameters

color W.set_color(*color c*) sets the foreground color parameter to c and returns its previous value.

color W.set_fill_color(*color c*) sets the fill color parameter (used by $\ll$ operators) to c and returns its previous value.

color W.set_bg_color(*color c*) sets the background color parameter to c and returns its previous value.

char∗ W.set_bg_pixmap(*char ∗ pr*)
sets the background pixmap to pr and returns its previous value.

int W.set_line_width(*int pix*)
sets the line width parameter to pix pixels and returns its previous value.

line_style W.set_line_style(*line_style s*)
sets the line style parameter to s and returns its previous value.

int W.set_node_width(*int pix*)
sets the node width parameter to pix pixels and returns its previous value.

text_mode W.set_text_mode(*text_mode m*)
sets the text mode parameter to m and returns its previous value.

drawing_mode W.set_mode(*drawing_mode m*)
sets the drawing mode parameter to m and returns its previous value.

int W.set_cursor(*int cursor_id* $= -1$)
sets the mouse cursor of W to *cursor_id*. Here *cursor_id* must be one of the constants predefined in <*X_11/cursorfont.h*> or -1 for the system default cursor. Returns the previous cursor.

void	W.set_show_coordinates(*bool* b)	
		sets the show coordinates flag to b.
void	W.set_frame_label(*string* s)	
		makes s the window frame label.
void	W.set_icon_label(*string* s)	
		makes s the window icon label.
void	W.reset_frame_label( )	restores the standard LEDA frame label.
void	W.set_window_delete_handler(*void* $(*F)$(*window*$*$))	
		sets the window delete handler function parameter to F.
void	W.set_show_coord_handler(*void* $(*F)$(*window*$*$, *double*, *double*))	
		sets the show coordinate handler function parameter to F.
void	W.set_redraw(*void* $(*F)$(*window*$*$))	
		sets the redraw function parameter to F.
void	W.set_redraw(*void* $(*F)$(*window*$*$, *double*, *double*, *double*, *double*) $= 0$)	
		sets the redraw function parameter to F.
void	W.set_bg_redraw(*void* $(*F)$(*window*$*$, *double*, *double*, *double*, *double*) $= 0$)	
		sets the background redraw function parameter to F.
void	W.start_timer(*int* *msec*, *void* $(*F)$(*window*$*$))	
		starts a timer that runs F every *msec* milliseconds with a pointer to W.
void	W.stop_timer( )	stops the timer.
void	W.set_flush(*bool* b)	sets the flush parameter to b.
void	W.set_icon_pixrect(*char* $*$ *pr*)	
		makes *pr* the new icon of W.
void$*$	W.set_client_data(*void* $*$ p, *int* $i = 0$)	
		sets the i-th client data pointer of W to p and returns its previous value. *Precondition*: $i < 16$.

3.3 Reading parameters

int	W.get_line_width( )	returns the current line width.
line_style	W.get_line_style( )	returns the current line style.

int	W.get_node_width( )	returns the current node width.
text_mode	W.get_text_mode( )	returns the current text mode.
drawing_mode	W.get_mode( )	returns the current drawing mode.
int	W.get_cursor( )	returns the id of the current cursor, i.e, one of the constants predefined in ¡X11/cursorfont.h¿ or -1 for the default cursor.
double	W.xmin( )	returns the minimal x-coordinate of the drawing area of W.
double	W.ymin( )	returns the minimal y-coordinate of the drawing area of W.
double	W.xmax( )	returns the maximal x-coordinate of the drawing area of W.
double	W.ymax( )	returns the maximal y-coordinate of the drawing area of W.
double	W.scale( )	returns the scaling factor of the drawing area of W, i.e. the number of pixels of a unit length line segment.
double	W.get_grid_dist( )	returns the width of the current grid (zero if no grid is used).
grid_style	W.get_grid_style( )	returns the current grid style.
int	W.get_grid_mode( )	returns the width of the current grid in pixels (zero if no grid is used).
void∗	W.get_client_data(*int* $i = 0$)	returns the i-th client data pointer of of W. *Precondition*: $i < 16$.
GraphWin∗	W.get_graphwin( )	returns a pointer to the *GraphWin* (see 14.6) that uses W as its display window or *NULL* if W is not used by any *GraphWin*.
( )		returns a pointer to the *GeoWin* (see **??**) that uses W as its display window or *NULL* if W is not used by any *GeoWin*.
int	W.width( )	returns the width of W in pixels.
int	W.height( )	returns the height of W in pixels.
int	W.xpos( )	returns the x-coordinate of the upper left corner of the frame of W.

int	W.ypos( )	returns the y-coordinate of the upper left corner of the frame of W.
int	W.get_state( )	returns the state of W.
void	W.set_state(*int stat*)	sets the state of W to *stat*.
bool	W.contains(*point p*)	returns true if p lies in the drawing area.

3.4 Drawing Operations

All drawing operations have an optional color argument at the end of the parameter list. If this argument is omitted the current foreground color (cf. section 14.2) of W is used.

3.4.1 Drawing points

void W.draw_point(*double x*, *double y*, *color c = window::fgcol*)

draws the point (x, y) (a cross of two short segments).

void W.draw_point(*point p*, *color c = window::fgcol*)

draws point p.

void W.draw_pixel(*double x*, *double y*, *color c = window::fgcol*)

sets the color of the pixel at position (x, y) to c.

void W.draw_pixel(*point p*, *color c = window::fgcol*)

sets the color of the pixel at position p to c.

void W.draw_pixels(*list<point> L*, *color c = window::fgcol*)

sets the color of all pixels in L to c.

void W.draw_pixels(*int n*, *double * xcoord*, *double * ycoord*, *color c = window::fgcol*)

draws all pixels $(xcoord[i], ycoord[i])$ for $0 \le i \le n - 1$.

3.4.2 Drawing line segments

void W.draw_segment(*double x_1*, *double y_1*, *double x_2*, *double y_2*, *color c = window::fgcol*)

draws a line segment from (x_1, y_1) to (x_2, y_2).

void W.draw_segment(*point p*, *point q*, *color c = window::fgcol*)

draws a line segment from point p to point q.

void W.draw_segment(*segment s*, *color c = window::fgcol*)

draws line segment s.

void W.draw_segment(*point p*, *point q*, *line l*, *color c = window::fgcol*)

draws the part of the line l between p and q. This version of *draw_segment* should be used if p or q may lie far outside W. *Precondition*: p and q lie on l or at least close to l.

void W.draw_segments(*list*<*segment*> L, *color* $c = window::fgcol$)

draws all segments in L.

3.4.3 Drawing lines

void W.draw_line(*double* x_1, *double* y_1, *double* x_2, *double* y_2, *color* $c = window::fgcol$)

draws a straight line passing through points (x_1, y_1) and (x_2, y_2).

void W.draw_line(*point* p, *point* q, *color* $c = window::fgcol$)

draws a straight line passing through points p and q.

void W.draw_line(*segment* s, *color* $c = window::fgcol$)

draws the line supporting segment s.

void W.draw_line(*line* l, *color* $c = window::fgcol$)

draws line l.

void W.draw_hline(*double* y, *color* $c = window::fgcol$)

draws a horizontal line with y-coordinate y.

void W.draw_vline(*double* x, *color* $c = window::fgcol$)

draws a vertical line with x-coordinate x.

3.4.4 Drawing Rays

void W.draw_ray(*double* x_1, *double* y_1, *double* x_2, *double* y_2, *color* $c = window::fgcol$)

draws a ray starting in (x_1, y_1) and passing through (x_2, y_2).

void W.draw_ray(*point* p, *point* q, *color* $c = window::fgcol$)

draws a ray starting in p and passing through q.

void W.draw_ray(*segment* s, *color* $c = window::fgcol$)

draws a ray starting in $s.source()$ containing s.

void W.draw_ray(*ray* r, *color* $c = window::fgcol$)

draws ray r.

void W.draw_ray(*point* p, *point* q, *line* l, *color* $c = window::fgcol$)

draws the part of the line l on the ray with source p and passing through q. This version of *draw_ray* should be used if p may lie far outside W. *Precondition*: p and q lie on l or at least close to l.

3.4.5 Drawing Arcs and Curves

void W.draw_arc(*point* p, *point* q, *point* r, *color* c = *window*::*fgcol*)

draws a circular arc starting in p passing through q and ending in r.

void W.draw_bezier(*list*<*point*> C, *int* n, *color* c = *window*::*fgcol*)

draws the bezier curve with control polygon C by a polyline with n points.

void W.draw_spline(*list*<*point*> L, *int* n, *color* c = *window*::*fgcol*)

draws a spline curve through the points of L. Each segment is approximated by a polyline with m points.

void W.draw_closed_spline(*list*<*point*> L, *int* n, *color* c = *window*::*fgcol*)

draws a *closed* spline through the points of L.

void W.draw_spline(*polygon* P, *int* n, *color* c = *window*::*fgcol*)

draws a *closed* spline through the vertices of P.

3.4.6 Drawing arrows

void W.draw_arrow(*double* x_1, *double* y_1, *double* x_2, *double* y_2, *color* c = *window*::*fgcol*)

draws an arrow pointing from (x_1, y_1) to (x_2, y_2).

void W.draw_arrow(*point* p, *point* q, *color* c = *window*::*fgcol*)

draws an arrow pointing from point p to point q.

void W.draw_arrow(*segment* s, *color* = *window*::*fgcol*)

draws an arrow pointing from s.start() to s.end().

void W.draw_polyline_arrow(*list*<*point*> *lp*, *color* c = *window*::*fgcol*)

draws a polyline arrow with vertex sequence *lp*.

void W.draw_arc_arrow(*point* p, *point* q, *point* r, *color* c = *window*::*fgcol*)

draws a circular arc arrow starting in p passing through q and ending in r.

void W.draw_bezier_arrow(*list*<*point*> C, *int* n, *color* c = *window*::*fgcol*)

draws the bezier curve with control polygon C by a polyline with n points, the last segment is drawn as an arrow.

void W.draw_spline_arrow(*list*<*point*> L, *int* n, *color* c = *window*::*fgcol*)

draws a spline curve through the points of L. Each segment is approximated by a polyline with n points. The last segment is drawn as an arrow.

point W.draw_arrow_head(*point* p, *double* *dir*, *color* c = *window*::*fgcol*)

draws an arrow head at position p pointing to direction *dir*, where *dir* is an angle from $[0, 2\pi]$.

3.4.7 Drawing circles

void W.draw_circle(*double x*, *double y*, *double r*, *color c = window::fgcol*)

draws the circle with center (x, y) and radius r.

void W.draw_circle(*point p*, *double r*, *color c = window::fgcol*)

draws the circle with center p and radius r.

void W.draw_circle(*circle C*, *color c = window::fgcol*)

draws circle C.

void W.draw_ellipse(*double x*, *double y*, *double r_1*, *double r_2*, *color c = window::fgcol*)

draws the ellipse with center (x, y) and radii r_1 and r_2.

void W.draw_ellipse(*point p*, *double r_1*, *double r_2*, *color c = window::fgcol*)

draws the ellipse with center p and radii r_1 and r_2.

3.4.8 Drawing discs

void W.draw_disc(*double x*, *double y*, *double r*, *color c = window::fgcol*)

draws a filled circle with center (x, y) and radius r.

void W.draw_disc(*point p*, *double r*, *color c = window::fgcol*)

draws a filled circle with center p and radius r.

void W.draw_disc(*circle C*, *color c = window::fgcol*)

draws filled circle C.

void W.draw_filled_ellipse(*double x*, *double y*, *double r_1*, *double r_2*,
color c = window::fgcol)

draws a filled ellipse with center (x, y) and radii r_1 and r_2.

void W.draw_filled_ellipse(*point p*, *double r_1*, *double r_2*, *color c = window::fgcol*)

draws a filled ellipse with center p and radii r_1 and r_2.

3.4.9 Drawing polygons

void W.draw_polyline(*list<point> lp*, *color c = window::fgcol*)

draws a polyline with vertex sequence *lp*.

void W.draw_polyline(*int n*, *double * xc*, *double * yc*, *color c = window::fgcol*)

draws a polyline with vertex sequence $(xc[0], yc[0]), \ldots, (xc[n-1], yc[n-1])$.

void W.draw_polygon(*list<point> lp*, *color c = window::fgcol*)

draws the polygon with vertex sequence *lp*.

void W.draw_oriented_polygon(*list<point> lp*, *color c = window::fgcol*)

draws the polygon with vertex sequence *lp* and indicates the orientation by an arrow.

void W.draw_polygon(*polygon* P, *color* $c = window::fgcol$)
draws polygon P.

void W.draw_oriented_polygon(*polygon* P, *color* $c = window::fgcol$)
draws polygon P and indicates the orientation by an arrow.

void W.draw_filled_polygon(*list*<*point*> *lp*, *color* $c = window::fgcol$)
draws the filled polygon with vertex sequence *lp*.

void W.draw_filled_polygon(*polygon* P, *color* $c = window::fgcol$)
draws filled polygon P.

void W.draw_polygon(*gen_polygon* P, *color* $c = window::fgcol$)
draws polygon P.

void W.draw_oriented_polygon(*gen_polygon* P, *color* $c = window::fgcol$)
draws polygon P and indicates the orientation by an arrow.

void W.draw_filled_polygon(*gen_polygon* P, *color* $c = window::fgcol$)
draws filled polygon P.

void W.draw_rectangle(*double* *x_0*, *double* *y_0*, *double* *x_1*, *double* *y_1*, *color* $= window::fgcol$)
draws a rectangle with lower left corner (x_0, y_0) and upper right corner (x_1, y_1).
Precondition: $x_0 < x_1$ and $y_0 < y_1$.

void W.draw_rectangle(*point* p, *point* q, *color* $= window::fgcol$)
draws a rectangle with lower left corner p and upper right corner q.
Precondition: $p < q$.

void W.draw_rectangle(*rectangle* R, *color* $= window::fgcol$)
draws rectangle R.

void W.draw_box(*double* *x_0*, *double* *y_0*, *double* *x_1*, *double* *y_1*, *color* $c = window::fgcol$)
draws a filled rectangle with lower left corner (x_0, y_0) and upper right corner (x_1, y_1).
Precondition: $x_0 < x_1$ and $y_0 < y_1$.

void W.draw_filled_rectangle(*point* p, *point* q, *color* $= window::fgcol$)
draws a filled rectangle with lower left corner p and upper right corner q.
Precondition: $p < q$.

void W.draw_filled_rectangle(*rectangle R*, *color* = *window*::*fgcol*)

draws rectangle R.

void W.draw_box(*point p*, *point q*, *color c* = *window*::*fgcol*)

same as *draw_filled_rectangle*(p, q, c).

void W.draw_box(*rectangle R*, *color c* = *window*::*fgcol*)

same as *draw_filled_rectangle*(p, q, c).

void W.draw_roundrect(*double x_0*, *double y_0*, *double x_1*, *double y_1*, *double rndness*, *color col* = *window*::*fgcol*)

draws a rectangle (x_0, y_0, x_1, y_1) with round corners. The *rndness* argument must be real number in the intervall $[0, 1]$ and defined the "roundness" of the rectangle.

void W.draw_rectangle(*point p*, *point q*, *double rndness*, *color col* = *window*::*fgcol*)

draws a round rectangle with lower left corner p, upper right corner q, and roundness *rndness*.

void W.draw_roundbox(*double x_0*, *double y_0*, *double x_1*, *double y_1*, *double rndness*, *color col* = *window*::*fgcol*)

draws a filled rectangle (x_0, y_0, x_1, y_1) with round corners. The *rndness* argument must be real number in the intervall $[0, 1]$ and defined the "roundness" of the rectangle.

void W.draw_roundbox(*point p*, *point q*, *double rndness*, *color col* = *window*::*fgcol*)

draws a round filled rectangle with lower left corner p, upper right corner q, and roundness *rndness*.

void W.draw_triangle(*point a*, *point b*, *point c*, *color* = *window*::*fgcol*)

draws triangle (a, b, c).

void W.draw_filled_triangle(*point a*, *point b*, *point c*, *color* = *window*::*fgcol*)

draws filled triangle (a, b, c).

3.4.10 Drawing functions

void W.plot_xy(*double x_0*, *double x_1*, *win_draw_func F*, *color c* = *window*::*fgcol*)

draws the graph of function F in the x-range $[x_0, x_1]$, i.e., all pixels (x, y) with $y = F(x)$ and $x_0 \leq x \leq x_1$.

void W.plot_yx(*double y_0*, *double y_1*, *win_draw_func F*, *color c* = *window*::*fgcol*)

draws the graph of function F in the y-range $[y_0, y_1]$, i.e., all pixels (x, y) with $x = F(y)$ and $y_0 \leq y \leq y_1$.

3.4.11 Drawing text

void W.draw_text(*double x*, *double y*, *string s*, *color c* = *window*::*fgcol*)

writes string s starting at position (x, y).

void W.draw_text(*point* p, *string* s, *color* $c = window::fgcol$)
writes string s starting at position p.

void W.draw_ctext(*double* x, *double* y, *string* s, *color* $c = window::fgcol$)
writes string s centered at position (x, y).

void W.draw_ctext(*point* p, *string* s, *color* $c = window::fgcol$)
writes string s centered at position p.

void W.draw_ctext(*string* s, *color* $c = window::fgcol$)
writes string s centered in window W.

double W.text_box(*double* x_0, *double* x_1, *double* y, *string* s, *bool* $draw = true$)
formats and writes string s into a box with its left border at x-coordinate $x0$, its right border at $x1$, and its upper border at y-coordinate y. Some LaTeX-like formatting commands can be used: `\bf`, `\tt`, `\rm`, `\n`, `\c`, `\<color>`, ... returns y-coordinate of lower border of box. If the optional last parameter *draw* is set to *false* no drawing takes place and only the lower y-coordinate of the box is computed.

void W.text_box(*string* s) as above with $x\_0 = W.xmin(\,)$, $x\_1 = W.xmax(\,)$, and $y = W.ymax(\,)$.

void W.message(*string* s) displays the message s (each call adds a new line).

void W.del_message( ) deletes the text written by all previous message operations.

3.4.12 Drawing nodes

Nodes are represented by circles of diameter *node_width*.

void W.draw_node(*double* x_0, *double* y_0, *color* $c = window::fgcol$)
draws a node at position (x_0, y_0).

void W.draw_node(*point* p, *color* $c = window::fgcol$)
draws a node at position p.

void W.draw_filled_node(*double* x_0, *double* y_0, *color* $c = window::bgcol$)
draws a filled node at position (x_0, y_0).

void W.draw_filled_node(*point* p, *color* $c = window::bgcol$)
draws a filled node at position p.

void W.draw_text_node(*double* x, *double* y, *string* s, *color* $c = window::bgcol$)
draws a node with label s at position (x, y).

void W.draw_text_node(*point p*, *string s*, *color c* = *window*::*bgcol*)

draws a node with label s at position p.

void W.draw_int_node(*double x*, *double y*, *int i*, *color c* = *window*::*bgcol*)

draws a node with integer label i at position (x, y).

void W.draw_int_node(*point p*, *int i*, *color c* = *window*::*bgcol*)

draws a node with integer label i at position p.

3.4.13 Drawing edges

Edges are drawn as straigth line segments or arrows with a clearance of *node_width*/2 at each end.

void W.draw_edge(*double x_1*, *double y_1*, *double x_2*, *double y_2*, *color c* = *window*::*fgcol*)

draws an edge from (x_1, y_1) to (x_2, y_2).

void W.draw_edge(*point p*, *point q*, *color c* = *window*::*fgcol*)

draws an edge from p to q.

void W.draw_edge(*segment s*, *color c* = *window*::*fgcol*)

draws an edge from s.start() to s.end().

void W.draw_edge_arrow(*double x_1*, *double y_1*, *double x_2*, *double y_2*, *color c* = *window*::*fgcol*)

draws a directed edge from (x_1, y_1) to (x_2, y_2).

void W.draw_edge_arrow(*point p*, *point q*, *color c* = *window*::*fgcol*)

draws a directed edge from p to q.

void W.draw_edge_arrow(*segment s*, *color c* = *window*::*fgcol*)

draws a directed edge from s.start() to s.end().

3.4.14 Bitmaps and Pixrects

*char** W.create_bitmap(*int w*, *int h*, *char * bm_data*)

creates a bitmap (monochrome pixrect) of width w, height h, from the bits in *data*.

*char** W.create_pixrect(*const char **xpm_str*)

creates a pixrect from the **xpm** data string *xpm_str*.

*char** W.create_pixrect(*string xpm_file*)

creates a pixrect from the **xpm** file *xpm_file*.

char∗ W.create_pixrect(*int* w, *int* h, *char* ∗ *bm_data*, *int fg* = *window*::*fgcol*, *int bg* = *window*::*bgcol*)

creates a pixrect of width w, height h, foreground color fg, and background color bg from bitmap *data*.

char∗ W.get_pixrect(*double* x_1, *double* y_1, *double* x_2, *double* y_2)

creates a color pixrect of width $w = x_2 - x_1$, height $h = y2 - y1$, and copies all pixels from the rectangular area (x_1, x_2, y_1, y_2) of W into it.

char∗ W.get_window_pixrect( )

creates a pixrect copy of the current window contents.

int W.get_width(*char* ∗ *pr*) returns the width (number of pixels in a row) of pixrect *pr*.

int W.get_height(*char* ∗ *pr*)

returns the height (number of pixels in a column) of pixrect *pr*.

void W.put_pixrect(*double* x, *double* y, *char* ∗ *pr*)

copies the contents of pixrect *pr* with lower left corner at position (x, y) into W.

void W.put_pixrect(*point* p, *char* ∗ *pr*)

copies the contents of pixrect *pr* with lower left corner at position p into W.

void W.center_pixrect(*double* x, *double* y, *char* ∗ *pr*)

copies the contents of pixrect *pr* into W such that its center lies on position (x, y).

void W.put_pixrect(*char* ∗ *pr*)

copies the contents of pixrect *pr* with lower left corner at position $(W.xmin(\,), W.ymin(\,))$ into W.

void W.put_bitmap(*double* x, *double* y, *char* ∗ *bm*, *color* c = *window*::*fgcol*)

draws all pixels corresponding to 1-bits in *bm* with color c, here the lower left corner of *bm* corresponds to the pixel at position (x, y) in W.

void W.put_pixrect(*double* x, *double* y, *char* ∗ *pr*, *int* x_0, *int* y_0, *int* w, *int* h)

copies (pixel) rectangle $(x0, y0, x0 + w, y0 + h)$ of *pr* with lower left corner at position (x, y) into W.

void W.del_bitmap(*char* ∗ *bm*)

destroys bitmap *bm*.

void W.del_pixrect(*char* ∗ *pr*)

destroys pixrect *pr*.

void W.copy_rect(*double* x_0, *double* y_0, *double* x_1, *double* y_1, *double* x, *double* y)

copies all pixels of rectangle (x_0, y_0, x_1, y_1) into the rectangle $(x, y, x+w, y+h)$, where $w = x_1 - x_0$ and $h = y_1 - y_0$.

3.4.15 Buffering

void W.start_buffering( )

starts buffering mode for W. If W has no associated buffer a buffer pixrect *buf* of the same size as the current drawing area of W is created. All subsequent drawing operations draw into *buf* instead of W until buffering mode is ended by calling *W.stop_buffering*( ).

void W.flush_buffer( )

copies the contents of the buffer pixrect into the drawing area of W.

void W.flush_buffer(*double* dx, *double* dy)

copies the contents of the buffer pixrect translated by vector (dx, dy) into the drawing area of W.

void W.flush_buffer(*double* x_0, *double* y_0, *double* x_1, *double* y_1)

copies the contents of rectangle $(x0, y0, x1, y1)$ of the buffer pixrect into the corresponding rectangle of the drawin area.

void W.flush_buffer(*double* dx, *double* dy, *double* x_0, *double* y_0, *double* x_1, *double* y_1)

copies the contents of rectangle $(x0, y0, x1, y1)$ of the buffer pixrect into the corresponding rectangle of the drawin area translated by vector (dx, dy).

void W.stop_buffering( )

ends buffering mode.

void W.stop_buffering(*char* ∗ & *prect*)

ends buffering mode and returns the current buffer pixrect in *prect*.

3.4.16 Clipping

void W.set_clip_rectangle(*double* x_0, *double* y_0, *double* x_1, *double* y_1)

sets the clipping region of W to rectangle $(x0, y0, x1, y1)$.

void W.set_clip_ellipse(*double* x_0, *double* y_0, *double* r_1, *double* r_2)

sets the clipping region of W to ellipse with center (x, y) and horizontal radius $r1$ and vertical radius $r2$.

void W.reset_clipping( )

restores the clipping region to the entire drawing area of W.

3.5 Input

The main input operation for reading positions, mouse clicks, and buttons from a window W is the operation W.read_mouse(). This operation is blocking, i.e., waits for a button to be pressed which is either a “real” button on the mouse device pressed inside the drawing area of W or a button in the panel section of W. In both cases, the number of the selected button is returned. Mouse buttons have pre-defined numbers MOUSE_BUTTON(1) for the left button, MOUSE_BUTTON(2) for the middle button, and MOUSE_BUTTON(3) for the right button. The numbers of the panel buttons can be defined by the user. If the selected button has an associated action function or sub-window this function/window is executed/opened (cf. 14.2 for details).

There is also a non-blocking version W.get_mouse() which returns the constant NO_BUTTON if no button was pressed.

The window data type also provides two more general input operations W.read_event() and W.get_event() for reading events. They return the event type (enumeration in <LEDA/impl/x_window.h>), the value of the event, the position of the event in the drawing section, and a time stamp of the event.

3.5.1 Read Mouse

int W.read_mouse( )

waits until a mouse button is pressed inside of the drawing area or until a button of the panel section is selected. In both cases, the number n of the button is returned which is one of the predefined constants MOUSE_BUTTON(i) with $i \in \{1, 2, 3\}$ for mouse buttons and a user defined value (defined when adding the button with W.button()) for panel buttons. If the button has an associated action function this function is called with parameter n. If the button has an associated window M it is opened and M.read_mouse() is returned.

int W.read_mouse(*double*& x, *double*& y)

If a button is pressed inside the drawing area the current position of the cursor is assigned to (x, y). The operation returns the number of the pressed button (see W.read_mouse().)

int W.read_mouse(*point*& p)

If a button is pressed inside the drawing area the current position of the cursor is assigned to p. The operation returns the number of the pressed button (see W.read_mouse().)

int W.read_mouse_seg(*double* x_0, *double* y_0, *double*& x, *double*& y)

displays a line segment from (x_0, y_0) to the current cursor position until a mouse button is pressed inside the drawing section of W. When a button is pressed the current position is assigned to (x, y) and the number of the pressed button is returned.

int W.read_mouse_seg(*point* p, *point*& q)

displays a line segment from p to the current cursor position until a mouse button is pressed inside the drawing section of W. When a button is pressed the current position is assigned to q and the number of the pressed button is returned.

int W.read_mouse_rect(*double* x_0, *double* y_0, *double*& x, *double*& y)

displays a rectangle with diagonal from (x_0, y_0) to the current cursor position until a mouse button is pressed inside the drawing section of W. When a button is pressed the current position is assigned to (x, y) and the number of the pressed button is returned.

int W.read_mouse_rect(*point* p, *point*& q)

displays a rectangle with diagonal from p to the current cursor position until a mouse button is pressed inside the drawing section of W. When a button is pressed the current position is assigned to q and the number of the pressed button is returned.

int W.read_mouse_circle(*double* x_0, *double* y_0, *double*& x, *double*& y)

displays a circle with center (x_0, y_0) passing through the current cursor position until a mouse button is pressed inside the drawing section of W. When a button is pressed the current position is assigned to (x, y) and the number of the pressed button is returned.

int W.read_mouse_circle(*point* p, *point*& q)

displays a circle with center p passing through the current cursor position until a mouse button is pressed inside the drawing section of W. When a button is pressed the current position is assigned to q and the number of the pressed button is returned.

int W.get_mouse( ) non-blocking read operation, i.e., if a button was pressed its number is returned, otherwise the constant NO_BUTTON is returned.

int W.get_mouse(*double*& x, *double*& y)

if a mouse button was pressed the corresponding position is assigned to (x, y) and the button number is returned, otherwise the constant NO_BUTTON is returned.

int W.get_mouse(*point*& p) if a mouse button was pressed the corresponding position is assigned to p and the button number is returned, otherwise the constant NO_BUTTON is returned.

int W.read_mouse(*double x_0*, *double y_0*, *int timeout1*, *int timeout2*, *bool*& *double_click*, *bool*& *drag*)

...

int W.read_mouse(*point p*, *int timeout1*, *int timeout2*, *bool*& *double_click*, *bool*& *drag*)

...

3.5.2 Events

int W.read_event(*int*& *val*, *double*& x, *double*& y, *unsigned long*& t)

waits for next event in window W and returns it. Assigns the button or key to *val*, the position in W to (x, y), and the time stamp of the event to t. Possible events are (cf. <LEDA/impl/x_window.h>): key_press_event, key_release_event, button_press_event, button_release_event, configure_event, motion_event, destroy_event.

int W.read_event(*int*& *val*, *double*& x, *double*& y, *unsigned long*& t, *int timeout*)

as above, but waits only *timeout* milliseconds; if no event occured the special event *no_event* is returned.

int W.read_event(*int*& *val*, *double*& x, *double*& y)

waits for next event in window W and returns it. Assigns the button or key to *val* and the position in W to (x, y).

int W.get_event(*int*& *val*, *double*& x, *double*& y)

if there is an event for window W in the event queue a *W.read_event* operation is performed, otherwise the integer constant *no_event* is returned.

bool W.shift_key_down( ) returns *true* if a *shift* key was pressed during the last handled mouse button event.

bool W.ctrl_key_down( ) returns *true* if a *ctrl* key was pressed during the last handled mouse button event.

bool W.alt_key_down( ) returns *true* if an *alt* key was pressed during the last handled mouse button event.

int W.button_press_time( ) returns the time-stamp (in msec) of the last button press event.

int W.button_release_time( )

returns the time-stamp (in msec) of the last button release event.

3.6 Panel Input

The operations listed in this section are useful for simple input of strings, numbers, and Boolean values.

bool W.confirm(*string* s) displays string s and asks for confirmation. Returns true iff the answer was "yes".

void W.acknowledge(*string* s)

displays string s and asks for acknowledgement.

int W.read_panel(*string* h, *int* n, *string* $* S$)

displays a panel with header h and an array of n buttons with labels $S[0..n-1]$, returns the index of the selected button.

int W.read_vpanel(*string* h, *int* n, *string* $* S$)

like read_panel with vertical button layout.

string W.read_string(*string* p) displays a panel with prompt p for string input, returns the input.

double W.read_real(*string* p) displays a panel with prompt p for double input returns the input.

int W.read_int(*string* p) displays a panel with prompt p for integer input, returns the input.

3.7 Input and output operators

For input and output of basic geometric objects in the plane such as points, lines, line segments, circles, and polygons the << and >> operators can be used. Similar to C++input streams windows have an internal state indicating whether there is more input to read or not. Its initial value is true and it is turned to false if an input sequence is terminated by clicking the right mouse button (similar to ending stream input by the eof character). In conditional statements, objects of type *window* are automatically converted to boolean by returning this internal state. Thus, they can be used in conditional statements in the same way as C++input streams. For example, to read a sequence of points terminated by a right button click, use " **while** ($W >> p$) { ... } ".

3.7.1 Output

window& $W \ll$ *point* p like W.draw_point(p).

window&	$W \ll$ *segment s*	like W.draw_segment(s).
window&	$W \ll$ *ray r*	like W.draw_ray(r).
window&	$W \ll$ *line l*	like W.draw_line(l).
window&	$W \ll$ *circle C*	like W.draw_circle(C).
window&	$W \ll$ *polygon P*	like W.draw_polygon(P).
window&	$W \ll$ *gen_polygon P*	like W.draw_polygon(P).
window&	$W \ll$ *rectangle R*	like W.draw_rectangle(R).

3.7.2 Input

window&	$W \gg$ *point& p*	reads a point p: clicking the left button assigns the current cursor position to p.
window&	$W \gg$ *segment& s*	reads a segment s: use the left button to input the start and end point of s.
window&	$W \gg$ *ray& r*	reads a ray r: use the left button to input the start point and a second point on r.
window&	$W \gg$ *line& l*	reads a line l: use the left button to input two different points on l.
window&	$W \gg$ *circle& C*	reads a circle C: use the left button to input the center of C and a point on C.
window&	$W \gg$ *polygon& P*	reads a polygon P: use the left button to input the sequence of vertices of P, end the sequence by clicking the right button.
list<point>	W.read_polygon( )	as above, however, returns list of vertices.

As long as an input operation has not been completed the last read point can be erased by simultaneously pressing the shift key and the left mouse button.

3.8 Non-Member Functions

int read_mouse(*window* $*$ & w, *double*& x, *double*& y)

waits for mouse input, assigns a pointer to the corresponding window to w and the position in $*w$ to (x, y) and returns the pressed button.

int	get_mouse(*window* ∗ & *w*, *double*& *x*, *double*& *y*)	
		non-blocking variant of *read_mouse*, returns NO_BUTTON if no button was pressed.
void	put_back_event( )	puts last handled event back to the event queue.

3.9 Panel Operations

The panel section of a window is used for displaying text messages and for updating the values of variables. It consists of a list of panel items and a list of buttons. The operations in this section add panel items or buttons to the panel section of W. Note that they have to be called before the window is displayed the first time.

In general, a panel item consists of a string label and an associated variable of a certain type (int, bool, string, double, color). The value of this variable can be manipulated through the item. Each button has a label (displayed on the button) and an associated number. The number of a button is either defined by the user or is the rank of the button in the list of all buttons. If a button is pressed (i.e. selected by a mouse click) during a *read_mouse* operation its number is returned.

Action functions can be associated with buttons and some items (e.g. slider items) whenever a button with an associated action function is pressed this function is called with the number of the button as actual parameter. Action functions of items are called whenever the value of the corresponding variable is changed with the new value as actual parameter. All action functions must have the type *void func*(*int*).

Another way to define a button is to associate another window with it. In this case the button will have a menu sign and as soon as it is pressed the attached window will open. This method can be used to implement pop-up menues. The return value of the current *read_mouse* operation will be the number associated with the button in the menu.

3.9.1 General Settings

void	W.set_panel_bg_color(*color c*)	
		sets the background color of the panel area to c.
void	W.buttons_per_line(*int n*)	
		defines the maximal number n of buttons per line.
void	W.set_button_space(*int s*)	
		sets the space between to adjacent buttons to s pixels.
void	W.set_item_height(*int h*)	
		sets the vertical size of all items to h pixels.

void W.set_item_width(*int* w)
sets the horizontal size of all *slider* and *string* items to w pixels.

void W.set_bitmap_colors(*int* c_0, *int* c_1)
sets the unpressed/pressed colors used for drawing the pixels in bitmap buttons to c_0 and c_1.

3.9.2 Simple Panel Items

panel_item W.text_item(*string* s) adds a text_item s to W.

panel_item W.bool_item(*string* s, *bool*& x, *const char* $*$ *hlp* $= 0$)
adds a boolean item with label s and variable x to W.

panel_item W.bool_item(*string* s, *bool*& x, *void* $(*F)(int)$, *const char* $*$ *hlp* $= 0$)
as above with action function F.

panel_item W.int_item(*string* s, *int*& x, *const char* $*$ *hlp* $= 0$)
adds an integer item with label s and variable x to W.

panel_item W.string_item(*string* s, *string*& x, *void* $(*F)(char*)$, *const char* $*$ *hlp* $= 0$)
as above with action function F.

panel_item W.string_item(*string* s, *string*& x, *const char* $*$ *hlp* $= 0$)
adds a string item with label s and variable x to W.

panel_item W.double_item(*string* s, *double*& x, *const char* $*$ *hlp* $= 0$)
adds a real item with label s and variable x to W.

panel_item W.color_item(*string* s, *color*& x, *const char* $*$ *hlp* $= 0$)
adds a color item with label s and variable x to W.

panel_item W.color_item(*string* s, *color*& x, *void* $(*F)(int)$, *const char* $*$ *hlp* $= 0$)
as above with action function F.

panel_item W.pstyle_item(*string* s, *point_style*& x, *const char* $*$ *hlp* $= 0$)
adds a point style item with label s and variable x to W.

panel_item W.pstyle_item(*string* s, *point_style*& x, *void*$(*F)(int)$, *const char* $*$ *hlp* $= 0$)
as above with action function F.

panel_item W.lstyle_item(*string* s, *line_style*& x, *const char* $*$ *hlp* $= 0$)
adds a line style item with label s and variable x to W.

panel_item W.lstyle_item(*string* s, *line_style*& x, *void*$(*F)($*int*$)$, *const char* $*$ *hlp* $= 0$)

as above with action function F.

panel_item W.lwidth_item(*string* s, *int*& x, *const char* $*$ *hlp* $= 0$)

adds a line width item with label s and variable x to W.

panel_item W.lwidth_item(*string* s, *int*& x, *void*$(*F)($*int*$)$, *const char* $*$ *hlp* $= 0$)

as above with action function F.

3.9.3 Integer Choice Items

panel_item W.int_item(*string* s, *int*& x, *int* l, *int* h, *int step*, *const char* $*$ *hlp* $= 0$)

adds an integer choice item with label s, variable x, range $l,\ldots,h$, and step size *step* to W.

panel_item W.int_item(*string* s, *int*& x, *int* l, *int* h, *int step*, *void* $(*F)($*int*$)$, *const char* $*$ *hlp* $= 0$)

adds an integer choice item with label s, variable x, range $l,\ldots,h$, and step size *step* to W. Function $F(x)$ is executed whenever the value of x is changed.

panel_item W.int_item(*string* s, *int*& x, *int* l, *int* h, *const char* $*$ *hlp* $= 0$)

adds an integer slider item with label s, variable x, and range $l,\ldots,h$ to W.

panel_item W.int_item(*string* s, *int*& x, *int* l, *int* h, *void* $(*F)($*int*$)$, *const char* $*$ *hlp* $= 0$)

adds an integer slider item with label s, variable x, and range $l,\ldots,h$ to W. Function $F(x)$ is executed whenever the value of x has changed by moving the slider.

3.9.4 String Menu Items

panel_item W.string_item(*string* s, *string*& x, *list*<*string*> L, *const char* $*$ *hlp* $= 0$)

adds a string item with label s, variable x, and menu L to W.

panel_item W.string_item(*string* s, *string*& x, *list*<*string*> L, *void* $(*F)($*char*$*)$, *const char* $*$ *hlp* $= 0$)

as above with action function F.

panel_item W.string_item(*string* s, *string*& x, *list*<*string*> L, *int sz*, *const char* $*$ *hlp* $= 0$)

menu L is displayed in a scroll box of height *sz*.

panel_item W.string_item($string\ s$, $string\&\ x$, $list$<$string$> L, $int\ sz$, $void\ (*F)(char*)$, $const\ char * hlp = 0$)
as above with action function F.

void W.set_menu($panel\_item\ it$, $list$<$string$> L, $int\ sz = 0$)
replaces the menu of string menu item it by a menu for list L (table style if $sz = 0$ and scroll box with sz entries otherwise).

3.9.5 Choice Items

panel_item W.choice_item($string\ s$, $int\&\ x$, $list$<$string$> L, $void\ (*F)(int) = 0$, $const\ char * hlp = 0$)
adds an integer item with label s, variable x, and choices from L to W.

panel_item W.choice_item($string\ s$, $int\&\ x$, $string\ s\_1$, ..., $string\ sk$)
adds an integer item with label s, variable x, and choices $s_1, \ldots, s_k$ to W $(k \leq 8)$.

panel_item W.choice_item($string\ s$, $int\&\ x$, $int\ n$, $int\ w$, $int\ h$, $char * *bm$, $const\ char * hlp = 0$)
adds an integer item with label s, variable x, and n bitmaps $bm[0], \ldots, bm[n-1]$ each of width w and height h.

panel_item W.choice_item($string\ s$, $int\&\ x$, $int\ n$, $int\ w$, $int\ h$, $char * *bm$, $void\ (*F)(int)$, $const\ char * hlp = 0$)

3.9.6 Multiple Choice Items

panel_item W.choice_mult_item($string\ s$, $int\&\ x$, $list$<$string$> L, $const\ char * hlp = 0$)

panel_item W.choice_mult_item($string\ s$, $int\&\ x$, $list$<$string$> L, $void\ (*F)(int)$, $const\ char * hlp = 0$)

panel_item W.choice_mult_item($string\ s$, $int\&\ x$, $int\ n$, $int\ w$, $int\ h$, $char * *bm$, $const\ char * hlp = 0$)

panel_item W.choice_mult_item($string\ s$, $int\&\ x$, $int\ n$, $int\ w$, $int\ h$, $char * *bm$, $void\ (*F)(int)$, $const\ char * hlp = 0$)

3.9.7 Buttons

The first occurence of character '&' in a button label makes the following character c an *accelerator character*, i.e., the button can be selected by typing ALT-c from the keyboard.

int W.button(*string s*, *int n*, *const char* ∗ *hlp* = 0)

adds a button with label s and number n to W.

int W.fbutton(*string s*, *int n*, *const char* ∗ *hlp* = 0)

as above but makes this button the focus button of W, i.e., this button can be selected by pressing the return key.

int W.button(*string s*, *const char* ∗ *hlp* = 0)

adds a new button to W with label s and number equal to its position in the list of all buttons (starting with 0).

int W.fbutton(*string s*, *const char* ∗ *hlp* = 0)

as above but makes this button the focus button.

int W.button(*int w*, *int h*, *char* ∗ *bm*, *string s*, *int n*, *const char* ∗ *hlp* = 0)

adds a button with bitmap bm, label s, and number n to W.

int W.button(*char* ∗ *pr1*, *char* ∗ *pr2*, *string s*, *int n*, *const char* ∗ *hlp* = 0)

adds a button with pixrects $pr1$ and $pr2$, label s, and number n to W.

int W.button(*int w*, *int h*, *char* ∗ *bm*, *string s*, *const char* ∗ *hlp* = 0)

adds a new button to W with bitmap bm, label s, and number equal to its position in the list of all buttons (starting with 0).

int W.button(*string s*, *int n*, *void* (∗*F*)(*int*), *const char* ∗ *hlp* = 0)

adds a button with label s, number n and action function F to W. Function F is called with actual parameter n whenever the button is pressed.

int W.fbutton(*string s*, *int n*, *void* (∗*F*)(*int*), *const char* ∗ *hlp* = 0)

as above but makes this button the focus button.

int W.button(*int w*, *int h*, *char* ∗ *bm*, *string s*, *int n*, *void* (∗*F*)(*int*), *const char* ∗ *hlp* = 0)

adds a button with bitmap bm, label s, number n and action function F to W. Function F is called with actual parameter n whenever the button is pressed.

int W.button(*char* ∗ *pr1*, *char* ∗ *pr2*, *string s*, *int n*, *void* (∗*F*)(*int*), *const char* ∗ *hlp* = 0)

as above, but with pixrect $pr1$ and $pr2$.

int W.button(*string* s, *void* $(*F)(int)$, *const char* $*$ *hlp* $= 0$)

adds a button with label s, number equal to its rank and action function F to W. Function F is called with the value of the button as argument whenever the button is pressed.

int W.button(*int* w, *int* h, *char* $*$ *bm*, *string* s, *void* $(*F)(int)$, *const char* $*$ *hlp* $= 0$)

adds a button with bitmap *bm*, label s, number equal to its rank and action function F to W. Function F is called with the value of the button as argument whenever the button is pressed.

int W.button(*char* $*$ *pr1*, *char* $*$ *pr2*, *string* s, *void* $(*F)(int)$, *const char* $*$ *hlp* $= 0$)

as above, but with pixrect $pr1$ and $pr2$.

int W.button(*string* s, *int* n, *window*& M, *const char* $*$ *hlp* $= 0$)

adds a button with label s, number n and attached sub-window (menu) M to W. Window M is opened whenever the button is pressed.

int W.button(*int* w, *int* h, *char* $*$ *bm*, *string* s, *int* n, *window*& M, *const char* $*$ *hlp* $= 0$)

adds a button with bitmap *bm*, label s, number n and attached sub-window (menu) M to W. Window M is opened whenever the button is pressed.

int W.button(*char* $*$ *pr1*, *char* $*$ *pr2*, *string* s, *int* n, *window*& M, *const char* $*$ *hlp* $= 0$)

as above, but with pixrect $pr1$ and $pr2$.

int W.button(*string* s, *window*& M, *const char* $*$ *hlp* $= 0$)

adds a button with label s and attached sub-window M to W. The number returned by *read_mouse* is the number of the button selected in sub-window M.

int W.button(*int* w, *int* h, *char* $*$ *bm*, *string* s, *window*& M, *const char* $*$ *hlp* $= 0$)

adds a button with bitmap *bm*, label s and attached sub-window M to W. The number returned by *read_mouse* is the number of the button selected in sub-window M.

int W.button(*char* $*$ *pr1*, *char* $*$ *pr2*, *string* s, *window*& M, *const char* $*$ *hlp* $= 0$)

as above, but with pixrect $pr1$ and $pr2$.

void	W.make_menu_bar( )	inserts a menu bar at the top of the panel section that contains all previously added menu buttons (buttons with an attached subwindow).
window∗	*window* :: get_call_window( )	A static function that can be called in action functions attached to panel items or buttons to retrieve a pointer to the window containing the corresponding item or button.
panel_item	*window* :: get_call_item( )	A static function that can be called in action functions attached to panel items to retrieve the corresponding item.
int	*window* :: get_call_button( )	A static function that can be called in action functions attached to panel buttons to retrieve the number of the corresponding button.

3.9.8. Manipulating Panel Items and Buttons

Disabling and Enabling Items or buttons

void	W.disable_item(*panel_item it*)	disables panel item *it*.
void	W.enable_item(*panel_item it*)	enables panel item *it*.
bool	W.is_enabled(*panel_item it*)	tests whether item *it* is enabled or not.
void	W.disable_button(*int b*)	disables button *b*.
void	W.enable_button(*int b*)	enables button *b*.
bool	W.is_enabled(*int b*)	tests whether button *b* is enabled or not.
void	W.disable_panel( )	disables the entire panel section of W.
void	W.enable_panel( )	enables the entire panel section of W.

Accessing and Updating Item Data

void	W.set_text(*panel_item it*, *string s*)	replaces the text of text item *it* by *s*.
panel_item	W.get_item(*string s*)	returns the item with label *s* and *NULL* if no such item exists in W.

int	W.get_button(*string* s)	returns the button with label s and -1 if no such button exists in W.
string	W.get_button_label(*int* *but*)	returns the label of button *but*.
void	W.set_button_label(*int* *but*, *string* s)	sets the label of button *but* to s.
window∗	W.get_window(*int* *but*)	returns a pointer to the subwindow attached to button *but* (*NULL* if *but* has no subwindow)
window∗	W.set_window(*int* *but*, *window* ∗ M)	associates subwindow (menu) $*M$ with button *but*. Returns a pointer to the window previously attached to *but*.
void	W.set_function(*int* *but*, *void* (∗F)(*int*))	assign action function F to button *but*.
void	W.redraw_panel( )	redraw the panel area of W.
void	W.redraw_panel(*panel_item* *it*)	redraw item i in the panel area of W.
void	W.display_help_text(*string* *fname*)	displays the help text contained in *name.hlp*. The file *name.hlp* must exist either in the current working directory or in \$*LEDAROOT*/*incl*/*Help*.

4. Example

Example programs can be found on LEDAROOT/demo/win and LEDAROOT/test/win.

14.3 Panels (panel)

1. Definition

Panels are windows consisting of a panel section only (cf. section 14.2). They are used for displaying text messages and updating the values of variables.

#include < *LEDA/panel.h* >

2. Creation

panel P; creates an empty panel P.

panel $P(string\ s)$; creates an empty panel P with header s.

panel $P(int\ w,\ int\ h)$;
creates an empty panel P of width w and height h.

panel $P(int\ w,\ int\ h,\ string\ s)$;
creates an empty panel P of width w and height h with header s.

3. Operations

All window operations for displaying, reading, closing and adding panel items are available (see section 14.2). There are two additional operations for opening and reading panels.

int P.open$(int\ x\ =\ window::center,\ int\ y\ =\ window::center)$
$P.display(x,y)$ + $P.read\_mouse(\ )$ + $P.close(\ )$.

int P.open$(window\&\ W,\ int\ x = window::center,\ int\ y = window::center)$
$P.display(W,x,y)$ + $P.read\_mouse(\ )$ + $P.close(\ )$.

14.4 Menues (menu)

1. Definition

Menues are special panels consisting only of a vertical list of buttons.

#include < *LEDA/menu.h* >

2. Creation

menu M; creates an empty menu M.

3. Operations

int M.button(*string* s, *int* n) adds a button with label s and number n to M.

int M.button(*string* s) adds a new button to M with label s and number equal to its position in the list of all buttons (starting with 0).

int M.button(*string* s, *int* n, *void* $(*F)$(*int*))
adds a button with label s, number n and action function F to M. Function F is called with actual parameter n whenever the button is pressed.

int M.button(*string* s, *void* $(*F)$(*int*))
adds a button with label s, number equal to its rank and action function F to M. Function F is called with the number of the button as argument whenever the button is pressed.

int M.button(*string* s, *int* n, *window*& W)
adds a button with label s, number n, and attached window W to M.Whenever the button is pressed W is opened.

int M.button(*string* s, *window*& W)
adds a button with label s and attached window W to M. Whenever the button is pressed W is opened and W.read_mouse() is returned.

void M.separator( ) inserts a separator (horizontal line) at the current position.

int M.open(*window*& W, *int* x, *int* y)
open and read menu M at position (x, y) in window W.

14.5 Postscript Files (ps_file)

1. Definition

The date type *ps_file* is a graphical input/output interface for the familiar LEDA drawing operations of two-dimensional geometry. Unlike the data type *window*, the output produced by a *ps_file* object is *permanent*, i.e., it is not lost after exiting the C++-program as it is saved in an output file.

An instance of type *ps_file* is (as far as the user takes notice of it) an ordinary ASCII file that contains the source code of the graphics output in the PostScript description language. After running the C++-program, the file is created in the user's current working directory and can later be handled like any other PostScript file, i.e., it may be viewed, printed etc.

Of course, features like a panel section (as in *window* type instances) don't make sense for a representation that is not supposed to be displayed on the screen and interactively worked with by the user. Therefore, only drawing operations are applicable to a *ps_file* instance.

ps_file was implemented by

Thomas Wahl
Lehrstuhl fr Informatik I
Universität Würzburg

The complete user manual can be found in LEDAROOT/Manual/contrib.

#include < *LEDA/ps_file.h* >

14.6 Graph Windows (GraphWin)

1. Definition

GraphWin combines the two types *graph* and *window* and forms a bridge between the graph data types and algorithms and the graphics interface of LEDA. *GraphWin* can easily be used in LEDA programs for constructing, displaying and manipulating graphs and for animating and debugging graph algorithms.

- The user interface of GraphWin is simple and intuitive. When clicking a mouse button inside the drawing area a corresponding default action is performed that can be redefined by users. With the initial default settings, the left mouse button is used for creating and moving objects, the middle button for selecting objects, and the right button for destroying objects. A number of menues at the top of the window give access to graph generators, modifiers, basic algorithms, embeddings, setup panels, and file input and output.
- Graphwin can display and manipulate the data associated with the nodes and edges of LEDA's parameterized graph type $GRAPH < vtype, etype >$. When a GraphWin is opened for such a graph the associated node and edge labels of type *vtype* and *etype* can be displayed and edited.
- Most of the actions of GraphWin can be customized by modifying or extending the menues of the main window or by defining call-back functions. So the user can define what happens if a node or edge is created, selected, moved, or deleted.
- Graphwin offers a collection of graph generators, modifiers and tests. The generators include functions for constructing random, planar, complete, bipartite, grid graphs, connected graphs, biconnected, graphs ...

  There are also methods for modifying existing graphs (e.g. by removing or adding a certain set of edges) to fit in one of these categories and for testing whether a given graph is planar, connected, bipartite ...
- The standard menu includes a choice of fundamental graph algorithms and basic embedding algorithms.

For every node and edge of the graph GraphWin maintains a set of parameters.

With every node is associated

- a *position* of type *point*,
- a *shape* of type *gw_node_shape*,

- a *color*, *border color*, and *label color* of type *color*,
- a *pixmap* of type *char∗*,
- a *width* and *height* in pixels of type *int*,
- a *radius1* and *radius2* in real world coordinates of type *double*,
- a *border width* in pixels of type *int*,
- a *label type* of type *gw_label_type*,
- a *user label* of type *string*, and
- a *label position* of type *gw_position*.

With every edge is associated

- a list of *bends* of type *list*<*point*>,
- a *color* and *label color* of type *color*,
- a *shape* of type *gw_edge_shape*,
- a *style* of type *gw_edge_style*,
- a *direction* of type *gw_edge_dir*,
- a *width* of type *int*,
- a *label type* of type *gw_label_type*,
- a *user label* of type *string*, and
- a *label position* of type *gw_position*,
- a *list of bends* of type *list*<*point*>,
- a *source anchor* of type *point*,
- a *target anchor* of type *point*.

The corresponding types are:

```
gw_node_shape = { circle_node, ellipse_node, square_node, rectangle_node }
gw_edge_shape = { poly_edge, circle_edge, bezier_edge, spline_edge }

gw_position = { central_pos, northwest_pos, north_pos,
                northeast_pos, east_pos, southeast_pos,
```

```
                south_pos, southwest_pos, west_pos }

gw_label_type = { no_label, user_label, data_label, index_label }

gw_edge_style = { solid_edge, dashed_edge, dotted_edge, dashed_dotted_edge }
gw_edge_dir   = { undirected_edge, directed_edge, bidirected_edge, rdirected_edge };
```

#include < *LEDA/graphwin.h* >

2. Creation

GraphWin $gw(graph\&\ G,\ int\ w,\ int\ h,\ const\ char * win\_label = "")$;

creates a graph window for graph G with a display window of size w pixels $\times$ h pixels. If *win_label* is not empty it is used as the frame label of the window, otherwise, a default frame label is used.

GraphWin $gw(graph\&\ G,\ const\ char * win\_label = "")$;

creates a graph window for graph G with a display window of default size and frame label *win_label*.

GraphWin $gw(int\ w,\ int\ h,\ const\ char * win\_label = "")$;

creates a graph window for a new empty graph with a display window of size w pixels $\times$ h pixels, and frame label *win_label*.

GraphWin $gw(const\ char * win\_label = "")$;

creates a graph window for a new empty graph with a display window of default size and frame label *win_label*.

GraphWin $gw(window\&\ W)$;

as above, but W is used as display window.

GraphWin $gw(graph\&\ G,\ window\&\ W)$;

as above, but makes G the graph of *gw*.

3. Operations

a) Window Operations

void	*gw*.display(*int* x, *int* y)	displays gw with upper left corner at (x, y). The predefined constant *window*::*center* can be used to center the window horizontally (if passed as x) or vertically (if passed as y).
void	*gw*.display( )	displays gw at default position.

bool	*gw*.edit( )	enters the edit mode of *GraphWin* that allows to change the graph interactively by operations associated with certain mouse events or by choosing operations from the windows menu bar (see section **??** for a description of the available commands and operations). Edit mode is terminated by either pressing the *done* button or by selecting *exit* from the file menu. In the first case the result of the edit operation is *true* and in the latter case the result is *false*.
bool	*gw*.open(*int* x, *int* y)	displays the window at position (x, y), enters edit mode and return the corresponding result.
bool	*gw*.open( )	as above, but displays the window at default position.
void	*gw*.close( )	closes the window.
void	*gw*.message(*const char* ∗ *msg*)	displays the message *msg* at the top of the window.
void	*gw*.del_message( )	deletes a previously written message.
double	*gw*.get_xmin( )	returns the minimal x-coordinate of the window.
double	*gw*.get_ymin( )	returns the minimal y-coordinate of the window.
double	*gw*.get_xmax( )	returns the maximal x-coordinate of the window.
double	*gw*.get_ymax( )	returns the maximal y-coordinate of the window.
void	*gw*.win_init(*double xmin*, *double xmax*, *double ymin*)	sets the coordinates of the window to $(xmin, xmax, ymin)$.
void	*gw*.redraw( )	redraws the graph. If the *flush* parameter of *gw* is set to false (see *set_flush*) this operation can be used to display the current state of the graph after a number of update operations.
void	*gw*.set_frame_label(*const char* ∗ *label*)	makes *label* the frame label of the window.
int	*gw*.open_panel(*panel*& P)	displays panel P centered on the drawing area of *gw*, disables the menu bar of *gw* and returns the result of *P.open*( ).

window&	*gw*.get_window( )	returns a reference to the window of *gw*.
void	*gw*.finish_menu_bar( )	this operation has to called before additional buttons are added to the panel section of *gw.get_window*( ).

b) Graph Operations

node	*gw*.new_node(*point p*)	adds a new node at position p to *gw*.
void	*gw*.del_node(*node v*)	deletes v and all edges incident to v from *gw*.
edge	*gw*.new_edge(*node v*, *node w*)	adds a new edge (v, w) to *gw*.
edge	*gw*.new_edge(*node v*, *node w*, *list*<*point*> *P*)	adds a new edge (v, w) with bend sequence P to *gw*.
void	*gw*.del_edge(*edge e*)	deletes edge e from *gw*.
void	*gw*.clear_graph( )	deletes all nodes and egdes.
graph&	*gw*.get_graph( )	returns a reference of the graph of *gw*.
void	*gw*.update_graph( )	this operation has to be called after any update operation that has been performed directly (not by GraphWin) on the underlying graph, e.g., deleting or inserting nodes or edges.

c) Node Parameters

Node parameters can be retrieved or changed by a collection of *get*- and *set*- operations. We use *param_type* for the type and *param* for the value of the corresponding parameter.

Individual Parameters

param_type	*gw*.get_param(*node v*)	returns the value of parameter *param* for node v.
param_type	*gw*.set_param(*node v*, *param_type x*)	sets the value of parameter *param* for node v to x. and returns its previous value.
void	*gw*.set_param(*list*<*node*>& *L*, *param_type x*)	sets the value of parameter *param* for all nodes in L to x.

Default Parameters

param_type *gw*.get_node_param( ) returns the current default value of parameter *param*.

param_type *gw*.set_node_param(*param_type x*, *bool apply* = *true*)
sets the default value of parameter *param* to x. and returns its previous value. If *apply* == *true* the parameter is changed for all existing nodes as well.

d) Edge Parameters

Individual Parameters

param_type *gw*.get_param(*edge e*) returns the value of parameter *param* for edge e.

param_type *gw*.set_param(*edge e*, *param_type x*)
sets the value of parameter *param* for edge e to x. and returns its previous value.

void *gw*.set_param(*list*<*edge*>& *L*, *param_type x*)
sets the value of parameter *param* for all edges in L to x.

Default Parameters

param_type *gw*.get_edge_param( ) returns the current default value of parameter *param*.

param_type *gw*.set_edge_param(*param_type x*, *bool apply* = *true*)
sets the default value of parameter *param* to x. and returns its previous value. If *apply* == *true* the parameter is changed for all existing edges as well.

e) Global Options

int *gw*.set_gen_nodes(*int n*) sets the default number of nodes n for all graph generator dialog panels.

int *gw*.set_gen_edges(*int m*) sets the default number of edges m for all graph generator dialog panels.

int *gw*.set_edge_distance(*int d*)
sets the distance of multi-edges to d pixels.

grid_style *gw*.set_grid_style(*grid_style s*)
sets the grid style to s.

int *gw*.set_grid_dist(*int d*) sets the grid distance to d.

int	*gw*.set_grid_size(*int n*)	sets the grid distance such that n vertical grid lines lie inside the drawin area.
bool	*gw*.set_show_status(*bool b*)	display a status window (b=*true*) or not (b=*false*).
color	*gw*.set_bg_color(*color c*)	sets the window background color to c.
char∗	*gw*.set_bg_pixmap(*char ∗ pr*, *double xorig* = 0, *double yorig* = 0)	sets the window background pixmap to *pr* and the tiling origin to $(xorig, yorig)$.
void	*gw*.set_bg_xpm(*const char ∗ ∗xpm_data*)	sets the window background pixmap to the pixmap defined by *xpm_data*.
void	*gw*.set_bg_redraw(*void* (∗*f*)(*window∗*, *double*, *double*, *double*, *double*))	sets the window background redraw function to f.
void	*gw*.set_node_label_font(*gw_font_type t*, *int sz*)	sets the node label font type and size. Possible types are *roman_font*, *bold_font*, *italic_font*, and *fixed_font*.
void	*gw*.set_node_label_font(*string fn*)	sets the node label font to the font with name *fn*.
void	*gw*.set_edge_label_font(*gw_font_type t*, *int sz*)	sets the edge label font type and size. *roman_font*, *bold_font*, *italic_font*, and *fixed_font*.
void	*gw*.set_edge_label_font(*string fn*)	sets the edge label font to the font with name *fn*.
string	*gw*.set_node_index_format(*string s*)	sets the node index format string to s.
string	*gw*.set_edge_index_format(*string s*)	sets the edge index format string s.

Animation and Zooming

int	*gw*.set_animation_steps(*int s*)	move a node in s steps to its new position.
bool	*gw*.set_flush(*bool b*)	show operations on *gw* instantly (b=*true*) or not (b=*false*).

double *gw*.set_zoom_factor(*double* f)

sets the zoom factor to f used when zooming from menu.

bool *gw*.set_zoom_objects(*bool* b)

resize nodes and edges when zooming ($b == true$) or not ($b == false$).

bool *gw*.set_zoom_labels(*bool* b)

resize labels when zooming ($b == true$) or not ($b == false$).

f) Node and Edge Selections

void	*gw*.select(*node* v)	adds v to the list of selected nodes.
void	*gw*.select_all_nodes( )	selects all nodes.
void	*gw*.deselect(*node* v)	deletes v from the list of selected nodes.
void	*gw*.deselect_all_nodes( )	clears the current node selection.
bool	*gw*.is_selected(*node* v)	returns *true* if v is selected and *false* otherwise.
list<*node*>	*gw*.get_selected_nodes( )	returns the current node selection.
void	*gw*.select(*edge* e)	adds e to the list of selected edges.
void	*gw*.select_all_edges( )	selects all edges.
void	*gw*.deselect(*edge* e)	deletes e from the list of selected edges.
void	*gw*.deselect_all_edges( )	clears the current node selection.
bool	*gw*.is_selected(*edge* e)	returns *true* if e is selected and *false* otherwise.
list<*edge*>	*gw*.get_selected_edges( )	returns the current edge selection.

g) Layout Operations

void *gw*.set_position(*node_array*<*point*> *pos*)

for every node v of G the position of v is set to $pos[v]$.

void *gw*.set_position(*node_array*<*double*> x, *node_array*<*double*> y)

for every node v of G the position of v is set to $(x[v], y[v])$.

void gw.get_position(*node_array*<*point*>& *pos*)

for every node v of G the position of v is assigned to $pos[v]$.

void gw.set_layout(*node_array*<*point*> *pos*, *node_array*<*double*> *r1*, *node_array*<*double*> *r2*, *edge_array*<*list*<*point*> > *bends*, *edge_array*<*point*> *sanch*, *edge_array*<*point*> *tanch*)

for every node v the position is set to $pos[v]$ and $radius_i$ is set to $r_i[v]$. For every edge e the list of bends is set to $bends[e]$ and source (target) anchor is set to $sanch[e]$ ($tanch[e]$).

void gw.set_layout(*node_array*<*point*> *pos*, *edge_array*<*list*<*point*> > *bends*, *bool reset_anchors = true*)

for every node v the position is set to $pos[v]$ and for every edge e the list of bends is set to $bends[e]$.

void gw.set_layout(*node_array*<*point*> *pos*)

for every node v the position is set to $pos[v]$ and for every edge e the list of bends is made empty.

void gw.set_layout(*node_array*<*double*> *x*, *node_array*<*double*> *y*)

for every node v the position is set to $(x[v], y[v])$ and for every edge e the list of bends is made empty.

void gw.set_layout( ) same as *gw.remove_bends*( ).

void gw.place_into_box(*double x0*, *double y0*, *double x1*, *double y1*)

moves and stretches the graph to fill the given rectangular box $(x0, y0, x1, y1)$ by appropriate scaling and translating operations.

void gw.place_into_win( ) moves and stretches the graph to fill the entire window by appropriate scaling and translating operations.

void gw.adjust_coords_to_box(*node_array*<*double*>& *xpos*, *node_array*<*double*>& *ypos*, *edge_array*<*list*<*double*> >& *xbends*, *edge_array*<*list*<*double*> >& *ybends*, *double x0*, *double y0*, *double x1*, *double y1*)

transforms the layout given by *xpos*, *ypos*, *xbends*, and *ybends* in such way as a call of *place_into_box*(*x0*, *y0*, *x1*, *y1*) would do. However, the actual layout of the current graph is not changed by this operation.

void *gw*.adjust_coords_to_box(*node_array*<*double*>& *xpos*, *node_array*<*double*>& *ypos*, *double x0*, *double y0*, *double x1*, *double y1*)

transforms the layout given by *xpos*, *ypos* in such way as a call of *place_into_box*(*x0*, *y0*, *x1*, *y1*) would do ignoring any edge bends. The actual layout of the current graph is not changed by this operation.

void *gw*.adjust_coords_to_win(*node_array*<*double*>& *xpos*, *node_array*<*double*>& *ypos*, *edge_array*<*list*<*double*> >& *xbends*, *edge_array*<*list*<*double*> >& *ybends*)

same as *adjust_coords_to_box*(*xpos*, *ypos*, *xbends*, *ybends*, *wx0*, *wy0*, *wx1*, *wy1*) for the current window rectangle $(wx0, wy0, wx1, wy1)$.

void *gw*.adjust_coords_to_win(*node_array*<*double*>& *xpos*, *node_array*<*double*>& *ypos*)

same as *adjust_coords_to_box*(*xpos*, *ypos*, *wx0*, *wy0*, *wx1*, *wy1*) for the current window rectangle $(wx0, wy0, wx1, wy1)$.

void *gw*.remove_bends(*edge e*) removes all bends from edge e.

void *gw*.remove_bends( ) removes the bends of all edges of the graph.

void *gw*.reset_edge_anchors( ) resets all edge anchor positions to $(0, 0)$.

int *gw*.load_layout(*istream*& *istr*)

read layout from stream *istr*.

bool *gw*.save_layout(*ostream*& *ostr*)

save layout to stream *ostr*.

bool *gw*.save_layout(*string fname*, *bool ask_override* = *false*)

save layout to file *fname*.

h) Zooming

void *gw*.zoom(*double f*) zooms the window by factor f.

void *gw*.zoom_area(*double x0*, *double y0*, *double x1*, *double y1*)

performs a zoom operation for the rectangular area with current coordinates $(x0, y0, x1, y0)$.

void *gw*.zoom_graph( ) performs a zoom operation, such that the graph fills the entire window.

void *gw*.unzoom( ) undoes last zoom operation.

i) Operations in Edit-mode

Before entering edit mode ...

gw_action *gw*.set_action(*long mask*, *gw_action func*)

sets action associated with condition *mask* to *func* and returns previous action for this condition. Here *gw_action* is the type *void* (*∗func*)(*GraphWin*&, *const point*&). For *func* = *NULL* the corresponding action is deleted.

gw_action *gw*.get_action(*long mask*)

returns the action associated with condition *mask*.

void *gw*.reset_actions( ) resets all actions to their defaults.

void *gw*.clear_actions( ) deletes all actions.

void *gw*.add_node_menu(*string label*, *gw_action func*)

appends action function *func* with label *label* to the context menu for nodes (opened by clicking with the right mouse button on a node).

void *gw*.add_edge_menu(*string label*, *gw_action func*)

appends action function *func* with label *label* to the context menu for edges (opened by clicking with the right mouse button on an edge).

void *gw*.set_new_node_handler(*bool* (*∗f*)(*GraphWin*& , *point*))

$f(gw, p)$ is called every time before a node is to be created at position p.

void *gw*.set_new_node_handler(*void* (*∗f*)(*GraphWin*& , *node*) = *NULL*)

$f(gw, v)$ is called after node v has been created.

void *gw*.set_new_edge_handler(*bool* (*∗f*)(*GraphWin*& , *node*, *node*))

$f(gw, v, w)$ is called before the edge (v, w) is to be created.

void *gw*.set_new_edge_handler(*void* (*∗f*)(*GraphWin*& , *edge*) = *NULL*)

$f(gw, e)$ is called after the edge e has been created.

void *gw*.set_start_move_node_handler(*bool* (*∗f*)(*GraphWin*& , *node*) = *NULL*)

$f(gw, v)$ is called before node v is to be moved.

void *gw*.set_move_node_handler(*void* $(*f)$(*GraphWin*& , *node*) = *NULL*)

$f(gw, v)$ is called every time node v reaches a new position during a move operation.

void *gw*.set_end_move_node_handler(*void* $(*f)$(*GraphWin*& , *node*))

$f(gw, v)$ is called after node v has been moved.

void *gw*.set_del_node_handler(*bool* $(*f)$(*GraphWin*& , *node*))

$f(gw, v)$ is called before the node v is to be deleted.

void *gw*.set_del_node_handler(*void* $(*f)$(*GraphWin*&) = *NULL*)

$f(gw)$ is called every time after a node was deleted.

void *gw*.set_del_edge_handler(*bool* $(*f)$(*GraphWin*& , *edge*))

$f(gw, e)$ is called before the edge e is to be deleted.

void *gw*.set_del_edge_handler(*void* $(*f)$(*GraphWin*&) = *NULL*)

$f(gw)$ is called every time after an edge was deleted.

void *gw*.set_start_edge_slider_handler(*void* $(*f)$(*GraphWin*& , *edge*, *double*) = *NULL*, *int* $sl = 0$)

$f(gw, e, pos)$ is called before slider sl of edge e is to be moved. Here pos is the current slider position.

void *gw*.set_edge_slider_handler(*void* $(*f)$(*GraphWin*& , *edge*, *double*) = *NULL*, *int* $sl = 0$)

$f(gw, e, pos)$ is called every time slider sl of edge e reaches a new position pos during a slider move.

void *gw*.set_end_edge_slider_handler(*void* $(*f)$(*GraphWin*& , *edge*, *double*) = *NULL*, *int* $sl = 0$)

$f(gw, e, pos)$ is called after slider sl of edge e has been moved to the final position pos.

void *gw*.set_init_graph_handler(*bool* $(*f)$(*GraphWin*&))

f is called every time before the entire graph is replaced, e.g. by a clear, generate, or load operation.

void *gw*.set_init_graph_handler(*void* $(*f)$(*GraphWin*&) = *NULL*)

f is called every time after the entire graph was replaced.

j) Menus

The default menu ...

void *gw*.set_default_menu(*long mask*)

...

void *gw*.add_menu(*long menu_id*)

...

void *gw*.del_menu(*long menu_id*)

...

Extending menus by new buttons and sub-menus ...

int *gw*.add_menu(*string label*, *int menu_id* = 0, *char* ∗ *pmap* = 0)

...

int *gw*.add_simple_call(*void* (∗*func*)(*GraphWin*&), *string label*, *int menu_id* = 0, *char* ∗ *pmap* = 0)

...

int *gw*.add_member_call(*void* (*GraphWin*:: ∗ *func*)( ), *string label*, *int menu_id* = 0, *char* ∗ *pmap* = 0)

...

void *gw*.add_separator(*int menu_id*)

...

void *gw*.display_help_text(*string fname*)

displays the help text contained in *name.hlp*. The file *name.hlp* must exist either in the current working directory or in $\$LEDAROOT/incl/Help$.

void *gw*.add_help_text(*string name*)

adds the help text contained in *name.hlp* with label *name* to the help menu of the main window. The file *name.hlp* must exist either in the current working directory or in $\$LEDAROOT/incl/Help$. Note that this operation must be called before *gw.display*( ).

int *gw*.get_menu(*string label*)

returns the number of the submenu with label *label* or -1 if no such menu exists.

void *gw*.enable_call(*int id*) enable call with id *id*.

void *gw*.disable_call(*int id*) disable call with id *id*.

bool *gw*.is_call_enabled(*int id*) check if call with *id* is enabled.

void *gw*.enable_calls( ) ...

void *gw*.disable_calls( ) ...

k) Input/Output

int *gw*.read_gw(*istream*& *in*)
reads graph in *gw* format from stream *in*.

int *gw*.read_gw(*string fname*)
reads graph in *gw* format from file *fname*.

bool *gw*.save_gw(*ostream*& *out*)
writes graph in *gw* format to output stream *out*.

bool *gw*.save_gw(*string fname*, *bool ask_override* = *false*)
saves graph in *gw* format to file *fname*.

int *gw*.read_gml(*istream*& *in*)
reads graph in GML format from stream *in*.

int *gw*.read_gml_string(*string s*)
reads graph in GML format from string *s*.

int *gw*.read_gml(*string fname*, *bool ask_override* = *false*)
reads graph in GML format from file *fname*. Returns 1 if *fname* cannot be opened, 2 if a parser error occurs, and 0 on success.

bool *gw*.save_gml(*ostream*& *out*)
writes graph in GML format to output stream *out*.

bool *gw*.save_gml(*string fname*, *bool ask_override* = *false*)
saves graph to file *fname* in GML format.

bool *gw*.save_ps(*string fname*, *bool ask_override* = *false*)
saves a postscript representation of the graph to *fname*.

bool *gw*.save_latex(*string fname*, *bool ask_override* = *false*)
saves a postscript/latex representation of the graph to *fname*.

bool *gw*.unsaved_changes( ) returns true if the graph has been changed after the last save (gw or gml) operation.

bool *gw*.save_defaults(*string fname*)
saves the default attributes of nodes and edges to file *fname*.

bool *gw*.read_defaults(*string fname*)
reads the default attributes of nodes and edges from file *fname*.

l) Miscellaneous

void *gw*.set_window(*window*& *W*)
makes W the window of *gw*.

void *gw*.set_graph(*graph*& *G*) makes G the graph of *gw*.

void *gw*.undo_clear( ) empties the undo and redo stacks.

bool *gw*.wait( ) waits until the done button is pressed (*true* returned) or exit is selected from the file menu (*false* returned).

bool *gw*.wait(*const char* ∗ *msg*)
displays *msg* and waits until the done button is pressed (*true* returned) or exit is selected from the file menu (*false* returned).

bool *gw*.wait(*float sec*, *const char* ∗ *msg* = "")
as above but waits no longer than *sec* seconds returns ?? if neither button was pressed within this time interval.

void *gw*.acknowledge(*string s*)
displays string s and asks for acknowledgement.

node *gw*.ask_node( ) asks the user to select a node with the left mouse button. If a node is selected it is returned otherwise nil is returned.

edge *gw*.ask_edge( ) asks the user to select an edge with the left mouse button. If an edge is selected it is returned otherwise nil is returned.

bool *gw*.define_area(*double*& *x0*, *double*& *y0*, *double*& *x1*, *double*& *y1*, *const char* ∗ *msg* = "")
displays message *msg* and returns the coordinates of a rectangular area defined by clicking and dragging the mouse.

list<*node*> *gw*.get_nodes_in_area(*double x0*, *double y0*, *double x1*, *double y1*)
returns the list of nodes intersecting the rectangular area $(x0, y0, x1, y1)$.

list<*edge*> *gw*.get_edges_in_area(*double x0*, *double y0*, *double x1*, *double y1*)
returns the list of edges intersecting the rectangular area $(x0, y0, x1, y1)$.

void *gw*.save_node_attributes( )
...

void *gw*.save_edge_attributes( )
...

void *gw*.save_all_attributes( ) ...

void *gw*.restore_node_attributes( )
...

void *gw*.restore_edge_attributes( )
...

void *gw*.restore_all_attributes( )
...

void *gw*.reset_nodes(*long mask* = *N_ALL*)
reset node parameters to their default values.

void *gw*.reset_edges(*long mask* = *E_ALL*)
reset edge parameters to their default values.

void *gw*.reset( ) reset node and edge parameters to their default values.

void *gw*.reset_defaults( ) resets default parameters to their original values.

node *gw*.get_edit_node( ) returns a node under the current mouse pointer position (*nil* if there is no node at the current position)

edge *gw*.get_edit_edge( ) returns an edge under the current mouse pointer position (*nil* if there is no edge at the current position).

int *gw*.get_edit_slider( ) returns the number of the slider under the current mouse pointer position (0 if there is no edge slider at the current position).

void *gw*.get_bounding_box(*double& x0*, *double& y0*, *double& x1*, *double& y1*)
computes the coordinates $(x0, y0, x1, y1)$ of a minimal bounding box for the current layout of the graph.

void *gw*.get_bounding_box(*list<node> V*, *list<edge> E*, *double& x0*, *double& y0*, *double& x1*, *double& y1*)
computes the coordinates $(x0, y0, x1, y1)$ of a minimal bounding box for the current layout of subgraph (V, E).

14.7 Geometry Windows (GeoWin)

1. Definition

An instance of data type *GeoWin* is an editor for sets of geometric objects. It can be used for the visualization of result and progression of geometric algorithms.

GeoWin manages the geometric objects in so called scenes. A scene contains a container storing geometric objects (the contents of the scene) and provides all operations that *GeoWin* needs to edit it. Every scene maintains a set of parameters:

- *name* is the name of the scene (type *string*)
- *color* is the color for drawing the boundaries of the objects (type *color*)
- *color2* is the color for drawing the selected boundaries of the objects (type *color*)
- *fill_color* is the color for drawing the interior of the objects (type *color*)
- *text_color* is the color for drawing text labels
- *line_width* is the line width used for drawing the objects (type *int*)
- *active_line_width* is the line width used for drawing the objects if the scene is active (type *int*)
- *line_style* is the line style used for drawing the objects (type *line_style*)
- *point_style* is the point style used for drawing points (type *point_style*)
- *visible* is *true* if the scene is visible in his *GeoWin*, *false* otherwise (type *bool*)
- *active* is *true* if the scene has input focus (is currently edited) in his *GeoWin*, *false* otherwise (type *bool*)

There are three kinds of scenes:

1. *Basic Scenes*

   The basic scene type *GeoBaseScene*<*CONTAINER*> works on every container type that provides an interface as the list of the STL library. More precisely, *CONTAINER* has to support the following type definitions and operations:

   - *value_type* - the type T of values the container holds
   - *iterator*
   - operations *begin*( ) and *end*( ) returning an iterator that can be used for begining (ending) the traversal of the container

- *void push_back*(*const T*&) for inserting an element at the end of the container
- *iterator insert*(*iterator*, *const T*&) for inserting an element
- *void erase*(*iterator it*) for erasing an element at position *it*
- operation *bool empty*( ) returning *true* if the container is empty, false otherwise

That means, that LEDA lists can be used as well as containers.

2. *Edit Scenes*

   A scene of type *GeoEditScene*<*CONTAINER*> is a user editable scene. *GeoWin* is an editor for this kind of scenes.

3. *Result Scenes*

   A scene of type *GeoResultScene*<*I*, *R*> is not independently editable. Instead its contents (a container of type *R*) is computed by a user defined update function or update object with an update function. The update function computes the contents of the result scene every time when another scene (the input scene for the result scene) changes. The update function *void* (**f_update*)(*const I*& *input*, *R*& *result*) gets the contents of this input scene (a container *input* of type *I*) and computes the contents *result* of the result scene. We say that the result scene *depends* on its input scene.

Both Edit and Result scenes are derived from Basic scenes. The objects stored in the containers of the scenes have to support input and output operators for streams and the LEDA window and the output operator to the *ps_file*.

Please note that the type *geo_scene* is the scene item type used in *GeoWin*. You can interpret them as pointers to scenes.

#include < *LEDA/geowin.h* >

2. Creation

GeoWin *GW*(*const char* * *label* = "*GEOWIN*");

creates a GeoWin *GW*. *GW* is constructed with frame label *label*

GeoWin *GW*(*double x*, *double y*, *const char* * *label* = "*GEOWIN*");

creates a GeoWin *GW*. *GW* is constructed with frame label *label* and physical size $x \times y$.

3. Operations

a) Main Operations

The *new_scene* and *get_objects* operations use member templates. If your compiler does not support member templates, you should use instead the templated functions *geowin_new_scene* and *geowin_get_objects* with GW as an additional first parameter.

All *new_scene* operations can get an as optional last parameter a pointer to a function that is used to compute the three-dimensional output of the scene. The type of such a function pointer f is

void $(*f)$(*const T*&, *d3_window*&, *GRAPH*<*d3_point*, *int*>&))

where T is the type of the container used in the scene (for instance *list*<*point*>). The function gets a reference to the container of it's scene, a reference to the output *d3_window* and to the parametrized graph describing the three-dimensional output. The function usually adds new nodes and edges to this graph. Note that every edge in the graph must have a reversal edge (and the reversal information has to be set).
Example:

```
void segments_d3(const list&lt;segment&gt;& L,d3_window& W,
                 GRAPH<d3_point,int>& H)
{
 GRAPH<d3_point,int> G;
 segment iter;
 forall(iter,L) {
   node v1 = G.new_node(d3_point(iter.source().xcoord(),
                                 iter.source().ycoord(),0));
   node v2 = G.new_node(d3_point(iter.target().xcoord(),
                                 iter.target().ycoord(),0));
   edge e1 = G.new_edge(v1,v2);
   edge e2 = G.new_edge(v2,v1);
   G.set_reversal(e1,e2);
 }
 H.join(G);
}
```

In this simple example program the function gets a list of segments. For every segment in the list two new nodes and two new edges are created. The reversal information is set for the two edges. At the end the local graph G is merged into H.

The following templated *new_scene* operation can be used to create edit scenes. The *CONTAINER* has to be a *list*<*T*> , where T is a 2d LEDA kernel type, a *d3_point* or a *d3_rat_point*.

GeoEditScene<*CONTAINER*>$*$ GW.new_scene(*CONTAINER*& c,
D3_FCN f = *NULL*)

creates a new edit scene for c. c will be edited by GW.

The following *new_scene* operations can be used to create result scenes. Result scenes use the contents of another scene (the input scene) as the input for a function (the update function). This function computes the contents of the result scene. The update function is called every time when the contents of the input scene changes. Instead of using an update function you can use an update object that encapsulates an update function. The type of this update object has to be *geowin_update*$<I, R>$ (I - type of the container in the input scene, R - type of the container in the result scene) or a class derived from it. A derived class should overwrite the virtual update function

void update(*const I*& *in*, *R*& *out*)

of the base class to provide a user defined update function. The class *geowin_update*$<I, R>$ has 3 constructors getting function pointers as arguments:

geowin_update(*void* $(*f)$(*const I*& *in*, *R*& *res*)

geowin_update(*void* $(*f)$(*const I*& *in*, *R*:: *value_type*& *obj*)

geowin_update(*R*:: *value_type* $(*f)$(*const I*& *in*)

These functions $(*f)$ will be called in the update function of the update object. The first is the normal update function that gets the contents *in* of the input scene and computes the contents *res* of the output scene. In the second variant the contents of the result scene will first be cleared, then the update function will be called and *obj* will be inserted in the result scene. In the third variant the contents of the result scene will be cleared, and then the object returned by $(*f)$ will be inserted in the result scene. The class *geowin_update* has also the following virtual functions:

bool insert(*const InpObject*& *new*)

bool del(*const InpObject*& *new*)

bool change(*const InpObject*& *old_obj*, *const InpObject*& *new_obj*)

where *new* is a new inserted or deleted object and *old_obj* and *new_obj* are objects before and after a change. *InpObject* is the value type of the container of the input scene. With these functions it is possible to support incremental algorithms. The functions will be called, when in the input scene new objects are added (*insert*), deleted (*del*) or changed when performing a move or rotate operation (*change*). In the base class these functions return *false*. That means, that the standard update-function of the update object should be used. But in derived classes it is possible to overwrite these functions and provide user-defined update operations for these three incremental operations. Then the function has to return *true*. That means, that the standard update function of the update object should not be used.

It is also possible to provide user defined redraw for a scene. For this purpose we use redraw objects derived from *geowin_redraw*. The derived class has to overwrite the virtual redraw function *void draw(window& W, color c1, color c2, double x1, double y1, double x2, double y2)*

of the base class to provide a user defined redraw function. The first 3 parameters of this function are the redraw window and the first and second drawing color (*color* and *color2*) of the scene. The class *geowin_redraw* has also a virtual method

bool draw_container( )

that returns *false* in the base class. If you want the user defined redraw of the scene (provided by the redraw function *draw*) and the normal redraw of the scene as well (output of the objects stored in the container of the scene), you have to overwrite *draw_container* in a derived class by a function returning *true*.

In update- and redraw- functions and objects the following static member functions of the *GeoWin* class can be used:

GeoWin ∗ GeoWin::get_call_geowin( )
geo_scene GeoWin::get_call_scene( )
geo_scene GeoWin::get_call_input_scene( )

The first function returns a pointer to the *GeoWin* of the calling scene, the second returns the calling scene and the third (only usable in update functions/ objects) returns the input scene of the calling scene.

Note that S and R in the following operations are template parameters. S and R have to be a *list<T>*, where T is a 2d LEDA kernel type, a *d3_point* or a *d3_rat_point*. S is the type of the contents of the input scene, R the type of the contents of the created result scene.

GeoResultScene<S, R>∗ *GW*.new_scene(*void (∗f_update)(S, R&), geo_scene sc, string str, D3_FCN f = NULL*)

creates a new result scene with name *str*. The input scene for this new result scene will be *sc*. The update function will be *f_update*.

GeoResultScene<S, R>∗ *GW*.new_scene(*geowin_update<S, R>& up, geo_scene sc_input, string str, D3_FCN f = NULL*)

creates a new result scene with name *str*. The input scene for this new result scene will be *sc_input*. The update object will be *up*.

void *GW*.set_update(*geo_scene res, geowin_update<S, R>& up*)

makes *up* the update object of *res*.

void *GW*.set_update(*geo_scene res, void (∗f_update)(S, R&)*)

makes *f_update* the update function of *res*.

GeoResultScene<*S*, *R*>$*$ GW.new_scene(*geowin_update*<*S*, *R*>& *up*, *geowin_redraw*& *rd*, *geo_scene sc_input*, *string str*, *D3_FCN f = NULL*)

creates a new result scene with name *str*. The input scene for this new result scene will be *sc_input*. The update object will be *ub*. The redraw object will be *rd*.

GeoResultScene<*S*, *R*>$*$ GW.new_scene(*geowin_update*<*S*, *R*>& *up*, *geowin_redraw_container*<*R*>& *rd*, *geo_scene sc_input*, *string str*, *D3_FCN f = NULL*)

creates a new result scene with name *str*. The input scene for this new result scene will be *sc_input*. The update object will be *ub*. The redraw container object will be *rd*.

bool GW.get_objects(*CONTAINER*& *c*)

If the contents of the current edit scene matches type *CONTAINER*, then the contents of this scene is copied to *c*.

bool GW.get_objects(*geo_scene sc*, *CONTAINER*& *c*)

If the contents of scene *sc* matches type *CONTAINER*, then the contents of scene *sc* is copied to *c*.

void GW.get_selected_objects(*GeoEditScene*<*T*> $*$ *sc*, *T*& *cnt*)

returns the selected objects of scene *sc* in container *cnt*.

void GW.edit( ) starts the interactive mode of GW. The operation returns if either the *DONE* or *Quit* button was pressed.

bool GW.edit(*geo_scene sc*)

edits scene *sc*. Returns *false* when the Quit-Button was pressed, *true* otherwise.

b) Window Operations

void GW.close( ) closes GW .

double GW.get_xmin( ) returns the minimal x-coordinate of the drawing area.

double GW.get_ymin( ) returns the minimal y-coordinate of the drawing area.

double GW.get_xmax( ) returns the maximal x-coordinate of the drawing area.

double GW.get_ymax( ) returns the maximal y-coordinate of the drawing area.

void GW.display(*int x* = *window* :: *center*, *int y* = *window* :: *center*)
opens GW at (x, y).

window& GW.get_window( ) returns a reference to the drawing window.

void GW.init(*double xmin*, *double xmax*, *double ymin*)
same as *window*::init(*xmin*, *xmax*, *ymin*, *g*).

void GW.init(*double x1*, *double x2*, *double y1*, *double y2*,
int r = *GEOWIN_MARGIN*)
inializes the window so that the rectangle with lower left corner $(x1 - r, y1 - r)$ and upper right corner $(x2 + r, y2 + r)$ is visible. The window must be open. $GEOWIN\_MARGIN$ is a default value provided by *GeoWin*.

void GW.redraw( ) redraws the contents of GW (all visible scenes).

int GW.set_cursor(*int cursor_id* = −1)
sets the mouse cursor to *cursor_id*.

c) Scene and scene group Operations

void GW.destroy(*geo_scene sc*)
removes scene *sc* from GW and destroys it.

geo_scene GW.get_scene_with_name(*string nm*)
returns the scene with name *nm* or nil if there is no scene with name *nm*.

void GW.activate(*geo_scene sc*)
makes scene *sc* the active scene of GW.

geo_scene GW.get_active_scene( )
returns the active scene of GW.

bool GW.is_active(*geo_scene sc*)
returns true if *sc* is an active scene in a *GeoWin*.

The following *get* and *set* operations can be used for retrieving and changing scene parameters. All *set* operations return the previous value.

string GW.get_name(*geo_scene sc*)
returns the *name* of scene *sc*.

string GW.get_name(*geo_scenegroup gs*)
returns the *name* of scene group *gs*.

string GW.set_name(*geo_scene sc*, *string nm*)

gives scene *sc* the name *nm*. If there is already a scene with name *nm*, another name is constructed based on *nm* and is given to *sc*. The operation will return the given name.

color GW.get_color(*geo_scene sc*)

returns the boundary drawing color of scene *sc*.

color GW.set_color(*geo_scene sc*, *color c*)

sets the boundary drawing color of scene *sc* to *c*.

void GW.set_color(*geo_scenegroup gs*, *color c*)

sets the boundary drawing color of all scenes in group *gs* to *c*.

color GW.get_selection_color(*geo_scene sc*)

returns the boundary drawing color for selected objects of scene *sc*.

color GW.set_selection_color(*geo_scene sc*, *color c*)

sets the boundary drawing color for selected objects of scene *sc* to *c*.

void GW.set_selection_color(*geo_scenegroup gs*, *color c*)

sets the boundary drawing color for selected objects of all scenes in *gs* to *c*.

color GW.get_fill_color(*geo_scene sc*)

returns the fill color of *sc*.

color GW.set_fill_color(*geo_scene sc*, *color c*)

sets the fill color of *sc* to *c*. Use color *invisible* to disable filling.

void GW.set_fill_color(*geo_scenegroup gs*, *color c*)

sets the fill color of all scenes in *gs* to *c*. Use color *invisible* to disable filling.

color GW.get_text_color(*geo_scene sc*)

returns the text color of *sc*.

color GW.set_text_color(*geo_scene sc*, *color c*)

sets the text of *sc* to *c*.

void GW.set_text_color(*geo_scenegroup gs*, *color c*)

sets the text color of all scenes in *gs* to *c*.

int GW.get_line_width(*geo_scene sc*)
returns the line width of scene *sc* .

int GW.get_active_line_width(*geo_scene sc*)
returns the active line width of *sc* .

int GW.set_line_width(*geo_scene sc*, *int w*)
sets the line width for scene *sc* to *w*.

void GW.set_line_width(*geo_scenegroup gs*, *int w*)
sets the line width for all scenes in *gs* to *w*.

int GW.set_active_line_width(*geo_scene sc*, *int w*)
sets the active line width of scene *sc* to *w*.

void GW.set_active_line_width(*geo_scenegroup gs*, *int w*)
sets the active line width for all scenes in *gs* to *w*.

line_style GW.get_line_style(*geo_scene sc*)
returns the line style of *sc*.

line_style GW.set_line_style(*geo_scene sc*, *line_style l*)
sets the line style of scene *sc* to *l*.

void GW.set_line_style(*geo_scenegroup gs*, *line_style l*)
sets the line style of all scenes in *gs* to *l*.

bool GW.get_visible(*geo_scene sc*)
returns the visible flag of scene *sc*.

bool GW.set_visible(*geo_scene sc*, *bool v*)
sets the visible flag of scene *sc* to *v*.

void GW.set_visible(*geo_scenegroup gs*, *bool v*)
sets the visible flag of all scenes in *gs* to *v*.

void GW.set_all_visible(*bool v*)
sets the visible flag of all scenes that are currently in GW to *v*.

point_style GW.get_point_style(*geo_scene sc*)
returns the point style of *sc*.

point_style GW.set_point_style(*geo_scene sc*, *point_style p*)
sets the point style of *sc* to *p*

void GW.set_point_style(*geo_scenegroup gs*, *point_style p*)
sets the point style of all scenes in *gs* to *p*

bool GW.get_cyclic_colors(*geo_scene sc*)

returns the cyclic colors flag for editable scene *sc*.

bool GW.set_cyclic_colors(*geo_scene sc*, *bool b*)

sets the cyclic colors flag for editable scene *sc*. If the cyclic colors flag is set, the new inserted objects of the scene get color *counter*%16, where counter is the object counter of the scene.

The following operations can be used to set/get individual attributes of objects in scenes. All set operations return the previous value. The first parameter is the scene, where the object belongs to. The second parameter is a generic pointer to the object. Precondition: the object belongs to the scene (is in the container of the scene).

color GW.get_obj_color(*GeoBaseScene*<*T*> $*$ *sc*, *void* $*$ *adr*)

returns the boundary color of the object at $(*adr)$.

color GW.set_obj_color(*GeoBaseScene*<*T*> $*$ *sc*, *void* $*$ *adr*, *color c*)

sets the boundary color of the object at $(*adr)$ to c.

bool GW.get_obj_color(*GeoBaseScene*<*T*> $*$ *sc*, T :: *value_type obj*, *color*& *c*)

if there is an object o in the container of scene *sc* with $o == obj$ the boundary color of o is assigned to c and *true* is returned. Otherwise false is returned.

bool GW.set_obj_color(*GeoBaseScene*<*T*> $*$ *sc*, T :: *value_type obj*, *color c*)

if there is an object o in the container of scene *sc* with $o == obj$ the boundary color of o is set to c and true will be returned. Otherwise false will be returned.

color GW.get_obj_fill_color(*GeoBaseScene*<*T*> $*$ *sc*, *void* $*$ *adr*)

returns the interior color of the object at $(*adr)$.

color GW.set_obj_fill_color(*GeoBaseScene*<*T*> $*$ *sc*, *void* $*$ *adr*, *color c*)

sets the interior color of the object at $(*adr)$ to c.

bool GW.get_obj_fill_color(*GeoBaseScene*<*T*> $*$ *sc*, T :: *value_type obj*, *color*& *c*)

if there is an object o in the container of scene *sc* with $o == obj$ the interior color of o is assigned to c and *true* is returned. Otherwise false is returned.

bool GW.set_obj_fill_color(*GeoBaseScene*<*T*> $*$ *sc*, T :: *value_type obj*, *color c*)

if there is an object o in the container of scene *sc* with $o == obj$ the interior color of o is set to c and true will be returned. Otherwise false will be returned.

line_style GW.get_obj_line_style(*GeoBaseScene*<*T*> $*$ *sc*, *void* $*$ *adr*)

returns the line style of the object at $(*adr)$.

line_style GW.set_obj_line_style(*GeoBaseScene*<*T*> $*$ *sc*, *void* $*$ *adr*, *line_style* *l*)

sets the line style of the object at $(*adr)$ to l.

bool GW.get_obj_line_style(*GeoBaseScene*<*T*> $*$ *sc*, T :: *value_type* *obj*, *line_style*& *l*)

if there is an object o in the container of scene sc with $o == obj$ the line style of o is assigned to l and *true* is returned. Otherwise false is returned.

bool GW.set_obj_line_style(*GeoBaseScene*<*T*> $*$ *sc*, T :: *value_type* *obj*, *line_style* *l*)

if there is an object o in the container of scene sc with $o == obj$ the line style of o is set to l and true will be returned. Otherwise false will be returned.

int GW.get_obj_line_width(*GeoBaseScene*<*T*> $*$ *sc*, *void* $*$ *adr*)

returns the line width of the object at $(*adr)$.

int GW.set_obj_line_width(*GeoBaseScene*<*T*> $*$ *sc*, *void* $*$ *adr*, *int* *w*)

sets the line width of the object at $(*adr)$ to w.

bool GW.get_obj_line_width(*GeoBaseScene*<*T*> $*$ *sc*, T :: *value_type* *obj*, *int*& *l*)

if there is an object o in the container of scene sc with $o == obj$ the line width of o is assigned to l and *true* is returned. Otherwise false is returned.

bool GW.set_obj_line_width(*GeoBaseScene*<*T*> $*$ *sc*, T :: *value_type* *obj*, *int* *l*)

if there is an object o in the container of scene sc with $o == obj$ the line width of o is set to l and true will be returned. Otherwise false will be returned.

void GW.reset_obj_attributes(*GeoBaseScene*<*T*> $*$ *sc*)

deletes all individual attributes of objects in scene $(*sc)$.

d) Input and Output Operations

void GW.read(*geo_scene* *sc*, *istream*& *is*)

reads the contents of sc from input stream is. Before the contents of sc is cleared.

void GW.write(*geo_scene* *sc*, *ostream*& *os*)

writes the contents of sc to output stream os.

void GW.write_active_scene(*ostream*& *os*)

writes the contents of the active scene of GW to output stream os.

e) View Operations

void	GW.zoom_up( )	The visible range is reduced to the half.
void	GW.zoom_down( )	The visible range is doubled.
void	GW.fill_window( )	changes window coordinate system, so that the objects of the currently active scene fill the window.
string	GW.get_bg_pixmap( )	returns the name of the current background pixmap.
string	GW.set_bg_pixmap(*string pix_name*)	changes the window background pixmap to pixmap with name *pix_name*. Returns the name of the previous background pixmap.
color	GW.get_bg_color( )	returns the current background color.
color	GW.set_bg_color(*color c*)	sets the background color to c and returns its previous value.

f) Handling of events

GeoWin provides operations for changing its default handling of events. As in *GraphWin* (cf. 14.6) the user can define what action should follow a mouse or key event. Constants are defined as in *GraphWin* :

- A_LEFT (left mouse-button)
- A_MIDDLE (middle mouse-button)
- A_RIGHT (right mouse-button)
- A_SHIFT (shift-key)
- A_CTRL (control-key)
- A_ALT (alt-key)
- A_DOUBLE (double click)
- A_DRAG (button not released)
- A_IMMEDIATE (do it immediatly without dragging or double click check)
- A_OBJECT (editable object at mouse position).

and can be combined with OR (|).

void GW.set_action(*long mask*, *geo_action* $f = 0$)
set action on condition *mask* to f. *geo_action* is a function of type *void* $(*)(GeoWin\&,$ *const point*&). For $f == 0$ the corresponding action is deleted.

geo_action GW.get_action(*long mask*)
get action defined for condition *mask*.

void GW.reset_actions( ) set all actions to their default values.

Default values are defined as follows :

- A_LEFT or A_LEFT | A_OBJECT
  read a new object at mouse position.
- A_LEFT | A_DRAG
  scrolling the window.
- A_LEFT | A_DRAG | A_OBJECT
  move the object.
- A_LEFT | A_CTRL
  pin current scene at mouse position or delete the pin point if it is currently there.
- A_MIDDLE | A_OBJECT
  toggle the selection state of the object at mouse position.
- A_MIDDLE | A_DRAG
  toggle the selection state of the objects in the dragging area.
- A_RIGHT | A_IMMEDIATE
  set the options of the currently active scene.
- A_RIGHT | A_IMMEDIATE | A_OBJECT
  opens a menu for the object at mouse position.

void GW.clear_actions( ) clears all actions.

Scene events

The following event handling functions can be set for edit scenes:

- Pre add handler
- Pre add change handler
- Post add handler

- Pre delete handler
- Post delete handler
- Start, Pre, Post and End change handler

The add handlers will be called when a user tries to add an object to an edit scene in GeoWin, the delete handlers will be called when the user tries to delete an object and the change handlers will be called when the user tries to change an object (for instance by moving it). The templated set operations for setting handlers uses member templates. If your compiler does not support member templates, you should use instead the templated functions *geowin_set_HANDLER*, where *HANDLER* is one the following handlers. All handling functions get as the first parameter a reference to the *GeoWin*, where the scene belongs to.

bool GW.set_pre_add_handler(*GeoEditScene*<*T*> $*$ *sc*, *F handler*)

sets the handler that is called before an object is added to ($*sc$). *handler* must have type *bool* ($*$*handler*)(*GeoWin*&, *const T*:: *value_type* &). *handler* gets a reference to the added object. If *handler* returns false, the object will not be added to the scene.

bool GW.set_post_add_handler(*GeoEditScene*<*T*> $*$ *sc*, *F handler*)

sets the handler that is called after an object is added to ($*sc$). *handler* must have type *void* ($*$*handler*)(*GeoWin*&, *const T*:: *value_type* &). *handler* gets a reference to the added object.

bool GW.set_pre_del_handler(*GeoEditScene*<*T*> $*$ *sc*, *F handler*)

sets the handler that is called before an object is deleted from ($*sc$). *handler* must have type *bool* ($*$*handler*)(*GeoWin*&, *const T*:: *value_type* &). *handler* gets a reference to the added object. If *handler* returns true, the object will be deleted, if *handler* returns false, the object will not be deleted.

bool GW.set_post_del_handler(*GeoEditScene*<*T*> $*$ *sc*, *F handler*)

sets the handler that is called after an object is deleted from ($*sc$). *handler* must have type *void* ($*$*handler*)(*GeoWin*&, *const T*:: *value_type* &).

bool GW.set_start_change_handler(*GeoEditScene*<*T*> $*$ *sc*, *F handler*)

sets the handler that is called when a geometric object from ($*sc$) starts changing (for instance when you move it or rotate it). *handler* must have type *bool* ($*$*handler*)(*GeoWin*&, *const T*:: *value_type* &). The handler function gets a reference to the object.

bool GW.set_pre_move_handler(*GeoEditScene*<*T*> ∗ *sc*, *F handler*)

sets the handler that is called before every move operation. *handler* must have type *bool* (∗*handler*)(*GeoWin*&, *const T*:: *value_type* &, *double x*, *double y*). The handler gets as the second parameter a reference to the object, as the third parameter and fourth parameter the move vector. If the handler returns true, the change operation will be executed, if the handler returns false, it will not be executed.

bool GW.set_post_move_handler(*GeoEditScene*<*T*> ∗ *sc*, *F handler*)

sets the handler that is called after every move operation. *handler* must have type *void* (∗*handler*)(*GeoWin*&, *const T*:: *value_type* &, *double x*, *double y*). The handler gets as the second parameter a reference to the object, as the third parameter and fourth parameter the move vector.

bool GW.set_pre_rotate_handler(*GeoEditScene*<*T*> ∗ *sc*, *F handler*)

sets the handler that is called before every rotate operation. *handler* must have type *bool* (∗*handler*)(*GeoWin*&, *const T*:: *value_type* &, *double x*, *double y*, *double a*). If the handler returns true, the rotate operation will be executed, if the handler returns false, it will not be executed.

bool GW.set_post_rotate_handler(*GeoEditScene*<*T*> ∗ *sc*, *F handler*)

sets the handler that is called after every rotate operation. *handler* must have type *void* (∗*handler*)(*GeoWin*&, *const T*:: *value_type* &, *double x*, *double x*, *double a*).

bool GW.set_end_change_handler(*GeoEditScene*<*T*> ∗ *sc*, *F handler*)

sets the handler that is called when a geometric object from (∗*sc*) ends changing. *handler* gets the object as the second parameter. *handler* must have type *void* (∗*handler*)(*GeoWin*&, *const T*:: *value_type* &).

Generator functions: The following operation can be used to set a generator function for an edit scene. The operation uses member templates. If your compiler does not support member templates, you should use instead the templated function *geowin_set_generate_fcn*.

bool GW.set_generate_fcn(*GeoEditScene*<*T*> $*$ *sc*, *void* $(*f)$(*GeoWin*& *gw*, *T*& *L*))

sets the generator function for edit scene ($*sc$). The function gets the GeoWin where ($*sc$) belongs to and a reference to the container L of ($*sc$). The function should write the generated objects to L.

Input objects: The following operation can be used to set an input object for an edit scene. The operation uses member templates. If your compiler does not support member templates, you should use instead the templated functions prefixed with *geowin*. A GeoInputObject<GeoObj> has the following virtual functions:

void operator()(GeoWin& gw, list<GeoObj>& L);

This virtual function is called for the input of objects. The new objects have to be returned in L.

void options(GeoWin& gw);

This function is called for setting options for the input object.

bool GW.set_input_object(*GeoEditScene*<*T*> $*$ *sc*, *GeoInputObject*< *T*::*value_type*> *obj*, *string name*)

sets the input object *obj* for edit scene ($*sc$). The function gets the GeoWin where ($*sc$) belongs to and a reference to a list L. The function must write the new objects to L.

bool GW.add_input_object(*GeoEditScene*<*T*> $*$ *sc*, *GeoInputObject*< *T*::*value_type*> *obj*, *string name*)

adds the input object *obj* to the list of available input objects of edit scene ($*sc$) without setting *obj* as input object.

void GW.set_draw_object_fcn(*GeoBaseScene*<*T*> $*$ *sc*, *window*& $(*fcn)$(*window*& , *T*::*value_type*))

sets a function *fcn* for scene ($*sc$) that will be called for drawing the objects of scene ($*sc$). If no such function is set (the default), the output operator is used.

void GW.set_quit_handler(*bool* $(*f)$(*GeoWin gw*))

sets a handler function f that is called when the user clicks the quit menu button. f should return true for allowing quiting, false otherwise.

void GW.set_done_handler(*bool* $(*f)$(*GeoWin gw*))

sets a handler function f that is called when the user clicks the done menu button. f should return true for allowing quiting, false otherwise.

g) Scene group Operations *GeoWin* can manage scenes in groups. It is possible to add and remove scenes to/from groups. Various parameters and dependences can be set for whole groups. Note that *geo_scenegroup* is a pointer to a scene group.

geo_scenegroup GW.new_scenegroup(*string name*)

Creates a new scene group with name *name* and returns a pointer to it.

geo_scenegroup GW.new_scenegroup(*string name*, *list*<*geo_scene*> *LS*)

Cretes a new scene group *name* and adds the scenes in *LS* to this group.

void GW.insert(*geo_scenegroup GS*, *geo_scene sc*)

adds *sc* to scene group *GS* .

bool GW.del(*geo_scenegroup GS*, *geo_scene sc*)

removes *sc* from scene group *GS* and returns true, if the operation was succesful (false: *sc* was not in *GS*).

h) Further Operations

int GW.set_button_width(*int* w)

sets the width of the scene visibility buttons in GW and returns the previous value.

int GW.set_button_height(*int* h)

sets the height of the scene visibility buttons in GW and returns the previous value.

You can associate a) buttons with labels or b) bitmap buttons with the visibility of a scene in GeoWin. You cannot use a) and b) at the same time. The following operations allow you to use add such visibility buttons to GeoWin. Note that before setting bitmap buttons with the *set_bitmap* operation you have to set the button width and height.

void GW.set_label(*geo_scene sc*, *string label*)

associates a button with label *label* with the visibility of scene *sc*.

void GW.set_bitmap(*geo_scene sc*, *char* $*$ *bitmap*)

associates a button with bitmap *bitmap* with the visibility of scene *sc*.

void GW.add_scene_buttons(*list*<*geo_scene*> *Ls*, *list*<*string*> *Ln*)

add a multiple choice panel for visibility of the scenes in *Ls* to GW. The button for the n-th scene in *Ls* gets the n-th label in *Ln*.

void GW.add_scene_buttons(*list*<*geo_scene*> *Ls*, *int w*, *int h*, *char* $**$*bm*)

add a multiple choice panel for visibility of the scenes in *Ls* to GW. The button for the n-th scene in *Ls* gets the n-th bitmap in *bm*. The bitmaps have width w and height h.

list<*geo_scene*> GW.get_scenes( ) returns the scenes of GW.

list<*geo_scenegroup*> GW.get_scenegroups( )

returns the scene groups of GW.

list<*geo_scene*> GW.get_scenes(*geo_scenegroup gs*)

returns the scenes of group *gs*.

list<*geo_scene*> GW.get_visible_scenes( )

returns the visible scenes of GW.

void GW.add_dependence(*geo_scene sc1*, *geo_scene sc2*)

makes *sc2* dependent from *sc1*. That means that *sc2* will be updated when the contents of *sc1* changes.

void GW.del_dependence(*geo_scene sc1*, *geo_scene sc2*)

deletes the dependence of scene *sc2* from *sc1*.

void GW.set_frame_label(*const char* $*$ *label*)

makes *label* the frame label of GW.

int GW.open_panel(*panel*& P)

displays panel P centered on the drawing area of GW, disabels the menu bar of GW and returns the result of *P.open*( ).

void GW.enable_menus( ) enables the menus of GW.

void GW.disable_menus( ) disables the menus of GW, but not the *User* menu.

double GW.version( ) returns the *GeoWin* version number.

void GW.message(*string msg*)

displays message *msg* on top of the drawing area. If *msg* is the empty string, a previously written message is deleted.

void GW.msg_open(*string msg*)
displays message *msg* in the message window of GW. If the message window is not open, it will be opened.

void GW.msg_close( ) closes the message window.

void GW.msg_clear( ) clears the message window.

void GW.set_d3_fcn(*geo_scene sc*, *void* $(*f)$(*geo_scene gs*, *d3_window*& *W*, *GRAPH*<*d3_point*, *int*>& *H*))
sets a function for computing 3d output. The parameters of the function are the *geo_scene* for that it will be set and a function pointer. The function f will get the scene for that it was set and the reference to a *d3_window* that will be the output window.

D3_FCN GW.get_d3_fcn(*geo_scene sc*)
returns the function for computing 3d output that is set for scene *sc*. The returned function has pointer type *void* $(*)$(*geo_scene*, *d3_window*&, *GRAPH*<*d3_point*, *int*>&).

GeoWin can be pined at a point in the plane. As standard behavior it is defined that moves of geometric objects will be rotations around the pin point.

bool GW.get_pin_point(*point*& *p*)
returns the pin point in p if it is set.

void GW.set_pin_point(*point p*)
sets the pin point to p.

void GW.del_pin_point( ) deletes the pin point.

void GW.add_help_text(*string name*)
adds the help text contained in *name.hlp* with label *name* to the help menu of the main window. The file *name.hlp* must exist either in the current working directory or in $\$LEDAROOT/incl/Help$. Note that this operation must be called before *gw.display*( ).

The templated *add_user_call* operation uses member templates. If your compiler does not support member templates, you should use instead the templated function *geowin_add_user_call* with GW as an additional first parameter.

void GW.add_user_call(*string label*, *F f*)

adds a menu item *label* to the "User" menu of GW. User defined function *geo_call*(*GeoWin*& *gw*, *F f*, *string label*) is called whenever this menu button was pressed. This menu definition has to be finished before GW is opened.

4. Non-Member Functions

GeoWin∗ get_geowin(*geo_scene sc*)

returns a pointer to the *GeoWin* of *sc*.

Chapter 15

Implementations

15.1 List of data structures

This section lists the data structures for dictionaries, dictionary arrays, priority queues, and geometric data types currently contained in LEDA. For each of the data structures its name and type, the list of LEDA data types it can implement, and a literature reference are given. Before using a data structures *xyz* the corresponding header file <LEDA/impl/*xyz*.h> has to be included (cf. section 1.2 for an example).

15.1.1 Dictionaries

ab_tree	a-b tree	**dictionary**, d_array, sortseq	[11]
avl_tree	AVL tree	dictionary, d_array	[5]
bb_tree	BB[α] tree	dictionary, d_array, sortseq	[12]
ch_hashing	hashing with chaining	**h_array**	[52]
dp_hashing	dyn. perf. hashing	h_array	[21], [78]
pers_tree	persistent tree	**p_dictionary**	[22]
rb_tree	red-black tree	dictionary, d_array, sortseq	[38]
rs_tree	rand. search tree	dictionary, d_array, sortseq	[2]
skiplist	skip lists	dictionary, **d_array**, **sortseq**	[**?**]

15.1.2 Priority Queues

f_heap	Fibonnacci heap	**priority_queue**	[32]
p_heap	pairing heap	priority_queue	[75]
k_heap	k-nary heap	priority_queue	[52]
m_heap	monotonic heap	priority_queue	[52]
eb_tree	Emde-Boas tree	priority_queue	[26], [78]

15.1.3 Geometry

range_tree	range tree	**d2_dictionary**, **point_set**	[80], [51]
seg_tree	segment tree	**seg_set**	[7], [24]
ps_tree	priority search tree	—	[54]
iv_tree	interval tree	**interval_set**	[53], [24]
delaunay_tree	delaunay tree	**point_set**	[19]

15.2 User Implementations

In addition to the data structures listed in the previous section user-defined data structures can also be used as actual implementation parameters provided they fulfill certain requirements.

15.2.1 Dictionaries

Any class *dic_impl* that provides the following operations can be used as actual implementation parameter for the *_dictionary*< K,I,*dic_impl*> and the *_d_array*< I,E,*dic_impl*> data types (cf. sections 6.1 and 6.5).

```
class dic_impl {

  virtual int  cmp(GenPtr, GenPtr) const = 0;
  virtual int  int_type()          const = 0;
  virtual void clear_key(GenPtr&)  const = 0;
  virtual void clear_inf(GenPtr&)  const = 0;
  virtual void copy_key(GenPtr&)   const = 0;
  virtual void copy_inf(GenPtr&)   const = 0;

public:

typedef ... item;

  dic_impl();
  dic_impl(const dic_impl&);
  virtual ~dic_impl();

  dic_impl& operator=(const dic_impl&);

  GenPtr key(dic_impl_item)  const;
  GenPtr inf(dic_impl_item)  const;

  dic_impl_item insert(GenPtr,GenPtr);
  dic_impl_item lookup(GenPtr) const;
  dic_impl_item first_item() const;
  dic_impl_item next_item(dic_impl_item) const;

  dic_impl_item item(void* p) const
  { return dic_impl_item(p); }

  void change_inf(dic_impl_item,GenPtr);
  void del_item(dic_impl_item);
  void del(GenPtr);
```

```
  void clear();

  int size() const;
};
```

15.2.2 Priority Queues

Any class *prio_impl* that provides the following operations can be used as actual implementation parameter for the *_priority_queue*< K,I,*prio_impl*> data type (cf. section 7.1).

```
class prio_impl $\{$

  virtual int  cmp(GenPtr, GenPtr) const = 0;
  virtual int  int_type()          const = 0;
  virtual void clear_key(GenPtr&)  const = 0;
  virtual void clear_inf(GenPtr&)  const = 0;
  virtual void copy_key(GenPtr&)   const = 0;
  virtual void copy_inf(GenPtr&)  const = 0;

public:

typedef ... item;

  prio_impl();
  prio_impl(int);
  prio_impl(int,int);
  prio_impl(const prio_impl&);
  virtual ~prio_impl();

  prio_impl& operator=(const prio_impl&);

  prio_impl_item insert(GenPtr,GenPtr);
  prio_impl_item find_min() \ const;
  prio_impl_item first_item() const;
  prio_impl_item next_item(prio_impl_item) const;

  prio_impl_item item(void* p) const
  { return prio_impl_item(p); }

  GenPtr key(prio_impl_item) const;
  GenPtr inf(prio_impl_item) const;

  void del_min();
  void del_item(prio_impl_item);
  void decrease_key(prio_impl_item,GenPtr);
  void change_inf(prio_impl_item,GenPtr);
  void clear();

  int  size()  const;
};
```

15.2.3 Sorted Sequences

Any class *seq_impl* that provides the following operations can be used as actual implementation parameter for the *_sortseq*< K,I,*seq_impl*> data type (cf. section 6.3).

```
class seq_impl  {

  virtual int  cmp(GenPtr, GenPtr) const = 0;
  virtual int  int_type()          const = 0;
  virtual void clear_key(GenPtr&)  const = 0;
  virtual void clear_inf(GenPtr&)  const = 0;
  virtual void copy_key(GenPtr&)   const = 0;
  virtual void copy_inf(GenPtr&)   const = 0;

public:

typedef ... item;

  seq_impl();
  seq_impl(const seq_impl&);
  virtual ~seq_impl();

  seq_impl& operator=(const seq_impl&);
  seq_impl& conc(seq_impl&);

  seq_impl_item insert(GenPtr,GenPtr);
  seq_impl_item insert_at_item(seq_impl_item,GenPtr,GenPtr);
  seq_impl_item lookup(GenPtr) const;
  seq_impl_item locate(GenPtr) const;
  seq_impl_item locate_pred(GenPtr) const;
  seq_impl_item succ(seq_impl_item) const;
  seq_impl_item pred(seq_impl_item) const;
  seq_impl_item item(void* p) const
  { return seq_impl_item(p); }

  GenPtr key(seq_impl_item) const;
  GenPtr inf(seq_impl_item) const;

  void del(GenPtr);
  void del_item(seq_impl_item);
  void change_inf(seq_impl_item,GenPtr);
  void split_at_item(seq_impl_item,seq_impl&,seq_impl&);
  void reverse_items(seq_impl_item,seq_impl_item);
  void clear();

  int  size()  const;
};
```

Bibliography

[1] H. Alt, N. Blum, K. Mehlhorn, M. Paul: "Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{m/\log n})$". Information Processing Letters, Vol. 37, No. 4, 237-240, 1991

[2] C. Aragon, R. Seidel: "Randomized Search Trees". Proc. 30th IEEE Symposium on Foundations of Computer Science, 540-545, 1989

[3] A.V. Aho, J.E. Hopcroft, J.D. Ullman: "Data Structures and Algorithms". Addison-Wesley Publishing Company, 1983

[4] R.K. Ahuja, T.L. Magnanti, J.B. Orlin: "Network Flows", Section 10.2. Prentice Hall, 1993

[5] G.M. Adelson-Veslkii, Y.M. Landis: "An Algorithm for the Organization of Information". Doklady Akademi Nauk, Vol. 146, 263-266, 1962

[6] I.J. Balaban: "An Optimal Algorithm for Finding Segment Intersections". Proc. of the 11th ACM Symposium on Computational Geometry, 211-219, 1995

[7] J.L. Bentley: "Decomposable Searching Problems". Information Processing Letters, Vol. 8, 244-252, 1979

[8] J.L. Bentley: "Multi-dimensional Divide and Conquer". CACM Vol 23, 214-229, 1980

[9] R.E. Bellman: "On a Routing Problem". Quart. Appl. Math. 16, 87-90, 1958

[10] J.L. Bentley, T. Ottmann: "Algorithms for Reporting and Counting Geometric Intersections". IEEE Trans. on Computers C 28, 643-647, 1979

[11] R. Bayer, E. McCreight: "Organization and Maintenance of Large Ordered Indizes", Acta Informatica, Vol. 1, 173-189, 1972

[12] N. Blum, K. Mehlhorn: "On the Average Number of Rebalancing Operations in Weight-Balanced Trees". Theoretical Computer Science 11, 303-320, 1980

[13] C. Burnikel, K. Mehlhorn, and S. Schirra: "How to compute the Voronoi diagram of line segments: Theoretical and experimental results". In *LNCS*, volume 855, pages 227–239. Springer-Verlag Berlin/New York, 1994. Proceedings of ESA'94.

[14] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra: "A strong and easily computable separation bound for arithmetic expressions involving square roots". *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, 1997.

[15] C. Burnikel. “Exact Computation of Voronoi Diagrams and Line Segment Intersections”. PhD thesis, Universität des Saarlandes, 1996.

[16] T.H. Cormen, C.E. Leiserson, R.L. Rivest: “Introduction to Algorithms”. MIT Press/McGraw-Hill Book Company, 1990

[17] D. Cheriton, R.E. Tarjan: “Finding Minimum Spanning Trees”. SIAM Journal of Computing, Vol. 5, 724-742, 1976

[18] J. Cheriyan and K. Mehlhorn: “Algorithms for Dense Graphs and Networks on the Random Access Computer”. Algorithmica, Vol. 15, No. 6, 521-549, 1996

[19] O. Devillers: “Robust and Efficient Implementation of the Delaunay Tree”. Technical Report, INRIA, 1992

[20] E.W. Dijkstra: “A Note on Two Problems in Connection With Graphs”. Num. Math., Vol. 1, 269-271, 1959

[21] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, R. Tarjan: “Upper and Lower Bounds for the Dictionary Problem”. Proc. of the 29th Annual IEEE Symposium on Foundations of Computer Science, 1988

[22] J.R. Driscoll, N. Sarnak, D. Sleator, R.E. Tarjan: “Making Data Structures Persistent”. Proc. of the 18th Annual ACM Symposium on Theory of Computing, 109-121, 1986

[23] J. Edmonds: “Paths, Trees, and Flowers”. Canad. J. Math., Vol. 17, 449-467, 1965

[24] H. Edelsbrunner: “Intersection Problems in Computational Geometry”. Ph.D. thesis, TU Graz, 1982

[25] J. Edmonds and R.M. Karp: “Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems”. Journal of the ACM, Vol. 19, No. 2, 1972

[26] P.v. Emde Boas, R. Kaas, E. Zijlstra: “Design and Implementation of an Efficient Priority Queue”. Math. Systems Theory, Vol. 10, 99-127, 1977

[27] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr: “The CGAL kernel: A basis for geometric computation”. *First ACM Workshop on Applied Computational Geometry*, 1996.

[28] I. Fary: “On Straight Line Representing of Planar Graphs”. Acta. Sci. Math. Vol. 11, 229-233, 1948

[29] F.W. Floyd: “Algorithm 97: Shortest Paths”. Communcication of the ACM, Vol. 5, p. 345, 1962

[30] L.R. Ford, D.R. Fulkerson: “Flows in Networks”. Princeton Univ. Press, 1963

[31] S. Fortune and C. van Wyk: “Efficient exact arithmetic for computational geometry”. *Proc. of the 9th Symp. on Computational Geometry*, 163–171, 1993.

[32] M.L. Fredman, and R.E. Tarjan: "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms". Journal of the ACM, Vol. 34, 596-615, 1987

[33] H.N.Gabow: "Implementation of algorithms for maximum matching on nonbipartite graphs". Ph.D. thesis, Stanford Univ., Stanford, CA, 1974

[34] H.N.Gabow: "An efficient implementation of Edmond's algorithm for maximum matching on graphs". Journal of the ACM, Vol. 23, 221-234, 1976

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides: *Design patterns*. Addison-Wesley Publishing Company, 1995

[36] A. Goralcikova, V. Konbek: "A Reduct and Closure Algorithm for Graphs". Mathematical Foundations of Computer Science, LNCS 74, 301-307, 1979

[37] K.E. Gorlen, S.M. Orlow, P.S. Plexico: "Data Abstraction and Object-Oriented Programming in C++". John Wiley & Sons, 1990

[38] L.J. Guibas, R. Sedgewick: " A Dichromatic Framework for Balanced Trees". Proceedings of the 19th IEEE Symposium on Foundations of Computer Science, 8-21, 1978

[39] Goldberg, R.E. Tarjan: "A New Approach to the Maximum Flow Problem". Journal of the ACM, Vol. 35, 921-940, 1988

[40] J.E. Hopcroft, R.M. Karp: "An $O(n^{2.5})$ Algorithm for Matching in Bipartite Graphs". SIAM Journal of Computing, Vol. 4, 225-231, 1975

[41] J.E. Hopcroft, R.E. Tarjan: "Efficient Planarity Testing". Journal of the ACM, Vol. 21, 549-568, 1974

[42] M. Himsolt: "GML: A portable Graph File Format". Technical Report, Universität Passau, 1997, cf. `http://www.fmi.uni-passau.de/~himsolt/Graphlet/GML`

[43] T. Hagerup, C. Uhrig: "Triangulating a Planar Map Without Introducing multiple Arcs", unpublished, 1989

[44] A.B. Kahn: "Topological Sorting of Large Networks". Communications of the ACM, Vol. 5, 558-562, 1962

[45] D. Knuth and S. Levy: *The CWEB System of Structured Documentation, Version 3.0*. Addison-Wesley, 1993.

[46] J.B. Kruskal: "On the Shortest Spanning Subtree of a Graph and the Travelling Salesman Problem". Proc. American Math. Society 7, 48-50, 1956

[47] D. Kühl, M. Nissen, K. Weihe: "Efficient, adaptable implementations of graph algorithms". Workshop on Algorithm Engineering, Venice, Italy, September 15-17, 1997. `http://www.dsi.unive.it/~wae97/proceedings/ONLY_PAPERS/pap4.ps.gz`

[48] D. Kühl and K. Weihe: "Data access templates". *C++ Report*, 9/7, 15 and 18-21, 1997

[49] E.L. Lawler: "Combinatorial Optimization: Networks and Matroids". Holt, Rinehart and Winston, New York, 1976

[50] S.B. Lippman: "C++Primer". Addison-Wesley, Publishing Company, 1989

[51] G.S. Luecker: "A Data Structure for Orthogonal Range Queries". Proc. 19th IEEE Symposium on Foundations of Computer Science, 28-34, 1978

[52] K. Mehlhorn: "Data Structures and Algorithms". Vol. 1–3, Springer Publishing Company, 1984

[53] D.M. McCreight: "Efficient Algorithms for Enumerating Intersecting Intervals". Xerox Parc Report, CSL-80-09, 1980

[54] D.M. McCreight: "Priority Search Trees". Xerox Parc Report, CSL-81-05, 1981

[55] M. Mignotte: "Mathematics for Computer Algebra". Springer Verlag, 1992.

[56] K. Mehlhorn, S. Näher: "LEDA, a Library of Efficient Data Types and Algorithms". TR A 04/89, FB10, Universität des Saarlandes, Saarbrücken, 1989

[57] K. Mehlhorn, S. Näher: " LEDA, a Platform for Combinatorial and Geometric Computing". Communications of the ACM, Vol. 38, No. 1, 96-102, 1995

[58] K. Mehlhorn, S. Näher: "LEDA, a Platform for Combinatorial and Geometric Computing". book, in preparation. For sample chapters see the LEDA www-pages.

[59] K. Mehlhorn and S. Näher: "Implementation of a sweep line algorithm for the straight line segment intersection problem". Technical Report MPI-I-94-160, Max-Planck-Institut für Informatik, Saarbrücken, 1994.

[60] K. Mehlhorn and S. Näher: "The implementation of geometric algorithms". In *13th World Computer Congress IFIP94*, volume 1, pages 223–231. Elsevier Science B.V. North-Holland, Amsterdam, 1994.

[61] M. Mignotte: Mathematics for Computer Algebra. Springer Verlag, 1992

[62] K. Mulmuley: Computational Geometry: An Introduction Through Randomized Algorithms. Prentice Hall, 1994

[63] D.R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley Publishing Company, 1995

[64] S. Näher: "LEDA2.0 User Manual". Technischer Bericht A 17/90, Fachbereich Informatik. Universität des Saarlandes, Saarbrücken, 1990

[65] M. Nissen: "Design Pattern Data Accessor". Proceedings of the EuroPLoP 1999.

[66] M. Nissen. Graph Iterators: "Decoupling Graph Structures from Algorithms" (masters thesis). `http://www.mpi-sb.mpg.de/m̃arco/diplom.ps.gz`

[67] M. Nissen, K. Weihe: "Combining LEDA with customizable implementations of graph algorithms". Konstanzer Schriften in Mathematik und Informatik (no. 17), Universität Konstanz, 1996. Available at `ftp://ftp.informatik.uni-konstanz.de/pub/preprints/`

[68] M. Nissen, K. Weihe: "Attribute classes in Java and language extensions". Konstanzer Schriften in Mathematik und Informatik (no. 66), Universität Konstanz, 1998. Available at `ftp://ftp.informatik.uni-konstanz.de/pub/preprints/`

[69] M. H. Overmars: Designing the computational geometry algorithms library CGAL. *In Proceedings First ACM Workshop on Applied Computational Geometry*, 1996

[70] F.P. Preparata, M.I. Shamos: "Computational Geometry: An Introduction". Springer Publishing Company, 1985

[71] W. Pugh: "Skip Lists: A Probabilistic Alternative to Balanced Trees". Communications of the ACM, Vol. 33, No. 6, 668-676, 1990

[72] N. Ramsey: "Literate programming simplified". *IEEE Software*, pages 97–105, 1994

[73] M. Stoer and F. Wagner: "A Simple Min Cut Algorithm". Algorithms – ESA '94, LNCS 855, 141- 147, 1994

[74] B. Stroustrup: "The C++Programming Language, Second Edition". Addison-Wesley Publishing Company, 1991

[75] J.T. Stasko, J.S. Vitter: "Pairing Heaps: Experiments and Analysis". Communications of the ACM, Vol. 30, 234-249, 1987

[76] R.E. Tarjan: "Depth First Search an Linear Graph Algorithms". SIAM Journal of Computing, Vol. 1, 146-160, 1972

[77] R.E. Tarjan: "Data Structures and Network Algorithms". CBMS-NSF Regional Conference Series in Applied Mathematics, Vol. 44, 1983

[78] M. Wenzel: "Wörterbücher für ein beschränktes Universum". Diplomarbeit, Fachbereich Informatik, Universität des Saarlandes, 1992

[79] A.G. White: "Graphs,Groups, and Surfaces". North Holland, 1973

[80] D.E. Willard: "New Data Structures for Orthogonal Queries". SIAM Journal of Computing, 232-253, 1985

Index

B

C

D

I

M

S

T