Hinweise zur Benutzung von LEDA

1 Allgemeines

LEDA (Library of Efficient Data types and Algorithms) ist eine Bibliothek von C++-Klassen, die am MPI in Saarbrücken entwickelt wurde und die eine Vielzahl von höheren Datenstrukturen sowie Hilfsmittel zur Visualisierung und Animation zur Verfügung stellt. Seit Februar 2001 ist Algorithmic Solutions Software GmbH alleiniger Distributor für LEDA geworden. Für eine Version von LEDA muss mittlerweile auch für Lehre und Forschung eine Nutzungsgebühr entrichtet werden. Näheres zu LEDA ist im WWW unter der folgenden URL zu finden:

`http://www.algorithmic-solutions.com/`

Im Praktikum werden wir LEDA 5.2 mit dem Compiler g++ (Version 4.2.1) benutzen.

2 Installationen

LEDA ist unter `/usr/local/LEDA` auf den den LINUX Rechnern des Lehrstuhls installiert. In Unterverzeichnis `Manual` liegen die zugehörigen Dokumentationen. Um mit LEDA zu arbeiten, sind folgende Environment-Variablen zu setzen:

bash / ksh:

```
export LEDAROOT=/usr/local/LEDA
export LD_LIBRARY_PATH=$LEDAROOT:$LD_LIBRARY_PATH
```

tcsh / csh:

```
setenv LEDAROOT /usr/local/LEDA
setenv LD_LIBRARY_PATH ${LEDAROOT}:${LD_LIBRARY_PATH}
```

3 Verwendung

Um ein C++-Programm, das LEDA verwendet, zu übersetzen und zu linken, müssen dem Compiler das Verzeichnis mit den LEDA-Includes und die LEDA-Bibliotheken bekannt gemacht werden. Ein entsprechendes `Makefile` sowie ein Beispielprogramm `dfs.C` (mit zugehörigem `control.h`) erhalten Sie auf der Praktikumswebpage.

`http://www14.in.tum.de/lehre/2008WS/algoprak/data/`

Nach Kopieren der drei Dateien in ein eigenes Verzeichnis sollte sich `dfs.C` mittels `make dfs` übersetzen lassen. Ein selbst erstelltes Programm `foo.C` kann dann amalog mit dem Aufruf `make foo` übersetzt werden. Auf der Praktikumswebpage werden im Verlauf des Praktikums Testgraphen bereitgestellt, die als Eingabe für LEDA-Programme dienen können. Sie sollten in das Verzeichnis kopiert werden, in dem auch die LEDA-Programme gestartet werden (zwecks einfacheren Einladens mittels der Funktion "Load Graph").

Soll eine Klasse `foo` aus LEDA im eigenen C++-Programm verwendet werden, so muss nur mittels `#include <LEDA/foo.h>` der entsprechende Header eingebunden werden. Einige der Klassen, die wir im Praktikum verwenden werden, sind:

`string` (ähnlich `char *` in C++, aber mit zusätzlichen Features),
`random_source` (Erzeugung von Zufallszahlen), `stack`, `queue`, `list`,
`set` (Menge von Elementen), `partition` (Partition einer Menge),
`map` (Abbildung von einem Typ auf einen anderen),
`p_queue` (Prioritäts-Warteschlange), `graph` (LEDA-Graph),
`node/edge_array` (Zuordnung von Werten zu Knoten bzw. Kanten),
`node/edge_map` (dynamische Variante von `node/edge_array`),
`node/edge_set` (Menge von Knoten bzw. Kanten),
`node_partition` (Partition der Knotenmenge eines Graphen),
`node_pq` (Prioritäts-Warteschlange von Knoten eines Graphen),
`color` (Definition von Farben), `window` (Bildschirm-Fenster) und
`GraphWin` (Darstellung von Graphen auf dem Bildschirm und Interaktion mit dem Benutzer).

Näheres zur Funktionalität und Benutzung dieser Klassen sollte z.B. im Online-Manual-Viewer nachgelesen werden. Außerdem stehen in `LEDA/basic.h` verschiedene nützliche Funktionen zur Verfügung, die im Manual-Abschnitt "misc" erklärt sind.

Als einfaches Beispiel sei kurz die Benutzung einer Queue (Warteschlange) aus LEDA skizziert. Nach `#include <LEDA/queue.h>` steht der Template-Typ `queue<T>` zur Verfügung, wobei `T` einen beliebigen Typ für die Elemente der Queue spezifizieren kann. Etwa kann mit `queue<node> Q` eine Queue von Knoten (`node` ist der Typ für Knoten eines LEDA-Graphen) deklariert werden. An diese Queue kann mit `Q.append(v)` ein Knoten angehängt und mit `v=Q.pop()` ein Knoten herausgeholt werden. Mit `Q.empty()` kann Leerheit der Queue getestet werden.

Die komplexeste im Praktikum verwendete Klasse ist wohl `GraphWin`, die Klasse zur Darstellung von Graphen auf dem Bildschirm. Wir werden diese Klasse verwenden, um den Benutzer einen Graphen eingeben oder laden zu lassen und um anschließend die Arbeitsweise und das Ergebnis unserer Algorithmen zu visualisieren. Dazu können wir vielerlei Funktionen zur Veränderung der Darstellung und der Labels von Knoten und Kanten auf dem Bildschirm einsetzen. Zu beachten ist, dass GraphWin-Graphen intern immer gerichtet sind und man ungerichtete Graphen dadurch realisiert, dass man die Kanten am Bildschirm ungerichtet darstellen lässt und die Nachbarkanten eines Knotens mittels `forall_inout_edges(e,v)` durchläuft (siehe `dfs.C`-Beispielprogramm).

Das von `dfs.C` verwendete Include-File `control.h` realisiert ein kleines Kontrollfenster, das am Programmanfang mit `create_control()` sichtbar gemacht werden muss und am Ende mit `destroy_control()` wieder geschlossen werden sollte. Das Kontrollfenster realisiert eine Art "Fernbedienung", mit der der Animationsablauf kontrolliert werden kann (Stop, Continue, etc.), wenn im Programm die Funktion `leda_wait()` für Verzögerungen verwendet wird.

Beispielprogramm: dfs.C

```
// Animation einer Tiefensuche in ungerichteten Graphen
// (unbesuchter Teil wird gelb (Default) angezeigt, erledigter Teil blau,
// noch in Bearbeitung befindlicher Teil rot, Rückwärtskanten grün)
#include <iostream>
#include <LEDA/graphics/graphwin.h>
#include <LEDA/graphics/color.h>
#include <LEDA/system/basic.h>
#include "control.h" // Fernbedienung

// Definitionen um bei den jeweiligen Klassen nicht immer eigens den
// entsprechenden namespace angeben zu müssen
using leda::graph;
using leda::node;
using leda::edge;
using leda::node_array;
using leda::user_label;
using leda::GraphWin;
using leda::red;
using leda::blue;
using leda::green;
using leda::yellow;
using leda::string;

// rekursive Funktion zur Realisierung der Tiefensuche (dfs);
// Parameter:
//    parent: Knoten, von dem aus der aktuelle Knoten besucht wird
//            (parent == v, falls der aktuelle Knoten der Startknoten ist)
//    v: aktueller Knoten
//    g: zu durchsuchender Graph (als Referenz)
//    gw: Darstellungsfenster des Graphen (als Referenz)
//    dfsnum: Feld zur Zuordnung von DFS-Nummern zu Knoten (als Referenz)
//    akt: nächste freie Nummer (als Referenz)
void dfs(node parent,
         node v,
         graph &g,
         GraphWin &gw,
         node_array<int> &dfsnum,
         int &akt) {

    dfsnum[v] = akt++;                                 // DFS-Nummer zuweisen
    gw.set_user_label(v, string("%d", dfsnum[v]));  // DFS-Nummer anzeigen
    gw.set_color(v, red);                              // Knoten rot färben
    gw.redraw();                                       // Darstellung aktualisieren
    leda_wait(0.5);                                    // 0.5 sec warten

    // GraphWin-Graphen sind immer gerichtet, auch wenn auf dem Bildschirm
    // keine Pfeile sichtbar sind

    edge e;
    forall_inout_edges(e, v) {  // alle Nachbarkanten von v
        node w = g.opposite(v, e); // Knoten am anderen Ende von e
        if (w != parent) {     // die Kante zum parent ignorieren wir
            if (dfsnum[w] < 0) { // falls Knoten w noch nicht besucht
                gw.set_color(e, red);          // Kante zu w rot färben
                gw.set_width(e, 2);            // Kante fett anzeigen
                dfs(v, w, g, gw, dfsnum, akt); // rekursiver Aufruf fuer w
                gw.set_color(e, blue);         // Kante blau färben
                leda_wait(0.5);                // 0.5 sec warten
            } else {    // Knoten w war schon besucht
                if (dfsnum[w] < dfsnum[v]) { // Rückwärtskante
```

```
                    gw.set_color(e, green);    // grün färben
                    leda_wait(0.5);            // 0.5 sec warten
                }
            }
        }
    }
    gw.set_color(v, blue); // Knoten blau färben
    gw.redraw(); // Darstellung aktualisieren (zur Sicherheit)
}

//
// Haupt-Programm
//
int main(int argc, char *argv[]) {
    // Fenster der Größe 800 x 600 zur Graphendarstellung erzeugen
    GraphWin gw(800, 600);

    gw.display(); // Fenster auf den Bildschirm bringen
    create_control(); // "Fernbedienung" anzeigen
    gw.set_directed(false); // ungerichtete Darstellung (keine Pfeile an Kanten)
    if (argc > 1)    // falls Name als Parameter, Graph laden
        gw.read(argv[1]);

    gw.edit();   // in Editier-Modus gehen, bis der Benutzer "done" klickt

    // jetzt holen wir uns den Graphen, den der Benutzer eingegeben oder geladen hat
    graph &g = gw.get_graph();

    if (g.number_of_nodes() == 0) {  // Ende, wenn der Graph leer ist.
        gw.close(); destroy_control();
        exit(1);
    }

    // Jetzt deklarieren wir ein Feld, das jedem Knoten eine Nummer zuordnet,
    // und initialisieren es mit "-1"
    node_array<int> dfsnum(g, -1);

    // Nun zeigen wir fuer alle Knoten den dfsnum-Wert als User-Label an
    // sowie initialisieren den Graphen gelb.
    node v;
    forall_nodes(v, g) {
        gw.set_label_type(v, user_label);    // User-Label anzeigen (statt Index-Label)
        gw.set_user_label(v, string("%d", dfsnum[v])); // User-Label auf dfsnum[v] setzen
        gw.set_color(v, yellow);
    }
    edge e;
    forall_edges(e, g)
        gw.set_color(e, yellow);

    // in dieser Variable merken wir uns die nächste zu vergebende Nummer
    // (und gleichzeitig die Zahl der schon besuchten Knoten)
    int akt = 0;

    do {
        // jetzt lassen wir den Benutzer mit der Maus einen unbesuchten Knoten
        // auswählen (wenn er danebenklickt, wird NULL zurückgeliefert),
        while ((v = gw.read_node()) == NULL || dfsnum[v] >= 0) ;

        // nun rufen wir die rekursive DFS-Funktion auf
        dfs(v, v, g, gw, dfsnum, akt);
    } while (akt < g.number_of_nodes()); // bis alle Knoten besucht wurden
```

```
    gw.acknowledge("Ready!"); // Dialogbox anzeigen und bestätigen lassen
    gw.edit(); // nochmal in den Edit-Modus, zum Anschauen :)

    // Aufräumen und Ende
    gw.close();
    destroy_control();
    exit(0);
}
```