Chapter 6

Topics

In this chapter we study three topics: shortest paths, splay trees and the coloring of graphs [CLR90].

6.1 Shortest Paths

The basic problem is to find the "best" way of getting from point s to point t where s and t are vertices in a graph

Best is measured by summing the length (or weight) of the edges that are along the path from s to t. In shortest path problems, we have a weighted directed graph $G = (V, E)$ with a weight function that maps edges to real-valued weights. We will assume that weights are positive, at least for the first algorithms we study. There are three variants:

single-source shortest path: we want to find a shortest path from a given source $s \in V$ to every vertex $v \in V$

single-pair shortest path: find shortest path from $u \in V$ to $v \in V$, i.e. two fixed vertices u and v

all-pairs shortest paths: find shortest path for every pair $u \in V$ and $v \in V$

6.1.1 Single-Source Shortest Path Problem

Dijkstra's algorithm is based on the greedy technique. The algorithm maintains a set S of vertices whose shortest path from the source s is already known and the length $dis[u]$ of the shortest path from the source s to each vertex u that passes only through vertices in S.

Initially, S contains no vertices. At each stage, we add to S a remaining vertex x whose distance value $dis[x]$ is minimum. A tie is broken arbitrarily. At this point, $dis[u]$ for vertex u not in S may decrease, since there may be a shorter path through x of length $dis[x] + d(x, w)$. So at each iteration, we find a shortest path among paths from the source s to a vertex that passes only through vertices in S. The process is repeated until S contains all vertices of V.

Procedure Dijkstra finds a shortest path from the source s to every other vertex $v \in V \backslash \{s\}$ in a directed graph $G = (V, E)$. For each vertex $u \in V \backslash \{s\}$, $pred[u]$ denotes the predecessor of u in a shortest path tree. $dis[u]$ denotes the length of a shortest path from s to u. Procedure Dijkstra assumes that graph G is represented by adjacency lists. It also assumes that all edges have nonnegative weights.

The procedure maintains a priority queue Q that initially contains all the vertices in V by their dis value. More precisely, dis is used as a key where dis is initialized in the following fashion:

$$dis[v] := \infty \qquad \text{for all } v \in\ V\backslash\{s\} \text{ and}$$

$$dis[s] := 0$$

which forces s to be chosen in the first iteration of the while loop.

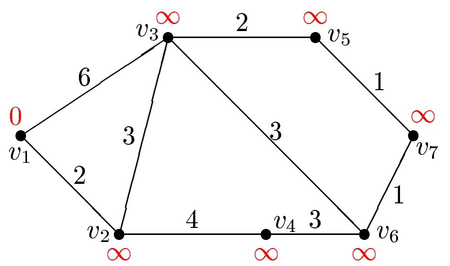
For a given graph $G = (V, E)$, a starting vertex s and a weight function $d : E \to \mathbf{R}^+$ procedure Dijkstra*(G, s)* is as follows:

Dijkstra(G, s)

---

begin
  $S \leftarrow \emptyset$
  for all $v \in V\backslash\{s\}$ **do**
    $pred[v] \leftarrow s$
initialize priority queue Q containing all vertices $v \in V\backslash\{s\}$ with key $dis[v] \leftarrow \infty$ and containing s with key $dis[s] \leftarrow 0$
  while $S \neq V$ **do**
    $v \leftarrow$ ExtractMin(Q)
    $S \leftarrow S \cup \{v\}$
    for each $w \in adj[v]$ **do**
      if $dis[v] + d(v, w) < dis[w]$
        then
          DecreaseKey$(w, dis[v] + d(v, w))$
          /* DecreaseKey changes $dis[w]$ to $dis[v] + d(v, w)$ */
          $pred[w] \leftarrow v$
end

Example:

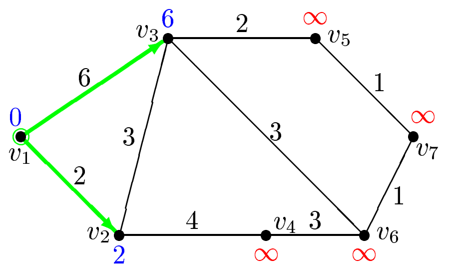


input graph G;

starting node v_1;

dis[v_1]=0;

dis[u]=$+\infty$
for all $u \in V - \{v_1\}$;

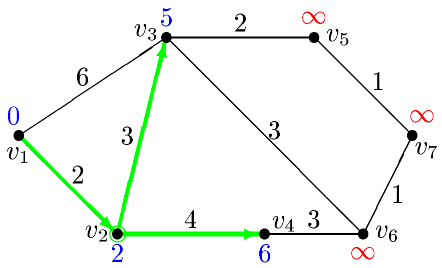


process v_1:

$adj[v_1] = \{v_2, v_3\}$

mark (v_1, v_2);
set dis[v_2]=2;

mark (v_1, v_3);
set dis[v_3]=6;



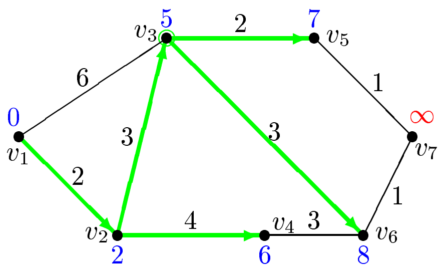
process v_2:

mark (v_2, v_3) and unmark (v_1, v_3), because $\text{dis}[v_2] + d(v_2, v_3) < \text{dis}[v_3]$;

set $\text{dis}[v_3]=2+3=5$;

mark (v_2, v_4);

set $\text{dis}[v_4]=2+4=6$;



process v_3;

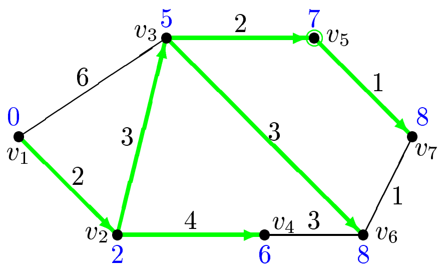
mark (v_3, v_5);

set $\text{dis}[v_5]=5+2=7$;

mark (v_3, v_6);

set $\text{dis}[v_6]=5+3=8$;

(no change when v_4 is processed)



process v_5;

mark (v_5, v_7);

set $\text{dis}[v_7]=7+1=8$;

(no more changes when v_6 and v_7 are processed)

Note that the strategy of Dijkstra's algorithm is similar to the one used for PRIM's minimum spanning tree algorithm. We grow a "dijkstra tree", starting at vertex s, by adding at each iteration a "cheap" edge whose non-tree endpoint is as close as possible to s. In PRIM's algorithm, the non-tree endpoint is as close as possible to any vertex in the current prim tree.

In both algorithms we insert one vertex at a time (build the tree by adding one edge at a time) and adjust the weights of the remaining vertices outside the "set" accordingly. Note that procedure DIJKSTRA also produces the actual minimum-weight paths themselves (not only the value of the paths stored in $dis[u]$). The edges in the "dijkstra tree" are of the form $(pred[u], u)$ for all $u \in V \backslash \{s\}$.

Proof of Dijkstra's algorithm

The key is to show that each time a vertex u is inserted into S, we have $dis[u] = \delta(s, u)$, where $\delta(s, u)$ denotes the distance of the shortest path from s to u.

The proof is based on the following facts.

Fact 1: *Shortest paths are composed of shortest subpaths*
Note that if this were not true and there exists a shorter path than any subpath, then the shorter path should replace that subpath to make the whole path shorter

Fact 2: *If $s \to \ \ldots \ \to u \to v$ is a shortest path from s to v, then after the insertion of u in S, we have $dis[u] = \delta(s, v)$, and $dis[v]$ does not change thereafter.* Note that at all times we have $dis[v] \geq \delta(s, v)$.

Theorem 6.1.1 *After the execution of procedure* DIJKSTRA*, we have* $dis[u] = \delta(s, u)$ *for all* $u \in V$*.*

Proof: The proof that $dis[u] = \delta(s, u)$ for all $u \in V$, after running the procedure, is by contradiction. Suppose that v is the first vertex added to S for which $dis[v] \neq \delta(s, v)$, i.e. $dis[v] > \delta(s, v)$. Since $dis[s] = 0$, v cannot be s. Since $dis[v] < \infty$ (v is added to S) then there must be a path from s to v. Also, since there is a path from s to v, there must be a shortest path from s to v.

Let $s \cdots x \to y \cdots v$ be a shortest path from s to v, where x is in S and y is the first vertex not in S. We know that $dis[x] = \delta(s, x)$ since we assume that v is the **first** vertex added to S for which $dis[v] \neq \delta(s, v)$. Since $y \in adj[x]$, when x is inserted in S, we perform the **if** statement, after which we have:

$$dis[y] = dis[x] + d(x, y)$$

So

$$\begin{aligned} dis[y] &= \delta(s, y) \\ &\leq \delta(s, v) \\ &\leq dis[v] \end{aligned}$$

So $dis[y] \leq dis[v]$.

But right before choosing v to insert in S, we had $v \in V \backslash S$ and $y \in V \backslash S$, and v was chosen over y. The choice was dictated by the fact that:

$$dis[v] \leq dis[y]$$

$$\left.\begin{array}{c} dis[y] \leq dis[v] \\ \text{and} \\ dis[v] \leq dis[y] \end{array}\right\} \Rightarrow \quad dis[v] = dis[y]$$

In other words,

$$\left.\begin{array}{ccc} dis[y] & = & \delta(s, y) \\ & \leq & \delta(s, x) \\ & \leq & dis[v] \\ & \text{and} & \\ dis[y] & = & dis[v] \end{array}\right\} \Rightarrow \quad dis[v] = \delta(s, v)$$

which contradicts our hypothesis.

So whenever a node u is chosen to be inserted in S, we have $dis[u] = \delta(s, u)$.

Analysis of Dijkstra's algorithm

Note that the vertices are scanned in increasing order of their shortest-path distance from s. Each vertex is scanned at most once. Its *dis*-value is updated at most once for each incoming edge, so DECREASEKEY is performed at most $|E|$ times. By implementing the priority queue Q with a fibonacci heap, we know that each call to DECREASEKEY will take only $O(1)$ amortized time to be executed.

Procedure DIJKSTRA performs EXTRACTMIN*(Q)* for each vertex in V. So if Q is implemented with a fibonacci heap, then we have $|V| \log_2 |V|$ work since each EXTRACTMIN*(Q)* operation requires $O(\log_2 |V|)$ amortized time. The total running time is therefore $O(|E| + |V| \log_2 |V|)$.

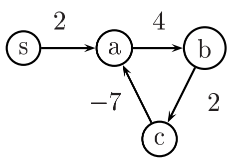
The fastest known implementation of Dijkstra's algorithm uses fibonacci heaps. The running times using other implementations are summarized in the following table:

heap implementation	ExtractMin	Insert	total
linked list	$O(\vert V\vert)$	$O(1)$	$O(\vert V\vert^2)$
binary heap	$O(\log_2 \vert V\vert)$	$O(\log_2 \vert V\vert)$	$O(\vert E\vert \log_2 \vert V\vert)$
d-ary heap	$O\left(\frac{d \log_2 \vert V\vert}{\log_2 d}\right)$	$O\left(\frac{\log_2 \vert V\vert}{\log_2 d}\right)$	$O\left((\vert V\vert d + \vert E\vert)\frac{\log_2 \vert V\vert}{\log_2 d}\right)$
fibonacci	$O(\log_2 \vert V\vert)$ *amortized*	$O(1)$ *amortized*	$O(\vert V\vert \log_2 \vert V\vert + \vert E\vert)$

A d-ary heap is similar to a binary heap, except that each node has d children instead of just two.

Negative-weight edges

In some instances of the single-source shortest-paths problem, there may be edges whose weights are negative. As long as the graph $G = (V, E)$ does not contain a negative-weight cycle reachable from the source s, then for all $v \in V$, the shortest path weight $\delta(s, v)$, defined in the previous section, remains well defined, even if some of the edges have a negative value.



Vertices a, b and c form a negative-weight cycle. It does not make any sense to ask for the shortest path from s to a for instance. Is it 2, going directly from s to a, or 1, after one cycle through a, b and c or 0 after two cycles, or -1 after three cycles ...?

Dijkstra's algorithm solves the single-source shortest-path problem when all edges have nonnegative weights. The Bellmann-Ford procedure will solve the problem even if some edges have negative weights as long as G contains no negative-weight cycles.

The Bellmann-Ford algorithm

The Bellmann-Ford algorithm solves the single-source shortest-path problem in the more general case in which edge weights can be negative. But the graph should not contain negative-weight cycles that can be reached from the source s. Procedure Bellman-Ford is based on the fact that if $s, v_1, v_2, \ldots, v_k = a$ is the shortest path between s and a for $1 \leq k \leq |V| - 1$, then if we perform

for $(v, w) \in E$
  if $dis[w] > dis[v] + d(v, w)$
  then
    $dis[w] := dis[v] + d(v, w)$
    $prev[w] := v$

first with $(v, w) = (s, v_1)$
then with $(v, w) = (v_1, v_2)$
then with $(v, w) = (v_2, v_3)$ and so on till $(v, w) = (v_{k-1}, a)$,

then we are sure that $dis[a]$ will contain the value of the shortest path between s and a and that the shortest path itself is enclosed in $prev$.

In essence, as was the case with Dijkstra's algorithm, $dis[v]$ is an upper bound on the value of the shortest distance between s and v, that we progressively decrease (with each iteration of the outer loop of the algorithm) until we achieve $dis[v] = \delta(s, v)$.

```
BELLMANN-FORD(G, s)
begin
  dis[v] := 0
  for all v ∈ V\{s} do
    dis[v] := ∞
    prev[v] := nil
  for i = 1 to |V| − 1 do
    for each (v, w) ∈ E do
      if dis[w] > dis[v] + d(v, w)
      then
        dis[w] := dis[v] + d(v, w)
        prev[w] := v
end
```

Note that by demanding that all edges go through the inner **for** loop $|V| - 1$ times, we are guaranteeing that edges $(s, v_1), (v_1, v_2), \ldots, (v_{k-1}, a)$, on the shortest path between s and a, got through the inner loop in the desired order (i.e. first (s, v_1), followed by (v_1, v_2) and so on till (v_{k-1}, a)).

The Bellman-Ford algorithm runs in time $O(|V||E|)$, since the initialization takes $\Theta(|V|)$ time, and each of the $|V| - 1$ passes over the edges in the main **for** loop takes $O(|E|)$ time.

Single-source shortest path in DAG's

A DAG is a **d**irected **a**cyclic **g**raph. For this important class of graphs, shortest paths can be computed more quickly, in linear time.

Shortest paths are always well defined in a DAG, since those graphs can never contain negative weight cycles, even if they have negative-weight edges. Procedure DAG-SHORTEST-PATHS first topologically sorts the DAG. A topological ordering of a DAG is such that if we have an edge from x to y, then x precedes y in the topological sort. A topological sort of a DAG can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right. The topological sort can be performed in $\Theta(|V| + |E|)$ time.

Procedure DAG-SHORTEST-PATHS then considers each vertex (in the topologically sorted order) and processes each adjacent vertex of the chosen vertex. Processing consists in updating $dis[w]$ for each $w \in adj[v]$ when v is the chosen vertex.

DAG-SHORTEST-PATHS$(G,\ s)$

```
begin
  dis[s] := 0
  for v ∈ V\{s}
    dis[v] ← ∞
    prev[v] ← nil
  topologically sort the vertices of G
  for each vertex v taken in topologically sorted order
    for each w ∈ adj[v]
      if dis[v] + d(v,w) < dis[w]
      then
        dis[w] ← dis[v] + d(v,w)
        prev[w] ← v
end
```

The topological sort – as mentioned already above – can be done in $\Theta(|V|+|E|)$ time. The outer **for** loop considers each vertex once and so the outer loop is performed $|V|$ times. The work done on each edge leaving w is constant. The total running time of the DAG-SHORTEST-PATHS procedure is $\Theta(|V|+|E|)$.

Topological sort

A topological sort of a directed acyclic graph $G=(V,E)$ is a linear ordering of all its vertices such that if G contains an edge (u,v), then u appears before v in the ordering.

Results:

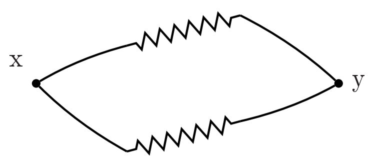
$$G=(V,E) \text{ has a topological ordering} \\ \Updownarrow \\ G \text{ is a directed acyclic graph}$$

Instead of the above, we show the equivalent result:

$$G \text{ has no topological ordering} \Leftrightarrow G \text{ contains a cycle}$$

Proof:

($\Leftarrow$): G contains a cycle, i.e.



x appears before y in the upper path
and
x appears after y in the lower path

$\Rightarrow G$ does not have a topological order

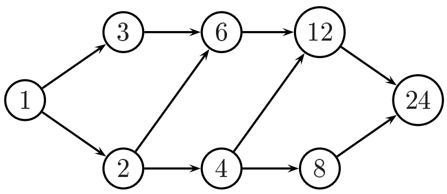
(⇒): consider the following procedure:

```
TOPOLOGICAL-SORT(G)
begin
  L := ∅;                                  holds the vertices in topological order
  S := ∅;                                  holds vertices without incoming edges
  for each vertex v ∈ V do
    I[v] := indegree[v];                   number of incoming edges to v
    if I[v] = 0
    then
      insert v in S;                       v is removed from V
  while S ≠ ∅ do
    remove a vertex x from S
    for each outgoing edge (x, w) do
      I[w] := I[w] − 1
      if I[w] = 0
      then
        insert w in S;                     w is removed from V
    insert x in L
end
```

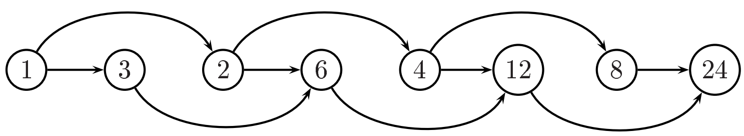
For example, Topological-Sort produces the following list L, in that order

$$L = 1\ 2\ 3\ 4\ 6\ 8\ 12\ 24$$

when the procedure is applied to the DAG shown below.



Many other solutions, including $L := 1\ 3\ 2\ 6\ 4\ 12\ 8\ 24$ are possible. Recall that a topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.



The diagram shows the graph topologically sorted ($L := 1\ 3\ 2\ 6\ 4\ 12\ 8\ 24$).

Back to the proof: If the procedure executes to completion (and does not get stuck), L will contain the elements of V topologically sorted since $u \in V$ is inserted in L only after all its incoming edges have been deleted (in other words, all predecessors of u are already in L). If procedure Topological-Sort does not execute to completion, it means that it cannot find a vertex with no incoming edges; in other words, $indegree[u] > 0$ for all u not in L. We want to show that in this case G has a cycle.

Pick one of the remaining nodes, w let us say, and find an incoming edge (t, w). We know that it exists since $indegree[w] > 0$. Likewise, let (m, t)

be an incoming edge of t. By continuing this process we are bound to have the following situation, where we eventually run into a vertex x we have seen before. This happens since there are only finitely many vertices (and all have indegrees greater than zero). So we have a cycle from x to itself. Therefore

$$G \text{ has no topological order} \Rightarrow G \text{ contains a cycle}$$

In summary, we have shown that

$$G \text{ has a topological ordering} \Leftrightarrow G \text{ is a directed acyclic graph}$$

Note that procedure TOPOLOGICAL-SORT finds a topological sorting of the graph when it exists, and can be used to find cycles when G is not a directed acyclic graph. The time complexity of the procedure is $\Theta(|V| + |E|)$.

6.1.2 Single-Pair Shortest-Path

To solve this problem, simply solve the single-source shortest-path problem from s to all vertices in V, and this will of course yield the shortest path from s to t. There are no algorithms that will find the shortest path from s to t without also finding the shortest path from s to all the other vertices in the worst case.

6.1.3 All-Pair Shortest Path

We have already studied procedure FLOYD that computes all-pairs shortest paths in $\Theta(|V|^3)$ time. The algorithm uses the dynamic programming technique.

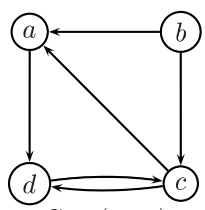
Note that this problem, as well as the other variants of the shortest path problem, has many applications including building a table of distances between all pairs of cities for a road atlas. One way of solving this problem is by running a single-source shortest paths algorithm $|V|$ times, one for each vertex as the source s. If the edges of G are nonnegative, we can use Dijkstra's algorithm. By using the fibonnaci heap implementation (as priority queue), we get a running time of $O(|V|^2 \log_2 |V| + |V||E|)$.

If on the other hand we have negative-weight edges, we could use BELLMAN-FORD procedure $|V|$ times, once from each vertex. The running time is then $O(|V|^2|E|)$.

Transitive closure of a directed graph

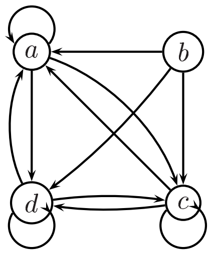
Given a directed Graph $G = (V, E)$, we would like to know whether there is a path in G from $u \in V$ to $v \in V$ for all vertex pairs $u, v \in V$. The transitive closure answers the question. The transitive closure of G is the graph $G^* = (V, E^*)$, where

$$E^* = \{(u, v)| \text{ there is a path from } u \text{ to } v \text{ in } G.\}$$



$G = (V, E)$

Procedure TRANSITIVE CLOSURE is similar to procedure FLOYD for all pairs shortest paths that uses the dynamic programming technique. The main difference is that here the edges do not have weights.



Transitive closure of G

We are not interested in finding the shortest paths, but rather if paths exist. We define a matrix D such that entry $d_{ij} = 1$ if there is a path from i to j, and $d_{ij} = 0$ if no such path exists. We are assuming that $V = \{1, 2, \ldots, n\}$. Procedure TRANSITIVE-CLOSURE*(G)* will progressively build D.

Initially, D = incidence matrix of G with ones in main diagonal. At the k^{th} iteration of procedure TRANSITIVE-CLOSURE*(G)*, the entry $d_{ij}^{(k)}$ of D will be

one if there is a path from vertex i to vertex j with all intermediate vertices in $\{1, 2, \ldots, k\}$, and

zero otherwise

$G^* = (V, E^*)$ is thus constructed by putting edge (i, j) in E^* if and only if $d_{ij}^{(n)} = 1$. The recursive definition of d_{ij}^{k} is :

$$d_{ij}^{(0)} = \begin{cases} 0 \text{ if } i \neq j \text{ and } (i,j) \notin E \\ 1 \text{ if } i = j \text{ or } (i,j) \in E \end{cases}$$

and

$$d_{ij}^{(k)} = d_{ij}^{(k-1)} \vee (d_{ik}^{(k-1)} \wedge d_{kj}^{(k-1)}) \quad \text{for } k \geq 1$$

As in the FLOYD algorithm, procedure TRANSITIVE-CLOSURE *(G)* computes the matrices D with entries $d_{ij}^{(k)}$ in order of increasing k.

TRANSITIVE-CLOSURE*(G)*

begin
  $n := |V|$
  for $i := 1$ **to** n
    for $j := 1$ **to** n
      if $i = j$ **or** $(i, j) \in E$
        then $d_{ij}^{(0)} := 1$
        else $d_{ij}^{(0)} := 0$
  for $k := 1$ **to** n
    for $i := 1$ **to** n
      for $j := 1$ **to** n
        $d_{ij}^{(k)} := d_{ij}^{(k-1)} \vee (d_{ik}^{(k-10)} \wedge d_{kj}^{(k-1)})$
  return D *entries of D are* $d_{ij}^{(n)}$
end

The time complexity of the procedure is $\Theta(n^3)$, where $|V| = n$.

Johnson's all-pair shortest-paths algorithm

Johnson's algorithm finds shortest paths between all pairs in $O(|V|^2 \log_2 V + |V||E|)$ time. It makes use of Dijkstra's algorithm and Bellman-Ford's algorithm.

Assume that $G = (V, E)$ is a weighted, directed graph and that $w : E \to \mathbf{R}$ is the weight function. If G has negative weight-edges, we use a new weight function $\overline{w}$ to recompute the weights of the edges of G.

Ideally we would like to define $\overline{w} : E \to \mathbf{R}$ such that:

1. for all pairs of vertices $u, v \in V$, a shortest path from u to v using weight function w is also a shortest path from u to v using $\overline{w}$.
2. for all edges (u, v), the new weight $\overline{w}(u, v)$ is nonnegative.

Let $\overline{\delta}$ denote the shortest-path weights derived from $\overline{w}$. The process of recomputing the weights (according to $\overline{w}$) is called reweighting. In other words, we would like to reweight the edges such that:

1. shortest paths are preserved, and
2. we don't have any negative weight edges.

Recall, $G = (V, E)$ is a weighted, directed graph with weight function $w : E \to \mathbf{R}$. Let $h : V \to \mathbf{R}$ be any function mapping, and define:

$$\overline{w}(u, v) = w(u, v) + h(u) - h(v)$$

for all edges (u, v) in G.

Claim: $\overline{w} : E \to \mathbf{R}$ as defined above preserves shortest paths.
In other words, if $v_0 \to v_1 \cdots \to v_k$ is a path P in G then

$$w(P) = \delta(v_0, u_k) \Leftrightarrow \overline{w}(P) = \overline{\delta}(v_0, v_k)$$

First, let us write $\overline{w}(P)$ as a function of $w(P)$.

$$\begin{aligned}
\overline{w}(P) &= \overline{w}(v_0, v_1) + \overline{w}(v_1, v_2) + \cdots + \overline{w}(v_{k-1}, v_k) = \\
&= [w(v_0, v_1) + h(v_0) - h(v_1)] + [w(v_1, v_2) + h(v_1) - h(v_2)] + \\
&\quad + \cdots + [w(v_{k-1}, v_k) + h(v_{k-1}) - h(v_k)] \\
&= [w(v_0, v_1) + h(v_0)] + [w(v_1, v_2)] + \cdots + [w(v_{k-1}, v_k) - h(v_k)] \\
&= w(P) + h(v_0) - h(v_k)
\end{aligned}$$

Now we show that reweighting does not change shortest paths, that is

$$w(P) = \delta(v_0, v_k) \Leftrightarrow \overline{w}(P) = \overline{\delta}(v_0, v_k)$$

Proof

($\Rightarrow$) the proof is by contradiction. Suppose that $\overline{w}(P) = \overline{\delta)}(v_0, v_k)$ is not the shortest path from v_0 to v_k under function $\overline{w}$.

So there is a shorter path P' from v_0 to v_k using $\overline{w}$. That is $\overline{w}(P') < \overline{w}(P)$. But with $\overline{w}(P') < \overline{P}$

$$\begin{aligned}
&\Rightarrow \quad w(P') + h(v_0) - h(v_k) < w(P) + h(v_0) - h(v_k) \\
&\Rightarrow \quad w(P') < w(P)
\end{aligned}$$

which contradicts the assumption that P is a shortest path from v_0 to v_k using w.

($\Leftarrow$) similar argument to the one above $\square$

Remark: We can use a similar argument to show that

G has a negative-weight cycle using w	$\Leftrightarrow$	G has a negative-weight cycle using $\overline{w}$

Reweighting starts by augmenting V. Given a weighted, directed graph $G = (V, E)$, and $w : E \rightarrow \mathbf{R}$, we construct $G' = (V', E')$ where:

$$V' = V \cup \{s\}, S \notin V \text{ and } E' = E \cup \{(s, v) | v \in V\}$$

with $w(s, v) = 0$ for all $v \in V$.

Note that:

G' has no negative–weight cycles	$\Leftrightarrow$	G has no negative-weight cycles

Let $h(v) = \delta(s, v)$ for all $v \in V'$. Recall that for all edges $(u, v) \in E$ in a directed graph, we have:

$$\delta(s, v) \leq \delta(s, u) + w(u, v).$$

So $h(v) \leq h(u) + w(u, v)$

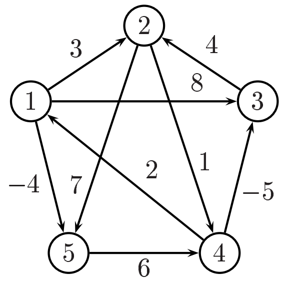
$$\begin{aligned} &\Rightarrow \quad w(u, v) + h(u) - h(v) \geq 0 \\ &\Rightarrow \quad \overline{w}(u, v) \geq 0 \end{aligned}$$

And so the second property (of not having negative-weight edges) is satisfied. Thus, $\overline{w} : E \rightarrow \mathbf{R}$, where $\overline{w}(u, v) = w(u, v) + h(u) - h(v)$ satisfies the two conditions mentioned earlier.

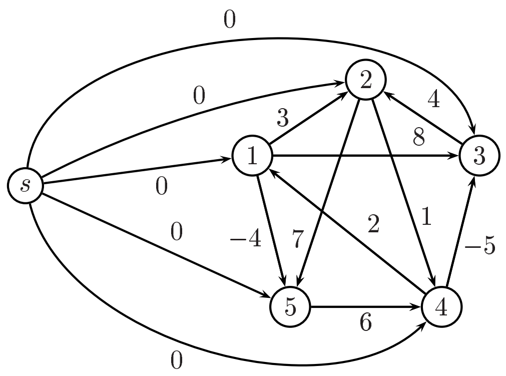
Procedure JOHNSON assumes that the edges are stored in an adjacency lists. The procedure returns the $|V| \times |V|$ matrix D where entry $d_{ij} = \delta(i, j)$ for $i, j = 1, 2, \ldots, |V|$. The procedure assumes, that the vertices are numbered from 1 to $|V|$.

```
JOHNSON(G)
begin
  construct G' = (V', E'), where V' = V ∪ {s} ,
  E' = E ∪ {(s,v)|v ∈ V} and w(s,v) = 0 for v ∈ V
  for each vertex v ∈ V'
    run the Bellman-Ford algorithm          since we might have
    to compute δ(s,v)                       negative-weight edges
    h(v) ← δ(s,v)
  for each edge (u,v) ∈ E'
    w̄(u,v) ← w(u,v) + h(u) − h(v)
  for each vertex u ∈ V
    run DIJKSTRA(G,u)                       with weights as
    to compute δ̄(u,v)                       computed by w̄
    for all v ∈ V
    for each vertex v ∈ V
      d_uv ← δ̄(u,v) + h(v) − h(u)
  return D
end
```

EXAMPLE: Consider the following graph $G = (V, E)$.



First we augment the graph $G = (V, E)$ by adding one vertex s.

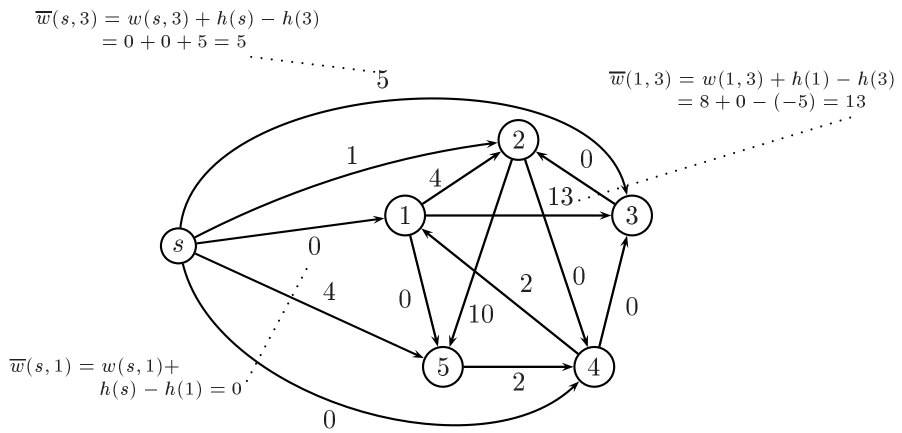


With procedure BELLMAN-FORD we obtain the following table

i	$h(i) = \delta(s, i)$
1	0
2	−1
3	−5
4	0
5	−4

and $h(s) = 0$.

The next figure shows G' after reweighting with weight function $\overline{w}$, where $\overline{w}(u, v) = w(u, v) + h(u) - h(v)$.



The matrix of all shortest paths produced by procedure JOHNSON with above example is

$$D = \begin{bmatrix} 0 & 2 & 2 & 2 & 0 \\ 2 & 0 & 0 & 2 & 2 \\ 2 & 0 & 0 & 0 & 2 \\ 2 & 0 & 0 & 0 & 2 \\ 4 & 2 & 2 & 2 & 0 \end{bmatrix}$$

If the priority queue in Dijkstra's algorithm is implemented by a Fibonacci-Heap, then the running time of Johnson's algorithm is

$$O(|V|^2 \log_2 |V| + |V||E|).$$

Note that Johnson's algorithm is efficient for sparse matrices.

In the next section, we study splay trees.

6.2 Splay Trees

A splay tree is a self-adjusting binary search tree. With self-adjusting data structures, we allow them to be in an arbitrary state, but during each operation, we apply a simple restructuring rule intended to improve the efficiency of future operations. Recall path compression for instance.

Self -adjusting data structures have several possible advantages over balanced or explicitly constrained data structures:

a) in an amortized sense, ignoring constant factors, they are never much worse than constrained structures, and since they adjust according to usage, they can be much more efficient if the usage pattern is skewed

b) they need less space, since no balance or other constraint information need to be stored

c) their access and update algorithms are easy to implement

Self-adjusting data structures have two potential disadvantages

a) they require more local adjustment, for example look-up operations during accesses; other structures need adjusting only during updates, not during accesses

b) individual operations within a sequence can be expensive, which is a problem for real time applications

Many variants of the simple binary search tree were developed to try and improve on the poor worst case behavior of search trees. The basic operations on binary search trees take $O(h)$ where h is the height of the tree. A complete binary tree with n elements has $h = \log_2 n$. But this is not always maintained as the tree undergoes various operations. One needs to balance the tree (and thus change the height of the tree) to achieve good performances so that the binary search tree can be an effective data structure.

AVL trees, $2-3$ trees and red-black trees are among a host of binary search trees that were developed so as to improve the performance of binary search trees. Splay trees were designed for that purpose, too. The goal of splay trees is to have the tree maintain a logarithmic depth in an amortized sense by adjusting its structure with each access.

Splay trees are reorganized whenever they are accessed or updated. The basic reorganizing operation (splaying) moves the accessed node towards the root by a sequence of rotations. Therefore, frequently accessed keys tend to be near the root. For the worst sequence of splaying, the number of operations is $O(\log_2 n)$ per node in the tree where n is the number of nodes.

While other variants of binary search trees often aggressively pursue maintaining a balanced tree, splay trees deal with the problem lazily, doing a little bit of work with each tree operation, and making little obvious effort to maintain a nicely balanced tree.

If we do an amount of work $c > O(\log_2 n)$, we show that there were a lot of operations in the recent past where the work done was less than allowed and so that we can amortize c by spreading it around to previous performed operations. In other words, any additional work that we do, beyond $O(\log_2 n)$, can be accounted for by our past laziness – a typical amortized example.

The two basic elements of splay trees are self-adjustment and rotation. Splay trees use a double rotation to distribute some of the weight below the queried node over its sibling. The total amount of imbalance, which is measured by the sum of ranks, is a potential function against which we charge the cost of long searches.

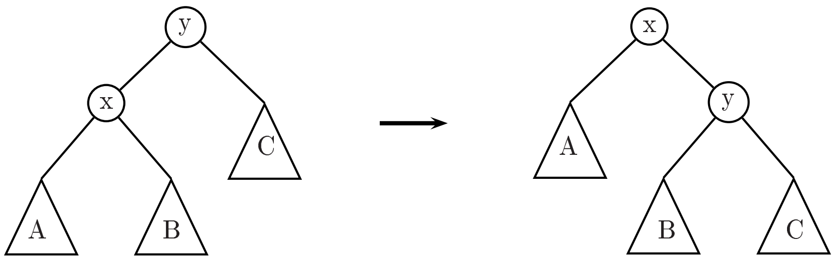
A splay tree is a binary search tree with the following tree splaying operations:

1. single rotation: ZIG
2. double rotations: ZIG-ZIG and ZIG-ZAG

6.2.1 Tree Rotations

Tree rotations can be thought of as a way to move a node to a higher position in a binary search tree without affecting the ordering properties.

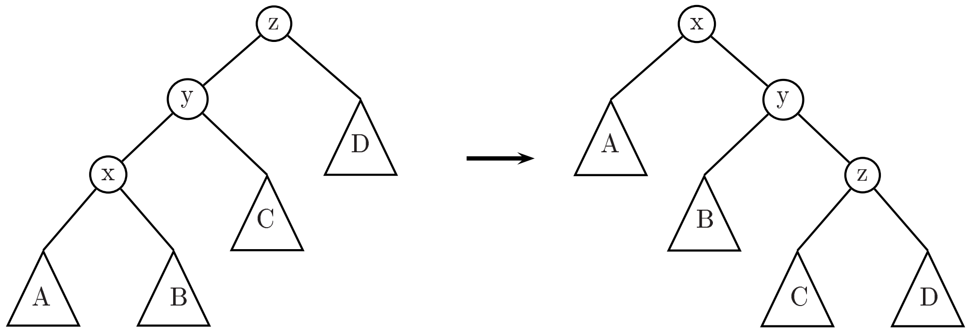
1. Single rotation: ZIG
   Splay step at x where x is the child of the rood node



   Single rotations involve two nodes and their corresponding subtrees. Note that the rotation brings node x to the top of the tree (and the resulting tree is still a binary search tree).

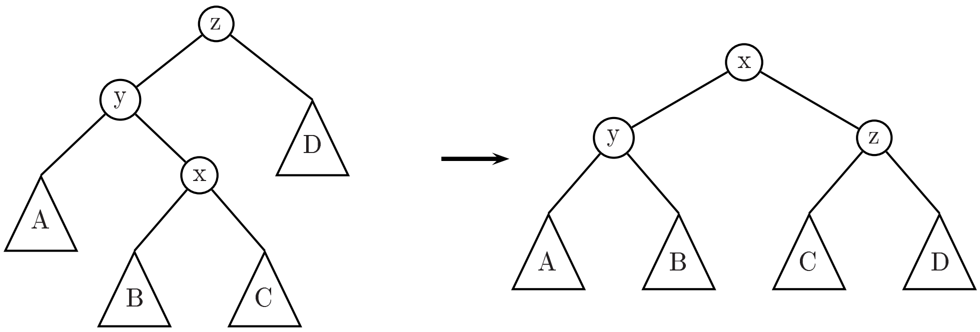
2. Double rotation
   Three nodes and their subtrees are involved with double rotations. A double rotation essentially performs single rotations in sequence.

   (a) ZIG-ZIG
   The parent of x is not the root node, and x and its parent are **left** children.



The node x is brought up two levels to the top of the tree (or subtree).

(b) ZIG-ZAG
The parent of x is the left child of the root node. x is the right child of its parent.

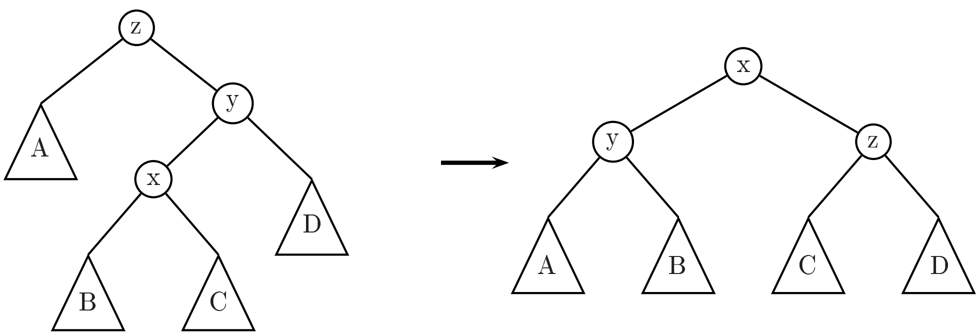


In this case, any weight below x is spread more evenly as x is brought to the root of the tree (subtree).

The double rotation is one of the key elements that make splay trees successful. While single rotations can move a queried element to the top of a tree without affecting key ordering, only double rotations produce the balancing properties of trees.

Remarks:

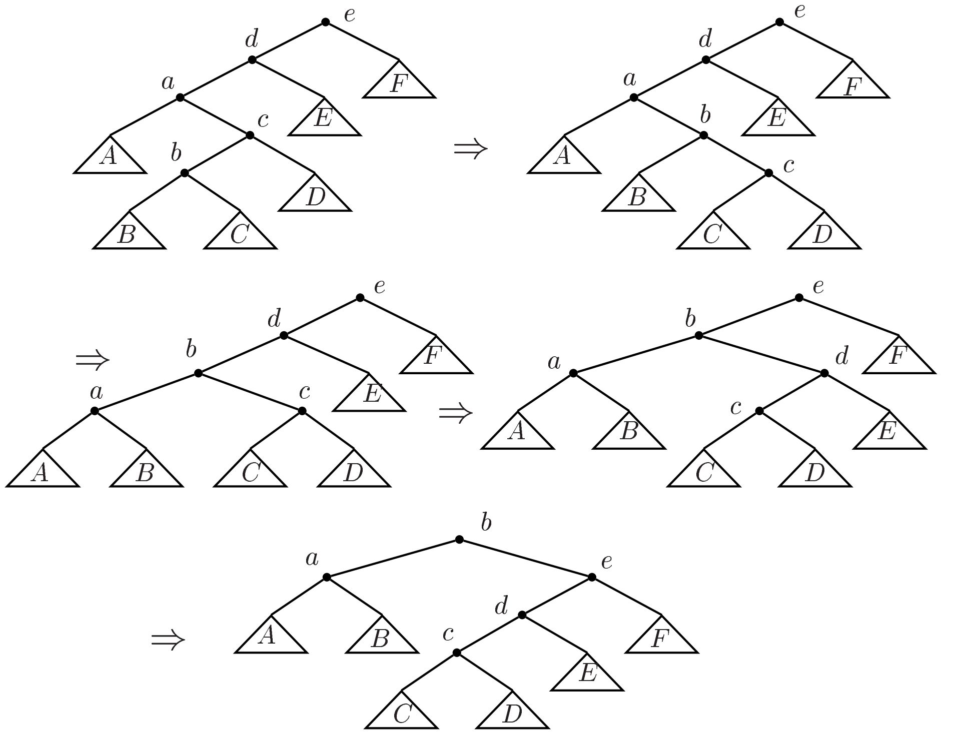
1. it is sufficient to look at the local structure of the tree defined by x, x's parent and x's grandparent to decide on one of the tree rules to follow. One has to continue applying the rules until x eventually reaches the top of the tree.
2. each rule has a mirror image variant, which covers all the cases. ("read" the figures from right to left for ZIG and ZIG-ZIG; for the ZIG-ZAG consider the following figure)



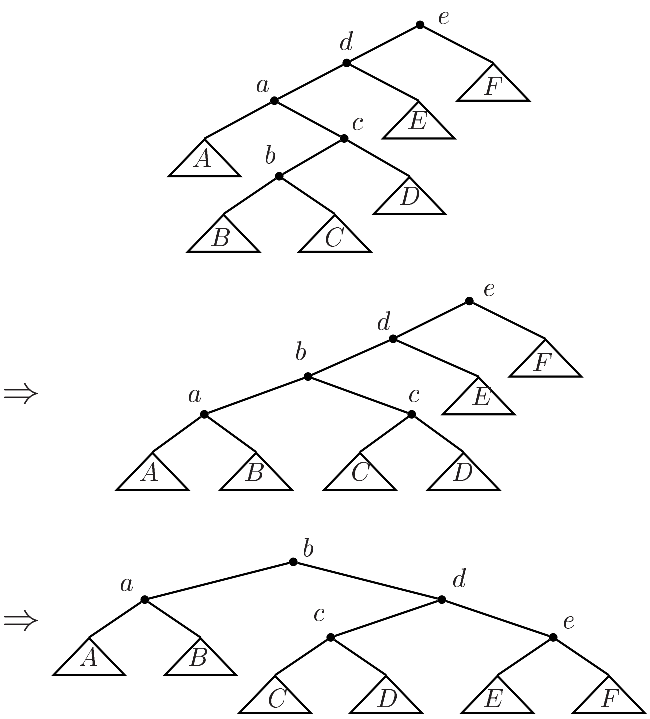
3. the ZIG-ZIG rule is the one that distinguishes splaying from just rotating x to the root of the tree

4. the three rules just studied are all bottom-up splaying rules

There is a big different between splaying and move–to–root.
Example: Move–to–root. Simple rotations are used to make node b the root node.



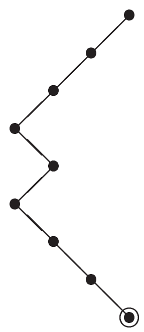
Example: Splaying. To splay a node b, repeat the splay step on b until it is the root.



EXAMPLE: Performing the following operations on each of the three trees (as indicated) will splay the encircled element to the root node.

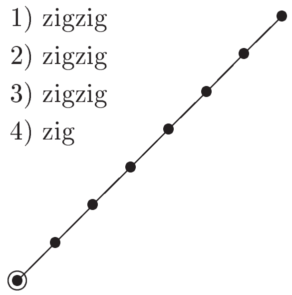
a)

1) zigzig
2) zigzag
3) zigzag
4) zigzig

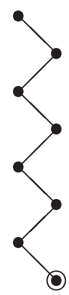


b)

1) zigzig
2) zigzig
3) zigzig
4) zig



c)



1) zigzag
2) zigzag
3) zigzag
4) zig

6.2.2 Analysis of Splay Tree Operations

Splay trees run all basic operations in $\log_2 n$ time. Splaying at a node of depth d takes $\Theta(d)$ time. We prove this through the manipulation of weights and a potential function. Let

c_k	denote the actual cost of operation k
$\overline{c_k}$	denote the amortized cost of the k^{th} operation
D_k	denote the state of the data structure (the tree) after applying the k^{th} operation on D_{k-1}
$\Phi(D_k)$	denote the potential associated with D_k

We know that

$$\begin{aligned} \overline{c_k} &= c_k + \Phi(D_k) - \Phi(D_{k-1}) \\ \Rightarrow\ c_k &= \overline{c_k} - \Phi(D_k) + \Phi(D_{k-1}) \end{aligned}$$

and the actual amount of work required is given by

$$\begin{aligned} \sum_{k=1}^{m} c_k &= \sum_{k=1}^{m} (\overline{c_k} - \Phi(D_k) + \Phi(D_{k-1})) \\ &= \Phi(D_0) - \Phi(D_m) + \sum_{k=1}^{m} \overline{c_k} \end{aligned}$$

and similarly

$$\sum_{k=1}^{m} \overline{c_k} = \Phi(D_m) - \Phi(D_0) + \sum_{k=1}^{m} c_k$$

Potential function for splay trees: Let

$w(x)$	weight of node x, a fixed but arbitrary value
T_x	tree rooted at x
$y \in T_x$	y is a node in tree T_x
$s(x) = \sum_{y \in T_x} w(y)$	s is the size of the weights of all descendants of x, including the weight of x itself
$r(x) = \log_2(s(x))$	r is the rank of x
$\Phi(T) = \sum_{x \in T} r(x)$	T is the data structure D and Φ is the potential function we use for proving properties about splay trees

Remarks:

1. performing a rotation between a pair of nodes x and y only affects the ranks of the nodes x and y, and no other nodes in the tree
2. if $w(y) = 1$ for all $y \in T$, then the potential of a balanced tree is $O(n)$, and at the other extreme, the potential of a long chain (most unbalanced tree) is $O(n \log_2 n)$

We shall need the following lemma later.

Lemma 1: *If b is a node in the tree with children a and c, then*

$$r(a) + r(c) - 2r(b) + 2 \leq 0$$

Proof: By definition,

$$s(b) = s(a) + s(c) + w(b).$$

Weights are positive, hence

$$\begin{aligned} & s(b) \geq s(a) + s(c) \\ \Rightarrow\ & [s(b)]^2 \geq (s(a) + s(b))^2 \\ \Rightarrow\ & [s(b)]^2 - 4s(a)s(c) \geq (s(a) - s(c))^2 \\ \Rightarrow\ & [s(b)]^2 - 4s(a)s(c) \geq 0 \quad \text{since } \ (s(a) - s(b))^2 \geq 0 \\ \Rightarrow\ & [s(b)]^2 \geq 4s(a)s(c) \\ \Rightarrow\ & \log_2 [s(b)]^2 \geq \log_2 (4s(a)s(c)) \\ \Rightarrow\ & 2\log_2 (s(b)) \geq 2\log_2 2 + \log_2 (s(a)) + \log_2 (s(c)) \end{aligned}$$

So by definition of $r(a)$, we get

$$2r(b) \geq 2 + r(a) + r(c)$$

so we have

$$r(a) + r(c) - 2r(b) + 2 \leq 0$$

$\square$

We consider the number of rotations. Let s and r denote the size and rank before a splay, and s' and r' denote the size and rank after the splay.

Lemma 2:(*access theorem*) *The amortized time for each rotation involved in the splaying of node x is at most*

$$3(r'(x) - r(x)) \ ,$$

and the time for a single rotation is at most

$$1 + 3(r'(x) - r(x)).$$

Proof:

Case 1: ZIG

The amortized time:

$$\begin{aligned} \overline{c_k} &= c_k + \Phi(T_k) - \Phi(T_{k-1}) \\ &= 1 + [r'(x) + r'(y)] - r(x) - r(y) \end{aligned}$$

since only x and y change ranks. But $r'(x) = r(y)$, hence

$$\overline{c_k} = 1 + r'(y) - r(x)$$

since y is below x in T_k, we have $r'(y) < r'(x)$. So $\overline{c_k} = 1 + r'(y) - r(x)$ with $r'(y) = r'(x) \Rightarrow \overline{c_k} < 1 + r'(x) - r(x)$. Since $r'(x) > r(x)$, we have

$$\begin{aligned} \overline{c_k} &< 1 + (r'(x) - r(x)) < 1 + 3(r'(x) - r(x)) \\ &\Rightarrow \overline{c_k} < 1 + 3(r'(x) - r(x)) \end{aligned}$$

Case 2: ZIG-ZIG

Here we have two rotations. So $c_k = 2$ and only x, y and z change ranks.

$$\begin{aligned} \overline{c_k} &= c_k + \Phi(T_k) - \Phi(T_{k-1}) = \\ &= 2 + [r'(x) + r'(y) + r'(z)] - [r(x) + r(y) + r(z)] \end{aligned}$$

We have $r'(x) = r(z)$, so $\overline{c_k} = 2 + r'(y) + r'(z) - r'(x) - r(y)$, but $r'(x) < r'(y)$ in T_k and $r(y) > r(x)$ in T_{k-1}, therefore

$$\begin{aligned} \Rightarrow \overline{c_k} &= 2 + r'(y) + r'(z) - r(x) - r(y) \\ &< 2 + r'(x) + r'(z) - r(x) - r(x) \\ \Rightarrow \overline{c_k} &< 2 + r'(x) + r'(y) - 2r(x) \end{aligned}$$

Add and subtract $2r'(x)$ and $r(x)$

$$\begin{aligned} \overline{c_k} &< 2 + r'(x) + r'(z) - 2r(x) + 2r'(x) - 2r'(x) + r(x) - r(x) \\ \overline{c_k} &< 2 + r(x) + r'(z) - 2r'(x) + 3r'(x) - 3r(x) \\ \Rightarrow \overline{c_k} &< [r(x) + r'(x) - 2r'(x) + 2] + 3\,[r'(x) - r(x)] \end{aligned}$$

but $r(x) + r'(x) - 2r'(x) + 2 \leq 0$ by Lemma 1. So

$$\overline{c_k} < 3(r'(x) - r(x))$$

Case 3: ZIG-ZAG

Here too $c_k = 2$, since we have 2 rotations. Only x, y and z change ranks and $r'(x) = r(z)$. So

$$\begin{aligned}
\overline{c_k} &= 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) = \\
&= 2 + r'(y) + r'(z) - r(x) - r(y) \\
\overline{c_k} &< 2 + r'(y) + r'(z) - r(x) - r(x) \text{ since } r(x) < r(y) \\
\Rightarrow \overline{c_k} &< 2 + r'(y) + r'(z) - 2r(x)
\end{aligned}$$

Add and subtract $2r'(x)$. Hence

$$\begin{aligned}
\overline{c_k} &< 2 + r'(x) + r'(x) - 2r(x) + 2r'(x) - 2r'(x) \\
&< [2 + r'(y) + r'(z) - 2r'(x)] + 2r'(x) - 2r(x) \\
&< [r'(y) + r'(z) - 2r'(x) + 2] + 2(r'(x) - r(x))
\end{aligned}$$

Again, by Lemma 1 we have:

$$r'(y) + r'(z) - 2r'(x) + 2 \leq 0$$

So $c_k < 2(r'(x) - r(x))$ and thus $\overline{c_k} < 3(r'(x) - r(x))$ □

We showed that the amortized cost of

- the ZIG step is bounded by

$$3(r'(x) - r(x)) + 1 \text{ , and of}$$

- the ZIG-ZIG and ZIG-ZAG steps by

$$3(r'(x) - r(x)).$$

We can now sum over all the rotations that are needed to splay x to the root.

Lemma 3: *The amortized time to splay a node x to the root is*

$$O\left(\log_2 \frac{s(t)}{w(x)}\right)$$

where $s(t)$ is the sum of all the weights in the tree rooted at t and $w(x)$ is the weight of x.

Proof: The amortized cost of splaying x to the root t is

$$1+3(r(t)-r(x_{k-1}))+3(r(x_{k-1}-r(x_{k-2}))+3(r(x_{k-3})-r(x_{k-4})+\cdots+3(r(x_1-r(x)) =$$

$$= 1 + 3(r(t) - r(x))$$

Note that $+1$ comes from the ZIG step that can happen only **once**. Here x, x_1, x_2, $x_3, \ldots, x_i, \ldots, x_{k-1}, t$ are roots of subtrees that are rotated. Recall $r(t) = \log_2(s(t))$ and $r(x) = \log_2(s(x))$. So

$$\begin{aligned}
3(r(t) - r(x)) &= O(r(t) - r(x)) = \\
&= O(\log_2(s(t)) - \log_2(s(x))) = \\
&= O\left(\log_2 \frac{s(t)}{s(x)}\right)
\end{aligned}$$

Since $w(x) \leq s(x)$ we have

$$\log_2 \frac{s(t)}{w(x)} < \log_2 \frac{s(t)}{w(x)}$$

$$\Rightarrow \; 3(r(t) - r(x)) = O\left(\log_2 \frac{s(t)}{w(x)}\right)$$

Let $W = s(t)$, i.e., the sum of all the weights of the nodes rooted at t. So the amortized time to splay x to the root is

$$O\left(\log_2 \frac{W}{w(x)}\right).$$

Lemma 4:(*balance theorem*) *A sequence of m splays in a tree of n nodes takes time*

$$O((m+n)\log_2 n).$$

Proof: From Lemma 3 we know that the amortized cost of splaying x to the root t is $1+3(r(t)-r(x))$. Let $w(y) = 1$ for all nodes $y \in T$. For a given splay, $r(t) \leq \log_2 n$ and $r(x) \geq 0$. So the amortized cost of the splay is at most $1 + 3\log_2 n$.

$$\sum_{k=1}^{m} \overline{c_k} = \Phi(T_m - \Phi(T_0) + \sum_{k=1}^{m} c_k$$

But

$$\begin{array}{rcll} \Phi(T_0) & \leq & n \log_2 n & \text{in case tree is skewed} \\ \Phi(T_m) & \geq & 0 & \end{array}$$

$$\Rightarrow \; \sum_{k=1}^{m} c_k \leq n\log_2 n + m(3\log_2 n + 1)$$

Therefore $\sum_{k=1}^{m} c_k$ is $O(m\log_2 n + n\log_2 n)$. So the sequence of m splays in a tree of n nodes takes time

$$O((m+n)\log_2 n)$$

Lemma 4, the balance theorem, states that on a sufficiently long sequence of accesses a splay tree is as efficient as any form of an uniformly balanced tree.

Remarks:

1. Note that by Lemma 3 we know that the amortized time to splay x to the root is $O\left(\log_2 \frac{W}{w(x)}\right)$. So if $w(y) = 1$ for all nodes in the tree, then $W = n$. Therefore in a tree with n nodes, the access time per operation is

   $$O\left(\log_2 \frac{W}{w(x)}\right) = O\left(\log_2 \frac{n}{1}\right) = O(\log_2 n)$$

2. If we now let $w(y) = \frac{1}{3^d}$ for all $y \in T$ where d is the depth of node y in the tree, then

   $$W = \sum_{y \in T} w(y) \leq \sum_{i=1}^{n} 2^i 3^{-d} \leq 1$$

   So the amortized cost to access a node that is found in depth d of a splay tree is

   $$O\left(\log_2 \frac{W}{w(x)}\right) = O\left(\log_2 \frac{1}{3^{-d}}\right) = O(d).$$

   This shows that splaying is competitive with any fixed binary search tree.

3. As a matter of fact, splaying is competitive with the best possible tree. Let m denote the total number of accesses of a tree. Let p_x denote the fraction of times x will be accessed (compared to all other accesses of the rest of the nodes in the tree). Note $p_x m$ denotes the access frequency of node x. It is known that the optimal access time is

$$\Omega(m \sum_{xinT} p_x \log_2 \frac{1}{p_x})$$

Now, if we go back to Lemma 3 and let $w(y) = p_y$, then the amortized cost for x is

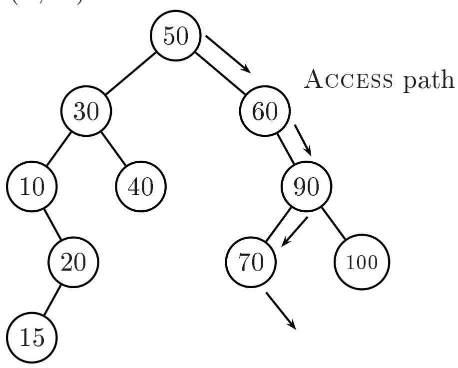
$$O\left(\log_2 \frac{W}{w(x)}\right) = O\left(\frac{1}{p_x}\right).$$

This sum is the entropy of a probability distribution. If we think of code lengths of items instead of node depths, then the above equation is the average code length required to send information about m items given the underlying probability distribution. This result, known as the static optimality theorem, implies that a splay tree is as efficient as any fixed search tree.

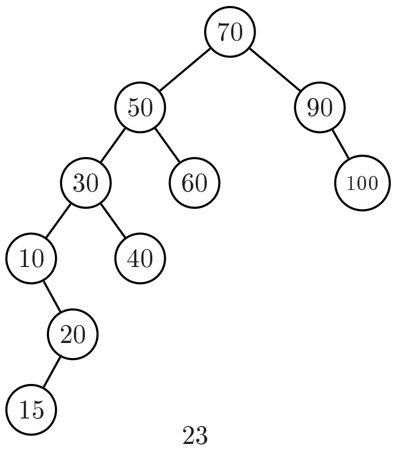
6.2.3 Update Operations on Splay Trees

Using splaying, we can implement all the standard update operations on binary search trees.

access(T, i) If item i is in the tree, return a pointer to its location, otherwise return a pointer to the null node. Search down the tree for i. If the search reaches a node x containing i, we complete the access by splaying at x and returning a pointer to x. If i is not in T, we splay the last nonnull node reached during the search and return a pointer to null. Splaying's effect of moving a designated node to the root considerably simplifies the updating of splay trees.
Example: Access$(T, 80)$

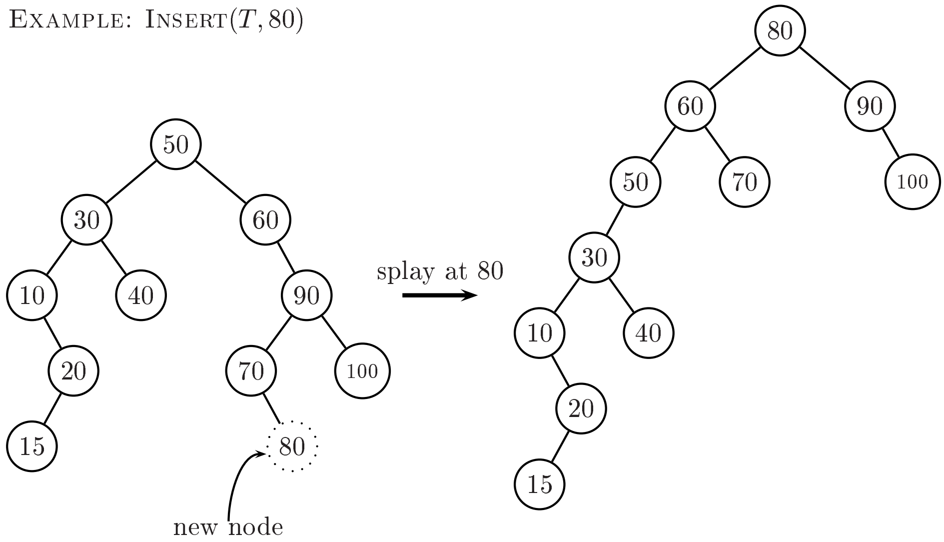


70 is the last nonnull node in the unsucessful search, so splay at 70.



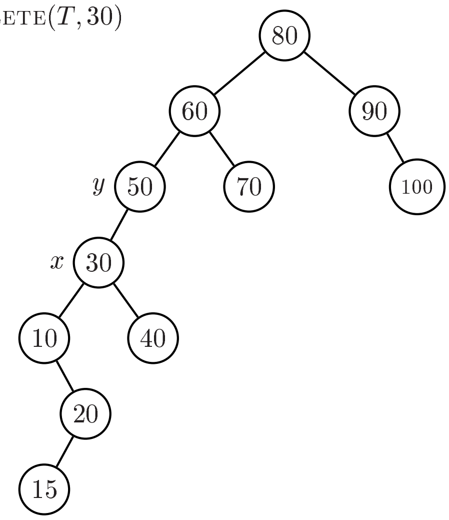
insert(T, i) Insert item i in tree T, assuming that it is not there already. One way of implementing $\text{INSERT}(T, i)$ is to search for i, then replace the pointer to null reached during the search by a pointer to a new node containing i, and finally splay the tree at the new node.

EXAMPLE: $\text{INSERT}(T, 80)$

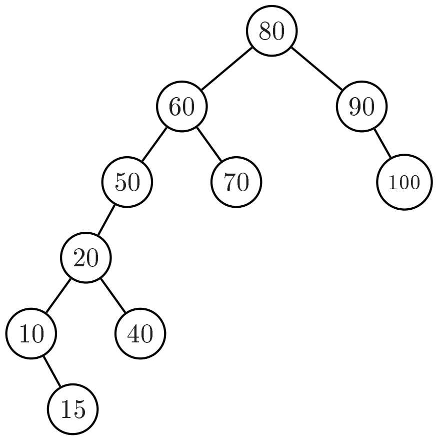


delete(T, i) Delete item i from T, assuming that it is present in T. One way of implementing $\text{DELETE}(T, i)$ is to search for the node containing i. If x is that node and y its parent, we replace x as a child of y by the join of the left and right subtrees of x, and then we splay at y.

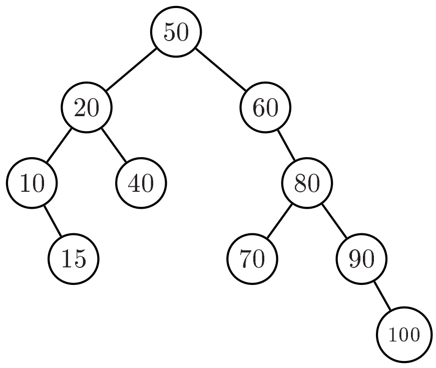
EXAMPLE: $\text{DELETE}(T, 30)$



We join the children of 30. Thus, we get



Now splay at 50 to obtain



join(T_1, i, T_2) Returns a tree where i is the root with T_1 as its left subtree and T_2 as its right subtree. It assumes that i is such that $t_1 < i < t_2$ for all $t_1 \in T_1$ and $t_2 \in T_2$. $\text{JOIN}(T_1, i, T_2)$ has amortized cost $O(\log_2 n)$. Note that we can also join two trees $T_1 < T_2$ by splaying the rightmost element of T_1 making T_2 its right subtree.

split(T, i) Returns two trees $T_1 \leq i < T_2$. Note that i simply defines a boundary between the two trees; i is not necessarily contained in T_1. $\text{SPLIT}(T, i)$ is implemented by splaying the greatest element less than i to the root. We rename the right subtree and call it T_3. The remaining tree is named T_1.

□

For any item i in a tree, let i^- and i^+, respectively, denote the item preceding i and the item following i in T. If i^- is undefined, $w(i^-) = \infty$ and if i^+ is undefined, let $w(i^+) = \infty$. The amortized times of the splay tree operations have the following upper bounds:

$$\begin{aligned} i^- &:= \text{largest element in T smaller than } i \\ i^+ &:= \text{smallest element in T larger than } i \end{aligned}$$

$$\text{ACCESS}(T, i) : \begin{cases} 3\log\left(\frac{W}{w(i)}\right) + O(1), \text{ if } i \in T \\ 3\log\left(\frac{W}{\min\{w(i^-),\ w(i^+)\}}\right) + O(1), \text{ if } i \notin T \end{cases}$$

$$\textsc{Insert}_{i\notin T}(T,i) : 3\log\left(\frac{W-w(i)}{min\{w(i^-),w(i^+)\}}\right)+\log\left(\frac{W}{w(i)}\right)+O(1)$$

$$\textsc{Delete}_{i\in T}(T,i) : 3\log\left(\frac{W}{w(i)}\right)+3\log\left(\frac{W-w(i)}{w(i^-)}\right)+O(1),\ \text{if } i \text{ not minimal in } T$$

$$\textsc{Join}(T_1,i,T_2) : 3\log\left(\frac{W}{w(i)}\right)+O(1),\ i \text{ maximal in } T_1$$

$$\textsc{Split}(T,i) : \begin{cases} 3\log\left(\frac{W}{w(i)}\right)+O(1), & \text{if } i\in T \\ 3\log\left(\frac{W}{\min\{w(i^-),\ w(i^+)\}}\right)+O(1), & \text{if } i\notin T \end{cases}$$

Remarks:

1. We defined $\textsc{Join}(T_1,x,T_2)$. If we use $\textsc{Join}(T_1,T_2)$, we begin by accessing the largest element, say i in T_1. After the access, the root of T_1 contains i and thus has a null right child. The join is completed by making T_2 the right subtree of this root and returning the resulting tree.

2. Alternative implementation of Delete and Insert using Split and Join.

   delete(T,i) We first perform $\textsc{Split}(T,i)$ to obtain two subtrees T_1 and T_2. Note that since i was an element of T_1, it must now be the root of T_1. Since the root of T_1 does not have a left subtree, we may remove i in constant time. We then join the two trees T_1 and T_2. Since we have done a constant number of $O(\log_2 n)$ operations, the amortized cost for Delete is $O(\log_2 n)$.

   insert(T,i) We first perform $\textsc{Split}(T,i)$ to obtain two subtrees, T_1 and T_2. Then we $\textsc{Join}(T_1,i,T_2)$ to obtain the resultant tree. The amortized cost for Insert is $O(\log_2 n)$.

□

In the next section, we study the problem of colosing the vertices and edges of graphs.

6.3 Graph Coloring

6.3.1 Vertex Coloring

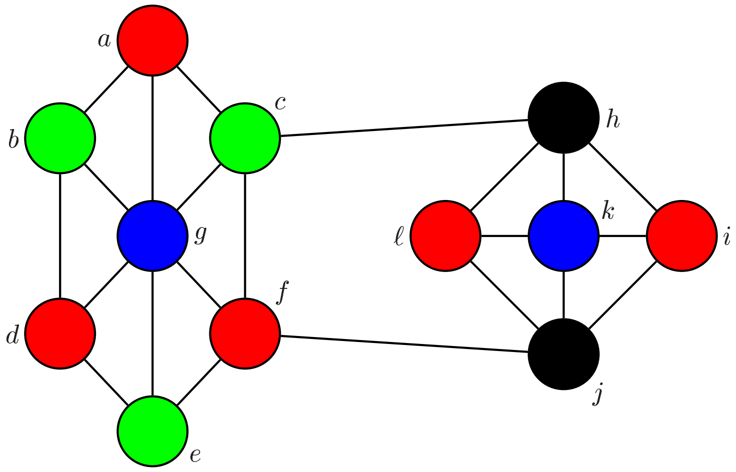
$G = (V,E)$ is an undirected graph. We are going to assign colors to vertices. The subset comprising all the vertices of a given color is a cell of a partition of V. Drawing the graph with colors assigned to the vertices is an intuitive way to represent such a partition.

Minimization problem of vertex coloring

Most applications involving vertex coloring are concerned with determining the minimum number of colors required under the condition that the endpoints of an edge of E cannot have the same color.

A vertex-coloring of G is an assignment $f : V \to C$ where C is the set of colors (usually denoted by $\{1,2,\ldots,m\}$. G is said to be vertex k–colorable if it has a vertex-coloring using exactly k colors and such that the endpoints of each edge in E are assigned two different colors.

Example: The following is an example of a vertex-coloring of a graph G.



In this example, we colored $\{a, d, f, l, i\}$ in red, $\{b, c, e\}$ in green, $\{g, k\}$ in blue and $\{h, j\}$ in black. But we can do better. By assigning green to k and the blue color to h and j we get a 3–coloring of G. □

Note that in the literature, the term "coloring" is used instead of "vertex coloring" since vertex coloring is more frequently encountered than edge coloring (or map-coloring).

The chromatic number of a graph, denoted by $\chi(G)$, is the minimum number of different colors required for the vertex-coloring of G such that 2 endpoints of edges have different colors. In other word, the chromatic number of G is the smallest k for which G is vertex k–colorable. So $\chi(G) = k$ if G is k–colorable but not $(k-1)$–colorable. A graph is k–chromatic if $\chi(G) = k$.

Remarks:

1. A graph G has $\chi(G) = 1 \Leftrightarrow$ G has no edges
2. The study of vertex-coloring of graphs is usually restricted to simple graphs, since:
   (a) If G has a self-loop, then it is uncolorable since the endpoint of the self-loop is adjacent to itself, and
   (b) A multiple adjacency has no more effect on the color of its endpoints than a single adjacency

A color class in a vertex-coloring of a graph G is a subset of V containing all the vertices of some color.

Modelling applications as vertex-coloring problems

When an application is modelled as a vertex-coloring problem, the vertices sharing the same color (in the same color class) generally represent items (objects or individuals) that do not compete (or conflict) with each other.

EXAMPLES:

1. **Assignment of radio frequencies**
   The vertices represent transmitters for radio stations. Two stations are considered adjacent when their broadcast areas overlap, which results in interference if they broadcast at the same frequency. So the adjacent stations should be assigned different transmission frequencies (in other words, different colors). In this application, each color class contains vertices representing stations with

no overlap, and $\chi(G)$ is the minimum number of transmission frequencies required to avoid broadcast interference.

2. **Combustible chemicals**
   The vertices represent different kinds of chemicals needed in some manufacturing process. But some chemicals cannot be stored together since the might explode. So each pair of chemicals that might explode if combined, has an edge between them. The chromatic number here is the minimum number of different storage areas required so that no two chemicals that mix explosively are stored together.

3. **University course scheduling**
   The vertices represent the courses at a university. Two courses are adjacent if at least one student preregisters for both courses. These two courses should not be scheduled at the same time. So the chromatic number here is the minimum number of time periods in which to schedule the classes so that no student has a conflict between two courses.

4. **Fast register allocation**
   Some computers have a limited number of special registers that permit faster execution of arithmetic operations than ordinary memory locations (or even other ordinary registers). The program variables that are frequently used should be "register" storage class variables. But we should not declare more "register" variables than the actual number of fast registers in hardware, otherwise we will waste time due to swapping.

   One solution is to have the programmer control the register designation, so that variables that are going to be simultaneously active are assigned to different registers. So the graph model has one vertex for each variable, and two vertices are adjacent if the endpoint variables can be simultaneously active in some computation (or operation). The chromatic number is the minimum number of fast registers needed to avoid the overswapping phenomenon.

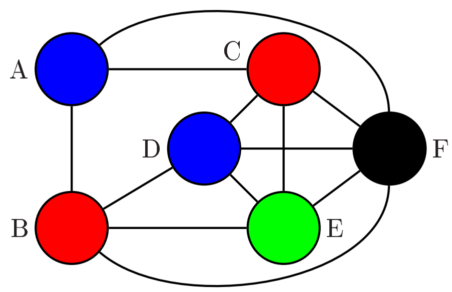
Lemma 1: *If G contains k mutually adjacent vertices, then $\chi(G) \geq k$.*
Proof: Using fewer than k colors would result in a pair from the mutually adjacent set of k vertices being assigned the same color. □

A clique in a graph $G = (V, E)$ is a maximal subset $S \subseteq V$ of mutually adjacent vertices. A clique number $\omega(G)$ of a graph $G = (V, E)$ is the number of vertices in a largest clique in G.

Lemma 2: *Let $G = (V, E)$ be a graph. Then $\chi(G) \geq \omega(G)$.*

Lemma 3: *If H is a subgraph of G then $\chi(G) \geq \chi(H)$.*

Example:



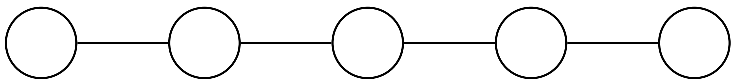
$S = \{C, D, E, F\}$ is a clique. $\omega(G) = 4$. So $\chi(G) \geq 4$. The graph is vertex 4–colorable since we can assign red to vertices $\{B, C\}$, blue to $\{A, D\}$, green to $\{E\}$ and black to $\{F\}$. So the graph is 4–chromatic and $\chi(G) = 4$. $\square$

Chromatic numbers for common graph families

1. Complete graphs
   A complete graph with n vertices, K_n is a simple graph such that every pair of vertices is joined by an edge. By Lemma 1, $\chi(K_n) \geq n$ since K_n has n mutually adjacent vertices. On the other hand, K_n is vertex n–colorable. So $\chi(K_n) = n$.

2. Bipartite graphs
   A bipartite graph $G = (V, E)$ is a graph in which V can be partitioned into two subsets U_1 and U_2, such that each edge $e \in E$ has one endpoint in U_1 and one endpoint in U_2. The pair U_1, U_2 is the bipartition of G, and are also known as the bipartition subsets. By Lemma 1, $\chi(G) \geq 2$, and G is 2–colorable since we can use one color for the vertices of U_1 and another color for the vertices of U_2. So if G is a bipartite graph, then $\chi(G) = 2$.

3. Path graphs
   A path graph $G = (V, E)$, denoted by P_n, is a simple graph with $|V| = |E| + 1$ $(|V| = n)$ that can be drawn so that all of its vertices and edges lie on a single straight line. Since path graphs are bipartite graphs, $\chi(P_n) = 2$ for $n \geq 2$.

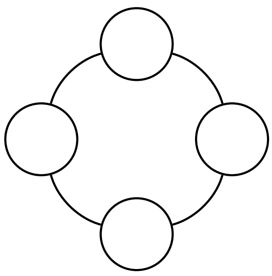
   

   path graph P_5

4. Trees
   Any nontrivial tree T is bipartite. So $\chi(T) = 2$.

5. Cycle graphs
   A cycle graph is a single vertex with a self-loop or a simple connected graph C with $|V| = |E|$ that can be drawn so that all of its vertices and edges lie on a circle. An n–vertex cycle graph is denoted by C_n.

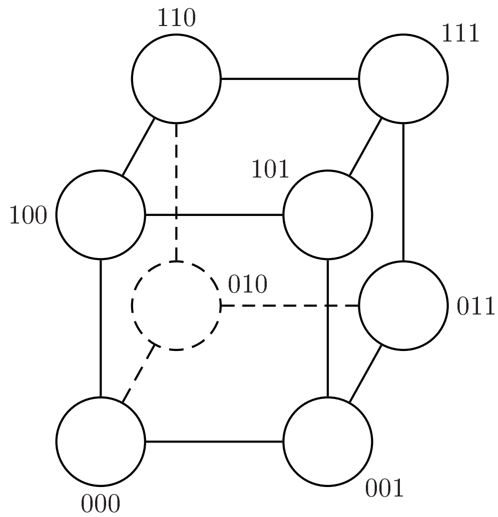
   (a) Even-cycle graphs
   Since an even-cycle graph, C_{2n}, is bipartite, $\chi(C_{2n}) = 2$.

   cycle graph C_4

   (b) Odd-cycle graphs
   If $v_1, v_2, \ldots, v_{2n+1}$ are the vertices of C_{2n+1} we would need 3 colors. If we use one color for $\{v_1, v_3, \ldots, v_{2n+1}\}$ and a second color for $\{v_2, v_4, \ldots, v_{2n}\}$, we would not have a 2–colorable graph, since v_1 and v_{2n+1} are adjacent. So we need 3 colors. Thus, $\chi(C_{2n+1}) = 3$.

6. Cube graphs
   A cube graph, Q_n, is a n–regular graph $Q_n = (V, E)$ (i.e. all vertices of V have degree n) whose vertex-set is the set of bitstrings of length n, and such that there is a n edge between two vertices if they differ in exactly one bit. Since the cube graph Q_n is bipartite, we have $\chi(Q_n) = 2$.

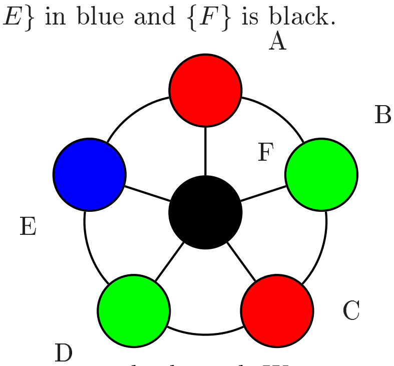


cube graph Q_3

7. Wheel graphs
   A wheel graph W_n, $n \geq 4$ is a cycle graph, C_{n-1}, to which we add one more vertex in the "center" and make it adjacent to all the vertices of C_{n-1}.

   (a) Odd-order wheel graph
   An odd-order wheel, W_{2n+1}, needs two colors for the even cycle graph C_{2n}, and then a third color to color the "center" vertex (since it is adjacent to every vertex of C_{2n}). Thus, $\chi(W_{n+1}) = 3$ for all $n \geq 1$.

   (b) Even-order wheel graph
   On the other hand, an even-order wheel, W_{2n}, needs three colors for its odd-cycle graph C_{2n-1}, plus a fourth color for the center vertex. So, $\chi(W_{2n}) = 4$.
   Example: The vertices of W_6 in the figure are colored: $\{A, C\}$ in red, $\{B, D\}$ in green, $\{E\}$ in blue and $\{F\}$ is black.



wheel graph W_6

The results are summarized in the following table:

Graph G	$\chi(G)$
Complete graph K_n	n
Bipartite graph	2
Path graph P_n	2
Tree	2
Cycle graphs (even)	2
Cycle graphs (odd)	3
Cube graphs Q_n	2
Wheel graphs (odd order)	3
Wheel graphs (even order)	4

The rest of this section deals with non-common graph families, where the aim is to find good and tight bounds for $\chi(G)$.

If $G = (V, E)$ is a graph, then an edge-deletion subgraph G' is obtained by removing an edge from E. So, $G' = (V, E')$ where $E' = E \backslash \{e\}, e \in E$. A connected graph $G = (V, E)$ is chromatically k–critical if $\chi(G) = k$ and the edge-deletion subgraph G' is $(k-1)$–colorable, for every edge $e \in E$.

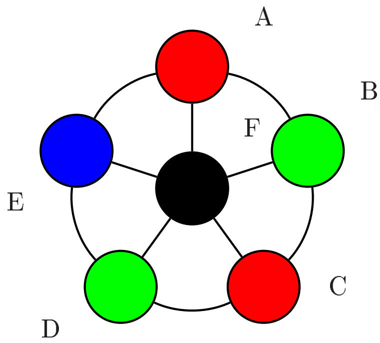
Examples:

1. An odd cycle is chromatically 3–critical, since the deletion of any edge will give a path graph and so the chromatic value goes from $\chi(G) = 3$ to $\chi(P) = 2$.

2. The even-order wheel W_6 is chromatically 4–critical. Recall $\chi(G) = 4$.

   even-order wheel W_6 W_6 is 4–colorable ($\{A, C\}$, $\{B, D\}$, $\{E\}$, $\{F\}$) and is chromatically 4–critical

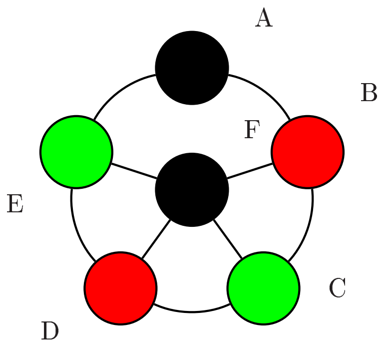
even-order wheel W_6



W_6 is 4-colorable and is chromatically 4-critical

remove any spoke-edge

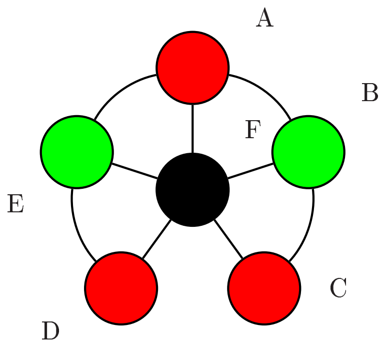
remove any rim-edge



G' is 3-colorable,

($\{A, F\}$, $\{B, D\}$, $\{E, C\}$)

$\chi(G') = 3$



G'' is 3-colorable,

($\{A, D, C\}$, $\{E, B\}$, $\{F\}$)

$\chi(G'') = 3$

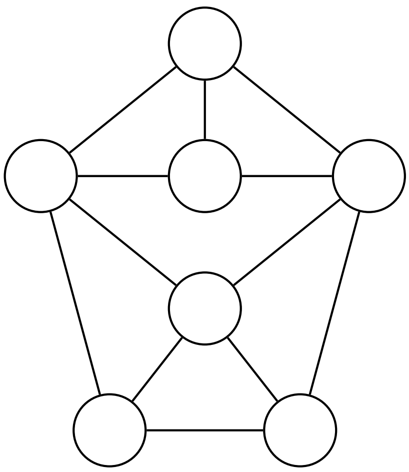
□

Similarly for vertex-deletion subgraphs. If $G = (V, E)$ is a graph, then a vertex-deletion subgraph G' is obtained by removing a vertex v from V, and consequently all edges in E that have v as endpoint. So $G' = (V', E')$ where $V' = V \backslash \{v\}$ and $E' = \{e \in E | v \text{ is not an endpoint of } e\}$. Since the deletion of a vertex will necessarily entail the removal of an edge, we have: If G is a chromatically k-critical graph and G' is a vertex-deletion subgraph obtained by removing any vertex of G, then G' is $(k-1)$-colorable.

Lemma 4: *If $G = (V, E)$ is a chromatically k-critical graph, then deg$(v) \geq k-1$ for all $v \in V$.*
Proof: The proof is by contradiction. Suppose that $\deg(v_1) < k-1$ for some $v_1 \in V$. Then the vertex-deletion subgraph G', obtained by removing v_1 from G is $k-1$-colorable. Since $\deg(v_1) < k-1$, v_1 has fewer than $k-1$ adjacent vertices and thus not all $k-1$ colors will be used on the adjacent vertices of v_1 in G'. Let c be a color not used on the adjacent vertices of v_1 in G'. Restore back G by adding v_1 to G' and color v_1 with color c. This means, that G is $(k-1)$-colorable which contradicts the fact that G is chromatically k-critical (and therefore $\chi(G) = k$). So no vertex of G has degree less than $k-1$. □

An independent set of vertices in a graph is a set of mutually nonadjacent vertices. The vertex independent number of a graph G is the maximum cardinality of an independent set of vertices. It is denoted by $ind(G)$.

EXAMPLE: The following graph has $ind(G) = 2$. This means that the maximum size of a color class of vertices is two.



Since G has 7 vertices, there must be at least 4 color classes. □

Lemma 5: *If G is any graph, then*

$$\chi(G) \geq \left\lceil \frac{|V|}{ind(G)} \right\rceil$$

Proof: Since each color class contains at most $ind(G)$ vertices, then the number of different color classes must be at least $\frac{|V|}{ind(G)}$ □

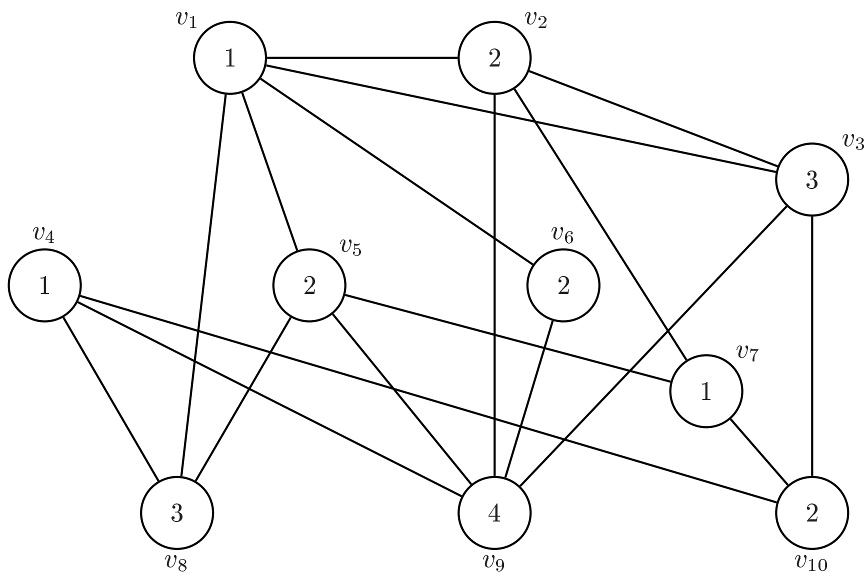
Sequential vertex-coloring algorithm

As mentioned in the chapter on Backtracking, the problem of calculating the chromatic number of a graph is known to be NP-hard. Deciding whether a graph is vertex 3-colorable is NP-complete.

Sequential Vertex-Coloring is a procedure that colors the vertices of a graph by always assigning the smallest color number that has still not been assigned to previously processed endpoints.

```
Sequential Vertex-Coloring
begin
  n ← |V|
  for i ← 1 to n
      let f(v_i) ← smallest color number
      not used on any of the smaller subscripted
      adjacent vertices of v_i
  return vertex-coloring f
end
```

The procedure assumes that $V = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices.
Example:



By applying Sequential Vertex-Coloring to the above graph, we obtain a 4-coloring using colors 1, 2, 3 and 4. □

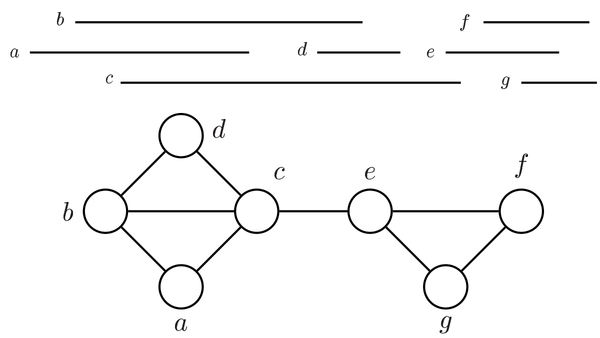
The color assignment in the example is not optimal. We could have used color 2 for v_4 and color 3 for v_7 (and thus not necessarily giving the smaller color number each time through the **for** loop). Then a 3-coloring can be obtained with the following color sets: $\{v_1, v_9, v_{10}\}$, $\{v_2, v_4, v_5, v_6\}$ and $\{v_3, v_7, v_8\}$.
Remarks:

1. Let us rename the vertices in $\{v_1, v_9, v_{10}\} \to \{v_1, v_2, v_3\}$ and $\{v_2, v_4, v_5, v_6\}$ to $\{v_4, v_5, v_6, v_7\}$ and $\{v_3, v_7, v_8\}$ to $\{v_8, v_9, v_{10}\}$. If the new graph with the renaming as described is used as input to Sequential Vertex-Coloring, the procedure will produce a minimum coloring.

2. Remark 1 implies that if we use all $n!$ possible labellings of $G = (V, E)$ where $|V| = n$, the procedure will output a minimum coloring (for one of the input data), but of course it is extremely inefficient.

3. Note that SEQUENTIAL VERTEX-COLORING is a greedy algorithm since at each iteration of the **for** loop we assign the "smallest" color available.

4. SEQUENTIAL VERTEX-COLORING yields an optimal coloring of interval graphs. Recall that with the fast register allocation example, we can allocate two variables not used at the same time, to the same register. For each variable, we compute the first and last time where it is used. A variable is active during the interval between this times. We define a graph with the variables as vertices, in which two vertices are adjacent if they are active at a common time. Recall that the number of registers needed is the chromatic number of the graph. Given a family of intervals, define a graph whose vertices are the intervals, and such that two vertices are adjacent when the intervals intersect. A graph formed in this way is an interval graph, and the family of intervals is an interval representation of the graph.



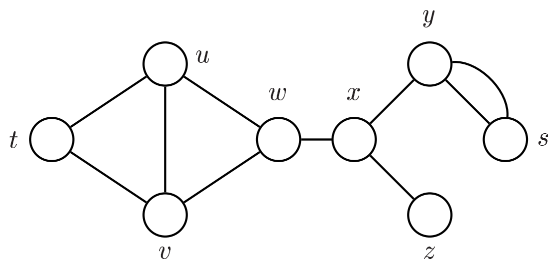
If the vertices are renamed $a \to v_1$, $b \to v_2$, $c \to v_3$, $d \to v_4$, $e \to v_5$, $f \to v_6$, $g \to v_7$ then SEQUENTIAL VERTEX-COLORING will yield an optimal coloring of 1, 2, 3, 1, 1, 2, 3 to v_1, v_2, $\ldots v_7$ respectively.

The maximum degree of a vertex in a graph G is denoted by $\delta_{max}(G)$ (or $\Delta(G)$).

Lemma 6: *For any graph,* $\chi(G) \le \delta_{max}(G) + 1$
Proof: Just apply SEQUENTIAL VERTEX-COLORING to the given graph, regardless of the labelling used. The procedure will not use more than $\delta_{max}(G) + 1$ colors since every vertex does not have more than $\delta_{max}(G)$ adjacent vertices $\square$

We can tighten up the upper bound to only $\delta_{max}(G)$, according to Brook's theorem. We need to define a few terms first.

A component of G is a maximal connected subgraph of G. In other words, a connected subgraph H is a component of a graph G if H is not a proper subgraph of any connected subgraph G. The only component of a connected graph is the entire graph. One can think of the components of a non-connected graph G as the "pieces" that make up G. If the vertex-deletion subgraph H, obtained by removing vertex x from V, is such that H has more components than G, then x is called a cut-vertex (or cut point). A block of a loopless graph is a maximal connected subgraph H such that no vertex of H is a cut-vertex of H.
EXAMPLE: The following graph has four blocks. Each block has the following set of vertices: $\{t, u, w, v\}$, $\{w, x\}$, $\{x, y, z\}$ and $\{y, s\}$



□

Note that a block H of a graph G has no cut-vertices (of H), but H may contain vertices that are cut-vertices of G. In the above example, w, x and y are cut-vertices of G.

Remarks:

1. Two different blocks of a graph can have at most one vertex in common.

2. $x \in G$ is a cut-vertex of G
$\Updownarrow$
x is in two different blocks of G

Result 1: Let B_1 and B_2 be distinct blocks of a connected graph G. Let $y_1 \in B_1$ and $y_2 \in B_2$ such that neither y_1 nor y_2 is a cut-vertex of G. Then vertex y_1 is not adjacent to vertex y_2. For example u in $\{t, u, w, v\}$ and z in $\{x, y, z\}$ are not adjacent (see figure of previous example).

A leaf block of a graph G is a block that contains exactly one cut-vertex of G.

Result 2: Let G be a connected graph with at least one cut-vertex. Then G has at least two leaf blocks. A graph G is k-connected if G is connected and it takes the removal of at least k vertices to either disconnect G or reduce G to a 1-vertex graph.

Lemma 7 *G is a non-complete, k-regular 2-connected graph with $k \geq 3$. Then G has a vertex x with two non-adjacent vertices y and z such that $G' = (V', E')$ with $V' = V \backslash \{y, z\}$ is a connected graph.*

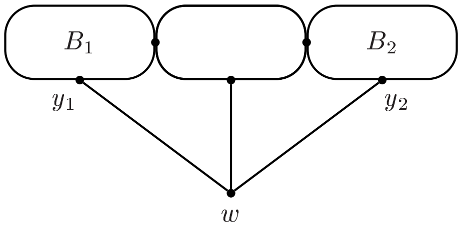
Proof: Let w be any vertex in G. $G \backslash w$ is either 2-connected or $G \backslash w$ is 1-connected.

Case 1: $G \backslash w$ is 2-connected

Let z be any vertex at distance 2 from w (which exists since G is regular and non-complete). If x is the vertex between w and z and y is taken to be w, we are done. In other words, G has 2 vertices, y and z that are non-adjacent to one another, but both adjacent to x such that $G \backslash \{y, z\}$ is connected

Case 2: $G \backslash w$ is 1-connected

By result 2, we know that we can find 2 leaf blocks, B_1 and B_2, of $G \backslash w$. Since G has no cut-vertices, it follows that w is adjacent to some vertex $y_1 \in B_1$ that is not a cut-vertex of $G \backslash w$, and likewise w is adjacent to some vertex $y_2 \in B_2$ that is not a cut-vertex of $G \backslash w$ (Note: $G \backslash \{w, y_1, y_2\}$ is connected since blocks have no cut-vertices).



By result 1, y_1 is not adjacent to y_2. If we let $w = x$, $y_1 = y$ and $y_2 = z$, we are done $\square$

Brook's Theorem: *If G is a non-complete, simple connected graph with $\delta_{max} \geq 3$ then $\chi(G) \leq \delta_{max}(G)$.*
Proof: The proof makes use of SEQUENTIAL VERTEX-COLORING, and is divided into 3 cases. Suppose $|V| = n$.

Case 1: G is not regular. Choose $v_n \in V$ such that its degree is less than $\delta_{max}(G)$. Since G is connected, we can grow a spanning tree of G from v_n, assigning indices in decreasing order as we reach vertices. Each vertex other than v_n in the resulting ordering $v_1, v_2, \ldots, v_n$ has a higher-indexed adjacent vertex along the path to v_n in the tree. Hence, each vertex has at most $\delta_{max}(G) - 1$ lower-indexed adjacent vertices. Therefore, SEQUENTIAL VERTEX-COLORING uses at most $\delta_{max}(G)$ colors.

In the remaining two cases, G is regular.

Case 2: G is regular and has a cut-vertex x
Let G' be a component of the graph $G \backslash x$ together with its edges to x. The degree of x in G' is less than $\delta_{max}(G)$. Thus, we can use the procedure of case 1 to color the vertices of G' with at most $\delta_{max}(G)$ colors. The same can be done for all other components. By permuting the names of colors in each subgraph so that vertex x always gets color c_1 (or 1) we can construct a coloring of the vertices of G with $\delta_{max}(G)$ colors.

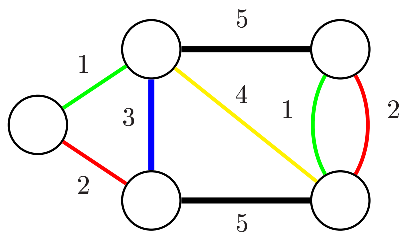
Case 3: G is regular and 2-connected by lemma 7, G has a vertex, which we call v_n with two non-adjacent vertices, which we denote by v_1 and v_2 such that $G \backslash \{v_1, v_2\}$ is connected.
Grow a spanning tree from v_n in $G \backslash \{v_1, v_2\}$ using numbers for colors $(3, \ldots, n)$ such that labels increase along paths to the root v_n. As in case 1, each vertex except v_n has at most $\delta_{max}(G) - 1$ lower-indexed adjacent vertices. So SEQUENTIAL VERTEX-COLORING uses at most $\delta_{max}(G)$ colors on all vertices except v_n. Moreover, the sequential coloring algorithm assigns the same color to v_1 and v_2 and at most $k - 2$ colors to the other $k - 2$ adjacent vertices of v_n. So, one of the $\delta_{max}(G)$ colors is available for coloring v_n. $\square$

6.3.2 Edge Coloring

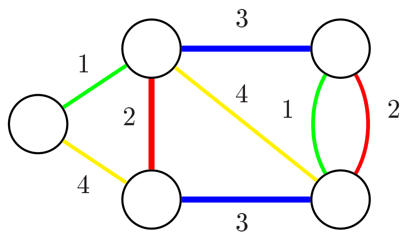
An edge-coloring of a graph $G = (V, E)$ is a mapping $f : E \rightarrow C$ where C is the set of colors usually denoted by $\{1, 2, \ldots, n\}$.

Two different edges are adjacent edges if they have at least one endpoint in common. G is said to be edge k-colorable if it has an edge-coloring using exactly k colors and such that adjacent edges are assigned different colors. The edge-chromatic number or chromatic index denoted by $\chi'(G)$ is the minimum k such

that G is edge k-colorable. Whereas multi-edges have no bearing on the vertex-colorings of a graph, they have an effect on edge colorings and should thus be taken into consideration. But graphs with self-loops will not be studied here.
EXAMPLE: The following is an example of an edge-coloring of a graph.



But we can do better



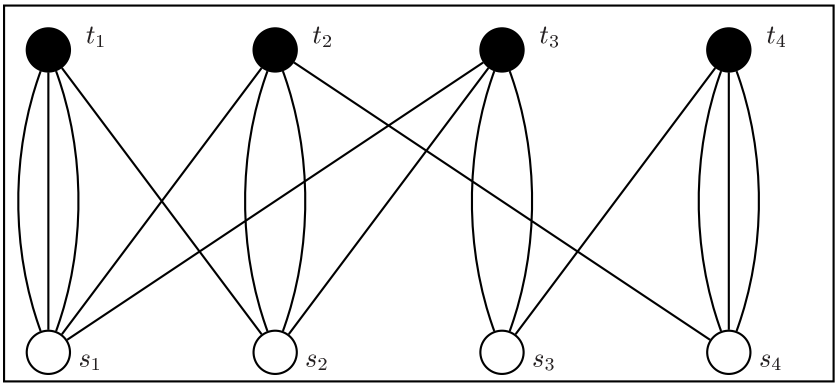
The graph is not edge 3-colorable since it contains 4 mutually adjacent edges. Therefore $\chi'(G) = 4$ □

Remark:

$$\text{a graph } G \text{ has } \chi'(G) = 1$$
$$\Updownarrow$$
$$\delta_{max}(G) = 1$$

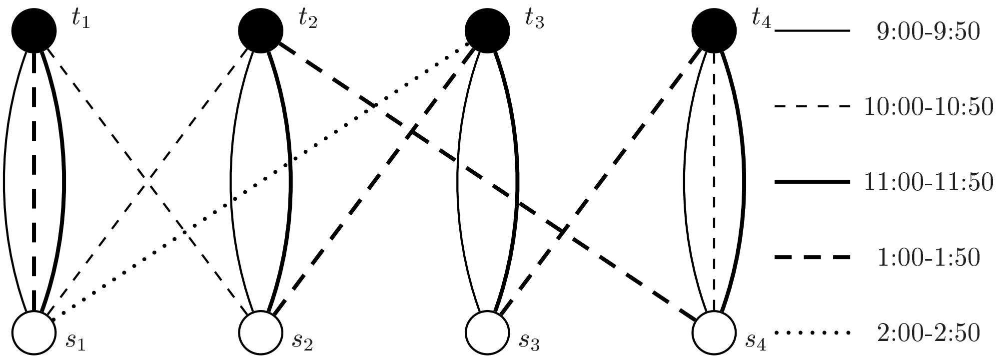
Modelling applications as edge-coloring problems

Example 1: Scheduling classes in High Schools.
$\{t_1, \ldots, t_m\}$ set of teachers
$\{s_1, \ldots, l_n\}$ set of courses.
Teacher t_j must teach $s_{j,k}$ sections of s_k.
Problem: Find a schedule with the minimum number of time periods such that no two sections of the same course are taught at the same time.
Solution: Construct a bipartite graph so that there are $s_{j,k}$ edges joining t_j and s_k for all j and k.



Representing a scheduling problem by a bipartite graph

Each edge-color represents a time slot in the schedule. The chromatic index represents the smallest number of time periods, which is equal to five in this example.



A minimum edge-coloring for the above graph

Example 2: Circuit boards.
The electronic devices $x_1, x_2, \ldots, x_n$ are on a board. The connecting wires emerging from each device must be colored differently, so that they can be distinguished. The least number of colors required is the chromatic index of the associated network. Since the edges incident to one vertex need different colors, we have

Result 1: For any graph,

$$\chi'(G) \geq \delta_{max}(G)$$

Result 2: If G has k mutually adjacent edges, then $\chi'(G) \geq k$.

Result 3: If H is a subgraph of G, then $\chi'(G) \geq \chi'(H)$.

Chromatic index for common graph families

Complete graph K_n, n even	$\chi'(G) = n - 1$
Complete graph K_n, n odd	$\chi'(G) = n$
Bipartite Graph G	$\chi'(G) = \delta_{max}(G)$
Path graph, P_n, $n \geq 3$	$\chi'(G) = 2$
Tree T	$\chi'(T) = \delta_{max}(T)$
Cube graph Q_n	$\chi'(Q_n) = n$
Cycle graph C_n, n even	$\chi'(C_n) = 2$
Cycle graph C_n, n odd	$\chi'(C_n) = 3$
Wheel graph W_n, $n > 4$	$\chi'(W_n) = n - 1$

Sequential edge-coloring algorithm

SEQUENTIAL EDGE-COLORING is analogous to sequential vertex-coloring and assumes that the input is the edge list: $e_1, e_2, \ldots, e_m$ where $e_i \in E$.

```
SEQUENTIAL EDGE-COLORING
begin
  m ← |E|
  for i ← to m
    Let f(e_i) ← smallest color number not used
    on any of the smaller-subscripted adjacent edges of e_i
  return edge-coloring f
end
```

As in the case with vertex-coloring, edge-coloring is an NP-hard problem. SEQUENTIAL EDGE-COLORING is a greedy procedure that does not necessarily deliver the chromatic index.

Vizing Theorem: *Let G be a simple graph; then either $\chi'(G) = \delta_{max}(G)$ or $\delta_{max}(G) = \delta_{max}(G) + 1$.* □

Multigraphs

The multiplicity $\mu(G)$ of a graph G is the maximum number of edges joining two vertices of G.

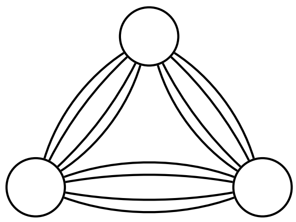
Result 4: For any loopless graph G, we have:

$$\delta_{max}(G) \leq \chi'(G) \leq \delta_{max}(G) + \mu(G)$$

Result 5: For any loopless graph G

$$\chi'(G) \leq \left\lfloor \frac{3\delta_{max}(G)}{2} \right\rfloor .$$

This result, due to Claude Shannon, is in terms of $\delta_{max}(G)$ only.
EXAMPLE:



In the multigraph, the edges are pairwise intersecting, and must receive distinct colors. In this example,

$$\chi'(G) = \frac{3\delta_{max}(G)}{2} = \delta_{max}(G) + \mu(G) = 12$$

□

Bibliography

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, 1990.