Chapter 4. Algorithms on Graphs

In this chapter we deal with efficient algorithms for many basic problems concerning graphs: topological sorting and transitive closure, connectivity and biconnectivity, least cost paths, least cost spanning trees, network flow and matching problems, and planarity testing. Most of these algorithms require methods for the systematic exploration of a graph. We will introduce such a method in Section 4.4 and then specialize it to breadth-first-search and depth-first-search. Novice readers should read Sections 4.1, 4.2, 4.4 and 4.5 before proceeding to higher numbered sections; readers, which have some familiarity with graph algorithms, may plunge into any section directly.

4.1. Graphs and their Representation in a Computer

A **directed edge** over a set V is an element of $V \times V$. Given a directed edge $e = (v, w)$, v is called its **tail** and w its **head**, both v and w are called **endpoints** of e. A directed edge $e = (v, w)$ is said to **leave** its tail v and to **enter** its head w. An **undirected edge** over V is a subset of V of cardinality exactly two. Given an undirected edge $e = \{v, w\}$, v and w are called its endpoints. A directed or undirected edge is said to be **incident** on its endpoints. If $e = (v, w)$ is a directed edge with $v \neq w$, the **reverse** e^{-1} of e is the directed edge (w, v), and the **undirected version** of e is the undirected edge $\{v, w\}$. The **directed versions** of an undirected edge $\{v, w\}$ are the two edges (v, w) and (w, v).

Given a finite, nonempty set V, a (directed, undirected) **graph** on the vertex set V is a pair $G = (V, E)$, where E is a set of (directed, undirected) edges over V. The elements of V are called the **vertices** or **nodes** of G and the elements of E are called the edges of G. We use graph to denote both undirected and directed graph and frequently use **digraph** instead of directed graph.

Given a directed graph $G = (V, E)$, the undirected version of G is the undirected graph $(V, \{\{v, w\};\ (v, w) \in E\})$. Given an undirected graph $G = (V, E)$, the directed version of G is the directed graph $(V, \{(v, w);\ \{v, w\} \in E\})$, i.e., each undirected edge is replaced by its two directed versions. The edges of the directed version of an undirected graph G are called the **darts** of G.

Let $G = (V, E)$ be a digraph. A **path** from v to w, where $v, w \in V$, is a sequence $v_0, v_1, \ldots, v_k$ of nodes such that $v_0 = v, v_k = w$ and $(v_i, v_{i+1}) \in E$ for $0 \leq i < k$; k is the **length** of the path. Note that there is always the path of length zero from v to v. A path is **simple** if $v_i \neq v_j$ for $0 \leq i < j < k$. A **cycle** is a path from v to v. If, in addition, the path is simple then the cycle is simple. A cycle in a directed graph is **trivial** if its length is 0, otherwise, it is non-trivial. A path (simple path, cycle, simple cycle) in an undirected graph G is a path (simple path, cycle, simple cycle) in the directed version of G. A cycle in an undirected graph is trivial if its length is either 0 or 2, otherwise, it is non-trivial.

A graph is **acyclic** if it does not contain any non-trivial cycles. Let $T \subseteq E$. We write $v \overset{*}{\underset{T}{\rightarrow}} w$ if there is a path from v to w using only edges in T. We write $v \overset{+}{\underset{T}{\rightarrow}} w$ if the length of the path is at least 1 and we use $v \underset{T}{\rightarrow} w$ to denote the existence of the edge $(v, w) \in T$.

The **indegree** of a node v in a directed graph is the number of edges entering v, $indeg_G(v) = |\{w;\ (w, v) \in E\}|$. Similarly, the **outdegree** of v is the number of edges starting in v, $outdeg_G(v) = |\{w;\ (v, w) \in E\}|$. The **degree** of a vertex v in an undirected graph is the number of edges incident to v.

A graph $G' = (V', E')$ is a **subgraph** of $G = (V, E)$, if $V' \subseteq V$ and $E' \subseteq E$. If $G = (V, E)$ is a graph and $V' \subseteq V$, then the subgraph **induced** (or **spanned**) by V' is (V', E'), where E' consists of those edges of E which have both endpoints in V'. $G - V'$ denotes the subgraph induced by $V - V'$. If $V' = \{v\}$ is a singleton, then we write $G - v$ instead of $G - \{v\}$.

A digraph $A = (V, T)$ is a **directed in-forest** (**out-forest**, respectively) if A is acyclic and $indeg_A(v) \leq 1$ ($outdeg_A(v) \leq 1$) for all $v \in V$. A node v with $indeg_A(v) = 0$ ($outdeg_A(v) = 0$) is called a **root** of the forest. Note that a directed forest has at least one root. If $|T| = |V| - 1$, then $A = (V, T)$ is a **directed tree**. In a directed tree there is a single root r. Also, an in-tree has a unique path from the root to any node v and an out-tree has a unique path from any node v to the root. Finally, if v is any node of an in-tree (out-tree), then the **subtree** A_v rooted at v is the subgraph induced by the descendants (predecessors) of v, i.e., A_v is the subgraph induced by $\{w;\ v \overset{*}{\underset{T}{\rightarrow}} w\}$ ($\{w;\ w \overset{*}{\underset{T}{\rightarrow}} v\}$).

Let $G = (V, E)$ be a digraph. A directed forest $A = (V, T)$ with $T \subseteq E$ is called a **spanning forest** of G. If A is a tree then it is called a **spanning tree** of G. A spanning forest (tree) of an undirected graph G is the undirected version of a spanning forest (tree) of the directed version of G.

Heaving laid out these basic definitions, we are now ready to discuss algorithmic questions. For the algorithmic treatment of graphs, we assume that the vertices of a graph $G = (V, E)$ are numbered from 1 to $|V|$, i.e., we assume $V = \{1, 2, \ldots, |V|\}$. We also set $n = |V|$ and $m = |E|$. The first question is the representation problem: how to store a graph in a computer. Two methods of storing a digraph are customary.

a) Adjacency matrix: A digraph $G = (V, E)$ is represented by a $n \times n$ boolean matrix $A_G = (a_{ij})_{1 \leq i,j \leq n}$ with

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E; \\ 0 & \text{if } (i,j) \notin E. \end{cases}$$

The storage requirement of this representation is clearly $\Theta(n^2)$.

b) Adjacency lists: A digraph $G = (V, E)$ is represented by n linear lists. The i-th list contains all nodes j with $(i, j) \in E$. The headers of the n lists are stored in an array. The storage requirement of this representation is $O(n + m)$. The lists are not necessarily arranged in sorted order.

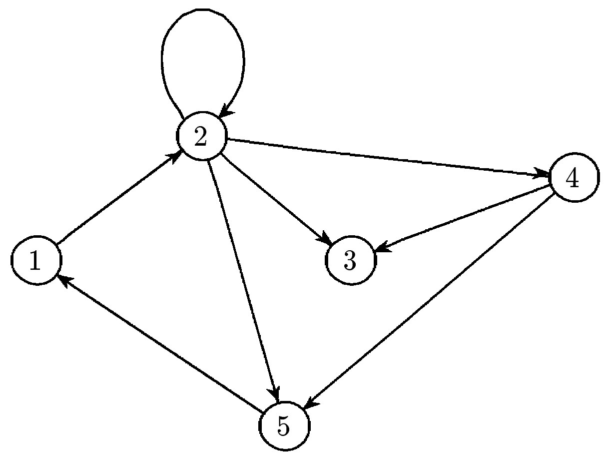


Figure 1. Graph G_{Ex}

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 2. Representation of G_{Ex} by adjacency matrix

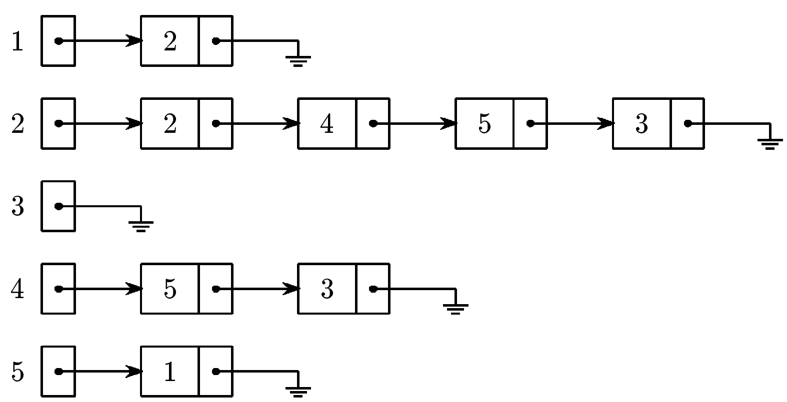


Figure 3. Representation of G_{Ex} by adjacency lists

Figures 1 to 3 show an example digraph and its representation by adjacency matrix and adjacency lists.

Since $0 \leq m \leq n^2$, we conclude that the adjacency list representation is often much smaller than the adjacency matrix representation and never much larger. Since most graphs which arise in applications are sparse, i.e., $m \ll n^2$, this is an important point to keep in mind. The fact that the choice of the representation can have a drastic influence on the time complexity of graph algorithms is even more important. In this chapter, we will see that many graph problems can be solved in linear time $O(n+m)$ if the adjacency list representation is used. However, any algorithm for these problems using the matrix representation must have running time $\Omega(n^2)$, cf. Section 4.2. For this reason we will always use the adjacency list representation if not explicitly stated otherwise (cf. Chapter 5).

To go into further detail, the adjacency list representation is based on the following declarations:

$$
\begin{array}{lll}
\textbf{type} & \mathit{edge} = \textbf{record} & \mathit{node}\colon [1\,..\,n]; \\
& & \vdots \\
& & \mathit{next} :\uparrow \mathit{edge} \\
& \quad\quad\quad \textbf{end} &
\end{array}
$$

and

$$\mathit{adjhead}\colon \text{array}[1\,..\,n] \text{ of } \uparrow \mathit{edge}.$$

Array *adjhead* contains the heads of the adjacency lists. The elements of the adjacency lists are of type *edge*, each element representing an edge. In some cases these elements will contain additional information, e.g., the cost of an edge in least cost path problems, the capacity of an edge in flow problems, the capacity and the cost of an edge in min cost flow problems, $\ldots$.

An undirected graph G is represented by its directed version. We also assume that each dart of G has a link to its reverse dart, i.e., there is a field *reverse*: $\uparrow$*edge* in the edge record and this field in the record representing the dart (v,w) points to the record representing the dart (w,v) for every $\{v,w\} \in E$.

Exercises 1) and 2) discuss the problem of how to convert other graph representations into the one postulated above.

4.2. Topological Sorting and the Representation Problem

A **topological sort** of a digraph $G = (V, E)$ is a mapping $ord : V \to \{1, \dots, n\}$ such that for all edges $(v, w) \in E$ we have $ord(v) < ord(w)$. Clearly, if a graph G has a topological sort then G is acyclic. The converse is also true and is easily proved by induction on the number of nodes. So suppose, $G = (V, E)$ is acyclic. If $n = 1$, then G has a topological sort. If $n > 1$, then G must have a node v with indegree 0. (Such a node can be found by starting at an arbitrary node w and traversing edges in reverse direction. Since the graph is acyclic, no node is entered twice in this process, and hence the process terminates. It terminates in a node with indegree 0.) By the deletion of v we obtain an acyclic graph G' with one node less. By the induction hypothesis G' has a topological sort and so has G.

Actually, the argument given above is an algorithm for computing the mapping *ord*. We formulate it in Program 1.

---

(1) $G_{current} \leftarrow G$; $count \leftarrow 0$;
(2) **while** $G_{current}$ has at least one node with no predecessor
(3) **do** let v be a node with no predecessor;
(4) $count \leftarrow count + 1$;
(5) $ord[v] \leftarrow count$;
(6) $G_{current} \leftarrow G_{current} - v$
(7) **od**;
(8) **if** $G_{current}$ is nonempty
(9) **then** G is cyclic **else** G is acyclic **fi**.

Program 1

The correctness of Program 1 follows immediately from the preceding discussion. With respect to complexity the crucial lines are lines (3) and (6). How do we efficiently find a node with indegree 0 in line (3)? A brute force approach would be a complete search of graph $G_{current}$. Since such a search would at least take time $\Omega(n)$, the entire algorithm would be $\Omega(n^2)$ at best.

A better approach is to exploit the interdependence of lines (3) and (6). In line (6) node v and all edges leaving v are deleted. This changes the indegrees of exactly those vertices which are heads of edges leaving v. It is therefore reasonable to use an array $indeg[1\,..\,n]$ to store the current indegree of all nodes. Array *indeg* is updated in line (6). In line (3) we need to know one node with indegree 0. The indegree of a node can only become zero in line (6) and it is easy to detect this fact there. It is therefore wise to keep all nodes of $G_{current}$ with indegree 0 in a set *zeroindeg*.

Program 2 refines our algorithm and makes use of the variables *indeg*: array $[1\,..\,n]$ of integer and *zeroindeg*: subset of V. The graph $G_{current}$ is not stored explicitly. Instead, we store it implicitly by using the fact that $G_{current}$ is the subgraph of G induced by the nodes which have not been given a number *ord* yet.

The set *zeroindeg* contains the nodes of zero indegree in $G_{current}$ and *indeg* maps each node to its indegree in $G_{current}$. Initially, $G_{current} = G$ and so *indeg* should be initialized to the indegrees in G. This can be done efficiently by traversing all adjacency lists.

```
(1.1)     count ← 0;
(1.2)     zeroindeg ← ∅; for all i ∈ V do indeg[i] ← 0 od;
(1.3)     for all i ∈ V
(1.4)     do for all j ∈ V with (i, j) ∈ E
(1.5)         do indeg[j] ← indeg[j] + 1
(1.6)         od
(1.7)     od;
(1.8)     for all i ∈ V
(1.9)     do if indeg[i] = 0 then add i to zeroindeg fi
(1.10)    od;
(2)       while zeroindeg ≠ ∅
(3.1)     do let v be any node in zeroindeg;
(3.2)         delete v from zeroindeg;
(4)           count ← count + 1;
(5)           ord[v] ← count;
(6.1)         for all w ∈ V with (v, w) ∈ E
(6.2)         do indeg[w] ← indeg[w] − 1;
(6.3)             if indeg[w] = 0
(6.4)             then add w to zeroindeg fi
(6.5)         od
(7)       od;
(8)       if count < n
(9)       then halt ("graph is cyclic") else halt ("graph is acyclic") fi.
```

Program 2

An implementation for set *zeroindeg* remains to be specified. On this set the following operations are performed: insertion of an arbitrary and deletion of an unspecified element, and test for emptiness. In Chapter 1 we saw that implementing *zeroindeg* by a stack or by a queue will allow us to execute each of these operations in time $O(1)$. We prefer the stack because of its simplicity and higher efficiency, so *zeroindeg* is a stack of elements of V (stack of $[1 \mathinner{..} n]$).

Finally, we need to explain lines (1.4) and (6.1) in detail. They are realized by traversing the adjacency list corresponding to the nodes i and v respectively and take time proportional to the outdegree of those nodes. Program 3 gives a detailed implementation of lines (1.4) and (1.5). (p is of type $\uparrow$*edge*.)

We are now able to determine the performance of our algorithm for topological sorting. Line (1.1) takes time $O(1)$, line (1.2) and lines (1.8)–(1.10) take time $O(n)$. The execution of lines (1.4) to (1.6) for a fixed i takes time $O(outdeg_G(i))$ and

$p \leftarrow adjhead[i]$;
while $p \neq$ nil
do $j \leftarrow p\uparrow.name$;
  $indeg[j] \leftarrow indeg[j] + 1$;
  $p \leftarrow p\uparrow.next$
od;

Program 3

hence lines (1.3)–(1.7) take time $O(n+m)$. Altogether, the initialization takes time $O(n+m)$. The main loop is executed $O(n)$ times and hence the total time spent in lines (3.1), (3.2), (4) and (5) is $O(n)$. For a fixed v, lines (6.1)–(6.5) take time $O(outdeg_G(v))$. Since every node v is deleted from *zeroindeg* at most once the total running time of that loop is $O(n+m)$. This shows that the running time of the entire algorithm is $O(n+m)$.

Theorem 1. *A topological sort of digraph $G = (V, E)$ can be computed in linear time $O(n+m)$.*

Proof: Given by the discussion above. ∎

Next we will show that any algorithm is doomed to inefficiency if we store the graph in the form of a matrix.

Theorem 2. *Any algorithm for topological sorting which receives the digraph as an adjacency matrix has running time $\Omega(n^2)$.*

Proof: Consider the behavior of any such algorithm on the empty graph, i.e., on the entire zero matrix. Suppose, there is a pair i, j of nodes, $i \neq j$, such that the algorithm neither inspects a_{ij} nor a_{ji}. Then we could change both entries to one and the algorithm would still return a topological sort. However, the graph is cyclic after having added edges (i, j) and (j, i). This shows that the algorithm has to inspect at least half of the entries of the matrix and hence has running time $\Omega(n^2)$. ∎

We saw that a topological sort of an acyclic digraph can be computed in linear time. Given the mapping $ord : V \to \{1, \ldots, |V|\}$ it is easy to rearrange the adjacency lists in increasing order as follows: Generate set $\{(ord(v), ord(w));\ (v, w) \in E\}$ and sort it using bucket sort. This takes time $O(n+m)$ by Section 2.2.1 and generates the adjacency lists in sorted order. An alternative linear time algorithm for topological sorting will be given in Section 4.5.

4.3. Transitive Closure of Acyclic Digraphs

Let $G = (V, E)$ be a digraph. The digraph $G^* = (V, E^*)$, where $(v, w) \in E^*$ if and only if there is a path from v to w in G, is called the **reflexive, transitive closure** of G or simply transitive closure. In this section we present an algorithm for computing the transitive closure of an acyclic digraph; the algorithm is extended to general digraphs at the end of Section 4.6. We will assume that the acyclic digraph is topologically sorted, i.e., $(i, j) \in E$ implies $i < j$ and that the adjacency lists are sorted in increasing order. We saw in the previous section that this can be achieved in linear time $O(n + m)$.

The idea underlying the algorithm is very simple. We consider the nodes of G in decreasing order. Suppose that we consider node i. Then for every $j > i$ we have already computed the set of nodes reachable from j, $\mathit{reach}[j] = \{k;\ j \xrightarrow{*} k\}$. Then

$$\mathit{reach}[i] = \{i\} \cup \bigcup_{(i,j)\in E} \mathit{reach}[j].$$

This equation demonstrates that $\mathit{reach}[i]$ can be computed by $\mathit{outdeg}(i)$ union operations on sets of nodes. For many graphs the number of union operations required to compute $\mathit{reach}[i]$ can be reduced considerably as follows. We consider the edges (i, j) emanating from i in increasing order of j. When edge (i, j) is considered, we first test whether $j \in \mathit{reach}[i]$ is at this stage already. If this is the case, then there must be a node $h \neq j$ with $i \to h \xrightarrow{*} j$ and hence $\mathit{reach}[h] \supset \mathit{reach}[j]$. Thus we do not have to add $\mathit{reach}[j]$ to $\mathit{reach}[i]$. Program 4 gives the complete algorithm.

---

(1) **for** i **from** n **downto** 1
(2) **do** $\mathit{reach}[i] \leftarrow \{i\}$;
(3)   **for** all j with $(i, j) \in E$ **co** in increasing order!! **oc**
(4)   **do if** $j \notin \mathit{reach}[i]$
(5)   **then** $\mathit{reach}[i] \leftarrow \mathit{reach}[i] \cup \mathit{reach}[j]$
(6)   **fi**
(7)   **od**
(8) **od**.

Program 4

How should we represent the set $\mathit{reach}[i]$? We recommend coding $\mathit{reach}[i]$ as a bit vector array$[1 \,..\, n]$ of boolean. Then lines (2) and (5) take time $O(n)$ each and line (4) takes time $O(1)$. Since line (2) is executed exactly once for each node and line (4) exactly once for each edge, the running time is $O(n^2 + m + m' \cdot n)$ where m' is the number of edges (i, j) for which line (5) is executed.

Definition: Let $G = (V, E)$ be an acyclic digraph. Let $E_{red} = \{(i, j) \in E$; there is no path of length at least two from i to j in $G\}$, let $G_{red} = (V, E_{red})$, and let $m_{red} = |E_{red}|$. G_{red} is called the **transitive reduction** of G. ∎

Lemma 1. *Let $G = (V, E)$ be an acyclic digraph.*

a) $G^* = (G_{red})^*$.

b) *The algorithm correctly computes the transitive closure.*

c) *If line* (5) *is executed for* (i, j) *iff* $(i, j) \in E_{red}$.

Proof: a) $(G_{red})^*$ is certainly a subgraph of G^*. In order to prove the converse consider any $(i, j) \in E^*$. Let $i_0, i_1, \ldots, i_k$ be a path of maximal length form $i = i_0$ to $j = i_k$. Then $(i_l, i_{l+1}) \in E_{red}$ for all l, $0 \le l < k$, and hence $(i, j) \in (E_{red})^*$.

b) It is obvious that our algorithm computes a subset of the transitive closure. Suppose that it computes a proper subset. Then let i be maximal such that a node h exists with $i \xrightarrow{*} h$ and h is never added to *reach*$[i]$. Consider a maximal length path $i_0, \ldots, i_k$ from $i = i_0$ to $h = i_k$. Then $h \in$ *reach*$[i_1]$ by definition of i. Also, $(i_0, i_1) \in E_{red}$. If the test in line (4) is executed with $j = i_1$ then $j \notin$ *reach*$[i]$ because there is no path of length at least two from i_0 to i_1; otherwise the path would not be of maximal length. Hence h is added to *reach*$[i]$ in line (5) if it is not there already.

c) Suppose that $(i, j) \in E - E_{red}$. Then h exists with $(i, h) \in E_{red}$ and $h \underset{E}{\xrightarrow{+}} j$. Hence j is added to *reach*$[i]$ when edge(i, h) is considered in loop (3) to (6). Thus $j \in$ *reach*$[i]$ when edge(i, j) is considered. Conversely, if $(i, j) \in E_{red}$ then line (5) is certainly executed. ∎

Theorem 1. *The transitive closure of an acyclic digraph $G = (V, E)$ can be computed in time $O(n \cdot (n + m_{red})) = O(n^3)$.*

Proof: We have argued above that the running time is $O(n^2 + m + m' \cdot n)$, where m' is the number of edges for which line (5) is executed. By Lemma 1 we have $m' = m_{red} \le m$. Also, $m \le n^2$. ∎

Of course, $m_{red} \le m$. Unfortunately, $m_{red} = m = O(n^2)$ can occur. Consider for example $V = \{1, \ldots, n\}$ and $E = \{(i, j);\ i \le n/2 < j\}$. In general however, m_{red} is considerably smaller than m. We will support this claim by an analysis of random digraphs below. Before doing so, we will slightly improve upon the running time of the algorithm by using packed bit vectors instead of just bit vectors for the sets *reach*$[i]$. We divide the bit vector *reach*$[i]$ into segments of length L where L is an integer to be determined later ($L \approx \log n$). The idea now is to perform the union operation in line (5) not bit by bit but rather segment by segment. In this way we may combine L bit operations into a single operation and hence speed up the computation by a factor L. For the analysis of this approach we need the following assumption:

($\mathcal{A}$): The arithmetic operations $+$, $-$, $\cdot$, $=$, $\ne$, $<$ on integers in the range $[0\,..\,n]$ take time $O(1)$.

Assumption ($\mathcal{A}$) is reasonable since node names are integers in that range and we assumed already that they can be handled in constant time. A bit string $a_{L-1} \ldots a_0$

of length L represents the integer $a = \sum_{l=0}^{L-1} a_l \cdot 2^l$ in the range $R = [0\,..\,2^L - 1]$. We define functions *union*: $R \times R \to R$, *power*: $[0\,..\,L-1] \to R$, *is_in*: $[0\,..\,L-1] \times R \to \{0,1\}$, *index*: $[1\,..\,n] \to [0\,..\,\lceil n/L \rceil - 1]$, and *bit*: $[1\,..\,n] \to [0\,..\,L-1]$ as follows: If $a = \sum_{\ell=0}^{L-1} a_\ell \cdot 2^\ell$ and $b = \sum_{\ell=0}^{L-1} b_\ell \cdot 2^\ell$ then

$$\begin{aligned} \mathit{union}(a,b) &= \sum_{\ell=0}^{L-1} \max(a_\ell, b_\ell) \cdot 2^\ell \\ \mathit{power}(l) &= 2^l \\ \mathit{is\_in}(l,a) &= a_\ell \\ \mathit{index}(j) &= (j-1) \text{ div } L \\ \mathit{bit}(j) &= (j-1) \bmod L. \end{aligned}$$

We can now represent the bit vector $\mathit{reach}[i]$ as $\mathit{reach}[i, \lceil n/L \rceil - 1], \ldots, \mathit{reach}[i, 0]$ where each $\mathit{reach}[i,h]$ is an integer in the range $R = [0\,..\,2^L - 1]$, i.e., a bit string of length L. Node j corresponds to the $\mathit{bit}(j)$-th bit in $\mathit{reach}[i, \mathit{index}(j)]$. With these definitions we can rewrite Program 4 as Program 5.

```
for i from n downto 1
do co  initialize reach[i] to empty set oc
   for l, 0 ≤ l < ⌈n/L⌉
   do reach[i,l] ← 0 od;
   co  add i to reach[i] oc
   reach[i, index(i)] ← power(bit(i));
   for all j with (i,j) ∈ E                co  in increasing order!! oc
   do if is_in(bit(j), reach[i, index(j)]) = 0
      then for l, 0 ≤ l < ⌈n/L⌉
               do reach[i,l] ← union(reach[i,l], reach[j,l]) od
      fi
   od
od.
```

Program 5

Lemma 2. *Under the assumption that functions union, power, bit, index and is_in can be evaluated in time $O(1)$, Program 5 runs in time $O((n + m_{red}) \cdot n/L) = O(n^3/L)$.*

Proof: Obvious. ∎

How can we evaluate these functions quickly? The easiest solution is to tabulate them.

Lemma 3. *The functions union, power, bit, index and is_in can be tabulated in time* $O(L \cdot 2^{2L})$.

Proof: We only discuss the functions *is_in* and *union* and leave the other functions to the reader. The value $is\_in(i,a)$ is true iff $a_i = 1$. This is exactly the case if $a = (2l+1) \cdot 2^i + h$ for some h, $0 \le h \le 2^i - 1$. The table *is_in*[,] is therefore filled correctly by Program 6. Program 6 runs in time $O(L \cdot 2^L)$.

$is\_in[i,a] \leftarrow 0$, for $0 \le i \le L-1$, $0 \le a \le 2^L - 1$;
for i, $0 \le i \le L-1$
do for h, $0 \le h \le power[i] - 1$
  do for l, $0 \le l \le power[L-i] - 1$
    do $is\_in[i, (2l+1) \cdot power[i] + h] \leftarrow 1$ **od**
  od
od.

Program 6

Using the table *is_in*[,], the entry $union[a,b]$ of table *union*[,] can now be computed in time $O(L)$ by Program 7.

$c \leftarrow 0$;
for j **from** 0 **to** $L-1$
do $c \leftarrow c + \max(is\_in[j,a], is\_in[j,b]) \cdot power[j]$ **od**;
$union[a,b] \leftarrow c$;

Program 7

The complete table *union*[,] can therefore be filled in time $O(2^{2L} \cdot L)$. ∎

Theorem 2. *Under assumption* ($\mathcal{A}$) *the transitive closure of an acyclic digraph can be computed in time* $O((n + m_{red})n/\log n) = O(n^3/\log n)$.

Proof: We choose $L = \log n - \log\log n = \Omega(\log n)$. Then

$$2^{2L} \cdot L \le 2^{2(\log n - \log\log n)} \cdot \log n \le n^2/\log n$$

and the computation of the various tables takes time $O(n^2/\log n)$ by Lemma 3. Also the transitive closure is computed in time $O((n + m_{red}) \cdot n/L) = O((n + m_{red}) \cdot n/\log n)$ after preprocessing by Lemma 2. ∎

Next, we turn to the average case analysis of the transitive closure algorithm. We postulate the following model of a **random acyclic digraph** on n nodes. Let ϵ be a real between 0 and 1. For $i < j$ the probability of the event "$(i,j) \in E$" is ϵ. For $(i,j) \neq (l,h)$ the events "$(i,j) \in E$" and "$(l,h) \in E$" are independent.

Theorem 3.

a) $\mathrm{E}(m^*) \geq \dfrac{n \cdot (n+3)}{2} - \dfrac{n \cdot (1 + \ln(1/\epsilon))}{\epsilon}$.

b) $\mathrm{E}(m_{red}) \leq \min(n \cdot (2 + \ln(1/\epsilon)), n^2 \cdot \epsilon)$.

c) *The expected running time of our algorithm on an n node random digraph is*

$$O(\min(n^2 \cdot (2 + \ln(1/\epsilon)), n^3 \cdot \epsilon)) = O(n^2 \cdot \ln n).$$

d) *Under assumption $(\mathcal{A})$ the expected running time on an n node random digraph is $O(n^2)$.*

Proof: a) and b): We prove parts a) and b) in four steps. In the first step (Lemma 4) we establish a connection between the expected size of the transitive closure and the expected size of the transitive reduction. This relation allows us to concentrate on the expected size of the transitive closure. We will next (Lemma 5) derive a recurrence for the probability that the number of vertices reachable from vertex 1 in a random digraph on n vertices is exactly l. This lemma in combination with a general property (Lemma 6) of this type of recurrence will allow us to get the desired bound on the size of the transitive closure (Lemma 7) and on the size of the transitive reduction.

We use the following notation. The random variable $outdeg_n^{red}(v)$ ($outdeg_n^*(v)$) denotes the outdegree of vertex v in the transitive reduction (transitive closure) of a random digraph on n vertices.

Lemma 4.

a) $\mathrm{prob}((1,n) \in E^{red}) = \dfrac{\epsilon}{1-\epsilon} \cdot \mathrm{prob}((1,n) \notin E^*)$.

b) $\mathrm{E}(m_{red}) = \dfrac{\epsilon}{1-\epsilon} \cdot \left(\dfrac{n(n-1)}{2} - \mathrm{E}(m^*) \right)$.

Proof: a) We have

$$\begin{aligned}
\mathrm{prob}((1,n) \in E^{red}) &= \mathrm{prob}((1,n) \in E) \cdot \mathrm{prob}(\underbrace{\forall\, v, v \neq 1, v \neq n, 1 \xrightarrow{*} v : (v,n) \notin E}_{:= A}) \\
&= \epsilon \cdot \sum_{l=1}^{n-1} \mathrm{prob}(A \mid outdeg_{n-1}^*(1) = l) \cdot \mathrm{prob}(outdeg_{n-1}^*(1) = l) \\
&= \epsilon \cdot \sum_{l=1}^{n-1} (1-\epsilon)^{l-1} \cdot \mathrm{prob}(outdeg_{n-1}^*(1) = l) \\
&\qquad [\text{since } \mathrm{prob}(A \mid outdeg_{n-1}^*(1) = l) \;=\; (1-\epsilon)^{l-1}]
\end{aligned}$$

$$= \frac{\epsilon}{1-\epsilon} \cdot \sum_{l=1}^{n-1} (1-\epsilon)^l \cdot \mathrm{prob}(outdeg^*_{n-1}(1) = l)$$

$$= \frac{\epsilon}{1-\epsilon} \cdot \mathrm{prob}((1,n) \notin E^*)$$

$$\left[\text{since } \mathrm{prob}\big(\forall\, v, 1 \xrightarrow{*} v\colon\ (v,n) \notin E | outdeg^*_{n-1}(1) = l\big) \;=\; (1-\epsilon)^l\right]$$

b)

$$\begin{aligned}
\mathrm{E}(m_{red}) &= \sum_{1 \le i < j \le n} \mathrm{E}((i,j) \in E^{red}) \\
&= \sum_{1 \le i < j \le n} \mathrm{prob}((i,j) \in E^{red}) \\
&= \sum_{1 \le i < j \le n} \frac{\epsilon}{1-\epsilon} \cdot \mathrm{prob}((i,j) \notin E^*) \qquad \text{[by part a)]} \\
&= \frac{\epsilon}{1-\epsilon} \cdot \sum_{1 \le i < j \le n} (1 - \mathrm{prob}((i,j) \in E^*)) \\
&= \frac{\epsilon}{1-\epsilon} \cdot \left(\frac{n(n-1)}{2} - \mathrm{E}(m^*) \right).
\end{aligned}$$

∎

We infer from Lemma 4b) that we can derive an upper bound on $\mathrm{E}(m_{red})$ by deriving a lower bound on $\mathrm{E}(m^*)$. Let $p_{n,l}$ denote the probability that $outdeg^*_n(1) = l$.

Lemma 5. *$p_{n,l} = 0$ for $l \notin \{1, \ldots, n\}$, $p_{1,1} = 1$ and for $n \ge 2$*

$$p_{n,l} = p_{n-1,l} \cdot (1-\epsilon)^l + p_{n-1,l-1} \cdot (1 - (1-\epsilon)^{l-1})$$

Proof: In a digraph with only a single vertex the number of vertices reachable from vertex 1 is always 1. Assume $n \ge 2$ next. If $outdeg^*_n(1) = l$, then either l vertices among the first $n-1$ are reachable from 1 and none of these vertices has an outgoing edge into vertex n (probability $p_{n-1,l} \cdot (1-\epsilon)^l$) or $l-1$ vertices among the first $n-1$ are reachable from 1 and at least one of these vertices has an outgoing edge into vertex n (probability $p_{n-1,l-1} \cdot (1-(1-\epsilon)^{l-1})$). Thus

$$p_{n,l} = p_{n-1,l} \cdot (1-\epsilon)^l + p_{n-1,l-1} \cdot (1 - (1-\epsilon)^{l-1}).$$

∎

With $\lambda_l = 1 - (1-\epsilon)^l$ the recurrence above can be rewritten as

$$\begin{aligned}
&p_{1,1} = 1; \\
&p_{n,l} = 0 \quad \text{for} \quad l \notin \{1, \ldots, n\}; \\
&p_{n,l} = (1-\lambda_l) \cdot p_{n-1,l} + \lambda_{l-1} \cdot p_{n-1,l-1}.
\end{aligned}$$

This type of recurrence is known as a pure birth process in probability theory. If one interprets n as time and l as size of a population, then the recurrence reads as follows. We start with a population of size 1. When the population has size l at time $n-1$ then it keeps at size l with probability $1-\lambda_l$ and grows to size $l+1$ with probability λ_l. We should expect that on the average size 2 is reached at time $1/\lambda_1$, size 3 is reached at time $1/\lambda_1 + 1/\lambda_2$, ... An exact formulation of this is

Lemma 6. *Let* $\varphi(l) = \sum_{j=1}^{l-1} 1/\lambda_j$ *for* $l \geq 1$. *Then*

$$\mathrm{E}(\varphi(\mathit{outdeg}_n^*(1))) = n-1$$

for all n.

Proof: We use induction on n. For $n=1$ we have

$$\begin{aligned} \mathrm{E}(\varphi(\mathit{outdeg}_1^*(1))) &= \sum_l \varphi(l) \cdot p_{1,l} \\ &= \varphi(1) \cdot p_{1,1} \qquad [\text{since } p_{1,l} = 0 \text{ for } l \neq 1] \\ &= 0 \qquad [\text{since } \varphi(1) = 0] \end{aligned}$$

and for $n \geq 2$

$$\begin{aligned} \mathrm{E}(\varphi(\mathit{outdeg}_n^*(1)) &= \sum_l \varphi(l) \cdot p_{n,l} \\ &= \sum_l \varphi(l) \cdot (\lambda_{l-1} \cdot p_{n-1,l-1} + (1-\lambda_l) \cdot p_{n-1,l}) \ [\text{by Lemma 5}] \\ &= \sum_l \varphi(l+1) \cdot \lambda_l \cdot p_{n-1,l} + \sum_l \varphi(l) \cdot p_{n-1,l} \\ &\quad - \sum_l \varphi(l) \cdot \lambda_l \cdot p_{n-1,l} \\ &= \sum_l \underbrace{(\varphi(l+1) - \varphi(l))}_{=\frac{1}{\lambda_l}} \cdot \lambda_l \cdot p_{n-1,l} + \sum_l \varphi(l) \cdot p_{n-1,l} \\ &= \sum_l p_{n-1,l} + \sum_l \varphi(l) \cdot p_{n-1,l} \\ &= 1 + n - 2, \end{aligned}$$

where $\sum_l \varphi(l) \cdot p_{n-1,l} = n-2$ by induction hypothesis. ∎

We can now use Lemma 6 to derive bounds on the expected size of the transitive closure.

Lemma 7.

a) $\mathrm{E}(\mathit{outdeg}_n^*(1)) \geq n + 1 - \dfrac{1+\ln(1/\epsilon)}{\epsilon}$

b) $\mathrm{E}(m^*) \geq n \cdot \dfrac{n+3}{2} - n \cdot \dfrac{1+\ln(1/\epsilon)}{\epsilon}$

c) $\mathrm{E}(m_{red}) \leq n \cdot \left(1 + \dfrac{\ln(1/\epsilon)}{1-\epsilon}\right) \leq n \cdot (2 + \ln(1/\epsilon))$

Proof: a) We first compute an upper bound Φ for the function φ. The bound Φ is a linear function and hence $\mathrm{E}(\Phi(\mathit{outdeg}_n^*(1))) = \Phi(\mathrm{E}(\mathit{outdeg}_n^*(1)))$. It is then easy to derive a lower bound on $\mathrm{E}(\mathit{outdeg}_n^*(1))$. Recall that $\lambda_j = 1 - (1-\epsilon)^j$ and $\varphi(l) = \sum_{j=1}^{l-1} 1/\lambda_j$.

Claim: $\varphi(l) \leq l - 2 + \dfrac{1+\ln(1/\epsilon)}{\epsilon}$

Proof: We have

$$\begin{aligned}
\varphi(l) &= 1/\epsilon + \sum_{j=2}^{l-1} \frac{1}{1-(1-\epsilon)^j} \\
&\leq 1/\epsilon + \int_1^{l-1} \frac{dx}{1-(1-\epsilon)^x} \\
&= 1/\epsilon + \left[x - \frac{\ln(1-(1-\epsilon)^x)}{\ln(1-\epsilon)} \right]_1^{l-1} \\
&= 1/\epsilon + l - 1 - \frac{\ln(1-(1-\epsilon)^{l-1})}{\ln(1-\epsilon)} - 1 + \frac{\ln \epsilon}{\ln(1-\epsilon)} \\
&\leq l - 2 + \frac{1+\ln(1/\epsilon)}{\epsilon},
\end{aligned}$$

since $\dfrac{\ln(1-(1-\epsilon)^{l-1})}{\ln(1-\epsilon)} > 0$ and $\ln(1-\epsilon) \leq -\epsilon$. ∎

Let $\Phi(l) = l - 2 + (1+\ln(1/\epsilon))/\epsilon$. Then

$$\begin{aligned}
n - 1 &= \mathrm{E}(\varphi(\mathit{outdeg}_n^*(1))) && \text{[Lemma 6]} \\
&\leq \mathrm{E}(\Phi(\mathit{outdeg}_n^*(1))) && \text{[since } \varphi \leq \Phi\text{]} \\
&= \sum_l \Phi(l) \cdot p_{n,l} \\
&= \sum_l (l - 2 + (1+\ln(1/\epsilon))/\epsilon) \cdot p_{n,l} \\
&= \mathrm{E}(\mathit{outdeg}_n^*(1)) - 2 + (1+\ln(1/\epsilon))/\epsilon.
\end{aligned}$$

Thus $\mathrm{E}(\mathit{outdeg}_n^*(1)) \geq n + 1 - (1+\ln(1/\epsilon))/\epsilon$.

b) We have

$$\begin{aligned}
\mathrm{E}(m^*) &= \mathrm{E}(\sum_{i=1}^{n} \mathit{outdeg}_n^*(i)) \\
&= \sum_{i=1}^{n} \mathrm{E}(\mathit{outdeg}_n^*(i)) \\
&= \sum_{i=1}^{n} \mathrm{E}(\mathit{outdeg}_{n-i+1}^*(1)) \\
&\geq \sum_{i=1}^{n} \left(n - i + 2 - \frac{1+\ln(1/\epsilon)}{\epsilon}\right) \\
&= n \cdot \frac{n+3}{2} - n \cdot \frac{1+\ln(1/\epsilon)}{\epsilon},
\end{aligned}$$

and part a) of Theorem 3 is completed.

Part c) of Lemma 7 follows from part b), Lemma 4b) and a simple computation:

$$\begin{aligned}
\mathrm{E}(m_{red}) &= \frac{\epsilon}{1-\epsilon} \cdot (n \cdot (n-1)/2 - \mathrm{E}(m^*)) && \text{[Lemma 4b)]} \\
&\leq \frac{\epsilon}{1-\epsilon} \cdot \left(n \cdot (n-1)/2 - n \cdot (n+3)/2 + n \cdot \frac{1+\ln(1/\epsilon)}{\epsilon}\right) && \text{[part b)]} \\
&= \frac{n}{1-\epsilon} \cdot (-2\epsilon + 1 + \ln(1/\epsilon)) \\
&\leq n \cdot \left(1 + \frac{\ln(1/\epsilon)}{1-\epsilon}\right) \\
&= n \cdot \left(1 + \frac{(1-\epsilon) + \frac{(1-\epsilon)^2}{2} + \frac{(1-\epsilon)^3}{3} + \cdots}{1-\epsilon}\right) && \text{[Taylor expansion]} \\
&\leq n \cdot (2 + \ln(1/\epsilon)).
\end{aligned}$$

∎

To complete part b) of Theorem 3 we only need the additional observation that $\mathrm{E}(m_{red}) \leq \mathrm{E}(m) \leq n^2 \cdot \epsilon$.

c) The expected running time of our algorithm is $\mathrm{E}(n \cdot (n + m_{red}))$ which is

$$O(\min(n^2 \cdot (2 + \ln(1/\epsilon)), n^3 \cdot \epsilon))$$

by part b). Next observe that for $\epsilon \geq (\ln n)/n$ we have

$$n^2 \cdot (2 + \ln(1/\epsilon)) = O(n^2 \cdot \ln n)$$

and that for $\epsilon \leq (\ln n)/n$ we have

$$n^3 \cdot \epsilon = O(n^2 \cdot \ln n).$$

∎

d) Under assumption $(\mathcal{A})$ the expected running time reduces to $E(n \cdot (n + m_{red})/\log n) = O(n^2)$ by the reasoning of part c) and Theorem 2. ∎

A closer look at Theorem 3 shows that the expected running time of our algorithm is optimal for dense digraphs. This can be seen as follows. Let $\epsilon_0 = 4 \cdot \ln n/n$. Then $E(m^*) = \Omega(n^2)$ for $\epsilon \geq \epsilon_0$ by part a) of Theorem 3 and hence the expected size of the output is quadratic for $\epsilon \geq \epsilon_0$. The expected running time is also quadratic and therefore optimal for $\epsilon \geq \epsilon_0$.

4.4. Systematic Exploration of a Graph

A fundamental requirement for most graph algorithms is the systematic exploration of a graph starting at some node s. The basic idea is quite simple.

Suppose that we have already visited some set S of nodes and have traversed some of the edges leaving nodes in S. Initially, $S = \{s\}$ and no edge has been traversed. At each step the algorithm selects one of the unused (= not traversed) edges incident to a node in S and explores it, i.e., the edge is marked used and the other endpoint of the edge is added to S. The algorithm terminates when no unused edges incident to nodes in S are left. We summarize in Program 8.

```
S ← {s};
mark all edges unused;
while there are unused edges leaving nodes in S
do choose any v ∈ S and an unused edge (v, w) ∈ E;
   mark (v, w) used;
   S ← S ∪ {w}
od.
```

Program 8

Lemma 1. *Let $G = (V, E)$ be a digraph. Then*

$$S = \{v;\ \text{there is a path from } s \text{ to } v \text{ in } G\}$$

on termination of Program 8.

Proof: If a node is added to S then it is certainly reachable from s. Suppose now that v is reachable from s, i.e., a path $v_0, \ldots, v_k$ exists from s to v. We show by induction on i that v_i is added to S. Since $s = v_0$, this is certainly true for $i = 0$. Suppose now that v_i is in S but v_{i+1} is not. Then edge (v_i, v_{i+1}) is unused and incident to a node in S. As long as this condition prevails the algorithm cannot terminate and hence v_{i+1} must be added to S. ∎

Of course, Program 8 leaves many implementation details unresolved. The major questions are how to mark edges used and unused, how to store the set S, and how to select an edge marked unused and leaving a node in S.

The first problem is easily solved. We have for each node i a pointer $p[i]$ into the adjacency list of node i. The edges to the left of the pointer are used and the other edges are unused. Initially, the pointer $p[i]$ points to the first entry of the i-th adjacency list, i.e., we mark all edges unused by executing Program 9.

```
for 1 ≤ i ≤ n
do p[i] ← adjhead[i] od.
```

Program 9

The selection of an unused edge leaving a node in S is also fairly easy. We only have to maintain the subset $\tilde{S} \subseteq S$ of all nodes $v \in S$, whose adjacency list is not yet exhausted. To select an unused edge we choose any node in $\tilde{S}$ and traverse any unused edge leaving that node. This leads to Program 10.

```
(1)   procedure explorefrom(s);
(2)   S ← {s}; mark all edges unused;
(3)   S̃ ← {s};
(4)   while S̃ ≠ ∅
(5)   do choose some node v ∈ S̃;
(6)      if there is an unused edge leaving v
(7)         then let (v, w) be any such edge;
(8)              mark (v, w) used;
(9)              if w ∉ S then add w to S;
(10)                           add w to S̃
(11)             fi
(12)        else delete v from S̃
(13)     fi
(14)  od
(15)  end.
```

Program 10

Lines (6) to (8) of Program 10 need to be refined further. Using the fact that the pointer $p[v]$ always points to the first unused edge in v's adjacency list, we can rewrite these lines as Program 11.

We still have to solve the representation question for sets S and $\tilde{S}$. On set S the operations Insert, Member and Initialize_to_Empty_Set are executed, on set $\tilde{S}$ the operations Empty?, Insert, Select_Some, Select_and_Delete_Some and Initialize_to_Empty_Set are executed. We saw in Section 3.8.1 that a boolean array is a

```
if p[v] ≠ nil
  then w ← p[v]↑.node;
       p[v] ← p[v]↑.next;
       if w ∉ S then ...
```

Program 11

good representation for S: Operations Insert and Member cost $O(1)$ time units and Initialize_to_Empty_Set costs $O(n)$ time units. For set $\tilde{S}$ we use either a stack or a queue (cf. Section 1.4.1). Then all operations on $\tilde{S}$ take $O(1)$ time units. We are now able to determine the efficiency of procedure *explorefrom*.

Lemma 2. *A call explorefrom(s) costs $O(n_s + m_s)$ time units (without counting the cost of initialization in line (2)), where $n_s = |V_s| = |\{v;\ s \xrightarrow{*} v\}|$ and m_s is the number of edges in the subgraph induced by V_s.*

Proof: One execution of the body of the while-loop takes $O(1)$ units of time. During each iteration either an edge is used up or an element is deleted from $\tilde{S}$. Since each node in V_s is added exactly once to $\tilde{S}$ (the test in line (9) avoids repetitions), the total time spent in the while-loop is $O(n_s + m_s)$. ∎

We will now put procedure *explorefrom* to its first use: determining the connected components of an undirected graph.

Definition: An undirected graph $G = (V, E)$ is **connected** if for every $v, w \in V$ there is a path from v to w. A **connected component** of an undirected graph G is a maximal (with respect to set inclusion) connected subgraph of G. ∎

The problem of determining the connected components of an undirected graph often arises in the following disguise. V is a set and $E \subseteq V \times V$ is a relation on V. Then the reflexive, symmetric, transitive closure of E is an equivalence relation. The problem is to determine the equivalence classes of this relation. In the language of graphs this amounts to determining the connected components of the undirected graph $G = (V, \{\{v, w\};\ (v, w) \in E \text{ or } (w, v) \in E\})$.

In an undirected graph the set of nodes reachable from s forms a connected component. This observation leads us to the following theorem.

Theorem 1. *The connected components of an undirected graph can be found in linear time $O(n + m)$.*

Proof: We embed procedure *explorefrom* into Program 12 and change line (2) in *explorefrom* from "$S \leftarrow \{s\}$; mark all edges unused" to "$S \leftarrow S \cup \{s\}$".

We infer from Lemma 2 that the cost of a call *explorefrom*(v) is proportional to the size of the connected component containing v. Since *explorefrom* is called exactly once for each connected component, the total running time is $O(n + m)$.

$S \leftarrow \emptyset$;
for all $v \in V$ **do** $p[v] \leftarrow adjhead[v]$ **od**;
for all $v \in V$
do if $v \notin S$ **then** $explorefrom(v)$ **fi od**.

Program 12

In what sense does this program determine the connected components of a graph? All nodes of a component are visited during one call of *explorefrom*. A list of the nodes of each component can be obtained as follows. Let *comp* be a variable of type "set of nodes" (implemented by a stack). We initialize *comp* to a singleton set $\{v\}$ before $explorefrom(v)$ is called and insert the instruction "add w to *comp*" in line (10) of *explorefrom*. Then *comp* contains all nodes of the component containing v after return from $explorefrom(v)$. ∎

Depending on the representation of set $\tilde{S}$, as stack or queue, we have two versions of the procedure *explorefrom* at hand. They are known by the names **depth-first-search** ($\tilde{S}$ is a stack) and **breadth-first-search** ($\tilde{S}$ is a queue). In depth-first-search the exploration always proceeds from the last node visited which still has unused edges, in breadth-first-search it proceeds from the first node visited which still has unused edges.

In either case *explorefrom* traverses the adjacency list of each node in a strictly sequential manner; the order of the edges in the adjacency lists has no influence on the running time. In Section 4.5 we will take a closer look at depth-first-search. In Section 4.6 we will apply depth-first-search to various connectivity problems. In Section 4.7 we will apply breadth-first-search to distance problems.

```
procedure dfs(v : V);
add v to S;
count1 ← count1 + 1; dfsnum[v] ← count1;
for all (v, w) ∈ E
do if w ∉ S
      then [add (v, w) to T];
            dfs(w)
     [else  if v -*->_T w then add (v, w) to F
                          else  if w -*->_T v then add (v, w) to B
                                                else  add (v, w) to C fi fi]
   fi
od;
count2 ← count2 + 1; compnum[v] ← count2;
end;

begin      co main program oc
S ← ∅; count1 ← 0; count2 ← 0;
[T ← F ← B ← C ← ∅;]
for all v ∈ V
do if v ∉ S
      then dfs(v)
   fi
od
end.
```

Program 13

4.5. A Close Look at Depth-First-Search

In this section we take a detailed look at depth-first-search of directed and undirected graphs. In the depth-first-search version of procedure *explorefrom* set $\tilde{S}$ is handled as a stack. It is convenient to make that stack implicit by formulating depth-first-search as a recursive procedure *dfs*, cf. Program 13. An execution of this program is called a depth-first-search or simply DFS on graph G.

Several remarks are to be made here. We have extended our basic algorithm in two respects. First of all, we number the nodes in two different ways. The first numbering *dfsnum* is with respect to the calling time of procedure *dfs*, the second numbering *compnum* is with respect to the completion time of procedure *dfs*. Second of all, we partition the edges of the graph into four classes: the tree edges T, the forward edges F, the backward edges B and the cross edges C. The partitioning process is only done conceptually (this fact is indicated by enclosing the corresponding statements in brackets); it will facilitate the discussion of depth-first-search.

Third of all, we assume that the reader knows by now how to represent set S

and how to realize line (4). S is represented as a boolean array as discussed in Section 4.4 and line (4) may be expanded into

(4a) $\quad p \leftarrow adjhead[v]$;
(4b) $\quad$ **while** $p \neq$ nil
(4c) $\quad$ **do** $w \leftarrow p\uparrow.node$; $p \leftarrow p\uparrow.next$

where $p :\uparrow edge$ and w : integer are local to procedure *dfs*.

In the example of Figure 4 tree edges are drawn solid, back edges are drawn dashed, cross edges are drawn squiggled and forward edges are drawn dash-dotted. Name (an element of $\{a, b, c, d, e\}$), depth-first-search numbers (*dfsnum*) and completion numbers (*compnum*) are indicated in each node in that order. It is assumed that the adjacency list for a is d, e, c and that *dfs*(a) is called first. In our examples we will always draw tree edges upwards and arrange the sons of a node (via tree edges) from left to right in increasing order of *dfsnum*.

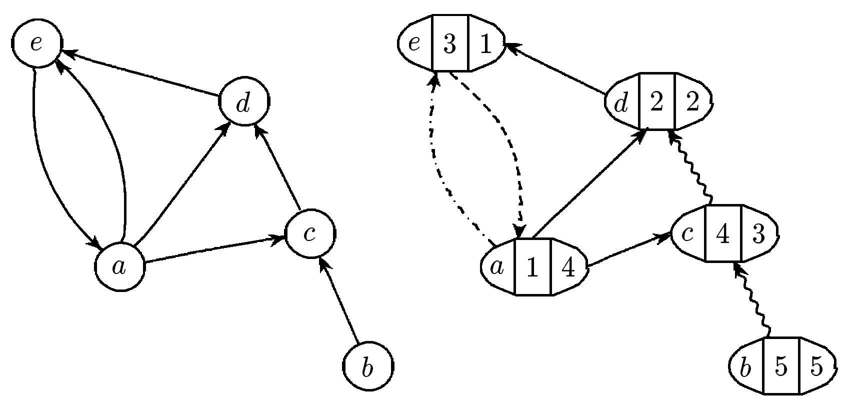


Figure 4. A graph and its *dfs*-tree

Lemma 1. *A depth-first-search on a digraph $G = (V, E)$ takes time $O(n + m)$.*

Proof: A call *dfs*(v) costs $O(outdeg_G(v))$ units of time; this accounts for the time spent in the body of *dfs* but does not account for further recursive calls. Since a node v is always added to S when the execution of *dfs*(v) starts and no node is ever removed from S, *dfs* is called at most once for each node. Hence the total time spent inside *dfs* is clearly $O(n + m)$; the total time spent outside *dfs* is $O(n)$. ∎

Next we will state some important properties of depth-first-search.

Lemma 2. *(DFS-Lemma). Let $G = (V, E)$ be a digraph and let T, F, B, C, dfsnum and compnum be defined by a depth-first-search on G.*

a) *Sets T, F, B, C form a partition of E.*

b) *$A = (V, T)$ is a spanning forest of G.*

c) *$v \xrightarrow[T]{*} w$ iff $dfsnum[v] \leq dfsnum[w]$ and $compnum[w] \leq compnum[v]$.*

d) *For all* $(v, w) \in E$*:* $(v, w) \in T \cup F$ *iff* $\mathit{dfsnum}[v] \le \mathit{dfsnum}[w]$*.*

e) *Let* v, w, z *be nodes such that* $v \xrightarrow[T]{*} w$*,* $\neg(v \xrightarrow[T]{*} z)$*, and* $(w, z) \in E$*. Then* $\mathit{dfsnum}[z] < \mathit{dfsnum}[v]$ *and* $(w, z) \in B \cup C$*. Moreover, if* $\mathit{compnum}[z] < \mathit{compnum}[v]$ *then* $(w, z) \in C$*; if* $\mathit{compnum}[z] > \mathit{compnum}[v]$ *then* $(w, z) \in B$*.*

f) *For all* $(w, z) \in E$*:*

$(w, z) \in B$ *iff* $\mathit{dfsnum}[w] > \mathit{dfsnum}[z]$ *and* $\mathit{compnum}[w] < \mathit{compnum}[z]$*.*

g) *For all* $(w, z) \in E$*:*

$(w, z) \in C$ *iff* $\mathit{dfsnum}[w] > \mathit{dfsnum}[z]$ *and* $\mathit{compnum}[w] > \mathit{compnum}[z]$*.*

Proof: a) Follows from the fact that each edge is handled exactly once during the depth-first-search on graph G.

b) When edge (v, w) is added to T in line (6) of *dfs* then w is added to S in line (2) in the following recursive call. This shows that $\mathit{indeg}_A(w) \le 1$ for all $w \in V$ and that A is acyclic.

c) Observe first that the forest A corresponds to the calling history of procedure *dfs*, i.e., $v \xrightarrow[T]{*} w$ iff the call $\mathit{dfs}(w)$ is nested within the call $\mathit{dfs}(v)$. Observe next that the call $\mathit{dfs}(w)$ is nested within the call $\mathit{dfs}(v)$ iff $\mathit{dfsnum}[v] \le \mathit{dfsnum}[w]$ and $\mathit{compnum}[w] \le \mathit{compnum}[v]$.

d) "$\Rightarrow$": If $(v, w) \in T \cup F$ then $v \xrightarrow[T]{*} w$ by definition of T and F and hence $\mathit{dfsnum}[v] \le \mathit{dfsnum}[w]$ by part c).
"$\Leftarrow$": Consider the exact instant when edge (v, w) is handled in $\mathit{dfs}(v)$. Either exploration of that edge will lead to call $\mathit{dfs}(w)$ or $v = w$ or $v \ne w$ and $\mathit{dfs}(w)$ was called before edge (v, w) is handled. In the first case we have $(v, w) \in T$. In the second case the edge (v, v) is added to F. Consider the third case. Since $\mathit{dfsnum}[v] < \mathit{dfsnum}[w]$, call $\mathit{dfs}(w)$ was started after call $\mathit{dfs}(v)$. Hence call $\mathit{dfs}(w)$ is nested within call $\mathit{dfs}(v)$ and therefore $v \xrightarrow[T]{*} w$ by part c). Thus $(v, w) \in F$ in this case.

e) The situation is visualized in Figure 5. Since $(w, z) \in E$, the call $\mathit{dfs}(z)$ is started before the call $\mathit{dfs}(w)$ is completed and hence before the call $\mathit{dfs}(v)$ is completed; since $\neg(v \xrightarrow[T]{*} z)$ the call $\mathit{dfs}(z)$ is not nested within the call $\mathit{dfs}(v)$. Thus the call $\mathit{dfs}(z)$ starts before the call $\mathit{dfs}(v)$ and hence $\mathit{dfsnum}[z] < \mathit{dfsnum}[v]$. This also implies $(w, z) \in B \cup C$ by parts a) and d).

Observe next, that $z \xrightarrow[T]{*} v$ iff $\mathit{compnum}[z] > \mathit{compnum}[v]$ by part c). Hence $(w, z) \in B$ iff $z \xrightarrow[T]{*} w$ iff $z \xrightarrow[T]{*} v$ iff $\mathit{compnum}[z] > \mathit{compnum}[v]$. This completes the proof of part e).

f) and g) follow immediately from part e) with $v = w$ and the observation that $(w, z) \in B \cup C$ implies $\neg(w \xrightarrow[T]{*} z)$. ∎

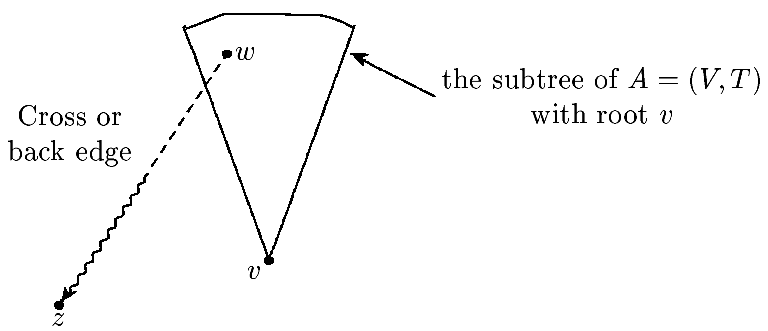


Figure 5. Situation in Lemma 2e)

It is worthwhile to restate the content of Lemma 2 in an informal way. Forward edges run from nodes to their descendants with respect to tree edges, backward edges run from nodes to their ancestors, and cross edges run from right to left in our drawings. Also, the depth-first-search number (completion number) of a node is smaller (larger) than that of its tree descendants. Cross edges run from larger to smaller depth-first-search and completion numbers. Finally, if (u, w) is a back edge, then all nodes which are completed between u and w are descendants of w.

In undirected graphs the situation is simpler; no cross edges exist and every forward edge is the reversal of a backward edge.

Lemma 3. *Let $G = (V, E)$ be an undirected graph and let T, F, B, C be defined by a depth-first-search on the directed version of G.*

a) $C = \emptyset$.

b) $(v, w) \in B$ *iff* $(w, v) \in T \cup F$ *for every dart* (v, w) *of* G.

c) *If G is connected then $A = (V, T)$ is a tree.*

Proof: a) (Indirect). Assume $C \neq \emptyset$. Let $(v, w) \in C$ be the first dart which was added to C in the depth-first-search on G. By Lemma 2g) the call *dfs*(w) is completed when dart (v, w) is handled in the call *dfs*(v). Since graph G is undirected, dart (w, v) was explored in *dfs*(w) and since (v, w) is the first dart added to C, we must have $(w, v) \in T \cup F \cup B$. In either case we have $w \xrightarrow[T]{*} v$ or $v \xrightarrow[T]{*} w$. Hence (v, w) is not added to C, contradiction.

b) Since $C = \emptyset$ by part a) this is an immediate consequence of Lemma 2, parts a), d) and f).

c) If $A = (V, T)$ were not a tree but a proper forest, then A would contain at least two trees A_1 and A_2. Since G is connected, there must be edges between A_1 and A_2. Since back edges and forward edges run parallel to paths of tree edges, such edges must be cross edges. However, depth-first-search on an undirected graph does not produce cross edges and hence A must be a single tree. ∎

In Program 13 the partitioning process on the edges is only done conceptually. In light of the DFS-Lemma it can also be done computationally. We only have to replace lines (8) to (10) by:

else if $dfsnum[v] \leq dfsnum[w]$
  then add (v, w) to F **co** part d) **oc**
  else **if** $compnum[w]$ is undefined
    then add (v, w) to B **co** part f) **oc**
    else add (v, w) to C **fi fi**   **co** part g) **oc**

We finally observe that depth-first-search gives us an alternative way of computing a topological sort of an acyclic digraph $G = (V, E)$. In an acyclic digraph there are no back edges and hence $compnum[w] < compnum[v]$ for all edges $(v, w) \in E$ by parts c) and g) of the DFS-Lemma. Hence $ord[v] = n + 1 - compnum[v]$ is a topological sort.

Theorem 1. *A topological sort of an acyclic digraph can be computed in time* $O(n + m)$. ∎

4.6. Strongly Connected Components of Directed Graphs and Biconnected Components of Undirected Graphs

We will describe linear time algorithms to determine the strongly connected components of a digraph and the biconnected components of an undirected graph. All algorithms are based on depth-first-search.

Definition:

a) A digraph $G = (V, E)$ is **strongly connected** if $v \xrightarrow{*} w \xrightarrow{*} v$ for $v, w \in V$.

b) A **strongly connected component** (**s.c.c.**) of a digraph G is a maximal strongly connected subgraph. ∎

The problem of determining the strongly connected components of a digraph often arises in the following disguise. V is a set and E a relation on V. Two elements $v, w \in V$ are called equivalent if $v \xrightarrow{*} w$ and $w \xrightarrow{*} v$. The equivalence classes of this equivalence relation are just the s.c.c.'s of $G = (V, E)$. Furthermore, shrinking the equivalence classes (s.c.c.'s) to single points leaves us with a partial order (an acyclic graph).

We will describe two linear time algorithms for computing s.c.c.'s. The first algorithm has a very simple correctness proof, but uses two passes of DFS, the second algorithm requires a more complicated proof, but uses only one pass of DFS.

Yet another one-pass algorithm is described in Exercise 8. Both one-pass algorithms can be modified for computing the biconnected components of an undirected graph.

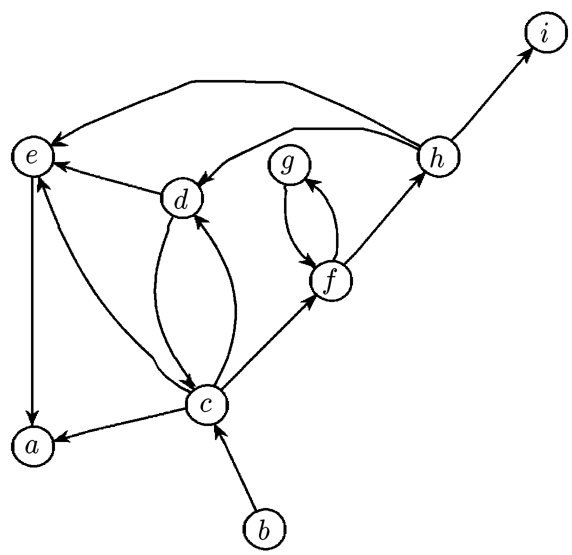


Figure 6. A graph with 5 s.c.c.'s

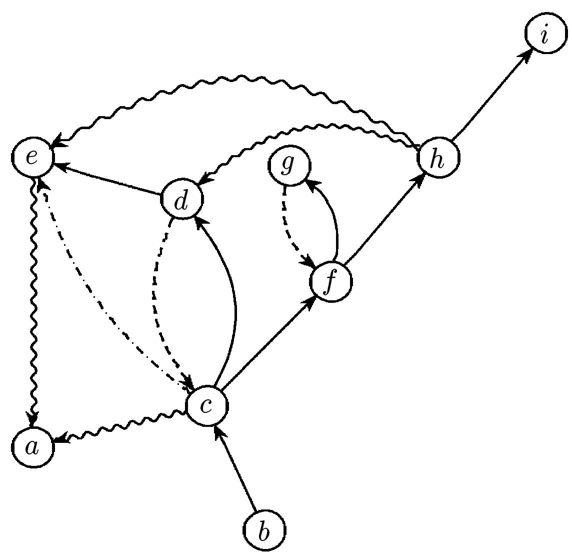


Figure 7. A DFS on the graph of Figure 6

Figure 6 shows a graph with five s.c.c.'s, Figure 7 shows a DFS on this graph, where the nodes were explored in the order a, b, c, d, e, f, g, h and i, and Figure 8 shows the acyclic graph obtained by shrinking s.c.c.'s to single nodes. The s.c.c.'s are $C_1, \ldots, C_5$, where C_i is the i-th component for which all calls $dfs(v)$, $v \in C_i$,

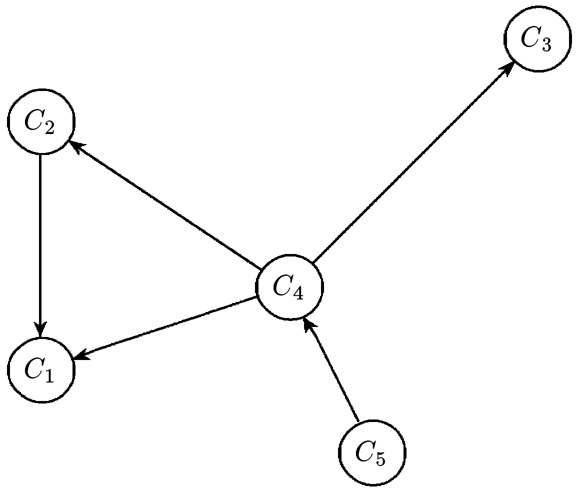


Figure 8. The graph obtained by shrinking the s.c.c.'s to single nodes

were completed. Thus $C_1 = \{a\}$, $C_2 = \{e\}$, $C_3 = \{i\}$, $C_4 = \{c,d,f,g,h\}$ and $C_5 = \{b\}$. The components $C_1, C_2, \ldots$ could be determined easily, if we knew nodes $v_1, v_2, \ldots$ such that v_i lies in C_i. Because then a call *dfs*(v_1) explores C_1, a subsequent call *dfs*(v_2) explores C_2, $\ldots$. Unfortunately, there seems to be no easy way to determine the desired sequence $v_1, v_2, \ldots$ of nodes. However, we know a node in the component completed last, namely the node v with highest completion number; this is node b in our example. A call *dfs*(v) on the reversed graph G^{-1} explores exactly the component completed last in the first pass. Let now be w the node with the highest completion number which has not yet been visited in the second pass. A call *dfs*(w) explores the component completed next to last, $\ldots$. The formal basis for this strategy is provided by

Lemma 1. *Let $C_i = (V_i, E_i)$, $1 \le i \le k$, be the s.c.c.'s of $G = (V, E)$, $n = |V|$, and let r_i be the node with largest completion number in C_i. Let us also assume that $compnum[r_1] < compnum[r_2] < \cdots < compnum[r_k]$.*

a) $compnum[r_k] = n$

b) *If* $v \overset{*}{\underset{E}{\rightarrow}} r_i$ *then* $v \in \bigcup_{j \ge i} V_j$

c) *If* $r_i \overset{*}{\underset{E^{-1}}{\rightarrow}} v$ *then* $v \in \bigcup_{j \ge i} V_j$

d) $r_i \overset{*}{\underset{E^{-1}}{\rightarrow}} v$ *for all* $v \in V_i$

Proof: a) The node with completion number n belongs to some s.c.c..

b) Let $v \in V_j$ and $v \overset{*}{\underset{E}{\rightarrow}} r_i$. Then $r_j \overset{*}{\underset{E}{\rightarrow}} r_i$ and hence *dfs*(r_i) is started before *dfs*(r_j) is completed. Thus, either $compnum[r_i] \le compnum[r_j]$ and hence $i \le j$ by assumption or call $dfs(r_j)$ is nested within call $dfs(r_i)$ and hence $r_i \xrightarrow[E]{*} r_j$ and hence $i = j$.

c) follows immediately from part b) and the observation that $v \xrightarrow[E]{*} r_i$ iff $r_i \xrightarrow[E^{-1}]{*} v$.

d) Let $v \in V_i$. Then $v \xrightarrow[E]{*} r_i$ and the claim follows. ∎

Assume that a DFS on G has been performed and that the completion numbers have been computed. Now consider a DFS on the graph G^{-1}. Assume that the first call made is $dfs(r_k)$. Note that this is easy to achieve since r_k is the node with completion number equal to n. The call $dfs(r_k)$ reaches exactly the vertices in V_k by parts c) and d) of Lemma 1. Also, at this point r_{k-1} is the node with the highest completion number which has not been reached by the search yet. A call $dfs(r_{k-1})$ will now reach all vertices in $V_{k-1}, \ldots$. This suggests the following algorithm for computing s.c.c.'s.

1) Perform DFS on G and compute the completion number of each vertex (equivalently, compute an array *ord* with $ord[i] = v$ iff $compnum[v] = i$).
2) Compute the graph G^{-1}.
3) Perform DFS on G^{-1}, where the main program considers the nodes in decreasing order of completion number, i.e., lines (18) to (20) of Program 13 are replaced by

```
(18')    for i from n downto 1
(19')    do if ord[i] ∉ S
(20')         then dfs(ord[i])
```

Lemma 2. *Under the hypothesis of Lemma 1 the following holds: the calls made in line (20) of Program 13 are exactly the calls $dfs(r_i)$, $k \geq i \geq 1$, and furthermore exactly the nodes in V_i are added to S during the call $dfs(r_i)$.*

Proof: It suffices to show that for all i the call $dfs(r_i)$ is made by the main program and that after completion of $dfs(r_i)$ the equality $S = \bigcup_{j \geq i} V_j$ holds. But this follows from Lemma 1, parts c) and d) and the observation that

$$compnum[r_i] = \max\{compnum[v];\ v \in V - \bigcup_{j>i} V_j\}.$$ ∎

We summarize in

Theorem 1. *The strongly connected components of a digraph G can be computed in linear time $O(n + m)$.*

Proof: Steps 1) and 3) are linear by Lemma 1 of Section 4.5, and step 2) is linear by Exercise 1. ∎

We now turn to an alternative algorithm which uses only one pass of DFS. The idea underlying the one-pass algorithm is to maintain the s.c.c.'s of $G_{cur} = (V_{cur}, E_{cur})$ which is the subgraph spanned by the set E_{cur} of explored edges. Initially, $V_{cur} = \{1\}$, $E_{cur} = \emptyset$ and there is only one s.c.c.. Suppose now that we explore an edge $e = (v, w)$. If $e \in T$, then w is added to V_{cur} and the node w by itself forms a s.c.c., if $e \notin T$, then the exploration of e may merge several s.c.c.'s into one. The main difficulty is to perform this merging process efficiently.

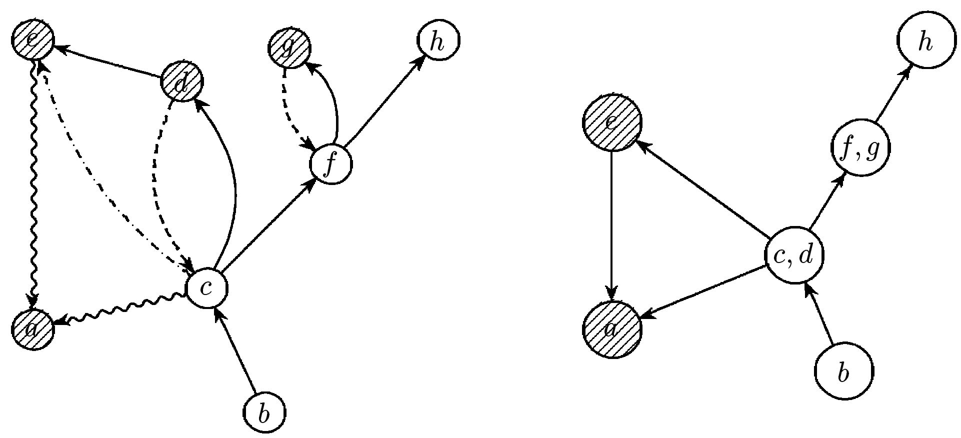


Figure 9. A snapshot of the execution of DFS on the graph of Figure 6. Nodes for which the call of *dfs* is completed are shaded. The shrunken graph is also shown. Completed components are shaded.

Figure 9 shows a typical situation for the example graph of Figure 6. In this situation the calls $dfs(a)$, $dfs(e)$, $dfs(d)$ and $dfs(g)$ have been completed and we are currently exploring edges out of node h. The s.c.c.'s of G_{cur} are $\{b\}$, $\{a\}$, $\{e\}$, $\{c, d\}$, $\{f, g\}$ and $\{h\}$. A s.c.c. C of G_{cur} is said to be **completed** if $dfs(v)$ is completed for all $v \in C$. Otherwise, it is called uncompleted. In our example, the components $\{a\}$ and $\{e\}$ are completed and the components $\{b\}$, $\{c, d\}$, $\{f, g\}$ and $\{h\}$ are uncompleted. Figure 9 also shows the graph obtained by shrinking the s.c.c.'s of G_{cur} to single nodes. We make two observations (which will be invariants of our algorithm):

I1: There are no edges (x, y) with x belonging to a completed component and y belonging to an uncompleted component.

I2: The uncompleted components form a path and we are currently exploring edges out of the last component of this path.

We can now further develop our basic idea. If we explore a tree edge (v, w), then a new uncompleted component $\{w\}$ is created and added to the path of uncompleted components. If we explore a non-tree edge (v, w) and w belongs to a completed component, then no action is required because the edge (v, w) cannot close a cycle by invariant I1. If w belongs to an uncompleted component then some final segment of the path of uncompleted components collapses to a single s.c.c. (cf. Fig. 10).

Note that in all three cases the Invariants I1 and I2 are preserved. Finally, observe that the invariants are also maintained when we complete a s.c.c. because the component to be completed is always the last component on the path of uncompleted components. The main algorithmic problem to be resolved at this point is the representation of the path of uncompleted components.

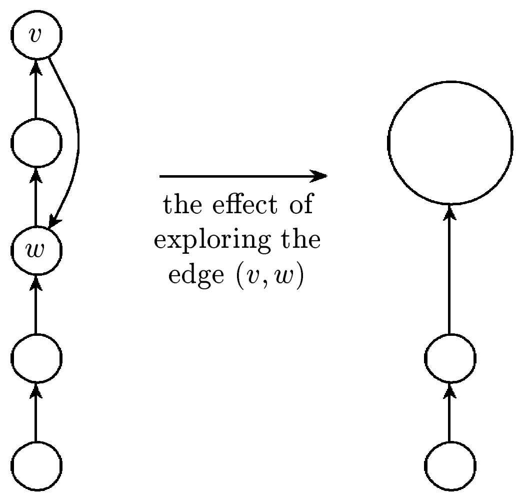


Figure 10. The path of uncompleted s.c.c.'s and the effect of exploring an edge (v, w), where w belongs to an uncompleted component.

Let *unfinished* denote the sequence of nodes belonging to uncompleted components of G_{cur} in increasing order of DFS-number. In the example of Figure 9, *unfinished* $= (b, c, d, f, g, h)$. We observe:

I3: The nodes of each uncompleted s.c.c. form a contiguous subsequence of the sequence *unfinished*.

For each s.c.c. C let us call the node with the smallest DFS-number in C the **root** of C and let *roots* be the sequence of roots of uncompleted s.c.c.'s in increasing order of DFS-number. Of course, *roots* is a subsequence of *unfinished*. In our example, *roots* $= (b, c, f, h)$. With these definitions we can reformulate (and refine) invariants I2 and I3 as follows:

Let *unfinished* $= (v_1, v_2, \ldots, v_s)$ and let *roots* $= (v_{i_1}, v_{i_2}, \ldots, v_{i_k})$, where $1 = i_1 < i_2 < \cdots < i_k$, be the subsequence of roots.

I2: The nodes in *roots* lie on a single tree path, i.e., $v_{i_l} \xrightarrow[T]{*} v_{i_{l+1}}$ for $1 \le l < k$, and we are currently exploring edges out of v_p, where $p \ge i_k$.

I3: The nodes in the uncompleted s.c.c. with root v_{i_l} are the nodes $v_{i_l}, v_{i_l+1}, \ldots,$ $v_{i_{l+1}-1}$ (with the convention $i_{k+1} = s + 1$). Moreover, all these nodes are tree descendants of the root v_{i_l}.

Let us reconsider the exploration of edges and the completion of calls. If (v, w) is the edge to be explored, let $G'_{cur} = (V_{cur} \cup \{w\}, E_{cur} \cup \{(v, w)\})$ be the new graph spanned by the explored edges. Of course, $w \in V_{cur}$ if (v, w) is not a tree edge.

- Exploration of a tree edge (v, w):

In G'_{cur} the node w is a s.c.c. by itself, of course, an uncompleted one; all other s.c.c.'s stay the same. We can reflect this change by adding the node w at the end of sequences *unfinished* and *roots*. Note that this preserves all our invariants. I1 is preserved since the node v belongs to an uncompleted component by I2; I2 is preserved since v is a tree descendant of the last element of sequence *roots* (= *top*(*roots*) if we implement *roots* as a stack) by I2, I3 and the fact that (v, w) is a tree edge; I3 is preserved since w is a s.c.c. by itself. Also, the sequences *unfinished* and *roots* are still ordered by DFS-number.

In Program 14, lines (3) and (4) implement the actions described above. The sequence *roots* and *unfinished* are realized as pushdown stores; in addition, *unfinished* is also represented as a boolean array *in_unfinished*.

- Exploration of a non-tree edge (v, w):

We have to distinguish two cases: either w belongs to a completed component (*in_unfinished*$[w]$ = *false*) or it does not. The case distinction is made in line (8) of Program 14.

Case 1: w belongs to a completed component, i.e., *in_unfinished*$[w]$ = *false*.
In this case no path exists from w to v, since v belongs to an uncompleted component of G_{cur} by I2 and no edge exists from a node in a completed component to a node in an uncompleted component by I1. Thus G'_{cur} and G_{cur} have the same s.c.c.'s and no action is required. The three invariants are clearly preserved.

Case 2: w belongs to an uncompleted component, i.e., *in_unfinished*$[w]$ = *true*.
Let *unfinished* $= (v_1, v_2, \ldots, v_s)$ and let *roots* $= (v_{i_1}, v_{i_2}, \ldots, v_{i_k})$, where $1 = i_1 < i_2 < \cdots < i_k$. Let $v = v_p$, where $p \geq i_k$ by I2, and $w = v_q$ where $i_l \leq q < i_{l+1}$, i.e., v_{i_l} is the root of the s.c.c. containing w. We claim that we can obtain the s.c.c.'s of G'_{cur} by merging the s.c.c.'s of G_{cur} with roots $v_{i_l}, v_{i_{l+1}}, \ldots, v_{i_k}$ into a single s.c.c. with root v_{i_l} and leaving all other s.c.c.'s unchanged. This can be seen as follows. Note first that completed s.c.c.'s remain the same by I1. Consider any node z in an uncompleted component next, i.e., $z = v_r$ for some r. If $r \geq i_l$, say $i_h \leq r < i_{h+1}$ with $l \leq h \leq k$, then

$$v_{i_l} \underset{E_{cur}}{\overset{*}{\to}} v_{i_h} \underset{E_{cur}}{\overset{*}{\to}} v_r \underset{E_{cur}}{\overset{*}{\to}} v_{i_h} \underset{E_{cur}}{\overset{*}{\to}} v_{i_k} \underset{E_{cur}}{\overset{*}{\to}} v \to w \underset{E_{cur}}{\overset{*}{\to}} v_{i_l}$$

where the existence of the first, the fourth and the fifth path follows from I2 and I3, the existence of the second and third path follows from the fact that v_{i_h} and v_r belong to the same s.c.c., and the existence of the seventh path follows from the fact that w and v_{i_l} belong to the same s.c.c.. Thus $z = v_r$ and v_{i_l} belong to the same s.c.c. of G'_{cur} if $r \geq i_l$.

```
procedure dfs(v : node);
count1 ← count1 + 1; dfsnum[v] ← count1; add v to S;
push v onto unfinished; in_unfinished[v] ← true;
push v onto roots;
for all w with (v, w) ∈ E
do if w ∉ S
   then dfs(w)
   else  if in_unfinished[w]
         then co we now merge components oc
              while dfsnum[top(roots)] > dfsnum[w]
              do pop(roots) od
         fi
   fi
od;
if v = top(roots)
then repeat w ← pop(unfinished); in_unfinished[w] ← false;
                co w is an element of the s.c.c. with root v oc
     until v = w;
     pop(roots)
fi
end;

begin      co main program oc
unfinished ← roots ← empty_stack; S ← ∅;
count1 ← 0;
for all v ∈ V do in_unfinished[v] ← false od;
for all v ∈ V do if v ∉ S then dfs(v) fi od
end.
```

Program 14: A one-pass s.c.c. algorithm

If $r < i_l$, say $i_h \le r < i_{h+1}$ with $h < l$, then $v_r \xrightarrow[E_{cur}]{*} v_{i_h} \xrightarrow[E_{cur}]{*} v_{i_l} \xrightarrow[E_{cur}]{*} w$, since v_r and v_{i_h} (v_{i_l} and w respectively) belong to the same s.c.c. and $v_{i_h} \xrightarrow[E_{cur}]{*} v_{i_l}$ by I2. Since $h < l$ no path exists from v_{i_l} to $v_r = z$ in G_{cur}. If did exist such a path in G'_{cur}, then it would have to use the edge (v, w) and hence there must be a path from w to v_r in G_{cur}. Thus w and v_r would belong to the same s.c.c. of G_{cur}, a contradiction. This shows that uncompleted s.c.c.'s with roots v_{i_h}, $h < l$, remain unchanged.

We have now shown that the s.c.c.'s of G'_{cur} can be obtained from the s.c.c.'s of G_{cur} by merging the s.c.c.'s with roots $v_{i_l}, \ldots, v_{i_k}$ into a single s.c.c.. The newly formed s.c.c. has clearly root v_{i_l} and hence the merge can be achieved by simply deleting the roots $v_{i_{l+1}}, \ldots, v_{i_k}$ from *roots*. Next note that $i_l \le q < i_{l+1} < \cdots < i_k$, where $w = v_q$ and hence $\mathit{dfsnum}[v_{i_l}] \le \mathit{dfsnum}[w] < \mathit{dfsnum}[v_{i_{l+1}}] < \ldots <$

dfsnum$[v_k]$ since *unfinished* and *roots* are ordered by DFS-number. This shows that the merge can be achieved by popping all roots from *roots* which have a DFS-number larger than w. That is exactly what lines (10) and (11) of Program 14 do. The three invariants are preserved by the arguments above. This finishes the description of how edges are explored. We now turn to the completion of calls *dfs*(v).

- Completion of a call *dfs*(v):

By I2 the node v is a tree descendant of $top(roots)$. If it is a proper tree descendant, i.e., $v \neq top(roots)$, then the completion of *dfs*(v) does not complete a s.c.c.. We return to *dfs*(w) where w is the parent of v. Clearly, w is still a tree descendant of $top(roots)$ and also $w \underset{E_{cur}}{\rightarrow} v \underset{E_{cur}}{\overset{*}{\rightarrow}} top(roots)$ and therefore w and $top(roots)$ belong to the same s.c.c.. This shows that I2 and I3 are preserved; I1 is also preserved since we do not complete a component.

If $v = top(roots)$ then we complete a component. By I3 this component consists of exactly those nodes in *unfinished* which do not precede $top(roots)$ and hence these nodes are easily enumerated as shown in lines (16) through (18) of Program 14. Of course, $top(roots)$ ceases to be a root of an uncompleted s.c.c. and hence has to be deleted from *roots*; line (19). We still need to prove that the invariants are preserved. For I1 this follows from the fact that all edges leaving the just completed s.c.c. must terminate in previously completed s.c.c.'s, since the uncompleted s.c.c.'s form a path by I2. The invariants I2 and I3 are also maintained by a similar argument as in the case $v \neq top(roots)$.

We have now completed the correctness proof of Program 14 and summarize in

Theorem 2. *Program 14 computes the strongly connected components of a digraph in time* $O(n + m)$.

Proof: Having already proved correctness, we still have to prove the time bound. The time bound follows directly from the linear time bound for DFS and the fact that every node is pushed onto and hence popped from *unfinished* and *roots* exactly once. This implies that the time spent in lines (11) and (16) is $O(n)$. The time spent in all other lines is $O(n + m)$ by Lemma 1 of Section 4.5. ∎

We now turn to biconnected components of undirected graphs.

Definition:

a) A connected undirected graph $G = (V, E)$ is **biconnected** if $G - v$ is connected for every $v \in V$.

b) A **biconnected component** (**b.c.c.**) of an undirected graph is a maximal biconnected subgraph.

c) A vertex $a \in V$ is an **articulation point** of G if $G - a$ is not connected. ∎

We start with a simple observation on biconnected components. Let $G_1 = (V_1, E_1)$, $\ldots, G_m = (V_m, E_m)$ be the biconnected components of an undirected graph $G =$

(V, E). Then $E = E_1 \cup \cdots \cup E_m$ and $E_i \cap E_j = \emptyset$ for $i \neq j$. To see this, note that for each edge $(v, w) \in E$ the graph consisting of vertices v and w and the single edge (v, w) is biconnected, and hence contained in one of the biconnected components of G. Thus $(v, w) \in E_h$ for some h. It remains to be shown that $E_i \cap E_j = \emptyset$ for $i \neq j$. Assume otherwise, say $(v, w) \in E_i \cup E_j$ for some $i \neq j$. Since G_i and G_j are maximal biconnected subgraphs, the subgraph $G' = (V_i \cup V_j, E_i \cup E_j)$ is not biconnected. Thus a vertex $a \in V_i \cup V_j$ must exist such that $G' - a$ is not connected. Let x and y be vertices in different components of $G' - a$. Since $G_i - a$ and $G_j - a$ are connected, we must have $x \in V_i$ and $y \in V_j$ (or vice versa). Since a cannot be equal to both v and w, we may assume $v \neq a$. Since $G_i - a$ ($G_j - a$) is connected, a path exists from x to v (y to v) in $G_i - a$ ($G_j - a$). Hence a path exists from x to y in $G' - a$, and we have reached a contradiction. We have thus shown that the b.c.c.'s of a graph give a partition of the edges of the graph.

One further observation on b.c.c.'s will be useful in the sequel. If there is a simple cycle through nodes v and w then v and w belong to the same b.c.c.. In the example of Figure 11 there are four b.c.c.'s, namely $\{f, c\}$, $\{a, b, c\}$, $\{e, g\}$ and $\{b, d, e\}$. The articulation points are c, e and b. A depth-first-search on the graph of Figure 11 could yield the structure of Figure 12; nodes are explored in the order a, b, c, f, d, e, g.

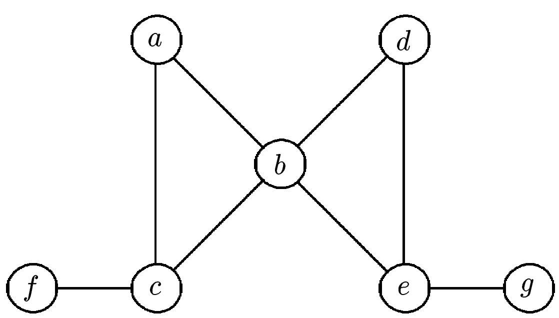


Figure 11. A graph with 4 b.c.c.'s

The biconnected components are easily recognized in Figure 12. The first edge of each b.c.c. which is explored is a tree edge; we call the endpoint of that tree edge the center of the component (formally, the node with the second smallest *dfsnum* in the b.c.c.). All back edges leaving the subtree rooted at the center of a b.c.c. end in the parent node of the center or in a tree descendant of the center. The parent of the center is always an articulation point or the root of the *dfs*-tree. The b.c.c. with center v consists of the parent of v and the nodes which are reachable from v via tree edges without going through another center. In our example the centers of the four components are f, g, d and b. The b.c.c. with center d consists of $b = parent[d]$, and the tree descendants d and e of d.

Formally, we define the **center** of a b.c.c. as the node with the second smallest *dfsnum* in the b.c.c.. The centers of b.c.c.'s play a role similar to the roots of s.c.c.'s. In fact, Program 14 is easily modified to compute the b.c.c.'s of undirected graphs,

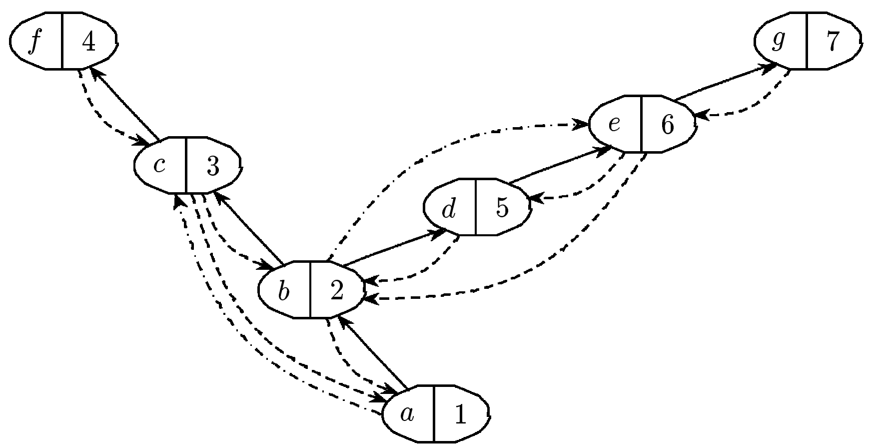


Figure 12. A DFS on the graph of Figure 11

cf. Exercise 11. Here we describe an algorithm which is closely related to the s.c.c. algorithm outlined in Exercise 8. For each node v let

$$lowpt[v] = \min(\{dfsnum[v]\} \cup \{dfsnum[z];\ \text{there is } w \text{ such that } v \xrightarrow[T]{*} w \xrightarrow[B]{} z\}).$$

Lemma 3. *Let $G = (V, E)$ be a connected undirected graph, let T, F, B and dfsnum be determined by a depth-first-search on the directed version of G and let lowpt be defined as above.*

a) *$lowpt[v] \leq dfsnum[parent[v]]$ for all v with $dfsnum[v] \geq 2$.*

b) *v is the center of a b.c.c. iff $lowpt[v] = dfsnum[parent[v]]$ and $dfsnum[v] \geq 2$.*

c) *Let $G' = (V', E')$ be a b.c.c. with center v. Then*

$$V' = \{parent[v]\} \cup \{w;\ v \xrightarrow[T]{*} w \text{ and there is no center } \neq v \text{ on the tree path from } v \text{ to } w\}.$$

d)

$$lowpt[v] = \min(\{dfsnum[v]\} \cup \{dfsnum[z];\ (v,z) \in B\} \cup \{lowpt[u];\ (v,u) \in T\}$$

for all $v \in V$.

Proof: a) If $dfsnum[v] \geq 2$ then $parent[v]$ exists and edge $(v, parent[v])$ is a back edge. Hence $lowpt[v] \leq dfsnum[parent[v]]$.

b) "$\Rightarrow$": Let v be the center of a b.c.c.. Then certainly $dfsnum[v] \geq 2$. Suppose $lowpt[v] = dfsnum[u] < dfsnum[parent[v]]$. Then a path $v \xrightarrow[T]{*} w \xrightarrow[B]{} u$ exists for some w. Also, u is a tree ancestor of v and since $u \neq parent[v]$ a tree ancestor of $parent[v]$ as well. Hence u, $parent[v]$, v and w lie on a simple cycle and hence all belong to the same b.c.c.. This b.c.c. contains at least two nodes, namely u and $parent[v]$, whose $dfsnum$'s are smaller than v's $dfsnum$, a contradiction.

"$\Leftarrow$": Suppose $dfsnum[v] \geq 2$ and $lowpt[v] = dfsnum[parent[v]]$. Consider the b.c.c. $G' = (V', E')$ containing the edge $\{parent[v], v\}$. We will show that v is the center of G'. Assume the existence of $u \in V'$, $u \neq parent[v]$ and $dfsnum[u] < dfsnum[v]$. Since $G' - parent[v]$ is connected, there must be a simple path $v_0, \ldots, v_k$ from $v = v_0$ to $v_k = u$ avoiding node $parent[v]$. Let v_i be the first node on that path which is not a tree descendant of v, i.e., $v \xrightarrow[T]{*} v_{i-1}$ and $\neg(v \xrightarrow[T]{*} v_i)$. Then edge (v_{i-1}, v_i) must be a back edge. Also, $lowpt[v] \leq dfsnum[v_i]$ by the definition of $lowpt$. Furthermore, since v_i is a tree ancestor of v_{i-1} and since v_i is not a tree descendant of v and $v_i \neq parent[v]$, v_i must be a proper ancestor of $parent[v]$. Hence $lowpt[v] \leq dfsnum[v_i] < dfsnum[parent[v]]$, a contradiction.

c) Let $G' = (V', E')$ be a b.c.c. with center v. In the proof of the second half of part b) it was shown that $parent[v] \in V'$. Also, by the definition of center $dfsnum[v] < dfsnum[w]$ for all $w \in V' - \{v, parent[v]\}$. Since $G' - parent[v]$ is connected, all vertices $w \in V' - \{v, parent[v]\}$ are reached by the search before $dfs(v)$ is completed. None of them is reached before $dfs(v)$ is started and hence $v \xrightarrow[T]{*} w$ for all $w \in V' - \{parent[v]\}$. This proves $V' \subseteq \{parent[v]\} \cup \{w;\ v \xrightarrow[T]{*} w\}$.

Next suppose $v \xrightarrow[T]{*} z$ and $z \notin V'$. Let c be the center of the b.c.c. G'' containing the edge $\{parent[z], z\}$. Then $c \xrightarrow[T]{*} z$ by the first part of the proof of part c) and hence either $v \xrightarrow[T]{*} c$ or $c \xrightarrow[T]{*} v$. $c = v$ is impossible since $z \notin V'$. In the first case we have finished. So suppose $c \xrightarrow[T]{*} v$. Since $G'' - parent[v]$ is connected ($parent[v]$ might not even be a node of G''), a path $z_0, \ldots, z_j$ must exist from z to c avoiding $parent[v]$. Because of this fact and since $v \xrightarrow[T]{*} z$ a path $v_0, \ldots, v_k$ must exist from $v = v_0$ to $c = v_k$ avoiding $parent[v]$. Let v_i be the first node of that path which is not a tree descendant of v. As in the proof of part b) one shows $lowpt[v] \leq dfsnum[v_i] < dfsnum[parent[v]]$, a contradiction. This completes the proof of part c).

d) Follows directly from the definition of $lowpt$. ∎

Lemma 3 directly leads to the algorithm shown in Program 15. Lines (4), (10) and (11) compute the $lowpt$-values; note that in line (11) the test whether $(v, w) \in B$, i.e., $dfsnum[w] < dfsnum[v]$, is unnecessary, since the line has no effect if $(v, w) \in F$, i.e., $dfsnum[v] \leq dfsnum[w]$. In line (14) the centers of b.c.c.'s are recognized by Lemma 3, part b). By part c) the b.c.c. with center v consists of $parent[v]$ and

```
(1)  procedure dfs(v : node);
(2)  count1 ← count1 + 1; dfsnum[v] ← count1;
(3)  add v to S;
(4)  lowpt[v] ← dfsnum[v];
(5)  push v on unfinished;
(6)  for all (v, w) ∈ E
(7)  do if w ∉ S
(8)      then parent[w] ← v;
(9)           dfs(w);
(10)          lowpt[v] ← min(lowpt[v], lowpt[w])
(11)     else lowpt[v] ← min(lowpt[v], dfsnum[w])
(12)     fi
(13) od;
(14) if dfsnum[v] ≥ 2 and lowpt[v] = dfsnum[parent[v]]
(15) then repeat w ← pop(unfinished);
(16)        until  w = v      co the nodes popped together with the
                                 node parent[v] form the b.c.c. with center v oc
(17) fi
(18) end;

(19) begin                     co main program oc
(20) S ← ∅; unfinished ← empty_stack;
(21) count1 ← 0;
(22) for all v ∈ V do if v ∉ S then dfs(v) fi od
(23) end.
```

Program 15

all tree descendants w of v which are not tree descendants of another center. These nodes w were not added to *unfinished* before v. Conversely, if w was not added to *unfinished* not before v and is on *unfinished* when line (14) is executed then the call $dfs(w)$ is nested within the call $dfs(v)$ (observe that we are about to complete the call $dfs(v)$) and hence w is a tree descendant of v but not a tree descendant of any other center. Thus lines (15) and (16) correctly enumerate the nodes in the b.c.c. with center v. This establishes the correctness of our algorithm.

For the running time we only have to observe that each node is popped from *unfinished* exactly once and hence the time spent in lines (15) and (16) is $O(n)$. The time spent in all other lines is $O(n+m)$ by Lemma 1 of Section 4.5. We summarize in

Theorem 3. *The biconnected components of an undirected graph can be determined in time $O(n+m)$.* ∎

In our example we have $lowpt[f] = 3$, $lowpt[g] = 6$, $lowpt[e] = lowpt[d] = 2$ and $lowpt[c] = lowpt[b] = lowpt[a] = 1$. The first center found is f. Just prior to the execution of line (15) in *dfs*(f) the content of *unfinished* is a, b, c, f. In line (15) f is deleted. Then d, e and g are added to *unfinished* and so prior to the execution of line (15) in *dfs*(g) the content of *unfinished* is a, b, c, d, e, g. In line (15) g is removed. The next center found is d and so d and e will be removed. Finally, center b is found and c and b are removed in line (15) of *dfs*(b).

We end this section with an application of the s.c.c. algorithm to the computation of the transitive closure of digraphs. Let $G = (V, E)$ be a digraph. Let $V_1, V_2, \ldots, V_k$ be the (node sets of the) s.c.c.'s of G. Let $G' = (V', E')$ be defined by

$$V' = \{V_1, \ldots, V_k\} \text{ and}$$
$$E' = \{(V_i, V_j);\ v \in V_i, w \in V_j \text{ exist such that } (v, w) \in E\}.$$

Then $G' = (V', E')$ is an acyclic digraph. Let $G'^* = (V', E'^*)$ be the transitive closure of G'. Then $G^* = (V, E^*)$, where

$$E^* = \{(v, w) \in V \times V;\ v \in V_i, w \in V_j \text{ and } (V_i, V_j) \in E'^* \text{ for some } i \text{ and } j\}$$

is the transitive closure of G. The process described above is easily turned into an algorithm. First, determine $V_1, \ldots, V_k$ in time $O(n + m)$. Secondly, construct G' in time $O(n + m)$. Thirdly, compute the transitive closure of G' in time $O(k^3)$ by the methods described in Section 4.3. Finally, E^* can be computed from G'^* in time $O(k + m^*)$, where $m^* = |E^*|$. We summarize in

Theorem 4. *Let $G = (V, E)$ be a digraph. Then the transitive closure of G can be computed in time $O(n + m^* + k^3)$, where m^* is the number of edges in the transitive closure and k is the number of s.c.c.'s of G.* ∎

4.7. Least Cost Paths in Networks

A **network** N is a directed graph $G = (V, E)$ together with a **cost function** $c : E \to \mathbb{R}$. We are interested in determining the least cost path from a fixed vertex s (the source) to all other nodes (the **single source problem**) or from each node to every other node (the **all pairs problem**). The latter problem is also treated in Chapter V.

A **path** p from v to w is a sequence $v_0, v_1, \ldots, v_k$ of nodes with $v = v_0$, $w = v_k$ and $(v_i, v_{i+1}) \in E$ for $0 \le i < k$. The **length** of the path p is k and the **cost** $c(p)$ of the path is $\sum_{i=0}^{k-1} c(v_i, v_{i+1})$. The cost of the path of length 0 is 0. The path above is **simple** if $v_i \neq v_j$ for $0 \le i < j < k$. We define the cost $\mu(u, v)$ of the least cost path from u to v by

$$\mu(u, v) = \inf\{c(p);\ p \text{ is a path from } u \text{ to } v\},$$

the infimum over the empty set being ∞.

Example: In the example of Figure 13 we have $\mu(a,e) = 1$, $\mu(e,a) = +\infty$, $\mu(a,b) = \mu(a,c) = \mu(a,d) = -\infty$. (Note that the path $ab(cdb)^i$ from a to b has length $1+3i$ and cost $3-i$.)

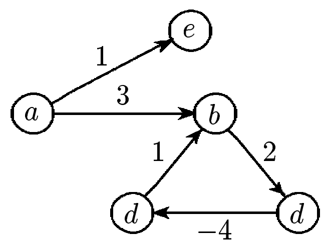


Figure 13. Graph with costs $+\infty$ and $-\infty$

We concentrate on the single source problem first, i.e., we are given a network $N = (V,E,c)$, $c: E \to \mathbb{R}$, and a node $s \in V$ and we have to determine $\mu(s,v)$ for all $v \in V$. Our algorithm for this problem is based on the following observation: The costs $\mu(s,v)$ certainly satisfy the triangle inequalities

$$\forall (u,v) \in E : \mu(s,u) + c(u,v) \geq \mu(s,v),$$

i.e., a path from s to v which consists of a least cost path from s to u followed by the edge (u,v) can certainly be of no smaller cost than the least cost path from s to v. Furthermore, for every $v \neq s$ there must be at least one edge $(u,v) \in E$ such that $\mu(s,u) + c(u,v) = \mu(s,v)$, e.g., let (u,v) be the last edge on a least cost path from s to v.

These observations lead to the following algorithm for determining least cost paths. We start with a function $cost[v]$, $v \in V$, which overestimates $\mu(s,v)$, e.g., the function $cost[s] = 0$ and $cost[v] = +\infty$ for $v \neq s$ will do. Then we look for an edge (u,v) such that *cost* does not satisfy the triangle inequality with respect to edge (u,v), i.e., $cost[u] + c(u,v) < cost[v]$. Whenever such an edge is found we use it to reduce $cost[v]$ to $cost[u] + c(u,v)$.

In the first formulation of our algorithm by Program 16 we do not only compute the costs of the least cost paths but also the paths themselves, i.e., $path[v]$ contains a path of cost $cost[v]$ from s to v stored as a sequence of nodes. We use (s) to denote the path of length zero from s to s and we use "$path[u]$ **cat** v" for extending a path from s to u to a path from s through u to v.

Program 16 is nondeterministic. Any edge violating the triangle inequality can be chosen in line (6). We will show that the correctness of the algorithm does not depend on the sequence of choices made. However, the running time depends heavily on it as the following example shows.

Example: Let $N_n = (V_n, E_n, c_n)$ be the following network:

(1) $cost[s] \leftarrow 0$; $path[s] \leftarrow (s)$;
(2) **for** all $v \in V$, $v \neq s$ **do** $cost[v] \leftarrow +\infty$;
(3) $path[v] \leftarrow$ undefined
(4) **od**;
(5) **while** $\exists (u,v) \in E : cost[u] + c(u,v) < cost[v]$
(6) **do** choose any edge $(u,v) \in E$ with $cost[u] + c(u,v) < cost[v]$;
(7) $cost[v] \leftarrow cost[u] + c(u,v)$;
(8) $path[v] \leftarrow path[u]$ **cat** v
(9) **od**.

Program 16

$$\begin{aligned}
&V_n = \{v_i, u_i, s_i;\ 0 \leq i < n\} \cup \{s_n\},\\
&E_n = \{(s_{i+1}, v_i), (s_{i+1}, u_i), (u_i, s_i), (v_i, s_i);\ 0 \leq i < n\} \text{ and}\\
&c_n : E_n \to \mathbb{R} \text{ with}\\
&c_n((s_{i+1}, u_i)) = c_n((u_i, s_i)) = c_n((v_i, s_i)) = 0\\
&c_n((s_{i+1}, v_i)) = 2^i.
\end{aligned}$$

Also $s = s_n$. Figure 14 shows N_3.

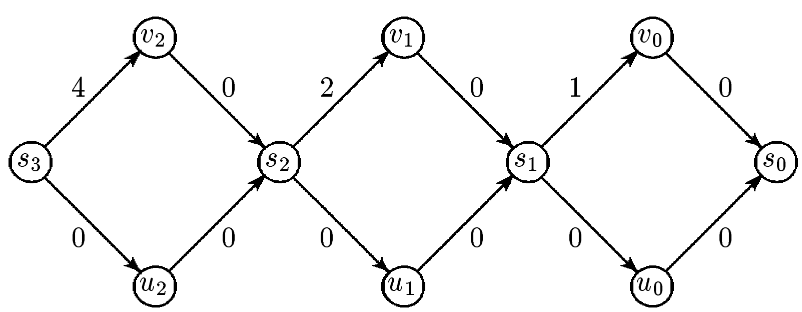


Figure 14. N_3

If the edges are chosen in the order (s_n, u_{n-1}), (u_{n-1}, s_{n-1}), (s_n, v_{n-1}), (s_{n-1}, u_{n-2}), (u_{n-2}, s_{n-2}), (s_{n-1}, v_{n-2}), … in line (6) then the body of the while loop is executed exactly $|V_n| - 1 = 3n$ times.

Now consider the following inductively defined sequence S_n of choices. On N_1 we use $S_1 = (s_1, v_0), (v_0, s_0), (s_1, u_0), (u_0, s_0)$ of length 4 and on N_n we use $S_n = (s_n, v_{n-1}), (v_{n-1}, s_{n-1}), S_{n-1}, (s_n, u_{n-1}), (u_{n-1}, s_{n-1}), S_{n-1}$ of length $|S_n| = 4 + 2 \cdot |S_{n-1}| = 4 \cdot (2^n - 1)$. Note that after using edges $(s_n, v_{n-1}), (v_{n-1}, s_{n-1}), S_{n-1}$ we will have $cost[s_n] = 0$, $cost[u_{n-1}] = \infty$, and $cost[v] \geq 2^{n-1}$ for all other nodes v. The choice of edges $(s_n, u_{n-1}), (u_{n-1}, s_{n-1})$ will reduce $cost[s_{n-1}]$ to zero. We can now run through sequence S_{n-1} again. ∎

Lemma 1.

a) *$\mu(s,v) > -\infty$ for all $v \in V$ iff the algorithm terminates.*

b) *If the algorithm terminates then $\mu(s,v) = cost[v]$ for all $v \in V$ on termination.*

Proof: a) "$\Rightarrow$": The following claim can easily be proved by induction on the number of iterations of the loop.

Claim: *Before any execution of line (6) the following holds true: path[v] is a path of cost cost[v] from s to v and if $path[v] = (v_0, \ldots, v_k)$ then for all $i < k$ we have that $(v_0, \ldots, v_i)$ was the content of $path[v_i]$ previously.* ∎

We will now show that $path[v]$ is always a simple path from s to v if $\mu(s,v) > -\infty$. Since the number of simple paths is finite and since no variable $path[v]$ can contain the same path twice this implies termination.

Assume to the contrary that $path[v]$ may be a non-simple path from s to v, i.e., $path[v] = (v_0, \ldots, v_i, \ldots, v_j, \ldots, v_k)$ and $u = v_i = v_j$ for some $i < j$. Then $p_1 = (v_0, \ldots, v_i)$ as well as $p_2 = (v_0, \ldots, v_i, \ldots, v_j)$ have been the content of $path[u]$ at some point during the execution of the algorithm. Also $c(p_2) < c(p_1)$ since $cost[u]$ decreases whenever $path[u]$ is changed. However, $c(p_2) - c(p_1)$ is the cost of the cycle $v_i, \ldots, v_j$. So the graph contains a cycle of negative cost. Going around that cycle sufficiently many times we can make the cost of a path to v as small as we want. Hence $\mu(s,v) = -\infty$, contradiction. This shows that $path[v]$ is always a simple path.

"$\Leftarrow$": When the algorithm terminates we obviously have $cost[v] > -\infty$ for all $v \in V$ on termination. Hence this direction follows from part b).

b) Suppose that the algorithm terminates and $\mu(s,v) < cost[v]$ for some $v \in V$ on termination. Then there must be a path $p = (v_0, \ldots, v_k)$ from $s = v_0$ to $v = v_k$ with $c(p) < cost[v]$. Let $p_i = (v_0, \ldots, v_i)$ be the prefix of p leading from s to v_i, $0 \le i < k$. There must be a minimal i such that $c(p_i) < cost[v_i]$ on termination. Since $c(p_0) = 0$ and $cost[v_0] \le 0$ (recall $s = v_0$) we deduce $i \ge 1$, and thus $cost[v_{i-1}] = c(p_{i-1})$ on termination. This implies

$$\begin{aligned} cost[v_i] > c(p_i) &= c(p_{i-1}) + c(v_{i-1}, v_i) \\ &= cost[v_{i-1}] + c(v_{i-1}, v_i) \end{aligned}$$

and hence the algorithm does not terminate because the triangle inequality for edge (v_{i-1}, v_i) is violated. ∎

Lemma 1 states that the algorithm will terminate with the correct costs whenever $\mu(s,v) > -\infty$ for all $v \in V$. However, we have also seen that the sequence of choices made in line (6) has a crucial influence on the running time. Also, we have to say more about the test in line (5). How do we find out whether some edge violates the triangle inequality?

Note first that when the loop is entered for the first time only the edges out of s are candidates for selection in line (6). Assume now that whenever an edge

(u, v) is selected in line (6) we will also check all other edges (u, v') going out of u for satisfaction of the triangle inequality. Then the edges going out of u do not have to be checked again until $cost[u]$ is reduced. This observation leads to Program 17, a refinement of our basic algorithm.

---

(1) $cost[s] \leftarrow 0$; $U \leftarrow \{s\}$;
(2) **for** all $v \neq s$ **do** $cost[v] \leftarrow +\infty$ **od**;
(3) **while** $U \neq \emptyset$
(4) **do co** if $u \notin U$ then $cost[u] + c(u, v) \geq cost[v]$ for all $(u, v) \in E$ **oc**
(5)   select any $u \in U$ and delete u from U;
(6)   **for** all $(u, v) \in E$
(7)   **do if** $cost[u] + c(u, v) < cost[v]$
(8)   **then** $cost[v] \leftarrow cost[u] + c(u, v)$;
(9)   add v to U
(10)   **fi**
(11)   **od**
(12) **od**.

Program 17

Our main problem still remains: Which point u shall we select from U in line (5)? The following lemma states that U always contains at least one perfect choice: a node with $cost[u] = \mu(s, u)$.

Lemma 2.

a) *If $v \notin U$ then $cost[v] + c(v, w) \geq cost[w]$ for all $(v, w) \in E$.*

b) *Let $+\infty > \mu(s, v) > -\infty$ and let $v_0, \ldots, v_k$ be a least cost path from $s = v_0$ to $v = v_k$. If $cost[v] > \mu(s, v)$ then there is an i, $0 \leq i < k$, such that $v_i \in U$ and $cost[v_i] = \mu(s, v_i)$.*

c) *If $\mu(s, v) > -\infty$ for all $v \in V$ then either $U = \emptyset$ or there exists $u \in U$ with $cost[u] = \mu(s, u)$.*

d) *If a node u having $cost[u] = \mu(s, u)$ is always chosen in line (5) then the body of the loop is executed at most n times.*

Proof: a) (By induction on the number of executions of the while loop). The claim holds certainly true before the first execution of the loop. Now suppose that $v \notin U$ after execution of the body. Then either $v \notin U$ before execution and then $cost[v]$ was not changed and $cost[w]$ for $w \neq v$ was not increased in the body of the loop and hence $cost[v] + c(v, w) \geq cost[w]$ by induction hypothesis or $v \in U$ before execution and then edge (v, w) has been considered in lines (7) and (8) and hence $cost[v] + c(v, w) \geq cost[w]$ by the algorithm.

b) Let $i = \min\{j;\ cost[v_{j+1}] > \mu(s, v_{j+1})\}$. Then $i < k$ since $cost[v_k] > \mu(s, v_k)$ by assumption and $i \geq 0$ since $\mu(s,s) = 0$ (note that $\mu(s,s) < 0$ would imply $[\mu(s,v) = +\infty$ or $\mu(s,v) = -\infty]$ for all $v \in V$) and hence $\mu(s,s) = cost[s]$. Since $i \geq 0$ we have $cost[v_i] = \mu(s, v_i)$. If v_i were not in U then by part a) $\mu(s, v_{i+1}) = \mu(s, v_i) + c(v_i, v_{i+1}) = cost[v_i] + c(v_i, v_{i+1}) \geq cost[v_{i+1}]$, a contradiction to the definition of i. Thus $v_i \in U$.

c) Let $v \in U$ be arbitrary. If $cost[v] = \mu(s,v)$ then we are done. Otherwise there is a node $u \in U$ on the least cost path from s to v with $cost[u] = \mu(s,u)$ by part b).

d) If a node u with $cost[u] = \mu(s,u)$ is always chosen in line (5) then no node can reenter U after having left U, since $cost[u]$ is reduced whenever a node is added to U. Hence every node is deleted at most once from U, i.e., the body of the loop is executed at most n times. ∎

Lemma 2 states that U always contains at least one perfect choice for the selection process in line (5). Unfortunately, in the case of arbitrary real costs we do not know any efficient method for making a perfect choice. We treat two special cases: acyclic networks (the underlying graph is acyclic) and non-negative networks (the function $c : E \to \mathbb{R}_0^+$ assigns non-negative costs to every edge). In these cases we obtain $O(n+m)$ and $O(m + n \log n)$ algorithms, respectively. In the general case we can only make sure that a good choice is made at each $O(n)$-th iteration of the loop. This will lead to an $O(n \cdot m)$ algorithm.

4.7.1. Acyclic Networks

Let $G = (V, E)$ be an acyclic graph and $c : E \to \mathbb{R}$ be a cost function on the edges. We assume that G is topologically sorted, i.e., $V = \{1, \ldots, n\}$ and $E \subseteq \{(i,j);\ 1 \leq i < j \leq n\}$ and $s = 1$. In Section 4.2 we saw that a graph can be topologically sorted in time $O(n+m)$. We replace line (5) by

(5′) select and delete $u \in U$ with u minimal.

Then u is always a perfect choice, i.e., $cost[u] = \mu(s,u)$ when u is selected from U: Indeed, by Lemma 2 there must be a node $v \in U$ on the least cost path from s to u with $cost[v] = \mu(s,v)$. Since the graph is topologically sorted we must have $v \leq u$ and hence $v = u$ by the definition of u in line (5′).

Line (5′) steps through the nodes of G in increasing order. We can therefore do completely away with set U and reformulate the algorithm as Program 18.

Theorem 1. *In acyclic graphs the single source least cost paths problem can be solved in time $O(n+m)$.*

Proof: Topological sorting takes time $O(n+m)$. Program 18 also clearly runs in time $O(n+m)$. ∎

```
cost[1] ← 0;
for all v ≥ 2 do cost[v] ← +∞ od;
for u from 1 to n − 1
do for all (u, v) ∈ E
    do cost[v] ← min(cost[v], cost[u] + c(u, v))
    od
od.
```

Program 18

There is a generalization of acyclic graphs which still allows for a linear time shortest path algorithm: shortest-path-orderable graphs.

Definition: A digraph $G = (V, E)$ with source s is **shortest-path-orderable** with respect to s (or briefly, **sp-orderable**) if there is permutation $e_1, e_2, \ldots, e_m$ of E such that every simple path starting in s uses edges in increasing order. The sequence $e_1, \ldots, e_m$ is called an **sp-order**.

Figure 15 shows examples of sp-orderable and non-sp-orderable graphs. For the two orderable graphs the edges are labelled $a, b, c, \ldots$ corresponding to a possible ordering. The third graph of Figure 15 is not sp-orderable since there is a simple path using e before e' and a simple path using e' before e.

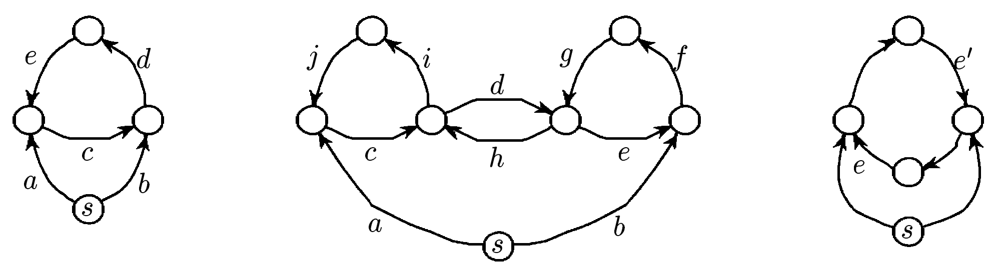


Figure 15. Two sp-orderable graphs and one graph which is not sp-orderable

Theorem 2. *Let $G = (V, E)$ with source s be sp-orderable and let $e_1, e_2, \ldots, e_m$ be an sp-order of E. Then the single source least cost path problem can be solved in linear time $O(n + m)$ for any cost function $c : E \to \mathbb{R}$.*

Proof: Consider Program 19. It runs in time $O(n + m)$ and computes a function $cost : V \to \mathbb{R}$. If this function satisfies the triangle inequality, a fact that can be checked in linear time, Program 19 corresponds to a run of Program 16 with the edges considered in the order $e_1, \ldots, e_m$ and hence $cost[v] = \mu(s, v)$ for all v by Lemma 1b. Suppose now that the resulting cost function does not satisfy the triangle inequality. Let W be a set of targets of edges violating the triangle inequality, i.e., $W = \{w;\ \exists(v, w) \in E, cost[w] > cost[v] + c(v, w)\}$, and let CW be the set of nodes reachable from a node in W.

$cost[s] \leftarrow 0$;
for al $v \neq s$ **do** $cost[v] \leftarrow +\infty$ **od**;
for $i \leftarrow 1$ **to** m
do let $e_i = (v, w)$;
$cost[w] = \min(cost[w], cost[v] + c(v, w))$
od.

Program 19

Claim: *$\mu(s, v) = cost[v]$ for $v \notin CW$ and $\mu(s, v) = -\infty$ for $v \in CW$.*

Proof: We show first that $\mu(s, v) > -\infty$ implies $\mu(s, w) = cost[w]$. If $\mu(s, w) > -\infty$ then $\mu(s, w)$ is the cost of a simple path from s to w. Since $e_1, \ldots, e_m$ is an sp-order of G, p consists off edges $e_{i_1}, e_{i_2}, \ldots, e_{i_k}$ with $i_1 < i_2 < \cdots < i_k$. It is now obvious that $cost[w] = \mu(s, w)$ on termination of Program 19. Consider any node w with $\mu(s, w) = -\infty$ next. Then there is a path $p_1 p_2 p_3$ from s to w such that p_2 is a cycle and $c(p_2) < 0$. Let p_2 consist of edges $e_{i_1}, e_{i_2}, \ldots, e_{i_k}$ where $e_{i_j} = (v_j, v_{j+1})$ and $v_{k+1} = v_1$. Then

$$\sum_{j=1}^{k}(cost[v_{j+1}] - cost[v_j] - c(e_{i_j})) = cost[v_{k+1}] - cost[v_1] - \sum c(e_{i_j})$$
$$= 0 - c(p_2) > 0$$

and hence there is some j with $cost[v_{j+1}] > cost[v_j] + c(e_{i_j})$, i.e., $v_{j+1} \in W$. Also w is clearly reachable from v_{j+1} and hence $w \in CW$.

We have now shown that $\mu(s, w) = -\infty$ implies $w \in CW$ and $w \notin CW$ implies $\mu(s, w) > -\infty$ and hence $\mu(s, w) = cost[w]$. It remains to show that $w \in CW$ implies $\mu(s, w) = -\infty$. Let $w \in W$, i.e., there is an edge $e = (v, w)$ with $cost[w] > cost[v] + c((v, w))$. Then either $cost[w] \neq \mu(s, w)$ or $cost[v] \neq \mu(s, v)$ and hence either $\mu(s, w) = -\infty$ or $\mu(s, v) = -\infty$. In either case we have $\mu(s, v) = -\infty$. This completes the proof of the claim and of Theorem 2. ▮▮

The best algorithm known to decide whether a directed graph is sp-orderable and, if need be, to compute an sp-order runs in time $O(n^2)$, cf. the bibliographic remarks. Together with Theorem 2 this yields an $O(n^2 + m)$ algorithm for the single source shortest path problem for sp-orderable graphs which compares favorably with the $O(n \cdot m)$ bound for arbitrary graphs; cf. Section 4.7.4. Furthermore, there are practically important subclasses of sp-orderable graphs where an order can be computed in linear time; cf. Exercises 1101 and 1102.

4.7.2. Non-negative Networks

A network $N = (V, E, c)$ is non-negative if $c : E \to \mathbb{R}_0^+$ assigns non-negative costs to every edge. Non-negative networks arise very frequently in practice and therefore the least cost path problem for these networks has been studied intensively. This section is divided into four parts. In the first part we reduce the least cost path problem to a data structure problem, namely the efficient realization of a priority queue. Different implementations of priority queues, which we treat in the second part, yield different running times. We show how to achieve time $O(m + n \log n)$, $O(m+n\sqrt{\log C})$, and $O(m \log \log C)$ respectively; for the latter bounds it is assumed that edge costs are integers in the range $[0 \mathinner{..} C]$. In the third part we deal with the one-pair problem and in the fourth part we describe a scaling approach for the least cost path problem. In this approach edge costs are also assumed to be integral. The method works in phases and computes successively better approximations to the final solution. Although the achieved running time is only $O(m \cdot \log_{m/n} C)$, the algorithm is very simple and serves as a first illustration of the scaling method. Further applications of the method will be seen in the sections on matching and network flow.

4.7.2.1. A Basic Algorithm for Non-Negative Networks

We replace line (5) of Program 16 by

$(5'')$ select and delete $u \in U$ with $cost[u]$ minimal.

Then u is always a perfect choice, i.e., $cost[u] = \mu(s,u)$ when u is selected from U: By Lemma 2 there must be a node $v \in U$ on the least cost path from s to u with $cost[v] = \mu(s,v)$. Since u is selected we have $cost[u] \le cost[v]$ and since v is on the least cost path from s to u and edge costs are non-negative we have $\mu(s,v) \le \mu(s,u)$. Therefore $cost[u] \le cost[v] = \mu(s,v) \le \mu(s,u)$ and since $cost[u] \ge \mu(s,u)$ always we even have $cost[u] = \mu(s,u)$.

How shall we implement set U? What operations are required on set U and function *cost*? In line $(5'')$ we need to select and delete $u \in U$ with $cost[u]$ minimal. In line (7) we need to obtain the value $cost[v]$ given node v and in line (8) we need to change the function value at argument v. In line (9) we need to add v to U if it is not already there. Finally, we need to initialize U and *cost* in lines (1) and (2). A data structure supporting these operations is called a priority queue. A precise definition is as follows.

Let K be any linearly ordered set with linear order $\le$ and let *INF* be any set. A **priority queue** (over K and *INF*) is a partial function $pq : I \to K \times INF$, where I is a set of items (in implementations of priority queues I is typically a set of array indices or a set of storage locations). For a pair $p = (k, inf) \in K \times INF$ let $\mathrm{key}(p) = k$ and $inf(p) = inf$. The following operations on priority queues are provided.

procedure *Create*$(PQ : \text{priority queue}, n : \text{integer})$
  co creates a set I of n items and gives PQ the value pq_0 where pq_0 is the function with empty domain. **oc**

function *Insert*(PQ : priority queue; $k : K, inf : INF$) : item
co let pq be the function denoted by PQ and let $i \in I - dom(pq)$ be any "unused" item. Then PQ is made to denote pq' where

$$pq'(j) = \begin{cases} (k, inf) & \text{if } j = i, \\ pq(j) & \text{if } j \neq i. \end{cases}$$

Also, the item i is returned by the function. **oc**

function *Findmin*(PQ : priority queue) : item
co Let pq be the function denoted by PQ and let $i \in dom(pq)$ be such that $\text{key}(pq(i)) \leq \text{key}(pq(j))$ for all $j \in dom(pq)$. Then i is returned. If $dom(pq) = \emptyset$ then this function is undefined. **oc**

function *is_empty*(PQ : priority queue) : boolean
co Let pq be the function denoted by PQ; returns true if $dom(pq) = \emptyset$ and false otherwise. **co**

function *get_key*(PQ : priority queue; i : item) : K
co Let pq be the function denoted by PQ;
then $\text{key}(pq(i))$ is returned if $i \in dom(pq)$. If $i \notin dom(pq)$ then the result is undefined. **oc**

function *get_inf*(PQ : priority queue; i : item) : K
Let pq be the function denoted by PQ; Then $inf(pq(i))$ is returned if $i \in dom(pq)$. If $i \notin dom(pq)$ then the result is undefined. **oc**

procedure *Delete*(PQ : priority queue; i : item)
co Let pq be the function denoted by PQ; then PQ is made to denote pq' where $dom(pq') = dom(pq) - \{i\}$ and $pq'(j) = pq(j)$ for all $j \in dom(pq')$. **oc**

procedure *Deletemin*(PQ : priority queue)
co $Delete(PQ, Findmin(PQ))$ **oc**

procedure *Decrease_key*(PQ : priority queue; i : item, k : key)
co Let pq be the function denoted by PQ. This operation assumes $i \in dom(pq)$ and $\text{key}(pq(i)) > k$. Then PQ is made to denote pq' where

$$pq'(j) = \begin{cases} (k, get\_inf(PQ, j)) & \text{if } j = i \\ pq(j) & \text{otherwise.} \end{cases}$$

oc

In the least cost path problem we use a priority queue with $K = \mathbb{R}_0^+$ and $INF = V$. Rewriting Program 17 yields Program 20. In line (0f) we make PQ a priority queue which can hold up to n items and initialize it with the empty function. In (1a) we insert the pair $(0, s)$ into PQ and remember the item created in $I[s]$. In (5a) we select the item of minimal key, extract the node associated with it in line (5b) and delete the item in (5c). In line (8a) we distinguish whether v was not added yet to

$PQ(cost[v] = \infty)$ or whether v already belongs to PQ. In the first case we insert the pair $(cost[v], v)$ into PQ and remember the new item in $I[v]$, in the second case we decrease the key of item $I[v]$ to the new cost.

```
(0a)   PQ : priority queue with K = ℝ₀⁺ and INF = V;
(0b)   I : array[1 .. n] of items;
(0c)   cost : array[1 .. n] of K = ℝ₀⁺ ∪ {∞};
(0d)   i, j : item, v, u : node, d : ℝ₀⁺ ∪ {∞};
(1f)   Create(PQ, n);
(1a)   I[s] ← Insert(PQ, 0, s);
(1b)   cost[s] ← 0;
(2)    for all v ≠ s do cost[v] ← +∞; I[v] ← nil od;
(3)    while ¬is_empty(PQ)
(4)    do
(5a)       i ← Findmin(PQ);
(5b)       u ← get_inf(PQ, i);
(5c)       Delete(PQ, i);
(6)        for all (u, v) ∈ E
(7)        do if cost[u] + c(u, v) < cost[v]
(8a)           then cost[v] ← cost[u] + c(u, v);
(8b)                if I[v] = nil
(9)                 then I[v] ← Insert(PQ, cost[v], v)
(8c)                 else  Decrease_key(PQ, I[v], cost[v])
(8e)                 fi
(10)           fi
(11)       od
(12)   od.
```

Program 20

Theorem 3. *Program 20 solves the single source least cost path problem in non-negative networks in time* $O(n + m + T_{Create}(n) + n \cdot (T_{Insert}(n) + T_{Findmin}(n) + T_{get\_inf}(n) + T_{Delete}(n)) + m \cdot T_{Decrease\_key}(n))$. *Here* $T_{XYZ}(n)$ *denotes the cost of priority queue operations* XYZ *on a queue of size at most* n.

Proof: For the correctness we only need to observe that Program 20 refines Program 18. For the time bound we only need to observe that there are never more than n items in PQ, that *Create* is executed once, that *Findmin*, *get_inf*, *Delete* and *Insert* are executed at most once for each node and that *Decrease_key* is executed at most once for each edge. ∎

Different implementations of priority queues are discussed in Section 4.7.2.2. We close this section with an important observation about the sequences of nodes selected in line $(5'')$.

Definition: The usage of priority queue PQ is **monotone** if all calls $Insert(PQ, k, \ldots)$ and $Decrease\_key(PQ, \ldots, k)$ satisfy $k \geq get\_key(i)$ where i is the item returned by the most recent *Findmin* operation. ∎

Lemma 3. *The least cost path algorithm uses its priority queue in a monotone way.*

Proof: Let i be an item returned in line (5a) of Program 20, let u be the associated node, and let $cost[u]$ be the associated cost. Then since edge costs are non-negative, the new value inserted in line (9) is no smaller than $cost[u]$ and the decreased value of line (8d) is no smaller than $cost[u]$. Hence the key of the item selected in the next iteration is at least $cost[u]$. This shows that the priority queue is monotone. ∎

4.7.2.2. Priority Queues

In this section we present several implementations of priority queues. Each implementation gives us a concrete algorithm for the least cost path problem. We divide the implementations into two groups: General priority queues and integer valued queues. For the implementations in the first group the set K of keys can be any linearly ordered set, for integer valued queues we have $K = [0..C]$ for some prespecified (in the *Create* operation) integer C.

4.7.2.2.1. General Priority Queues

The most simple implementation of priority queues (called the **array implementation**) uses three arrays $PQ.K$: array$[1..n]$ of K and $PQ.INF$: array$[1..n]$ of INF and $PQ.is\_used$: array$[1..n]$ ofboolean for a priority queue created by a call $Create(PQ, n)$. The set I of items is equal to $[1..n]$. Then *Create* allocates the three arrays and initializes $PQ.is\_used$ to false in time $O(n)$. $Insert(PQ, k, inf)$ determines an unused item i by linear search through the array $PQ.is\_used$, declares i used, stores k and inf and returns i. All of this takes time $O(n)$. *Findmin* scans through the arrays $PQ.is\_used$ and $PQ.K$ and determines an item of minimum key in time $O(n)$. *is_empty* takes also $O(n)$ by a scan through $PQ.is\_used$, and finally *get_key*, *get_inf*, *Delete* and *Decrease_key* take clearly time $O(1)$. We summarize in:

Theorem 4.

a) *The array-implementation of priority queues supports the priority queue operations with time bounds* $T_{Create}(n) = T_{Insert}(n) = T_{Findmin}(n) = T_{is\_empty}(n) = O(n)$ *and* $T_{get\_key}(n) = T_{get\_inf}(n) = T_{Delete}(n) = T_{Decrease\_key}(n) = O(1)$.

b) *The least cost path problem in non-negative networks can be solved within time complexity $O(n^2)$.*

Proof: Part a) follows from the discussion above and Part b) follows from Part a) and Theorem 3. $\blacksquare$

For almost complete graphs, i.e., $m = \Omega(n^2)$, the $O(n^2)$ running time provided by Theorem 4b is clearly optimal. If $m = O(n^2)$ then the running time is dominated by the n calls of *Insert*, *Findmin* and *is_empty*.

A second implementation (the **heap implementation**) of priority queues stores the pairs in $range(pq)$ in a heap, cf. Section 2.1.2, ordered according to key values. More precisely, let $a \geq 2$ be an integer to be chosen later. A priority queue PQ created by a call $Create(PQ, n)$ is realized by four arrays $PQ.K$: array$[1..n]$ of K, $PQ.INF$: array$[1..n]$ of INF, $PQ.location\_of\_item$: array$[1..n]$ of integer $PQ.item\_of\_location$, a list *unused_items*, and an integer $PQ.free$. The set I of items is equal to $[1..n]$. If pq is the function denoted by priority queue PQ then the following four conditions hold

(1) $PQ.free = |dom(pq)| + 1$,

(2) for $i \in dom(pq)$ and $j = PQ.location\_of\_item[i]$: $pq(i) = (PQ.K[j], PQ.INF[j])$, $1 \leq j < PQ.free$, and $PQ.item\_of\_location[j] = i$,

(3) *unused_items* is a list of the integers in $[1..n] - dom(pq)$,

(4) for $2 \leq j < PQ.free$: $PQ.K[\lceil (j-1)/a \rceil] \leq PQ.K[j]$.

The first two conditions state that pairs in $range(pq)$ are stored in locations $1..PQ.free-1$ of the arrays $PQ.K$ and $PQ.INF$ and that the arrays *location_of_item* and *locit* translate between items and locations. The third condition states that the list *unused_items* contains exactly those items which do not belong to the domain of pq and the fourth condition finally states that if we identify locations with the nodes of a complete tree of degree a, i.e., the root is labelled 1, its children 2, 3, ..., $a+1$, and so on, then this tree has the heap property, i.e., the key of the parent of any node is never larger than the key of the node itself. Figure 16 illustrates these notions for $a = 3$. Note that in complete a-ary tree the parent of node $j \geq 2$ is labelled $\lceil (j-1)/a \rceil$ and that the children of node j have labels $a \cdot (j-1) + 2, \ldots, a \cdot j + 1$.

Theorem 5.

a) *The heap-implementation with parameter a supports the priority queue operations with time bound $T_{Create}(n) = O(n)$, $T_{Findmin}(n) = T_{is\_empty}(n) = T_{get\_key}(n) = T_{get\_inf}(n) = O(1)$, $T_{Delete}(n) = O(a \cdot \log_a n)$ and $T_{Insert}(n) = T_{Decrease\_key}(n) = O(\log_a n)$.*

b) *For $a \geq 2$ the single source least cost path problem in non-negative networks can be solved in time $O(a \cdot n \log n / \log a + m \cdot \log n / \log a)$*

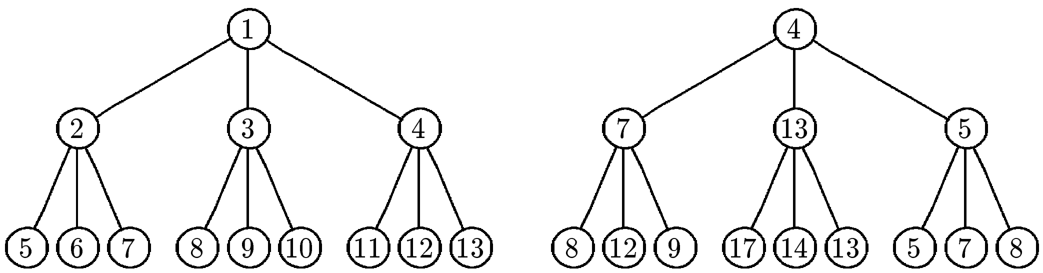


Figure 16. The first tree shows the numbering of the nodes in a ternary heap. The second tree shows key values satisfying the heap property. it corresponds to array $[4, 7, 13, 5, 8, 12, 9, 17, 14, 13, 5, 7, 8]$.

c) *The single source least cost path problem in non-negative networks can be solved in time* $O(n + m \cdot \log n / \max(1, \log m/n))$.

Proof: a) *Create* needs to allocate four arrays of size n, a linear list of n elements and takes therefore time $O(n)$. *is_empty*, *get_key* and *get_inf* take clearly time $O(1)$. *Findmin* takes also time $O(1)$ since the root of a heap always corresponds to an item of smallest key and hence *Findmin* can return $PQ.item\_of\_location[1]$. Operation $Decrease\_key(PQ, i, k)$ can be realized in time $O(\log_a n)$ as follows. We start in the location $loc = PQ.location\_of\_item[i]$ and walk towards the root. As long as the key stored in the parent location of *loc* is larger than k we move that key and the corresponding information into location *loc*, update the correspondence between locations and items, change *loc* to its parent and continue. The details are given by Program 21. Since the depth of an a-ary tree with n nodes is $O(\log_a n)$ the cost of a *Decrease_key* operation is $O(\log_a n)$. The operation $Insert(PQ, k, inf)$ is only slightly more complicated. We take the first item, say i, from the list *unused_items*, establish the correspondence between i and location *PQ.free*, increase *PQ.free* by one and then execute the program for $Decrease\_key(PQ, i, k)$. All of this takes time $O(\log_a n)$.

It remains to discuss $Delete(PQ, i)$. Let *loc* be the location corresponding to item i. If $loc = PQ.free - 1$ then we only have to decrement *PQ.free*. If $loc \neq PQ.free - 1$ we move the content of location $PQ.free - 1$ to location *loc* and then restore the heap property. If the content of that location became smaller then we essentially perform a *Decrease_key* and need time $O(\log_a n)$. If it became larger then we scan through the children of location *loc* and find the location, say *locnew*, with smallest key. This takes time $O(a)$. We then interchange the contents of locations *loc* and *locnew* and continue. Of course, we also update the correspondence between locations and items. All of this takes time $O(a \cdot \log_a n)$.

b) Follows immediately from part a) and Theorem 3.

c) Follows from part b) with $a = \max(2, m/n)$. ∎

It is worthwhile to look at Theorem 2 for some particular values of m. If $m = O(n)$ then running time is $O(n \log n)$, if $m = n \log n$ then running time is $O(m \cdot$

```
procedure Decrease_key(PQ : priority queue, i : item, k : key);
loc ← PQ.location_of_item[i]; co loc is the location corresponding to pq(i) oc
inf ← PQ.INF[loc];
while loc > 1 and PQ.K[⌈(loc − 1)/a⌉] > k
do locnew ← ⌈(loc − 1)/a⌉;
   co We move the content of location locnew into location loc
      and update the correspondence between items and locations. oc
   PQ.K[loc] ← PQ.K[locnew];
   PQ.INF[loc] ← PQ.INF[locnew];
   PQ.item_of_location[loc] ← PQ.item_of_location[locnew];
   PQ.location_of_item[PQ.location_of_item[loc]] ← loc;
   loc ← locnew
od
co We store the pair (k, inf) in location loc and establish the
   correspondence between item i and location loc. oc
PQ.K[loc] ← k;
PQ.INF[loc] ← inf;
PQ.item_of_location[loc] ← i;
PQ.location_of_item[i] ← loc;
```

Program 21

$\log n/\log\log n$), and if $m = n^{1+1/k}$ then running time $O(k \cdot m)$. So for sufficiently dense graphs the running time is linear, but for sparse graphs the running time is non-linear.

With $a = 2$ the heap implementation of priority queues supports all operations in time $O(\log n)$. We next describe the Fibonacci heap implementation which supports *Deletemin* and *Delete* in (amortized) time $O(\log n)$ whilst achieving (amortized) time $O(1)$ for all other operations. This will lead to an $O(m + n\log n)$ bound for the single source least cost path problem in non-negative networks.

A **Fibonacci heap** (**F-heap**) represents a priority queue pq as a collection of heap-ordered trees; each item $i \in dom(pq)$ uniquely corresponds to a node of one of the trees in the collection. A tree is **heap-ordered** if for each non-root node v the key of the item corresponding to v is no less than the key of the item corresponding to the parent of v.

The storage representation of F-heaps makes use of the following types:

```
type node= record key : K;
                  inf : INF;
                  parent, leftsib, rightsib, child ↑node;
                  rank : integer;
                  marked : boolean
           end;
     item=↑node;
```

priority queue=$\uparrow$*node*;

Each node contains a pointer to its parent (the value of the pointer is nil for a root) and to one of its children. The children of each node and also the roots of the trees in a F-heap form a doubly-linked circular list (pointers *leftsib* and *rightsib*). Finally, the *rank* field of each node contains the number of children of the node and the *marked* field is a boolean flag which will be explained later on. A F-heap is accessed by a pointer to a root of minimum key. Figure 17 shows a F-heap and its storage representation.

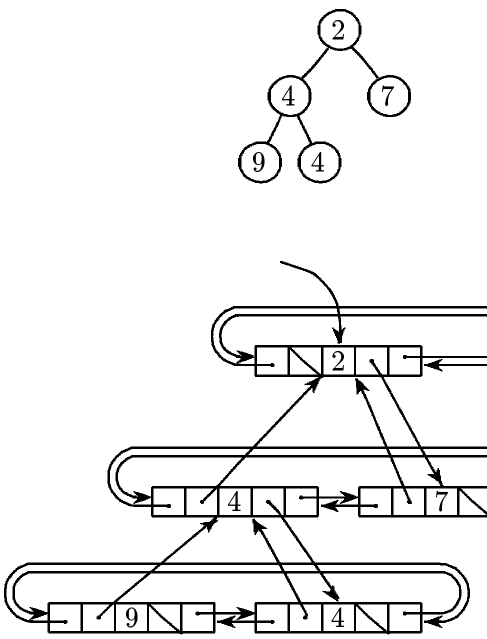


Figure 17. A F-heap and its storage representation. The information, *rank* and *marked* fields are not shown, keys are integers and nil-pointers are indicated by $\boxed{\diagdown}$.

We can now discuss the implementation of the various

4.7.3. General Networks

We will now treat the case of general networks $N = (V, E, c)$, $c : E \to \mathbb{R}$. In this case no efficient method exists which guarantees a perfect choice. However, if U is organized as a queue, then between any two selections of the same node we will have made one perfect choice.

More precisely, U is implemented as a queue UQ and a boolean array UB. We use UQ in order to select elements in line (4) of Program 22 on a first-in first-out basis. Furthermore, we use the boolean array representation of U in order to test in line (11) whether v is already present in U and if not we add v to the end of the queue. The complete algorithm is specified in Program 22.

```
(1)   cost[s] ← 0; UQ ← ∅; add s to the end of UQ;
      UB[s] ← true; count[s] ← 0;
(2)   for all v ≠ s do cost[v] ← +∞; count[v] ← 0; UB[v] ← false od;
(3)   while UQ ≠ ∅
(4)   do let u be the first element in UQ;
(5)     count[u] ← count[u] + 1;
(6)     if count[u] ≥ n + 1 then goto Exit fi;
(7)     delete u from UQ; UB[u] ← false;
(8)     for all (u, v) ∈ E
(9)     do if cost[u] + c(u, v) < cost[v]
(10)          then cost[v] ← cost[u] + c(u, v);
(11)               if ¬UB[v]
(12)                 then add v to the end of UQ;
(13)                      UB[v] ← true
(14)               fi
(15)       fi
(16)    od
(17)  od;
(18)  Exit: if count[v] = n + 1 for some v ∈ V
(19)         then "cycle of negative cost exists"
(20)         else  "cost[v] = μ(s, v) for all v ∈ V" fi.
```

Program 22

We have added the array of counters in order to ensure termination even in the presence of cycles of negative costs. We still have to show that the counters do not impede correctness. Queue UQ is implemented as described in I.4, i.e., either by a linear list or by an array.

Theorem 6. *In general networks the single source least cost path problem can be solved in time $O(n \cdot e)$.*

Proof: By virtue of the counters each node (except for maybe one) is selected at most n times in line (4). Whenever node v is chosen the time spent in lines (4) to (16) is $O(outdeg(v))$. Hence the total running time of the loop (3) to (17) is $O(n \cdot \sum_{v \in V} outdeg(v)) = O(n \cdot e)$. The cost of the statements outside the loop is clearly $O(n)$. Correctness remains to be shown.

Claim: *Assume $\mu(s, u) > -\infty$ for all $u \in V$. Let v be arbitrary. Then v is selected at most n times in line (4).*

Proof: Let U_i be set U when v is removed from U for the i-th time. Then U_i contains at least one element, say u_i, with $cost[u_i] = \mu(s, u_i)$ by Lemma 2c). Since U is organized as a queue u_i is deleted from U before v is deleted for the $(i+1)$-th time. Since u_i will never be added to U again (see proof of Lemma 2d)), we have $i \le n$. ▮▮

In Exercise 15 it is shown that the time bound may be reduced to $O(k_{max} \cdot e)$ where k_{max} is the maximal length (number of edges) of a least cost path from s to any $v \in V$. In Exercise 16 the algorithm above is related to dynamic programming. Alternative approaches to the single source least cost path problem are discussed in Exercises 18 and 19. A fast algorithm for planar graphs is described in Section 4.10.

Another improvement can be made for almost acyclic graphs. Let $G = (V, E)$ be a graph and let $V = V_1 \cup \cdots \cup V_k$ be the partition of V into strongly connected components. We order the s.c.c.'s such that $(v, w) \in E$, $v \in V_i$, $w \in V_j$ implies $i \le j$. Also we split the adjacency lists into two parts, the cyclic and the acyclic part. For each node $v \in V_i$, the cyclic part contains all edges (v, w) with $w \in V_i$, and the acyclic part contains all edges (v, w) with $w \in V_j$, $j > i$. We can now modify our algorithm as follows. There are k queues $UQ_1, \ldots, UQ_k$ one for each s.c.c.. In line (4) we always select the first element, say u, of the first (smallest index) non-empty queue. Then in line (8) we only step through the cyclic part of u's adjacency list. Once a queue UQ_i becomes empty we step through the acyclic parts of the adjacency lists of all nodes $v \in V_i$ and update the costs of the other endpoints. An argument similar to the one used in the proof of Theorem 5 shows that $v \in V_k$ is selected at most $|V_k|$ times from UQ_k (provided that $\mu(s, v) > -\infty$ for all v). Hence the running time is bounded by

$$O\left(e + \sum_{j=1}^{k} |V_j| \cdot |E_j|\right)$$

where (V_j, E_j), $1 \le j \le k$, are the s.c.c.'s of graph G. If the s.c.c.'s are small then this is a considerable improvement on Theorem 5. Also note that the modified algorithm will work in linear time on acyclic networks.

Theorem 7. *Let $N = (V, E, c)$ be a network and let (V_j, E_j), $1 \le j \le k$, be the strongly connected components of $G = (V, E)$. Then the single source least cost path problem can be solved in time $O(e + \sum_{j=1}^{k} |V_j| \cdot |E_j|)$.*

4.7.4. The All Pairs Problem

We now extend the solution of the previous section to a solution of the all pairs least cost path problem; an alternative solution can be found in Chapter V.

Let $N = (V, E, c)$ be a network. Suppose that we have a function $\alpha : V \to \mathbb{R}$ such that

$$\forall (u,v) \in E : \quad \alpha(u) + c(u,v) \geq \alpha(v).$$

Consider cost function $\bar{c} : E \to \mathbb{R}$ with

$$\bar{c}(u,v) = \alpha(u) + c(u,v) - \alpha(v)$$

for all $(u,v) \in E$. Then $\bar{c}$ is a non-negative cost function. Let $\bar{\mu}(x,y)$ be the cost of the least cost path from x to y with respect to cost function $\bar{c}$. There is a very simple relation between μ and $\bar{\mu}$.

Lemma 4. *Let μ and $\bar{\mu}$ be defined as above. Then $\bar{\mu}(x,y) = \mu(x,y) + \alpha(x) - \alpha(y)$.*

Proof: Let $p = (v_0, \ldots, v_k)$ be any path from $x = v_0$ to $y = v_k$. Then

$$\begin{aligned}
\bar{c}(p) &= \sum_{i=0}^{k-1} \bar{c}(v_i, v_{i+1}) \\
&= \sum_{i=0}^{k-1} (\alpha(v_i) + c(v_i, v_{i+1}) - \alpha(v_{i+1})) \\
&= \alpha(v_0) + \sum_{i=0}^{k-1} c(v_i, v_{i+1}) - \alpha(v_k) \\
&= c(p) + \alpha(x) - \alpha(y).
\end{aligned}$$

Since p is an arbitrary path from x to y we infer $\bar{\mu}(x,y) = \mu(x,y) + \alpha(x) - \alpha(y)$. ∎

Lemma 4 implies that we can reduce a general least cost path problem to a non-negative least cost path problem if we know a function α having the required properties. But solving one single source problem will give us (essentially) a function, namely $\mu(s,v)$ with the desired properties. There is only one problem we have to cope with: $\mu(s,v)$ might be infinite and the α's are required to be real. We will overcome this difficulty by augmenting the network as described below.

Theorem 8. *The all pairs least cost path problem can be solved in time*

$$O\left(n \cdot e \cdot \frac{\log n}{\max(1, \log(e/n))}\right).$$

Proof: Let $N = (V, E, c)$ be a network and let $s \in V$ be arbitrary. As a first step we will extend N to a network $N' = (V, E', c')$ by adding some edges, namely $E' = E \cup \{(s,v);\ v \in V, v \neq s\}$, and

$$c'(u,v) = \begin{cases} c(u,v) & \text{if } (u,v) \in E; \\ \textit{large} & \text{if } (u,v) \in E' - E, \end{cases}$$

where *large* $= \sum_{(u,v)\in E} |c(u,v)|$. Let $\mu(x,y)$ and $\mu'(x,y)$ be the cost of the least cost path from x to y in N and N' respectively. Then $\mu'(s,v) < +\infty$ for all $v \in V$ because of the augmentation. Also N' contains a cycle of negative cost iff N contains a cycle of negative cost. This can be seen as follows: If N' contains a cycle of negative cost, then N' contains a simple cycle of negative cost. Then each edge of E' is used at most once in this cycle. It cannot contain an edge of $E' - E$ because then the length of the cycle would be at least *large* $- \sum_{(u,v)\in E} |c(u,v)| \geq 0$. Hence N contains a cycle of negative cost.

Next we use the algorithm of Section 4.7.3 to find out whether N' (and hence N) has a cycle of negative cost and if not to determine $\mu(s,v)$ for all $v \in V$. In the first case the algorithm stops, in the second case we use $\alpha(v) = \mu(s,v)$ to transform the all pairs problem on a general network into a set of n single source problems on a non-negative network. Using the methods of 4.7.2 we obtain the time bound $O\left(e \cdot n + n \cdot e \cdot \log n / \max(1, \log(e/n))\right)$. ∎

4.8. Minimum Spanning Trees

Let $N = (V, E, c)$ be an **undirected network**, i.e., (V, E) is an undirected graph and c is symmetric ($c(v,w) = c(w,v)$ for all $(v,w) \in E$). A tree $A = (V,T)$ with $T \subseteq E$ and $|T| = n - 1$ is called a **spanning tree** of N. The cost of spanning tree A is $c(A) = \sum_{(v,w)\in T} c(v,w)$. It is a **minimum spanning tree** (or least cost spanning tree) if $c(A) \leq c(A')$ for all other spanning trees A'. Throughout this section we assume that (V, E) is connected. Thus $e \geq n - 1$.

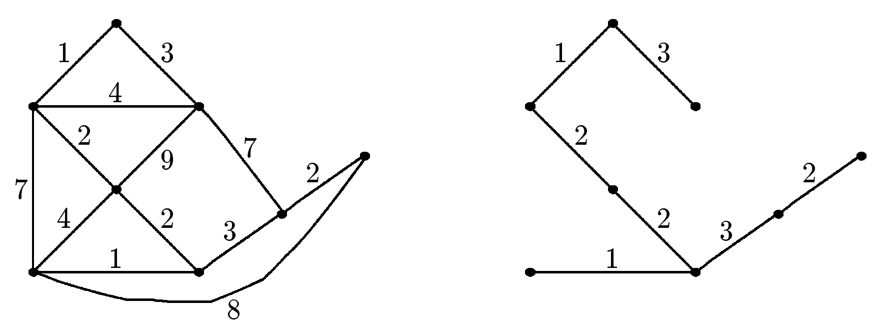


Figure 18. A network and one of its minimum spanning trees

Program 23 is a common skeleton for many algorithms for computing minimum spanning trees.

Lemma 1. *Program 23 computes a minimum spanning tree.*

Proof: We show by induction on $m = |T_1| + |T_2| + \ldots + |T_n|$ that there is a minimum spanning tree $A = (V,T)$ with $T_i \subseteq T$ for all i. If $m = 0$ then there is nothing

```
(1)   for all i ∈ V do V_i ← {i}; T_i ← ∅ od;
(2)   do n − 1 times
(3)       choose any non-empty V_i;
(4)       choose (v, w) ∈ E such that v ∈ V_i, w ∉ V_i and c(v, w) ≤ c(v', w')
          for all (v', w') ∈ E with v' ∈ V_i, w' ∉ V_i;
(5)       let j be such that w ∈ V_j;
(6)       V_i ← V_i ∪ V_j; V_j ← ∅;
(7)       T_i ← T_i ∪ T_j ∪ {(v, w)}; T_j ← ∅
(8)   od.
```

Program 23

to show. So let us turn to the induction step. By induction hypothesis there is a minimum spanning tree $A = (V, T)$ with $T_i \subseteq T$ for all i. Let $(v, w) \in E$ be the edge chosen in line (4). If $(v, w) \in T$ then we are done. If $(v, w) \notin T$ then $(V, T \cup \{(v, w)\})$ contains a cycle. Hence there must be an edge $(v', w') \in T$ such that $v' \in V_i$, $w' \notin V_i$. We have $c(v, w) \le c(v', w')$ by the choice of (v, w). Hence $T - \{(v', w')\} \cup \{(v, w)\}$ is also a minimum spanning tree. Finally, case $m = n - 1$ implies the correctness of the algorithm. ∎

Various details are to be filled in. What set V_i should we choose in line (3), how do we find (v, w) in line (4) and how do we represent sets V_i? Let us solve the latter problem first. We use the Union-Find data structure of Section III.8.3 to represent sets V_i. Then line (6) is a Union operation (and we execute $n - 1$ of them) and testing whether both endpoints of edge $(v, w) \in E$ belong to the same V_i corresponds to two Finds. Since this test has to be done at most once for every edge $(v, w) \in E$ the number of Finds is $O(e)$. Thus the total cost of handling sets V_i is $O(e \cdot \alpha(e, n))$ where α is defined in Section III.8.3.

The former questions are more difficult to solve. We discuss three strategies: considering edges in order of increasing weight, always growing component V_1 and growing components uniformly.

Theorem 1.

a) *Let $E = \{e_1, e_2, \ldots\}$ be sorted according to cost, i.e., $c(e_1) \le c(e_2) \le \cdots$. Then a minimum spanning tree can be constructed in time $O(e \cdot \alpha(e, n))$.*

b) *A minimum spanning tree can be constructed in time $O(e \log n)$.*

Proof: a) We replace lines (3) and (4) by

```
(3')   let (v, w) be the next edge on E;
(4')   while v and w belong to the same component
(4'')  do let (v, w) be the next edge on E od;
```

Correctness of this refinement follows immediately from Lemma 1. Also, the bound on the running time follows directly from the discussion above.

b) Follows from part a) and the fact that we can sort the set of edges in time $O(e \log e) = O(e \log n)$. $\blacksquare$

We show next that we can improve upon Theorem 1 for dense graphs.

Theorem 2. *A minimum spanning tree can be constructed in time*

$$O\left(e \cdot \frac{\log n}{\max(1, \log(e/n))}\right).$$

Proof: We always choose V_1 in line (3), i.e., we grow the spanning tree starting at node 1. In order to facilitate the selection of edge (v, w) in line (4) we maintain a priority queue PQ for set $\{(c(w), v, w);\ w \notin V_1\}$ ordered according to $c(w)$ where $c(w) = \min\{c(u, w);\ u \in V_1\}$ and v is such that $c(w) = c(v, w)$. Given this definition line (4) corresponds to operation Deletemin on priority queue PQ. Suppose that edge (v, x) is chosen in line (4). In line (6) we have to add point x to V_1 and we have to update priority queue PQ. More precisely, for every edge $(x, w) \in E$ with $w \notin V_1$ we have to check whether $c(x, w) < c(w)$ and if so we have to change element $(c(w),\ , w)$ of PQ to $(c(x, w), x, w)$. In order to do this efficiently we use an array $P[1..n]$ of pointers. Pointer $P[w]$ points to element $(c(w),\ , w)$ on PQ if $w \notin V_1$ and is nil otherwise. With the help of array $P[1..n]$ line (6) reduces to $O(deg(x))$ operations Demote* (cf. III.5.3.1) on priority queue PQ. Thus the cost of constructing a minimum spanning tree is the cost of n Deletemin, $O(n)$ *Insert* and $O(e)$ Demote* operations on PQ plus the time needed for initializing the priority queue. Initially, $PQ = \{(c(1, w), 1, w);\ w \neq 1 \text{ and } (1, w) \in E\}$ and hence the initialization corresponds to $deg(1) = O(n)$ Insert operations.

If we realize PQ as an unordered $(a, 2a)$-tree (cf. III.5.3.1) with $a = \max(2, e/n)$ then the cost of a Delete is $O(a \cdot \log n / \log a)$, of a Deletemin and an *Insert* operation $O(a \cdot \log n / \log a)$, and of a Demote* $O(\log n / \log a)$. Hence the total cost is $O(e \log n / \max(1, \log(e/n)))$. $\blacksquare$

Theorem 2 is most significant for dense graphs. If $e = n^{1+1/k}$ for $k \in \mathbb{N}$ then the running time is $O(k \cdot e)$. For sparse graphs, say $e = O(n)$, the running time is $O(n \log n)$. Can we do better for sparse graphs?

We end this section giving a brief description of an $O(e \log \log n)$ algorithm. This algorithm is based on two ideas, on a strategy for growing components uniformly and on a special purpose priority queue. Put sets $V_1, V_2, \ldots, V_n$ into a queue Q and replace lines (3) and (6) by

(3″) let V_i be the first element of queue Q;

and

(6″a) delete V_i and V_j from Q;
(6″b) $V_i \leftarrow V_i \cup V_j$; $V_j \leftarrow \emptyset$;
(6″c) add V_i to the end of Q;

The selection strategy described above selects components in a round-robin fashion. For the analysis we conceptually divide the algorithm into stages. Stages are defined as follows. We initially add a special marker to the end of Q and start stage 0. Whenever the special marker appears at the front of queue Q we finish a stage, move the marker to the end of the queue, and start the next stage.

Lemma 2.

a) *All sets selected at line* $(3'')$ *in stage* k *have size at least* 2^k *and all sets produced in line* $(6''b)$ *have size at least* 2^{k+1}.

b) *The number of stages is at most* $\log n$.

Proof:

a) We use induction on k. The claim is clearly true for $k = 0$. If V_i is chosen in stage $k > 0$ and combined with V_j then V_i and V_j are created in stage $k-1$ and hence have size at least 2^k each, by induction hypothesis. Thus $|V_i \cup V_j| \geq 2^{k+1}$.

b) The algorithm terminates when a set of size n is produced in line (6). Hence the maximal stage number k must satisfy $2^{k+1} \leq n$. ∎

Lemma 2 has an important consequence. Call a point v active during an iteration of loop (2) to (8) if v belongs to component V_i selected in line $(3'')$. Then any node v can be active at most once in a stage and hence can be active at most $\log n$ times by Lemma 2b). In other words, any fixed node v has to be considered at most $\log n$ times in line (4).

We can use this fact for deriving another $O(e \log n)$ algorithm as follows: In line (4) we consider all nodes $v \in V_i$ and determine the least cost edge (v, w) with $w \notin V_i$. This can certainly be done in time $O(\mathit{deg}(v))$. Since a node is active at most $\log n$ times the total cost of this algorithm is $O(\sum_v \mathit{deg}(v) \cdot \log n) = O(e \log n)$.

In order to obtain an $O(e \log \log n)$ algorithm we need two additional concepts: shrinking the graph and a special purpose priority queue. Suppose that we execute the algorithm above for $\log \log n$ stages; this will take $O(e \log \log n)$ time units and build up components of at least $2^{\log \log n} = \log n$ vertices each; let $U_1, \ldots, U_m$, $m \leq n/\log n$, be the components after stage $\log \log n$. Define network $N' = (V', E', c')$ as follows: $V' = \{1, \ldots, m\}$, $E' = \{(i,j);\ \exists v \in U_i, w \in U_j$ such that $(v,w) \in E\}$ and $c'(i,j) = \min\{c(v,w);\ v \in U_i, w \in U_j\}$. N' can be constructed from N in time $O(e)$; cf. Exercise 2. We still have to compute a minimum spanning tree of N'. For every node v of N' we divide the edges incident to v into $\mathit{deg}(v)/\log n$ groups of $\log n$ edges each. We sort each group according to *cost* within $O((\log n) \log \log n)$ time units per group. Thus the total preprocessing time is $O(e \log \log n)$.

In line (4) we proceed as follows. For every node $v \in V_i$ we inspect every group. For every group we inspect the edges in order of increasing cost and discard edges which do not lead outside V_i. When this process is finished we are left with $\lceil \mathit{deg}(v)/\log n \rceil$ edges leading from v to nodes outside V_i. We can certainly find the one of minimal cost in time $O(\lceil \mathit{deg}(v)/\log n \rceil)$. Thus the cost of finding minimum cost edges going out of v is $O(1 + \mathit{deg}(v)/\log n +$ number of discarded edges) per stage. Since every edge is discarded at most once, since there are only $\log n$ stages and since N' has only $n/\log n$ nodes the total cost is $O(n+e)$. We have

Theorem 3. *A minimum cost spanning tree of an undirected network can be computed in time $O(e \log\log n)$.*

Proof: By the discussion above. ∎

We finally come to an improvement for planar networks. In a planar graph we always have $e \le 3n - 6$, cf. 4.10, Lemma 2. Suppose that we apply the shrinking process after every stage. Let N_i be the network after stage i. Then N_i is planar and hence $e_i \le 3n_i - 6$ where e_i (n_i) is the number of edges (nodes) of network N_i. Also stage $i+1$ takes $O(e_i)$ time units and N_{i+1} can be constructed from N_i in time $O(e_i)$. Thus the total cost is

$$\sum_{i=0}^{\log n - 1} O(e_i) = O\left(\sum_{i=0}^{\log n - 1} n_i\right) = O\left(\sum_{i=0}^{\log n - 1} n/2^i\right) = O(n).$$

Theorem 4. *Let $N = (V, E, c)$ be a planar undirected network. Then a minimum cost spanning tree can be computed in time $O(n)$.* ∎

4.9. Maximum Network Flow and Applications

4.9.1. Algorithms for Maximum Network Flow

A directed network $N = (V, E, c)$ consists of a directed graph $G = (V, E)$ and a capacity function $c : E \to \mathbb{R}^+$. Let $s, t \in V$ be two designated vertices, the source s and the sink t. A function $f : E \to \mathbb{R}$ is a **legal (s,t)-flow function** (or legal flow for short) if it satisfies

a) the capacity constraints, i.e., $0 \le f(e) \le c(e)$ for all $e \in E$;
b) the conservation laws, i.e., $\sum_{e \in in(v)} f(e) = \sum_{e \in out(v)} f(e)$ for all nodes $v \in V - \{s,t\}$. Here $in(v)$ resp. $out(v)$ is the set of edges entering resp. leaving v.

If $f : E \to \mathbb{R}$ is a legal flow function then

$$val(f) = \sum_{e \in out(s)} f(e) - \sum_{e \in in(s)} f(e)$$

is the flow value of f. The **maximum network flow problem** is to compute a legal flow function with maximum flow value. In this section we will describe two algorithms for achieving this goal. On the way we will derive a powerful combinatorial result: the max flow-min cut theorem.

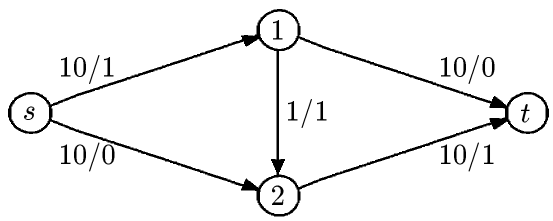


Figure 19. Graph with capacity/flow

Definition: An **(s,t)-cut** is a partition S, T of V, i.e., $V = S \cup T$, $S \cap T = \emptyset$, such that $s \in S$, $t \in T$. The capacity of cut (S,T) is given by

$$c(S,T) = \sum_{e \in E \cap (S \times T)} c(e).$$ ∎

The capacity of a cut (S,T) is thus the total capacity of all edges going from S to T. The easy direction of the min cut-max flow theorem is given by

Lemma 1. *Let f be a legal flow and let (S,T) be an (s,t)-cut. Then*

$$val(f) \leq c(S,T).$$

Proof: We have

$$\begin{aligned} val(f) &= \sum_{e \in out(s)} f(e) - \sum_{e \in in(s)} f(e) \\ &= \sum_{v \in S} \left[\sum_{e \in out(v)} f(e) - \sum_{e \in in(v)} f(e) \right] \\ &= \sum_{e \in E \cap (S \times T)} f(e) - \sum_{e \in E \cap (T \times S)} f(e) \\ &\leq c(S,T). \end{aligned}$$

Here the second equality follows since the conservation law holds for all $v \in S - \{s\}$. The third equality follows since every edge $e = (u,v) \in E \cap (S \times S)$ is counted twice, positively since $e \in out(u)$ and negatively since $e \in in(v)$ for some u and v. Finally, the inequality follows since $f(e) \leq c(e)$ for all $e \in E \cap (S \times T)$ and $f(e) \geq 0$ for all $e \in E \cap (T \times S)$. ∎

Most algorithms for maximum network flow work iteratively and are based on the concept of an **augmenting path**, i.e., they start with any legal initial flow, say the flow function which is zero everywhere, and then use augmenting paths to increase the flow. In the example of Figure 19 we use the edge label a/b to denote capacity a and flow b.

There are three augmenting paths in this example: $s, 1, t$ with bottleneck value 9; $s, 2, t$ with bottleneck value 9; $s, 2, 1, t$ with bottleneck value 1. Paths $s, 1, t$ and $s, 2, t$ can be used to increase the flow value by 9 in an obvious way. The use of path $s, 2, 1, t$ is more subtle. We might send one additional unit from s to 2. This relieves us from the obligation to push one unit from 1 to 2 and we can therefore send this unit directly from 1 to t. Augmentation along path $s, 2, 1, t$ changes the flow as shown in Figure 20.

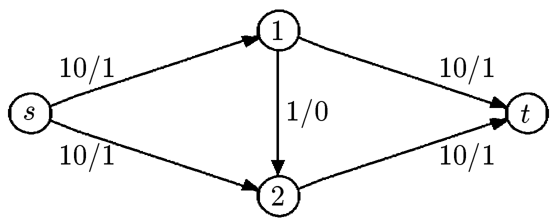


Figure 20. After augmentation by 1 along path $s, 2, 1, t$

All shortest (= minimum cardinality) augmenting paths are captured in the **layered network** LN with respect to the legal flow function f which is defined as follows. Let

$$E_1 = \{(v, w);\ (v, w) \in E \text{ and } f(e) < c(e)\}$$

and let

$$E_2 = \{(w, v);\ (v, w) \in E \text{ and } f(e) > 0\},$$

i.e., edges in E_1 can be used to push flow forward and the edges in E_2 can be used to push flow backward. If $e = (v, w) \in E$ then we use e_1 to denote edge $(v, w) \in E_1$ (if it is there) and e_2 to denote edge $(w, v) \in E_2$ (if it is there). Also $\bar{c} : E_1 \cup E_2 \to \mathbb{R}^+$ is given by

$$\bar{c}(e_1) = c(e) - f(e) \quad \text{for } e_1 \in E_1$$

and

$$\bar{c}(e_2) = f(e) \quad \text{for } e_2 \in E_2.$$

Note that $E_1 \cup E_2$ is a multiset because if $e = (v, w) \in E$ and $e' = (w, v) \in E$ then $e_1, e_2, e'_1, e'_2 \in E_1 \cup E_2$ is possible. For the example of Figure 19 we obtain the graph of Figure 21. Edges in E_1 are drawn solid and edges in E_2 are drawn dashed.

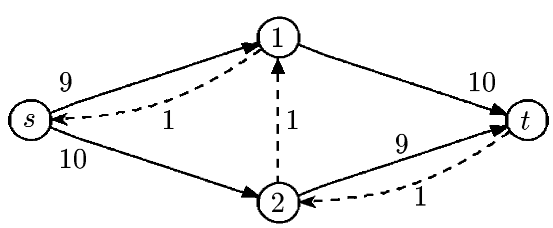


Figure 21. E_1, E_2 and $\bar{c}$ for original graph

Next, let $V_0 = \{s\}$ and

$$V_{i+1} = \{w \in V - (V_0 \cup \cdots \cup V_i);\ \exists v \in V_i : (v, w) \in E_1 \cup E_2\}$$

for $i \geq 0$, and let $\overline{V} = \bigcup_{i \geq 0} V_i$. Then $LN = (\overline{V}, (E_1 \cup E_2) \cap \bigcup_{i \geq 0}(V_i \times V_{i+1}), \bar{c})$ is the layered network with respect to flow function f. In our example we obtain Figure 22.

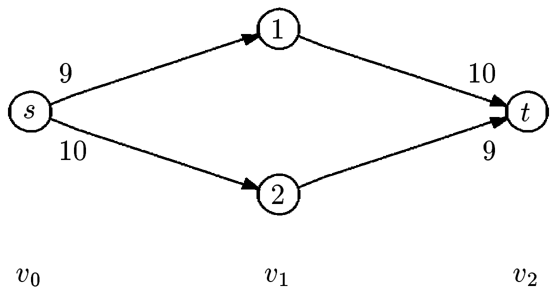


Figure 22. LN for original graph

Any path from s to t in the layered network is an augmenting path and can be used to increase the flow. More generally, we have

Lemma 2. *Let f be a legal (s,t)-flow in network N and let $LN = (\overline{V}, \overline{E}, \bar{c})$ be the layered network with respect to f.*

a) *f is a maximum flow iff $t \notin \overline{V}$.*

b) *Let $\bar{f}$ be a legal (s,t)-flow in LN. Then $f' : E \to \mathbb{R}$ with*

$$f'(e) = f(e) + \bar{f}(e_1) - \bar{f}(e_2)$$

is a legal flow in N with flow value $val(f) + val(\bar{f})$. Here $\bar{f}(e_i)$ is defined to be zero if $e_i \notin \overline{E}$.

Proof: b) We have to show that f_1 satisfies the capacity constraints and the conservation law. Let $e \in E$ be arbitrary. Then

$$\begin{aligned}
0 &\leq f(e) - \bar{f}(e_2) && (\text{since } f(e) = \bar{c}(e_2) \geq \bar{f}(e_2)) \\
&\leq f'(e) && (\text{since } \bar{f}(e_1) \geq 0) \\
&\leq f(e) + \bar{f}(e_1) && (\text{since } \bar{f}(e_2) \geq 0) \\
&\leq c(e) && (\text{since } \bar{f}(e_1) \leq \bar{c}(e_1) = c(e) - f(e)),
\end{aligned}$$

i.e., f' satisfies the capacity constraints. Next, let $v \in V - \{s,t\}$ be arbitrary. Then

$$\begin{aligned}
&\sum_{e \in out(v)} f'(e) - \sum_{e \in in(v)} f'(e) \\
&= \sum_{e \in out(v)} f(e) - \sum_{e \in in(v)} f(e) + \left[\sum_{e \in out(v)} \bar{f}(e_1) + \sum_{e \in in(v)} \bar{f}(e_2) \right] \\
&\quad - \left[\sum_{e \in in(v)} \bar{f}(e_1) + \sum_{e \in out(v)} \bar{f}(e_2) \right] \\
&= 0 + 0
\end{aligned}$$

since f and $\bar{f}$ satisfy the conservation laws. Note that $e_2 \in E_2$ emanates from node v if $e \in in(v)$ and that e_2 ends in node v if $e \in out(v)$. Finally, the flow value of f_1 is clearly $val(f) + val(\bar{f})$.

a) "$\Rightarrow$": If $t \in \overline{V}$ then there is a path from s to t in the layered network. Let p be any such path and let $\epsilon > 0$ be the minimal capacity of any edge of p. Then there is clearly a flow of value ϵ in LN, namely $\bar{f}(e) = \epsilon$ for all edges e of p and $\bar{f}(e) = 0$ otherwise. Hence f is not maximum by part b).

"$\Leftarrow$": Let $S = \overline{V}$ and let $T = V - S$. Then $s \in S$ and $t \in T$, i.e., (S,T) is an (s,t)-cut. Furthermore, $(E_1 \cup E_2) \cap (S \times T) = \emptyset$ since no node of T is added to the layered network. Thus $f(e) = c(e)$ for $e \in S \times T$ and $f(e) = 0$ for $e \in T \times S$. We conclude that the inequality in the proof of Lemma 1 turns into an equality and hence $val(f) = c(S,T)$. Since $val(g) \le c(S,T)$ for any legal flow g we infer that f is a flow with maximum flow value. ∎

It seems that we have not got very far. In order to increase the flow through network N we have to find a (large) legal flow through layered network LN. Fortunately, an approximation to the maximum flow in LN is good enough. More precisely, it suffices to compute a blocking flow in LN.

Definition: A legal flow $\bar{f}$ in layered network LN is **blocking** if for every path $s = v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} v_2 \xrightarrow{e_3} \ldots \xrightarrow{e_k} v_k = t$ from s to t at least one of the edges is saturated, i.e., $\bar{f}(e_i) = \bar{c}(e_i)$ for at least one i, $1 \le i \le k$. ∎

Now we outline the basic maximum flow algorithm in Program 24.

Two questions arise: How can we find a blocking flow in a layered network and how many iterations are required? We turn to the second question first.

Definition: Let f be a (non-maximum) legal flow in N and let LN be the layered network for f. Then k, where $t \in V_k$, is called the **depth** of LN. ∎

```
let f(e) ← 0 for all e ∈ E;
construct layered network LN = (V̄, Ē, c̄) from f;
while t ∈ V̄
do find a blocking flow f̄ in LN;
   update f by f̄ as described in Lemma 2b);
   construct layered network LN from f
od.
```

Program 24

Lemma 3. *Let k_i be the depth of the layered network used in the i-th iteration, $i = 1, 2, \ldots$. Then $k_i > k_{i-1}$ for $i \geq 2$.*

Proof: Let LN_i be the layered network used in the i-th iteration. In LN_i there is a path p of length k_i from s to t.

$$s = v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} v_2 \xrightarrow{e_3} \ldots \xrightarrow{e_{k_i-1}} v_{k_i-1} \xrightarrow{e_{k_i}} v_{k_i} = t$$

For $0 \leq j \leq k_i$, let d_j be the length (= number of edges) of the shortest path from s to v_j in LN_{i-1}, i.e., v_j belongs to the d_j's layer of LN_{i-1}. If v_j is not a node of LN_{i-1} then $d_j = \infty$.

Claim: *For all $i \geq 2$ holds:*

a) *If there is an edge from v_{j-1} to v_j in LN_{i-1} then $d_j = d_{j-1} + 1$.*

b) *If there is no edge from v_{j-1} to v_j in LN_{i-1} then $d_j \leq d_{j-1}$.*

c) *$k_{i-1} < k_i$.*

Proof: a) Obvious since network LN_{i-1} is layered, i.e., if v_{j-1} belongs to layer d_{j-1} and there is an edge from v_{j-1} to v_j in LN_{i-1} then v_j belongs to layer $d_{j-1}+1$.

b) Let us assume for the sake of contradiction that $d_j \geq d_{j-1} + 1$. Let f_{i-1} resp. f_i be the flow in network N which gives rise to the construction of layered network LN_{i-1} resp. LN_i. Then $f_{i-1}(v_{j-1}, v_j) = c(v_{j-1}, v_j) > f_i(v_{j-1}, v_j)$ if $(v_{j-1}, v_j) \in E$ or $f_{i-1}(v_j, v_{j-1}) = 0 < f_i(v_j, v_{j-1})$ if $(v_j, v_{j-1}) \in E$ because $(v_{j-1}, v_j) \notin \overline{E}_{i-1}$ and $(v_{j-1}, v_j) \in \overline{E}_i$. In either case we conclude that $(v_j, v_{j-1}) \in \overline{E}_{i-1}$ and hence $d_{j-1} = d_j + 1$. Thus $d_j = d_{j-1} - 1 \leq d_{j-1}$, contradiction.

c) Since $v_0 = s$ and hence $d_0 = 0$ we conclude from parts a) and b) that $d_j \leq j$. Also, $d_j = j$ for all $j \leq k_i$ is only possible if edge e_j from v_{j-1} to v_j is present in LN_{i-1} for all $j \geq 1$. Thus $d_j = j$ for all $j \leq k_i$ implies that there is some path p from s to t which exists in LN_{i-1} and LN_i. This contradicts the fact that f_i is obtained from f_{i-1} by "adding" a blocking flow with respect to layered network LN_{i-1}. We conclude that $d_j < j$ for some $j \leq k_i$ and hence $d_{k_i} < k_i$ by parts a) and b). We can now complete the proof of the claim and the lemma by observing that $k_{i-1} = d_{k_i}$ by definition. ▮▮

Corollary 1. *The number of iterations is at most n.*

Proof: Let k_i be the depth of the layered network used in the i-th iteration, $i \geq 1$. Then $k_1 \geq 1$ since $s \neq t$, $k_{i-1} < k_i$ by Lemma 3 and $k_i \leq n$ for all i. Hence the number of iterations is at most n. ∎

Better bounds on the number of iterations can be derived for restricted networks. In particular, we will derive considerably smaller bounds for (0,1)-networks in Section 4.9.2. We will next describe two algorithms for constructing blocking flows in layered networks, first an $O(n^2)$ algorithm and then an $O(e \cdot (\log n)^2)$ algorithm.

Let $LN = (V, E, c)$ be a layered network, i.e., $V = \bigcup_{0 \leq i \leq k} V_i$ for some k, $E \subseteq \bigcup_{0 \leq i < k}(V_i \times V_{i+1})$, $V_0 = \{s\}$ and $c : E \to \mathbb{R}^+$. We may assume w.l.o.g., i.e., the condition can be established in linear time by simple graph exploration, that every node $v \in V$ is reachable from s and that t can be reached from all nodes. In particular, $V_k = \{t\}$ in this case. The $O(n^2)$ algorithm is based on the concept of the potential of a node. Let f be a legal flow and let $v \in V$. The **potential** of node v with respect to flow f is given by

$$PO(v) = \min\left[\sum_{e \in out(v)} c(e) - f(e),\ \sum_{e \in in(v)} c(e) - f(e)\right],$$

i.e., the potential of node v is the maximum possible increase in flow through node v. Also

$$PO^* = \min\{PO(v);\ v \in V\}$$

is the minimal potential of any node in V. It is now quite simple to increase the flow by PO^*. Let v be any node with $PO(v) = PO^*$. Starting at node v we forward PO^* additional units from node v through higher layers to node t and we suck PO^* additional units flow into node v through lower layers. Forwarding the flow is done as follows. We proceed layer by layer, starting at the layer containing v. When we consider layer V_l we have determined a subset $S_l \subseteq V_l$ of nodes which holds an additional amount of PO^* units of flow, i.e., $PO^* = \sum_{x \in S_l} S(x)$ where $S(x)$ is the excess of flow available in node $x \in S_l$. We consider the nodes in S_l in turn and push their excess of flow into the next layer. Since $PO^* \leq PO(w)$ for all w no node can receive more flow than it can handle. We continue in this way until we have pushed the additional flow all the way to t. Similarly, we work our way back from node v towards source s and suck PO^* additional units of flow into the network.

In this way we increase the flow by PO^* units. After having done so, we simplify the network by deleting saturated edges and useless nodes, i.e., nodes which are not connected to either s or t, and edges incident to useless nodes. Note that at least node v will be deleted from the network. (Remark: It would simplify the algorithm if we forwarded additional flow starting at s. Correctness would not be impeded, however efficiency might suffer.) If the network is not empty after the simplification we repeat the process. Since the simplification deletes at least one node from the network the number of iterations is clearly $O(n)$.

We will next describe the algorithm in more detail. We assume that for every node $v \in V$ the set of ingoing and the set of outgoing edges are ordered in some way. Also set $S_l \subseteq V_l$ is realized as a bit vector and as a linear list. In this way we can test $v \in S_l$, add an element to S_l and delete some element from S_l in time $O(1)$. In addition, we store for every node $x \in X$ the excess (deficit) of flow available at node x in $S[x]$. The procedure *forward* of Program 25 is fundamental to the algorithm; it forwards the flow from node x into the next layer. There is a symmetric procedure *suck* which sucks the flow into node x from the previous layer.

---

(1) **procedure** *forward*(x, S, h);
co x is a node in layer V_h and there are S units of additional flow available in x. These S units are pushed into nodes in layer V_{h+1} **oc**
(2) **while** $S > 0$
(3) **do** let $e = (x, y)$ be the first edge out of x;
(4) $\quad$ *delta* $\leftarrow \min(S, c(e) - f(e))$;
(5) $\quad$ increase flow along e by *delta*, add y to S_{h+1} (if it is not already there), increase $S[y]$ by *delta*, and decrease $c(e)$ by *delta*;
(6) $\quad$ $S \leftarrow S -$ *delta*;
(7) $\quad$ **if** $c(e) = 0$ **then** delete e from the graph **fi**
(8) **od**;
(9) remove x from S_h and set $S[x]$ to zero;
(10) **if** $(out(x) = \emptyset$ **and** $x \neq t)$ **or** $(in(x) = \emptyset$ **and** $x \neq s)$
(11) **then** add x to set *del* **fi**
(12) **end**.

Program 25

In set *del* we collect all nodes which have to be deleted from the network because either they cannot be reached from s or t cannot be reached from them. The running time of a call of *forward* is $O(1+\#$ edges deleted in line (7)$)$, because at each execution of the loop body (except maybe the last) an edge is deleted and since the cost outside the loop is clearly $O(1)$. The complete algorithm for computing a blocking flow is given by Program 26.

Procedure *simplify*(*del*) removes all nodes (and edges incident to them) in *del* from the network. Also if some other node z loses its last ingoing (outgoing) edge during this process then z is also deleted. It is easy to see that *simplify* can be implemented to run in time proportional to the number of nodes and edges removed from the graph; an algorithm similar to Program 26 used for topological sorting will do. The details are left to the reader (Exercise 27).

Theorem 1. *Let LN be a layered network. Then a blocking flow can be computed in time* $O(n^2)$.

Proof: Correctness of Program 26 follows from the fact that nodes (edges) are removed only if all paths from s to t through that node (edge) are blocked.

(1) **for** all $x \in V$ **do** $S[x] \leftarrow 0$ **od**;
(2) **for** all l, $0 \le l \le k$, **do** $S_l \leftarrow \emptyset$ **od**;
(3) $del \leftarrow \emptyset$;
(4) **while** LN is not empty
(5) **do** compute $PO[v]$ for all $v \in V$, let $PO^* = \min\{PO[v];\ v \in V\}$ and
let $v \in V_l$ be such that $PO^* = PO[v]$;
(6) $S[v] \leftarrow PO^*$; $S_l \leftarrow \{v\}$;
(7) **for** h **from** l **to** $k-1$
(8) **do for** all $x \in S_h$ **do** *forward*$(x, S[x], h)$ **od od**;
(9) $S[v] \leftarrow PO^*$; $S_l \leftarrow \{v\}$;
(10) **for** h **from** l **step** -1 **to** 1
(11) **do for** all $x \in S_h$ **do** *suck*$(x, S[x], h)$ **od od**;
(12) *simplify*(del)
(13) **od**.

Program 26

The cost of lines (1) to (3) is clearly $O(n)$. Also, loop (4) to (13) is executed $O(n)$ times since at least one node, namely v, is removed from the graph in line (12). The cost of an execution of the loop body outside the calls of *forward*, *suck* and *simplify* is clearly $O(n)$ and hence $O(n^2)$ if summed over all $O(n)$ iterations. Since the cost of a call of *forward* (*suck*) is $O(1 + \#$ deleted edges$)$, since *forward* (*suck*) is called at most once for each node during an execution of the loop body and since every edge is deleted at most once the total cost of all calls of *forward* (*suck*) is $O(n^2 + e) = O(n^2)$. Finally, the total cost of all calls of *simplify* is $O(e)$. ∎

The algorithm can be made to run faster in (0,1)-networks, i.e., networks where $c(e) = 1$ for all $e \in E$. Exercise 28 describes an implementation with running time $O(e)$. We will later describe a simpler $O(e)$ algorithm for computing blocking flows in (0,1)-networks.

Theorem 2. *Let $N = (V, E, c)$, $s, t \in V$, be a network. Then a maximum flow from s to t can be computed in time $O(n^3)$.*

Proof: A maximum flow can be computed by $O(n)$ applications of the blocking flow algorithm to layered networks. The construction of the layered network and the computation of a blocking flow takes time $O(n^2)$. The time bound follows. ∎

It is now easy to derive the min-cut max-flow theorem.

Theorem 3. *Let $N = (V, E, c)$, $s, t \in V$, be a network. Let fmax be the maximum flow value of any legal (s,t)-flow function and let cmin be the minimal capacity of all (s,t)-cuts. Then*

$$fmax = cmin.$$

Proof: Note first that $cmin$ exists because there is only a finite number of (s,t)-cuts. Also, $fmax$ exists because we have an $O(n^3)$ algorithm for computing a maximum flow from s to t. $fmax = cmin$ remains to be shown. We have $fmax \leq cmin$ by Lemma 1. Finally, let f be a flow function with $val(f) = fmax$. If we construct the layered network with respect to f then t is not added to the network. The proof of Lemma 2a) shows how to construct an (s,t)-cut (S,T) such that $fmax = val(f) = c(S,T)$. Since $c(S,T) \geq cmin$, this proves $fmax \geq cmin$. ∎

Our second algorithm for computing blocking flows is based on *dfs* and is particularly well suited for sparse networks, i.e., $e \ll n^2$. The basic idea is quite simple. Starting at s we construct a path by always taking the first edge out of every node until we either reach t or reach a dead-end v, i.e., a node v with $out(v) = \emptyset$ and $v \neq t$. In the second case we back up one node, delete all edges leading into v from the graph and continue. In the first case we compute the bottleneck capacity ϵ of the path, i.e., the minimal capacity of any edge on the path, increase the flow along the path by ϵ, decrease the capacities by ϵ, and delete all saturated edges from the graph. Having done so, we construct the next path starting at node s.

Theorem 4. *The algorithm above constructs a blocking flow in a layered network in time $O(e \cdot n)$. In a (0,1)-network it runs in time $O(e)$.*

Proof: Correctness is obvious. The bound on the running time is derived as follows. As before k denotes the depth of the layered network. Observe first, that a path from s to t is constructed in time $O(k + \#$ of edges found to be ending in dead-ends) and that at least one edge on the path is saturated by increasing the flow. Hence at most $O(e)$ paths are constructed for a total cost of $O(k \cdot e + e) = O(e \cdot n)$.

One additional observation is needed for (0,1)-networks. In (0,1)-networks *all* edges on the constructed path are saturated and hence the cost of constructing a path from s to t is proportional to the number of deleted edges. The claimed time bound follows. ∎

We will next describe an improved implementation of the algorithm above which reduces the time bound to $O(e \cdot (\log n)^2)$. In the algorithm above, whenever we succeed in constructing a path from s to t we saturate edges and then forget about the constructed path. A more economical way to proceed is to keep the remnants of the path as path fragments (PF's). In the example of Figure 23 we split the path into three PF's p, pf_1 and pf_2.

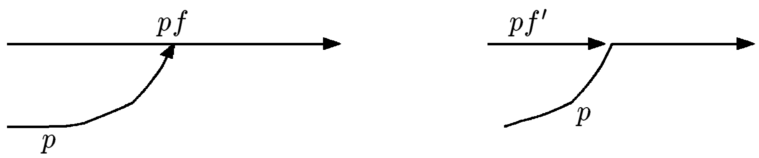


Figure 23. Splitting a path into three path fragments

We will always use p to denote the path fragment starting in s. We will maintain the invariant that at most one PF goes through every node v, i.e., that there is at most one path fragment pf such that v is a node of pf but not the last node of pf. In other words the PF's form a forest with edges directed towards the roots. We can now start to construct a new path from s to t starting at the last vertex $last(p)$ of PF p.

There are four ways of changing path p. If there is a path fragment which goes through $last(p)$, say pf, then we split pf at $last(p)$ and concatenate one of the parts to p. If $last(p)$ is the first vertex of pf then the splitting is trivial. See Figure 24.

Figure 24. The first two ways of changing path p

If there is an unblocked edge out of $last(p)$ then we add this edge to p. If $last(p)$ is a dead-end, i.e., there is neither a PF going through $last(p)$ nor an unblocked edge leaving $last(p)$, then we shrink p by deleting its last edge. Finally, if $last(p) = t$ then we saturate some of p's edges and split p into path fragments. The details are as described in Program 27.

```
(1)   p ← path consisting of s only;
(2)   while s is not a dead-end
(3)   do extend p by adding an unblocked edge out of last(p);
(4)       while a PF pf goes through last(p)
(5)       do split pf at last(p) into pf′ and pf″;
              co pf′ ends in last(p), pf″ starts in last(p) oc
              compute the capacities of pf′ and pf″;
(6)           concatenate pf″ to the end of p and update p's capacity
(7)       od;
(8)       if last(p) = t
(9)           then increase the flow along p by the capacity of p, split p into PF's
                   by deleting all saturated edges, compute the capacities
                   of the fragments and let p be the fragment starting in s
(10)      fi;
(11)      while last(p) is a dead-end and s ≠ last(p)
(12)      do delete the last edge from p and update p's capacity
(13)      od
(14)  od.
```

Program 27

There are two points which we have to deal with now. How do we maintain the invariant and can we always execute line (3)?

Lemma 4. *The following holds at all times during the execution of Program 27:*

a) *For every node v there is at most one path fragment going through v.*

b) *Whenever line (3) has to be executed there is an unblocked edge out of $last(p)$ and no PF goes through $last(p)$.*

Proof: (By induction on the number of steps executed.) Claims a) and b) are certainly true prior to the first execution of the loop body. Suppose now that a) and b) hold prior to execution of line (3). We will show that a) and b) hold at the end of the loop body. Since b) holds line (3) can be executed and execution of line (3) does not impede the truth of part a). Nor does the execution of lines (4) to (7). Before executing line (8) we know that no PF goes through $last(p)$. We claim that this is also true after executing lines (8) to (10). The claim is obvious if $last(p) \neq t$. If $last(p) = t$ then we reset p to an initial segment p' of p. Since the edge on p which emanates from $last(p')$ is saturated in (9) and since a) holds we conclude that no PF goes through $last(p')$. Thus a) holds prior to execution of line (11) and no PF goes through $last(p)$ at this point. Execution of lines (11) to (13) certainly does not affect a). Also these lines ensure that no PF goes through $last(p)$ (this fact is an invariant of line (12) because of a)) and that either $last(p) = s$ or that there is an unblocked edge out of $last(p)$. Thus b) and a) hold prior to the next execution of line (3) because of the test in line (2). ∎

Lemma 4 implies the correctness of Program 27. Let us turn to efficiency next. We need to discuss two points: how to represent path fragments so that the various operations on them can be done fast and how to derive bounds on the number of executions of the various statements.

Path fragments are stored as balanced trees. More precisely, we store the edges of a PF in the leaves of a (2,4)-tree in the natural order: Then every vertex z of the tree represents a path $pf(z)$ in the network, namely the path comprised of the edges stored in the subtree rooted at z. We store two informations about path $pf(z)$ in vertex z: $f(z)$ (flow) and $c(z)$ (capacity). The flow field $f(z)$ indicates that $f(z)$ units of flow have been pushed through $pf(z)$ without distributing these units over the subpaths. Thus, if e is an edge of the network, the flow through e is given by $\sum f(z)$ where the summation is over all vertices z on the path from the leaf representing edge e (note that this leaf is uniquely defined by Lemma 4a)) to the root of the tree representing the path fragment containing e. Field $c(z)$ is the minimal residual capacity (= capacity − flow) of any edge in $pf(z)$ ignoring the flow associated with proper ancestors of z. Thus

$$c(z) = \min_{e \in pf(z)} \left[c(e) - \sum_{y \in ver(e,z)} f(y) \right]$$

where $ver(e,z)$ is the set of vertices of the tree path from the leaf representing e to z (the leaf representing e and the vertex z are included). In particular, if z is the root of a tree then $c(z)$ is the residual capacity of $pf(z)$, i.e., $c(z)$ and no more additional units of flow can be pushed through $pf(z)$.

Lemma 5.

a) *If pf_1 and pf_2 are path fragments with $last(pf_1) = first(pf_2)$ then pf_1 and pf_2 can be concatenated in time $O(\log n)$.*

b) *Let pf be a PF represented as a balanced tree and let v be a node of pf. Then pf can be split at v in time $O(\log n)$. Also, if the residual capacity of pf is zero then a saturated edge of pf can be located in time $O(\log n)$.*

Proof: For both parts we need to push flow information into trees. If z is a vertex with sons z_i, $i = 1, 2, \ldots$, then

$$(f(z_i), c(z_i)) \leftarrow (f(z_i) + f(z), c(z_i) - f(z));$$
$$(f(z), c(z)) \leftarrow (0, c(z))$$

is a consistent change of the information fields associated with vertices $z, z_1, z_2, \ldots$. Also, it pushes flow from vertex z into the subpaths represented by vertices z_i, $i = 1, 2, \ldots$.

a) Let T_i of height h_i be the tree representing pf_i, $i = 1, 2$. Assume w.l.o.g. that $h_1 \le h_2$. Then we concatenate T_1 and T_2 by first pushing the flow down the left spine of T_2 for $h_2 - h_1 + 1$ levels and then concatenating T_1 and T_2 as described in Section III.5. Note that the flow and capacity field of the vertices affected by the operation are easily computed. More precisely, the flow field is set to zero and the capacity field is set to the minimum residual capacity of the sons. This shows that T_1 and T_2 can be concatenated in time $O(|h_2 - h_1| + 1)$.

b) Let T represent path fragment pf and let v be a node in pf. Note first, that v corresponds to a "gap" between two leaves of T in a natural way. Let e be the edge (leaf) following v in pf. We prepare the splitting by tracing the tree path from e to the root of T and then push the flow down this path. This changes the flow field of all vertices on the tree path to zero and therefore they can be safely removed. Splitting is completed by a sequence of concatenations as in ordinary (2,4)-trees.

Finally, we show how to find a saturated edge if the residual capacity of pf is zero. Push the flow from the root z of T into its sons. Then $0 = c(z) = \min(c(z_i))$ where z_i ranges over the sons of z. Therefore one of the sons has a zero capacity field. If we continue in this way we find a saturated edge in time $O(\log n)$. ∎

Lemma 6. *A single execution of lines (3), (5), (6) and (12) takes time $O(\log n)$. A single execution of line (9) takes time $O(d \log n)$ where d is the number of edges deleted in line (9).*

Proof: Follows immediately from Lemma 5. Note that line (3) can be visualized as constructing a path fragment consisting of a single edge and concatenating it with p. ∎

Now we are (almost) able to determine the running time of the improved algorithm. Note first that the total time spent outside lines (4) to (7) is $O(e \log n)$ because the number of executions of the loop body is at most e, because the total number of edges deleted in lines (9) and (12) is at most e, and because the cost of handling an edge is $O(\log n)$ by Lemma 6. It remains to bound the costs arising in lines (4) to (7), i.e., we need to bound the number of executions of lines (5) and (6). Call this number K. We show $K = O(e \log n)$ in a two step process. As a first step we rephrase the problem of bounding K as a game problem (which bears great resemblance to the Union-Find Problem studied in Section III.8.3) and as a second step we derive a bound on the number of moves in the game. The argument will be similar to the one used in Section III.8.3.

For step one we conceptually assign non-negative integers to path fragments as follows. To path fragment p (starting at s) no number is ever assigned. When p is split in line (9) into path fragments p, pf_1, pf_2, $\ldots$, pf_k (in this order from s to t) then we assign integer $L + i$ to pf_i, $1 \le i \le k$, where L is the largest integer given to a PF prior to that point. Also if we split PF pf into pf' and pf'' and concatenate pf'' to p, then pf' inherits the number of pf provided that pf' is non-trivial. We use $num(pf)$ to denote the number assigned to PF pf. We clearly have $1 \le num(pf) \le e$ for all path fragments pf since new numbers are assigned only in line (9) and assigning a new number corresponds to deleting an edge. We need one more property of path fragment numbers. If pf_1 and pf_2 are PF's then pf_1 **points to** pf_2 if pf_2 goes through $last(pf_1)$. We have

Lemma 7. *At all times during the execution holds:*

a) *If pf is a PF then pf points to at most one other PF.*

b) *If pf_1 points to pf_2 and $pf_2 \ne p$ then $num(pf_1) < num(pf_2)$.*

Proof: a) Follows directly from Lemma 4a) since there is at most one path fragment going through $last(pf)$ by that lemma.

b) (By induction on execution time.) New path fragments are created in lines (5) and (9). Line (9) ensures that b) holds true since "large" numbers are assigned to the newly created path fragments and since the newly created path fragments do not point to any other path fragment by part a). Line (5) keeps b) true since pf' inherits $num(pf)$, since pf' is a subpath of pf, and since pf' points to p after its creation and therefore to no other path fragment by part a). ∎

We can now view our algorithm as manipulating a set $S \subseteq \{(x, y);\ 1 \le x < y \le e\}$ of pairs, namely $S = \{(num(p), num(q));\ p, q$ are PF's and p points to $q\}$. Set S is manipulated in stages, where a stage corresponds to a single execution of the body (3) to (14) of the main loop. Thus the number of stages is at most e. In a stage we remove a number of pairs in lines (4) to (7), say (x_1, y_1), (x_2, y_2), $\dots$, (x_k, y_k) for $k \geq 0$. These pairs form a chain, i.e., $y_1 = x_2, y_2 = x_3, \dots, y_{k-1} = x_k$, because if $y_i = num(pf_i), x_1 = num(pf_0)$ then pf_i must point to pf_{i+1}, $0 \leq i < k$, and p must point to pf_0 prior to line (4). See Figure 25. Thus $K \leq D + e$ where D is the number of pairs removed in lines (4) to (7). In a stage we add some pairs to S in line (9). If $(x, y) = (num(pf_1), num(pf_2))$ is a pair added in line (9) then $y = num(pf_2)$ is a "new" number. In particular, if pair (x, y') was deleted at some previous stage and pair (u, v) was deleted at the same stage then $y > v$.

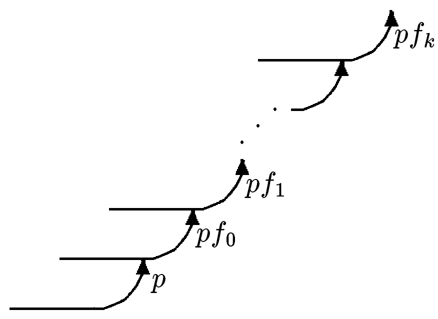


Figure 25. A chain of path fragments

Readers familiar with the Union-Find Problem, Section III.8.3, should see a similarity at this point. Consider the Union-Find data structure with path compression but without the weighted union rule. If one numbers nodes as they are created then upward links correspond to pairs (x, y) with $x < y$. Also path compression removes a chain of pairs and adds some new pairs with a "large" second component.

Theorem 5. *Let N and M be integers. Consider a process operating on the set $S \subseteq \{(x, y);\ 1 \leq x < y \leq M\}$ in N stages. Initially, S is a set of pairs satisfying $(x, y) \in S, (x, y') \in S \Rightarrow y = y'$. In each stage a chain $(x_1, x_2), (x_2, x_3), \dots, (x_{k-1}, x_k)$ of pairs is removed from S and some set of pairs (x, y) is added to S. Let the added pairs (x, y) satisfy*

(1) *If (x, y) is added to S then no (x, y') is currently in S and (x, y) never belonged to S previously.*

(2) *If (x, y') for some y' was deleted at some previous (including the present) stage and pair (u, v) was deleted at the same stage then $y \geq v$.*

Under these assumptions at most $(N + M)\lceil \log M \rceil$ pairs are removed from S.

Proof: The proof is based on the following idea. If we delete a large chain $(x_1, x_2), \dots, (x_{k-1}, x_k)$ at some stage then all pairs (x_i, y) added later on must satisfy $y > x_k$. Therefore all these edges must have a large "reach" $y - x_i$. But no pair can have a "reach" exceeding M and hence this cannot happen very often. The details are as follows. Let F be the set of pairs deleted. We divide F into classes according to the reach of the edges in F, namely

$$M_i = \{(x,y) \in F;\ 2^i \le y - x < 2^{i+1}\} \text{ for } 0 \le i \le \lceil \log M \rceil - 1.$$

Furthermore, let

$$L_i = \{(x,y) \in M_i; \text{ no } (u,v) \in M_i \text{ with } v > y \text{ is removed from } S \text{ at the same stage as } (x,y)\}.$$

Note that the definitions make sense since no pair can be added twice to S by property (1).

Claim 1. $|L_i| \le N$.

Proof: Obvious, since the edges removed at a stage form a chain and since there are only N stages. ∎

Claim 2. *For all x and i there is at most one y such that $(x,y) \in M_i - L_i$.*

Proof: Assume $(x,y_1),(x,y_2) \in M_i - L_i$ where $y_1 < y_2$. When (x,y_1) is removed from S a pair $(u,v) \in M_i$ with $v > y_1$ is also removed from S at the same stage, since $(x,y_1) \notin L_i$. Since the set of pairs removed at a stage form a chain we also have $y_1 \le u$. Next observe that $y_2 \ge v$ by property (2). Thus

$$y_2 - x \ge v - x \ge v - u + y_1 - x \ge 2^i + 2^i = 2^{i+1},$$

since $(x,y_1),(u,v) \in M_i$ and hence $v - u \ge 2^i$ and $y_1 - x \ge 2^i$. We conclude that $(x,y_2) \notin M_i$, contradiction. ∎

The proof is now easily completed. Claims 1 and 2 yield

$$|F| = \sum_{i=0}^{\lceil \log M \rceil - 1} |M_i - L_i| + |L_i| \le (M+N)\lceil \log M \rceil,$$

since $|M_i - L_i| \le M$ by Claim 2 and $|L_i| \le N$ by Claim 1. ∎

Theorem 6.

a) *Let LN be a layered network with n nodes and e edges. Then a blocking flow can be computed in time $O(e \cdot (\log n)^2)$.*
b) *Let $N = (V,E,c)$, $s,t \in V$, be a network with n nodes and e edges. Then a maximum (s,t)-legal flow can be computed in time $O(e \cdot n \cdot (\log n)^2)$.*

Proof:

a) Theorem 5 ($M = N = e$) implies that the number of executions of lines (5) and (6) is $O(e \log n)$. Thus the total running time is $O(e \cdot (\log n)^2)$ by Lemma 6 and by the discussion following it.
b) Follows from part a) and Corollary 1. ∎

Theorem 5 can be used to show an $O((n+m)\cdot\log(n+m))$ bound on the cost of n unions and m finds when path compression is used but the weighted union rule is not used (cf. Chapter III).

Theorem 6 can be slightly improved. Sleator/Tarjan (cf. Sleator (80)) have shown that a clever use of dynamic weighted trees (cf. III.6) instead of balanced trees reduces the cost of blocking flow computations to $O(e\log n)$ and hence the cost of the maximum flow problem to $O(e\cdot n\log n)$. Finally, the algorithm above can be used to compute the maximum flow in an (s,t)-planar network in time $O(n\log n)$; see Exercise 29 for a detailed discussion. The main additional insight required is that a single blocking flow computation suffices to compute a maximum flow.

4.9.2. (0,1)-Networks, Bipartite Matching and Graph Connectivity

In this section we will specialize the network flow algorithms to (0,1)-networks, or more generally bounded networks, and then apply it to compute maximum matchings in bipartite graphs and to compute the vertex connectivity of graphs.

Definition: Let $d\in\mathbb{N}$. A network $N=(V,E,c)$ is ***d*-bounded** if $c(e)\in\{1,2,\ldots,d\}$ for all $e\in E$. A 1-bounded network is commonly called **(0,1)-network**. ∎

If we apply any of our maximum flow algorithms to d-bounded networks then all intermediate flows f are integral, i.e., $f(e)\in\mathbb{N}_0$ for all $e\in E$. In particular, the maximum flow is integral. We already observed that a blocking flow in a (0,1)-network can be computed in linear time. More generally, we have

Lemma 8. *Let N be a d-bounded network. Then a blocking flow can be computed in time $O(d\cdot e)$.*

Proof: Use the proof of Theorem 4 and observe that an edge can be used at most d times in any path from s to t. ∎

From Lemma 8 we conclude that maximum flows in d-bounded networks can be computed in time $O(d\cdot e\cdot n)$. Actually, a much better time bound holds for small d and can be shown by a more careful analysis of our network flow algorithms. More precisely, we show improved bounds on the number of phases executed by the algorithm.

Let $N=(V,E,c)$ be a network. Let $s,t\in V$ and let f be a legal flow. Let $E_1=\{(v,w);\ f(v,w)<c(v,w)\}$ and let $E_2=\{(w,v);\ f(v,w)>0\}$; cf. the definition of layered network. Let $AN=(V,E_1\dot\cup E_2,\bar c)$ where $\bar c(v,w)=c(v,w)-f(v,w)$ for $(v,w)\in E_1$ and $\bar c(w,v)=f(v,w)$ for $(w,v)\in E_2$ and $E_1\dot\cup E_2$ is the disjoint union of E_1 and E_2. Then constructing the layered network with respect to N and f is the same as constructing the layered network with respect to AN and the flow function which is zero everywhere.

Lemma 9. *Let N be a network and let fmax be the value of a maximum (s,t)-flow. Let f be any legal (s,t)-flow, let AN be defined as above and let $\overline{fmax}$ be the value of a maximum (s,t)-flow in AN. Then*

$$fmax = \overline{fmax} + val(f).$$

Proof: Let $(S, V-S)$ be any (s,t)-cut. We use $c(S, V-S)$ and $\bar{c}(S, V-S)$ to denote the capacity of cut $(S, V-S)$ with respect to N and AN, respectively. We have

$$\begin{aligned}\bar{c}(S, V-S) &= \sum_{v\in S, w\in V-S} \bar{c}(v,w) \\ &= \sum_{v\in S, w\in V-S} [(c(v,w) - f(v,w)) + f(w,v)] \\ &= c(S, V-S) - \sum_{v\in V, w\in V-S} (f(v,w) - f(w,v)) \\ &= c(S, V-S) - val(f).\end{aligned}$$

Therefrom we conclude that $\overline{cmin} = cmin - val(f)$ where $cmin$ and $\overline{cmin}$ are the minimum capacities of any (s,t)-cut in N and AN, respectively. An application of Theorem 3 (min cut = max flow) completes the proof. ∎

Lemma 9 states that the augmenting network AN has the potential of increasing the flow to its maximum value. The layered network captures all shortest (s,t)-paths in AN.

Lemma 10. *Let N be a d-bounded network. Then the number of phases is at most $3 \cdot d^{1/3} \cdot n^{2/3}$.*

Proof: Let $fmax$ be the value of a maximum (s,t)-flow. If $fmax < d^{1/3} \cdot n^{2/3}$ then the claim holds true since every phase increases the flow by at least one. So let us assume $fmax \geq d^{1/3} \cdot n^{2/3}$. Consider the phase, say the l-th phase, which increases the flow to at least $fmax - (d^{1/2} \cdot n)^{2/3}$. Then there are at most $d^{1/3} \cdot n^{2/3}$ phases after phase l since every phase increases the flow by at least one. We complete the proof by showing that $l \leq 2 \cdot d^{1/3} \cdot n^{2/3}$, i.e., that flow value $fmax - d^{1/3} \cdot n^{2/3}$ is reached in at most $2 \cdot d^{1/3} \cdot n^{2/3}$ phases. Since the depth of the layered network grows by at least one in every phase (Lemma 3), it suffices to show that k_l, the depth of layered network LN used in phase l, is at most $2 \cdot d^{1/3} \cdot n^{2/3}$. Let $LN = (V_0 \cup V_1 \cup \cdots \cup V_k, \overline{E}, \bar{c})$, where $k = k_l$, $V_0 = \{s\}$, $t \in V_k$, and $\overline{E} \subseteq \bigcup_i (V_i \times V_{i+1})$ be the layered network used in phase l. LN is constructed with respect to flow f.

Let $W_i = V_0 \cup \cdots \cup V_i$, $0 \leq i < k$. Then $(W_i, V - W_i)$ is an (s,t)-cut and hence $\bar{c}(W_i, V - W_i) \geq \overline{fmax} = fmax - val(f) \geq d^{1/3} \cdot n^{2/3}$ by the proof of Lemma 9. Next observe that all edges of AN which emanate in W_i and end in $V - W_i$ actually start in

V_i and end in V_{i+1} by the definition of layered network LN. Hence $\bar{c}(W_i, V - W_i) \leq 2 \cdot d \cdot |V_i| \cdot |V_{i+1}|$ since there are at most $2 \cdot |V_i| \cdot |V_{i+1}|$ edges from V_i to V_{i+1}. (The 2 is due to the fact that $(v,w) \in E_1$ and $(v,w) \in E_2$ is possible.) Thus $|V_i| \cdot |V_{i+1}| \geq (n/d)^{2/3}/2$ and hence $|V_i| + |V_{i+1}| \geq (n/d)^{1/3}$ for $0 \leq i < k$. Summing this inequality for i, $0 \leq i < k$, we obtain

$$2 \cdot |V| \geq k \cdot (n/d)^{1/3}$$

or

$$k \leq 2 \cdot d^{1/3} \cdot n^{2/3}.$$ ∎

Theorem 7. *Let N be a d-bounded network. Then a maximum flow can be computed in time $O(d^{4/3} \cdot n^{2/3} \cdot e)$.*

Proof: Obvious from Lemmas 8 and 10. ∎

One restricted form of (0,1)-networks is particularly important, namely simple (0,1)-networks.

Definition: A network $N = (V, E, c)$ is **simple** if $indeg(v) \leq 1$ or $outdeg(v) \leq 1$ for all $v \in V$. ∎

Theorem 8. *Let $N = (V, E, c)$ be a simple (0,1)-network. Then a maximum flow can be computed in time $O(n^{1/2} \cdot e)$.*

Proof: A phase of the network flow algorithm takes time $O(e)$ by Theorem 4. It therefore suffices to show that the number of phases is $O(n^{1/2})$. We use an argument similar to Lemma 10.

Let $fmax$ be the value of a maximum (s,t)-flow. If $fmax < n^{1/2}$ then there is nothing to show. If $fmax \geq n^{1/2}$ then consider the phase, say the l-th, which increases the flow to $fmax - n^{1/2}$. Then there are at most $n^{1/2}$ phases following phase l. It remains to be shown that the layered network LN used in phase l has depth at most $n^{1/2}$.

Let f be the legal (s,t)-flow obtained by our algorithm just prior to phase l and let AN be the augmenting network with respect to f. We claim that AN is a simple network. This can be seen as follows. Let $v \in V$ be arbitrary. Assume that $indeg(v) = 1$, the case $outdeg(v) = 1$ is similar. If $f(e) = 0$ for $in(v) = \{e\}$ and hence $f(e') = 0$ for all $e' \in out(v)$ then v certainly has indegree one in AN. If $f(e) = 1$ for $in(v) = \{e\}$ and hence $f(e') = 1$ for exactly one $e \in out(v)$ then v has also indegree at most one in AN. This follows from the fact that the direction of e and e' is reversed for constructing the augmenting network. Thus AN is a simple network.

By Lemma 9, AN permits an (s,t)-flow of $fmax - val(f) \geq n^{1/2}$. Consider a maximum (s,t)-flow $\bar{f}$ in AN. We may assume that $\bar{f}$ is integral, i.e., $\bar{f}(\bar{e}) \in \{0,1\}$ for all edges of AN. Since AN is a simple network, $\bar{f}$ defines $val(\bar{f}) \geq n^{1/2}$ paths from s to t which have no common vertex other than s and t. Hence any one of these paths can have at most $n^{1/2}$ intermediate nodes. This shows that the layered network used in phase l has depth at most $n^{1/2}$.

We have thus shown that the number of phases is $O(n^{1/2})$ and hence the total running time is $O(n^{1/2} \cdot e)$. ∎

We end this section with two applications of simple (0,1)-network flow: bipartite matching and graph connectivity.

Let $G = (V,E)$ be an undirected graph. A **matching** M is a set of edges $M \subseteq E$ such that no two edges $e_1, e_2 \in M$, $e_1 \neq e_2$, share an endpoint. A maximum matching is a matching of maximum cardinality. An undirected graph $G = (V,E)$ is **bipartite** if there is a partition V_1, V_2 of V such that $E \subseteq V_1 \times V_2$. In bipartite graphs, the nodes of V_1 (V_2) are often called girls (boys). Then $(v,w) \in E$ can be interpreted as "girl v can go along with boy w". Matching in arbitrary graphs allows for homosexuality.

Theorem 9. *Let $G = (V_1 \cup V_2, E)$, $E \subseteq V_1 \times V_2$, be a bipartite graph. A maximum matching can be computed in time $O(n^{1/2} \cdot e)$.*

Proof: Define a simple (0,1)-network $N = (V_1 \cup V_2 \cup \{s,t\}, \overline{E}, c)$ as follows. Add two nodes s and t, connect s to all vertices in V_1, direct all edges in E from V_1 to V_2 and connect all vertices in V_2 to t. Also assign capacity one to all edges. Then integer-valued flows in N are in one-to-one correspondence to matchings in G. (Figure 26 shows a matching and the corresponding flow by wiggled edges.) By Theorem 8a) maximum flow in N can be computed in time $O(n^{1/2} \cdot e)$. ∎

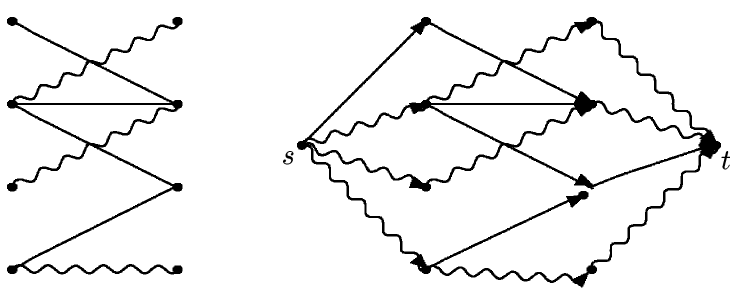


Figure 26. Bipartite matching reduced to a flow problem

We will next turn to vertex connectivity of undirected graphs. Let $G = (V,E)$ be an undirected graph and let $a, b \in V$ be such that $(a,b) \notin E$. Set $S \subseteq V - \{a,b\}$ is an $\boldsymbol{(a,b)}$**-vertex separator** if every path from a to b passes through a vertex of S. In other words a and b belong to different connected components of $G - S$. The minimum cardinality of any (a,b)-vertex separator is denoted by $N(a,b)$. If $(a,b) \in E$ we set $N(a,b) = +\infty$.

Lemma 11. *Let $G = (V, E)$ be an undirected graph and let $a, b \in V$ be such that $(a, b) \notin E$. Then $N(a, b)$ can be computed in time $O(n^{1/2} \cdot e)$.*

Proof: Construct a simple network $N = (\overline{V}, \overline{E}, \bar{c})$ as follows. Let $V' = \{v';\ v \in V\}$ and $V'' = \{v'';\ v \in V\}$; let $\overline{V} = V' \cup V''$ and $\overline{E} = \{(v', v'');\ v \in V\} \cup \{(v'', w'), (w'', v');\ (v, w) \in E\}$. Finally, let $\bar{c}(v', v'') = 1$ for $v \in V$ and let $\bar{c}(v'', w') = \bar{c}(w'', v') = \infty$ for $(v, w) \in E$. The construction is illustrated by Figure 27.

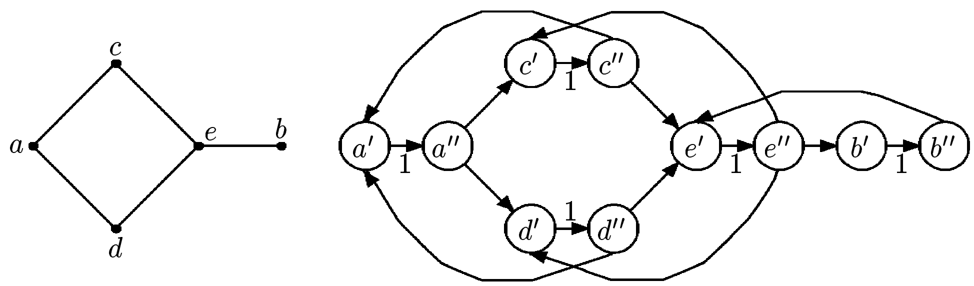


Figure 27. Computation of $N(a, b)$

Claim: *$N(a, b)$ is equal to the maximum flow from a'', the source, to b', the sink, in network N.*

Proof: "$\leq$": Let $(A, \overline{V} - A)$ be a minimal (a'', b')-cut in network N. Let $S = \{v;\ v' \in A, v'' \in \overline{V} - A\}$. Then clearly $\bar{c}(A, \overline{V} - A) \geq |S|$. Also, S is an (a, b)-vertex separator and hence $|S| \geq N(a, b)$. This can be seen as follows. Let $a = v_0, v_1, \ldots, v_k = b$ be a path from a to b in G. Consider path $v_0'', v_1', v_1'', v_2', \ldots, v_{k-1}'', v_k'$ from a'' to b' in N. At least one of its edges must go across the cut defined by A. It cannot be one of the edges (v_i'', v_{i+1}') because these edges have capacity ∞ and cut $(A, \overline{V} - A)$ has finite capacity. Note that cut $(A, \overline{V} - A)$ has finite capacity because it is a minimal cut and since there are cuts, e.g., $A = \{a''\} \cup (V' - \{b'\})$, of finite capacity.

"$\geq$": Let $S \subseteq V - \{a, b\}$ be an (a, b)-vertex separator with $|S| = N(a, b)$. Define $A = \{x \in \overline{V};\ x$ can be reached from a'' without using an edge (s', s'') for $s \in S\}$. Then $b' \notin A$ and hence $(A, \overline{V} - A)$ is an (a'', b')-cut of network N. Also, $v' \in A$ implies $v'' \in A$ for $v \in V - S$. Hence $\bar{c}(A, \overline{V} - A) \leq |S| = N(a, b)$. ∎

It is easy to see that the maximum flow from a'' to b' does not change if we change all capacities to 1. In this way we obtain a simple (0,1)-network. A maximum flow in this network and hence also $N(a, b)$ can be computed in time $O(n^{1/2} \cdot e)$ by Theorem 8. ∎

The **vertex connectivity** c of an undirected graph $G = (V, E)$ is the minimal connectivity number of any pair of unconnected vertices. More precisely

$$c = \begin{cases} n - 1 & \text{if } G \text{ is complete;} \\ \min\{N(a, b);\ (a, b) \notin E\} & \text{otherwise.} \end{cases}$$

Theorem 10. *Let $G = (V, E)$ be an undirected graph and let c be its vertex connectivity.*

a) *c can be computed in time $O(c \cdot n^{3/2} \cdot e) = O(n^{1/2} \cdot e^2)$.*

b) *Let $\epsilon > 0$ and assume $e \leq n^2/4$. Then there is a randomized algorithm which computes c in expected time $O((-\log \epsilon) \cdot n^{3/2} \cdot e)$ with probability of error at most ϵ.*

Proof: Both parts are based on the following simple observation.

Claim 1. *Assume $c < n - 1$. Let $c = N(x, y)$ for some nodes $x, y \in V$ and let S, $|S| = c$, be an (x, y)-vertex separator. Then $c = \min\{N(a, b);\ b \in V\}$ for all $a \in V - S$.*

Proof: $G - S$ consists of at least two components. Let b be a node which does not belong to the same component as a. Then S separates a from b and hence $N(a, b) \leq |S| = c$. Thus $c = N(a, b)$ by definition of c. ∎

a) Claim 1 suggests Program 28. Let $v_1, v_2, v_3, \ldots, v_n$ be some ordering of V. Correctness of Program 28 can be seen as follows. It is $c = \min\{c_1, c_2, \ldots, c_{c+1}\}$ by the claim above. Also $C \geq c$ and $C = \min\{c_1, \ldots, c_i\}$ always. The algorithm terminates with $c \leq C < i$ and hence $C = c$. Also $C = c$ whenever $i \geq c + 1$. Thus $c + 1$ iterations suffice.

(1) $C \leftarrow \infty;\ i \leftarrow 1;$
(2) **while** $C \geq i$
(3) **do** $c_i \leftarrow \min\{N(v_i, v);\ v \in V\};$
(4) $\quad C \leftarrow \min(C, c_i);\ i \leftarrow i + 1$
(5) **od**.

Program 28

The running time is determined by line (3). A single execution of line (3) takes time $O(n^{3/2} \cdot e)$ by Lemma 11. Hence the total running time is $O(c \cdot n^{3/2} \cdot e)$. Finally observe that $c \leq \min\{\deg(v);\ v \in V\} \leq 2 \cdot e/n$ since $\sum_{v \in V} \deg(v) = 2 \cdot e$.

b) Since $c \leq 2 \cdot e/n$, cf. the proof of part a), and $e \leq n^2/4$ by the assumption we have $c \leq n/2$.

Claim 2. *Choose $a \in V$ at random. Then*

$$\mathrm{prob}(c > \min\{N(a, b);\ b \in V\}) \leq 1/2.$$

Proof: This is almost obvious from Claim 1. Let S be defined as in Claim 1. Then $|S| \leq n/2$ and

$$\mathrm{prob}(c > \min\{N(a, b);\ b \in V\}) \leq \mathrm{prob}(a \in S) \leq 1/2.$$ ∎

Claim 2 suggests the randomized algorithm of Program 29 for computing c.

---

(1) $k \leftarrow -\log \epsilon$; $C \leftarrow \infty$;
(2) **do** k **times**
(3) choose $a \in V$ at random;
(4) $C \leftarrow \min(C, \min\{N(a,b);\ b \in V\})$
(5) **od**.

Program 29

The running time of Program 29 is clearly $O((-\log \epsilon) \cdot n^{3/2} \cdot e)$. Also, the probability that $C > c$ upon termination is at most $(1/2)^k = \epsilon$ by Claim 2. ∎

4.9.3. Weighted Network Flow and Weighted Bipartite Matching

A **weighted network flow problem** is given by $N = (V, E, cap, cost)$ and nodes $s, t \in V$. Here $cap : E \to \mathbb{R}^+$ gives the capacity of an edge (we used c instead of cap so far) and $cost : E \to \mathbb{R}$ is the cost of transporting one unit of flow across an edge. Throughout this section we assume that capacities are integers. Let f be a legal (s,t)-flow. Then the cost of f is given by

$$cost(f) = \sum_{e \in E} f(e) \cdot cost(e).$$

The weighted network flow problem is then to compute a legal (s,t)-flow with maximum flow value and (among these) minimal cost. More generally, we might look for a flow function f with $val(f) = v$ for some predefined v and minimal $cost$. In this section we will see that a minimal cost flow with flow value v can be computed in time $O(v \cdot e \cdot (\log n)/\log(e/n))$. At the end of this section we will apply this result to weighted bipartite matching and derive an $O(n \cdot e \cdot (\log n)/\log(e/n))$ algorithm for it.

The theory of weighted network flow is a natural extension of the theory of ordinary network flow. Let $N = (V, E, cap, cost)$ be a network, $s, t \in V$, and let $f : E \to \mathbb{R}$ be a legal (s,t)-flow. We define the augmenting network with respect to N and f as we did in the previous section. More precisely, $AN = (V, \overline{E}, \overline{cap}, \overline{cost})$, where $\overline{E} = E_1 \cup E_2$ and $E_1 = \{e \in E;\ f(e) < cap(e)\}$ and $E_2 = \{(w,v);\ (v,w) = e \in E \text{ and } f(e) > 0\}$. For $e \in E$ we use e_i to denote the edge corresponding to e in E_i, $i = 1, 2$. Also

$$\overline{cap}(\bar{e}) = \begin{cases} cap(e) - f(e) & \text{if } \bar{e} = e_1; \\ f(e) & \text{if } \bar{e} = e_2 \end{cases}$$

and

$$\overline{cost}(\bar{e}) = \begin{cases} cost(e) & \text{if } \bar{e} = e_1; \\ -cost(e) & \text{if } \bar{e} = e_2. \end{cases}$$

Our first lemma connects minimality in cost with the existence of cycles of negative cost in the augmenting network.

Lemma 12. *Let f be a legal (s,t)-flow with $val(f) = v$ and let AN be the augmenting network with respect to f. Then f has minimal cost among all (s,t)-flows with value v iff there is no cycle of negative cost in AN.*

Proof: "$\Rightarrow$": (Indirect.) It is clear that a negative cost cycle can be used to decrease the cost of f without changing the flow value.

"$\Leftarrow$": (Indirect.) Assume that f does not have minimal cost, i.e., there is a legal (s,t)-flow g with $val(g) = val(f)$ and $cost(g) < cost(f)$. Let $AN = (V, \overline{E}, \overline{cap}, \overline{cost})$ be the augmenting network with respect to f. Consider $h : \overline{E} \to \mathbb{R}$ defined by

$$h(\bar{e}) = \begin{cases} \max(0, g(e) - f(e)) & \text{if } \bar{e} = e_1; \\ \max(0, f(e) - g(e)) & \text{if } \bar{e} = e_2. \end{cases}$$

Claim 1. *h is a legal (s,t)-flow in AN with $val(h) = 0$ and $\overline{cost}(h) < 0$.*

Proof: We show first that h has negative cost. Note that for all $e \in E$ we have $h(e_1) \cdot \overline{cost}(e_1) + h(e_2) \cdot \overline{cost}(e_2) = (g(e) - f(e)) \cdot cost(e)$ and hence

$$\begin{aligned} \overline{cost}(h) &= \sum_{\bar{e} \in \bar{E}} h(\bar{e}) \cdot \overline{cost}(\bar{e}) \\ &= \sum_{e \in E} [h(e_1) \cdot \overline{cost}(e_1) + h(e_2) \cdot \overline{cost}(e_2)] \\ &= \sum_{e \in E} (g(e) - f(e)) \cdot cost(e) \\ &= cost(g) - cost(f) < 0. \end{aligned}$$

Next we show that h satisfies the conservation laws. Note that for all $e \in E$ $h(e_1) - h(e_2) = g(e) - f(e)$ and hence for all $v \in V$ ($\overline{out}(v)$ is the set of edges emanating from v in AN and similarly for $\overline{in}(v)$):

$$\begin{aligned} & \sum_{\bar{e} \in \overline{out}(v)} h(\bar{e}) - \sum_{\bar{e} \in \overline{in}(v)} h(\bar{e}) \\ = & \sum_{e \in out(v)} (h(e_1) - h(e_2)) - \sum_{e \in in(v)} (h(e_1) - h(e_2)) \\ = & \sum_{e \in out(v)} (g(e) - f(e)) - \sum_{e \in in(v)} (g(e) - f(e)) \\ = & \left[\sum_{e \in out(v)} g(e) - \sum_{e \in in(v)} g(e) \right] - \left[\sum_{e \in out(v)} f(e) - \sum_{e \in in(v)} f(e) \right] \\ = & \begin{cases} 0 - 0 & \text{if } v \neq s,t; \\ val(g) - val(f) & \text{if } v = s; \\ -val(g) + val(f) & \text{if } v = t. \end{cases} \end{aligned}$$

In any case, this shows that h satisfies the conservation laws and that $val(h) = 0$. Finally, it is trivial to see that h satisfies the capacity constraints. ∎

Flow function h has zero flow and negative cost. It is intuitively clear, that h is circular in some sense. More precisely, we show that h can be decomposed into a set of flows around cycles. It is then easy to conclude that one of the cycles must have negative cost.

Claim 2. *There are $h_1, h_2, \ldots, h_m : \bar{E} \to \mathbb{R}_0^+$, $m \le e$, such that*

1) $h(\bar{e}) = \sum_i h_i(\bar{e})$ *for all* $\bar{e} \in \bar{E}$.
2) *For every i, $1 \le i \le m$, there is a directed cycle $w_0, w_1, \ldots, w_k = w_0$ in AN such that $h_i(w_j, w_{j+1}) = h_i(w_l, w_{l+1})$ for $0 \le j < l < k$, and $h_i(\bar{e}) = 0$ for edges not on the cycle.*

Proof: (By induction on the number k of edges $\bar{e} \in \bar{E}$ with $h(\bar{e}) \neq 0$.) If $k = 0$ then there is nothing to prove. So let us assume $k \ge 0$. Let v_0 be any node such that there is an edge $(v_0, v_1) \in \bar{E}$ with $h(v_0, v_1) \neq 0$. Since $\sum_{\bar{e} \in \overline{out}(v_1)} h(\bar{e}) = \sum_{\bar{e} \in \overline{in}(v_1)} h(\bar{e})$ and $h(\bar{e}) \ge 0$ for all $\bar{e} \in \bar{E}$ there must be v_2 such that $(v_1, v_2) \in \bar{E}$ with $h(v_1, v_2) \neq 0$. Continuing in this fashion we construct a path $v_0, v_1, v_2, \ldots, v_r$ in AN with $v_r = v_j$ for some $j < r$.

We take $v_j, v_{j+1}, \ldots, v_r$ as the desired cycle and define $h_1 : E \in \mathbb{R}_0^+$ by $h_1(v_l, v_{l+1}) = \min\{h(v_l, v_{l+1});\ j \le l < r\}$ for $j \le l < r$ and $h_1(\bar{e}) = 0$ for all edges $\bar{e}$ not on the cycle.

Finally, let $h' = h - h_1$. Then h' is a legal (s,t)-flow with flow value 0. Also there is at least one edge $\overline{e}$ less with $h'(\bar{e}) \neq 0$. ∎

The proof of Lemma 12 is now readily completed. It is $\overline{cost}(h) = \sum_i \overline{cost}(h_i)$ and hence $\overline{cost}(h_i) < 0$ for some i, $1 \le i \le m$. Let C be the cycle underlying h_i and let ϵ be the flow along the edges of C. Then $\overline{cost}(h_i) = \epsilon \cdot \overline{cost}(C)$ where $\overline{cost}(C)$ is the cost of cycle C interpreted as a path in network AN. ∎

Lemma 12 can be used to design an algorithm for minimizing the cost without changing the flow value (Exercise 32). What is more important, we can use Lemma 12 to show that the augmenting along minimum cost paths does not destroy the cost minimality.

Lemma 13. *Let f be a minimal cost flow with $val(f) = v$ and let $AN = (V, \overline{E}, \overline{cap}, \overline{cost})$ be the augmenting network with respect to f. Let p be a minimum cost path from s to t in AN, let f' be a legal (s,t)-flow in AN which is non-zero only along p (i.e., f' sends some units of flow from s to t along p). Then f'' where*

$$f''(e) = f(e) + f'(e_1) - f'(e_2) \qquad \text{for all } e \in E$$

is a minimum cost flow of value $val(f) + val(f')$.

Proof: f'' is certainly a legal (s,t)-flow with value $val(f) + val(f')$. A formal proof can be given along the lines of Lemma 2b). It remains to be shown that f'' has minimal cost. Assume otherwise. Then there is a negative cost cycle C in the augmenting network AN'' constructed with respect to f''. We will derive a contradiction as follows.

If cycle C exists in AN then f was not optimal, contradiction. So C cannot exist in AN, i.e., there is at least one edge (v,w) on path p such that C uses this edge in reverse direction. Let (v,w) be the first such edge. See Figure 28.

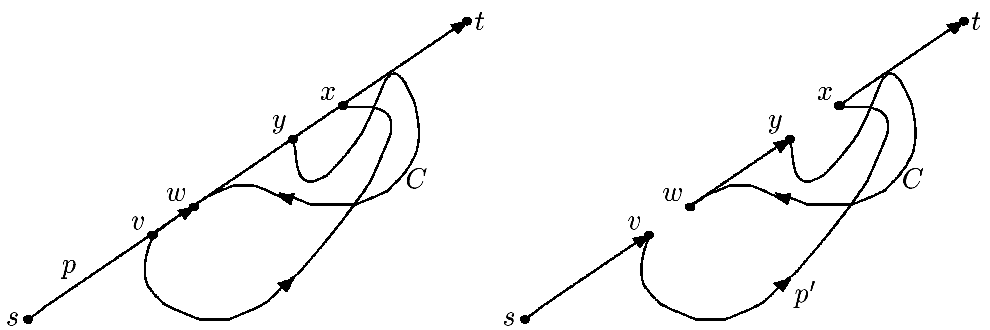


Figure 28. Construction for Lemma 13

Let path p' from s to t be constructed as follows. Follow p from s to v, then follow C until C intersects p for the next time, say in point x, then follow p from x to t. Let cycle C' be constructed as follows. Follow p from w to y, where y is the point following x on cycle C, and then follow C from y to w. Note that $\overline{cost}(p') + \overline{cost}(C') = \overline{cost}(p) + \overline{cost}(C)$ since the cost of edge (v,w) is the negative of the cost of edge (w,v). Continuing in this way we obtain an (s,t)-path p'' from s to t in AN and a cycle C'' such that $\overline{cost}(p'') + \overline{cost}(C'') = \overline{cost}(p) + \overline{cost}(C)$ and p'' and C'' use no edge in reverse order. Thus C'' is a cycle in network AN. Since $\overline{cost}(C) < 0$ we either have $\overline{cost}(C'') < 0$, a contradiction to the optimality of f, or $\overline{cost}(p'') < \overline{cost}(p)$, a contradiction to the fact, that p is a least cost path from s to t in AN. ∎

Lemma 13 gives rise to a minimum cost flow algorithm formulated in Program 30.

construct a minimal cost flow with flow value 0 (cf. Exercise 32);
while $val(f) < v$
do let AN be the augmenting network with respect to f;
    let p be a least cost path from s to t in AN;
    let ϵ be the minimal capacity of any edge in p;
    increase the flow along p by $\min(\epsilon, v - val(f))$
od.

Program 30

Theorem 11. *Let $N = (V, E, cap, cost)$, $s, t \in V$, be a network with integer capacities and let $v \in \mathbb{R}^+$. Then a minimum cost flow with value v (if it exists) can be computed in time $O((1 + v) \cdot n \cdot e)$.*

Proof: Correctness of the algorithm above is obvious from Lemma 13. A single execution of the loop body takes time $O(e)$ for lines (3), (5) and (6), plus $O(n \cdot e)$ for line (4). The time bound for line (4) follows from 4.7.3, Theorem 5. Finally, since capacities are integers, the flow is increased by at least one in every iteration (except maybe the last). Thus the total running time is $O((1 + v) \cdot n \cdot e)$. ∎

In line (4) of Program 30 one has to solve single source least cost problems. In Section 4.7.4 we saw that arbitrary edge costs can sometimes be transformed into non-negative edge costs by means of a potential function. More precisely, we proceeded as follows. Given a weighted graph $(V, E, cost)$ and $s \in V$ we computed $\mu(s, v)$, the cost of the least cost path from s to v. We used $\mu(s, v)$ as a potential function and turned all edge costs into non-negative edge costs by

$$\widetilde{cost}(v, w) = cost(v, w) + \mu(s, v) - \mu(s, w).$$

A similar approach works here. Let AN be the augmenting network with respect to minimal cost flow f and let p be a minimal cost path from s to t in AN. After increasing the flow along p we obtain flow f'. Let AN' be the augmenting network with respect to f'. Then AN and AN' are very similar. The only difference is that some edges of path p are removed from, and the reverse of some edges of path p are added to AN to obtain AN'. Also if a reverse edge is added then its cost is the negative of the cost of the edge. Let $\mu(s, v)$ be the cost of the least cost path from s to v in AN. We claim that we can use $\mu(s, v)$ as a potential function for least cost path computations in network AN'.

Lemma 14. *Let $AN' = (V, E', cap', cost')$ and let $\mu(s, v)$ be the cost of a least cost path from s to v in AN, $v \in V$. Let*

$$\widetilde{cost}(v, w) = cost'(v, w) + \mu(s, v) - \mu(s, w)$$

for all $(v, w) \in E'$. Then $\widetilde{cost}(v, w) \geq 0$ for all $(v, w) \in E'$.

Proof: We distinguish two cases: Edge (v, w) is the reverse of an edge of path p or it is not. If it is not then $cost'(v, w) = \overline{cost}(v, w)$ where $\overline{cost}$ is the cost function of network AN and the claim is true since the distances satisfy the triangle inequality. If (v, w) is the reverse of edge (w, v) and (w, v) belongs to p then $\mu(s, v) = \mu(s, w) + \overline{cost}(w, v)$ since p is a least cost path and $cost'(v, w) = -\overline{cost}(w, v)$ by definition of AN'. Hence $\widetilde{cost}(v, w) = 0$. ∎

Lemma 14 almost implies that we only have to solve single source least cost path problems with non-negative edge costs in line (4). However, there is still a small problem to solve. If we transform edge costs as described in Lemma 14, then we compute $\widetilde{\mu}(s, v)$, the least cost of a path from s to v in AN' with respect to cost function $\widetilde{cost}$, in line (4). However, we need to know $\mu'(s, v)$, the least cost of a path from s to v in AN' with respect to cost function $cost'$, in order to be able to transform edge costs for the next iteration. This difficulty is easily surmounted. Note that

$$\begin{aligned}\widetilde{\mu}(s, v) &= \mu'(s, v) + \mu(s, s) - \mu(s, v) \\ &= \mu'(s, v) - \mu(s, v)\end{aligned}$$

for all $v \in V$. Hence $\mu'(s, v)$ is easily computed from $\widetilde{\mu}(s, v)$ and $\mu(s, v)$. We summarize in

Theorem 12. *Let $N = (V, E, cap, cost)$, $s, t \in V$, be a network with integer capacities and let $v \in \mathbb{R}^+$. Then a minimum cost flow from s to t with value v can be computed in time $O(v \cdot e \cdot (\log n)/\max(1, \log(e/n)))$.*

Proof: Follows immediately from the discussion above, and 4.7.2, Theorem 2. ∎

We end this section with a short discussion of weighted bipartite matching. Let $G = (V_1 \cup V_2, E)$, $E \subseteq V_1 \times V_2$, be a bipartite (undirected) graph. Let $cost : E \to \mathbb{R}^+$ be a cost function. If $M \subseteq E$ is a matching then the cost of M is defined by

$$cost(M) = \sum_{e \in M} cost(e).$$

Theorem 13. *Let $G = (V_1 \cup V_2, E)$ be a weighted bipartite graph, let $cost : E \to \mathbb{R}^+$ be a cost function and let $v \leq n$, $v \in \mathbb{N}$. Then a matching of cardinality v (if it exists) and minimal cost can be computed in time*

$$O(n \cdot e \cdot (\log n)/\max(1, \log(e/n))).$$

Proof: The proof is very similar to the proof of Theorem 9. Define network $N = (\{s, t\} \cup V_1 \cup V_2, \overline{E}, \overline{cap}, \overline{cost})$ by $\overline{E} = (\{s\} \times V_1) \cup E \cup (V_2 \times \{t\})$, $\overline{cap}(\bar{e}) = 1$ for all $\bar{e} \in \overline{E}$ and $\overline{cost}(\bar{e}) = cost(\bar{e})$ for $\bar{e} \in E$ and $cost(\bar{e}) = 0$ for $\bar{e} \in \overline{E} - E$. Then matchings and flows are in one-to-one correspondence and hence a matching of cardinality v and minimal cost can be computed in time $O(n \cdot e \cdot (\log n)/\max(1, \log(e/n)))$ by Theorem 12. ∎

4.10. Planar Graphs

This section is devoted to planar graphs. We will treat five topics. We start with a linear time algorithm for 5-coloring planar graphs and then show how to test planarity and to construct a combinatorial embedding in linear time. The third topic is an $O(n \log n)$ algorithm for the construction of a straight-line embedding and the fourth topic is the planar separator theorem which makes the family of planar graphs amenable to divide-and-conquer algorithms. The final topic is one particular algorithm based on the divide-and-conquer paradigm: a single source least cost path algorithm for planar graphs.

A (**topological**) **planar embedding** of an undirected graph $G = (V, E)$ is a function $\mathcal{E}$ that maps the vertices of G to distinct points in $\mathbb{R}^2$ and each edge $\{u, v\} \in E$ to a Jordan curve in $\mathbb{R}^2$ from $\mathcal{E}(u)$ to $\mathcal{E}(v)$ such that for all $e = \{u, v\} \in E$, $\mathcal{E}(e) \cap (\mathcal{E}(V) \cup \mathcal{E}(E \backslash \{e\})) = \{\mathcal{E}(u), \mathcal{E}(v)\}$ (i.e., edges do not cross). G is **planar** if there exists a planar embedding of G.

Let $\mathcal{E}$ be a planar embedding of a planar graph $G = (V, E)$. The **faces** of $\mathcal{E}$ are the connected regions of $\mathbb{R}^2 \backslash \mathcal{E}(V \cup E)$. The **boundary** (induced by $\mathcal{E}$) of a face F of $\mathcal{E}$ is the subgraph of G consisting of those elements $\xi \in V \cup E$ for which there exist points in F arbitrarily close to $\mathcal{E}(\xi)$. If G is biconnected and $|V| \geq 3$, the boundary of each face of $\mathcal{E}$ is a simple cycle. Let D be the set of **darts** of the directed version of G, and for each dart $e = (u, v) \in D$, let $\Phi_{\mathcal{E}}(e)$ be the dart $e' = (u, w) \in D$ such that $\mathcal{E}(\{u, w\})$ is the first curve in $\mathcal{E}(E)$ with endpoint $\mathcal{E}(u)$ encountered after $\mathcal{E}(e)$ in a clockwise scan around $\mathcal{E}(u)$. $\Phi_{\mathcal{E}}$ is a permutation of D known as the combinatorial planar embedding corresponding to $\mathcal{E}$ or induced by $\mathcal{E}$. Note that $\Phi_{\mathcal{E}}$ is equivalent to a cyclic ordering of the edges incident to any vertex v. A permutation Φ of D is called a **combinatorial planar embedding** if $\Phi = \Phi_{\mathcal{E}}$ for some planar embedding $\mathcal{E}$. The graph $(D, \{((u, v), \Phi_{\mathcal{E}}((v, u)));\ (u, v) \in D\})$, which is the union of vertex-disjoint directed simple cycles, is called the **face cycle graph** of $\mathcal{E}$, and its cycles the **face cycles** of $\mathcal{E}$. The intuitive meaning of a face cycle C of $\mathcal{E}$ is that it corresponds to a walk inside a particular face F of $\mathcal{E}$ along the image R under $\mathcal{E}$ of the elements on the boundary of F, always keeping F to the left and R to the right (cf. Fig. 101). The face cycle C is said to be a face cycle of F. If G is connected, each face of $\mathcal{E}$ has precisely one face cycle. A face is said to be incident on the vertices and edges on its boundary, and the vertices and edges on its boundary are said to **border** the face. Among the faces there is exactly one which is unbounded. It is called the **outer** or **infinite face**.

Lemma 1. *Let $\mathcal{E}$ be any planar embedding of a graph G, let F be a face of $\mathcal{E}$ and let B be the boundary of F. Then there is an embedding $\mathcal{E}'$ of G in which B is the boundary of the outer face. Moreover, $\mathcal{E}$ and $\mathcal{E}'$ induce the same combinatorial embedding.*

Proof: Let $\mathcal{E}$ be any embedding of G and let F be any face of $\mathcal{E}$, which is not the outer face. Then there is a sequence $F_0, F_1, \ldots, F_k$ of faces such that F_0 is the outer face, $F_k = F$ and F_i and F_{i+1} have a common edge in their boundaries. Let e be

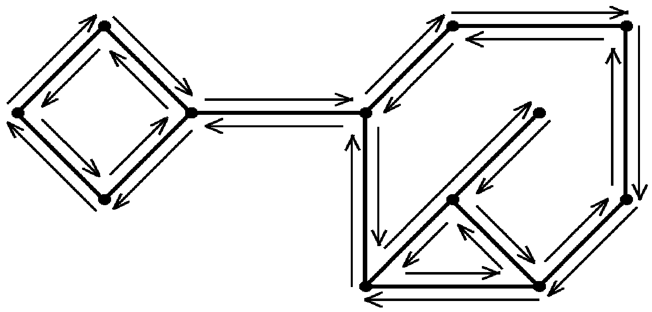


Figure 101. A planar embedding and its face cycles

an edge which is on the boundary of F_0 and F_1. By changing the embedding of e, cf. Figure 102, we can make the boundary of F_1 the boundary of the outer face. Continuing in this fashion we obtain the desired embedding $\mathcal{E}'$. ∎

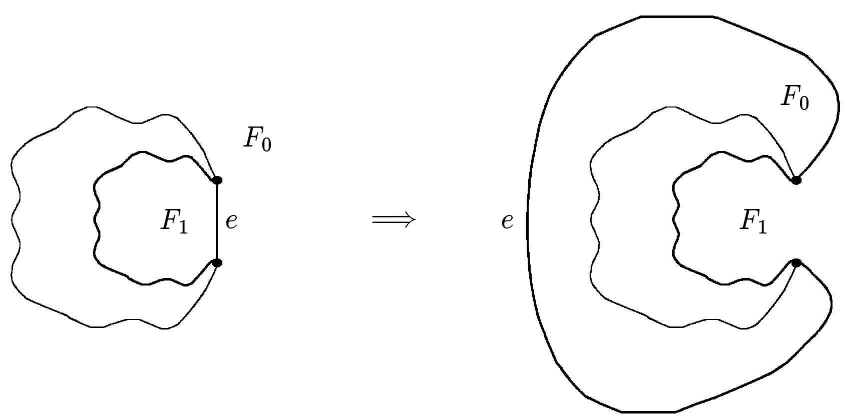


Figure 102. Changing the embedding of edge e

Lemma 2. *Let $G = (V, E)$ be a planar graph. Then $m \leq 3n - 6$ for $n \geq 3$.*

Proof: It clearly suffices to prove the lemma for connected planar graphs. We first prove Euler's formula which relates the number of edges, nodes and faces of a connected planar graph.

Claim: *Let $\mathcal{E}$ be a planar embedding of a connected planar graph G with n nodes and m edges and let f be the number of faces. Then $n + f = m + 2$.*

Proof: We use induction on m. If $m = 0$ then $n = f = 1$ and the claim holds. If $m > 0$ and there is a node of degree 1 then removal of this node yields an embedding of a connected graph with one less node, one less edge and the same number of faces and the claim follows immediately from the induction hypothesis. If $m > 0$ and there is no node of degree 1 then $\mathcal{E}$ contains a cycle. The removal of one of the edges of the cycle yields an embedding of a connected graph with the same number of nodes, one less face and one less edge, and again the claim follows immediately from the induction hypothesis. ∎

Let $\mathcal{E}$ be any embedding of G. Then the face cycle graph of $\mathcal{E}$ consists of one cycle for each face and this cycle consists of at least three darts of G since G is connected and $n \geq 3$. Thus $3f \leq 2m$ and hence $n + 2m/3 \geq m + 2$ or $m \leq 3n - 6$. ∎

Lemma 2 is helpful in at least two respects. First, it implies that an $O(n+m)$ algorithm on planar graphs is really an $O(n)$ algorithm and secondly it implies that a planar graph has a large fraction of nodes of small degree. More precisely, let n_d be the number of nodes of degree d. Then $\sum_j n_j \cdot j = 2m$ and $\sum_j n_j = n$ and hence $\sum_j n_j \cdot j \leq \sum_j 6n_j - 12$ by Lemma 2. Thus

$$(n_1 + \cdots + n_d) \geq \frac{\sum_{j \geq d+1}(j-6)n_j + 12}{6}.$$

In particular, $n_1 + \cdots + n_5 \geq 2$, i.e., there are at least two nodes of degree at most 5, and $n_1 + \cdots + n_6 \geq \sum_{j \geq 7} n_j/6 = (n - (n_1 + \cdots + n_6))/6$, i.e., at least 14% of the nodes have degree at most 6. The fact that a large number of nodes have small degree can sometimes be exploited to design divide-and-conquer algorithms for planar graphs. One such algorithm is treated in Section 8.3 on Voronoi diagrams and searching planar subdivisions. Another consequence of Lemma 2 is the following

Lemma 3. *Every planar graph is 5-colorable, i.e., if $G = (V, E)$ is planar then there is a mapping $c : V \to \{1, 2, 3, 4, 5\}$ such that $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$.*

Proof: We use induction on the number of nodes of G. Let $v \in V$ be any node of degree at most 5. If the degree of v is 4 or less then a 5-coloring of $G - v$ can clearly be extended to a 5-coloring of G. So let us assume that v has degree 5. Then there must be neighbors x and y of v such that $\{x, y\}$ is not an edge of G. Otherwise, the neighbors of v would form a K_5, i.e., a complete graph on 5 nodes, and hence G were not planar since a K_5 has 5 nodes and 10 edges and hence is not planar by Lemma 2. Consider the graph G' obtained from $G - v$ by identifying the nodes x and y, i.e., G' has the vertex set $V' = V - \{v, y\}$ and the edge set

$$\begin{aligned} E' = \{\{u,w\};\ u, w \in V' \text{ and } \{u,w\} \in E\} \cup \\ \{\{x,w\};\ w \in V' \text{ and } \{y,w\} \in E\}. \end{aligned}$$

The graph G' is clearly planar (Figure 103 indicates how a planar embedding of G' can be obtained from a planar embedding of G) and hence G' is 5-colorable by induction hypothesis. Thus $G - v$ is 5-colorable with nodes x and y having the same color and hence G is 5-colorable. ∎

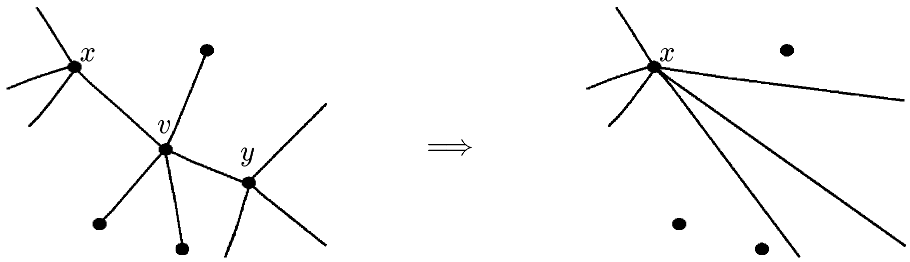


Figure 103. A planar embedding of G yields a planar embedding of G'

The proof of Lemma 3 directly yields a 5-coloring algorithm with $O(n^2)$ running time. The key idea for a linear time algorithm is the observation that we can require the nodes x and y to have small degree. Let us call a node v **small** if $deg(v) \leq 11$, and **large** otherwise and let us call a node v **good**, if either $deg(v) \leq 4$ or $deg(v) = 5$ and v has at most one large neighbor.

Lemma 4.

a) *A planar graph contains at least one good node.*

b) *Let v be a good node of degree 5. Then there are small neighbors x, y of v such that $\{x, y\} \notin E$.*

Proof: a) Let n_d be the number of nodes of degree d. If $n_1 + \cdots + n_4 > 0$ then we are done. Otherwise

$$\sum_{j \geq 5} j \cdot n_j \leq 6n - 12 < 6 \cdot \sum_{j \geq 5} n_j$$

by Lemma 2 and hence

$$n_5 > \sum_{j \geq 6} (j-6) \cdot n_j \geq \sum_{j \geq 12} (j-6) \cdot n_j \geq \frac{1}{2} \sum_{j \geq 12} j \cdot n_j .$$

Suppose now that a good node does not exist. Then every node of degree 5 has at least two large neighbors and hence the number of edges incident to large nodes is at least $2 \cdot n_5$, i.e., $\sum_{j \geq 12} j \cdot n_j \geq 2 \cdot n_5$. This contradicts the inequality derived above.

b) Let N be the set of small neighbors of v. Then $|N| \geq 4$ and if $\{x, y\} \in E$ for all $x, y \in N$ then $N \cup \{v\}$ contains a K_5 and hence G is not planar, a contradiction. ∎

Lemmas 3 and 4 suggest the following algorithm for 5-coloring a planar graph. We start by determining the set M of good nodes. This takes time $O(n)$. We keep the set M as a doubly linked linear list L. For each node $v \in M$ we have a pointer to the item on the list L which represents v. Then v can be deleted from M in constant time, an element of M can be selected in constant time and elements can be added to M in constant time.

Let v be any node in M. If the degree of v is four or less, then we delete v from M, update M (cf. below), color $G - v$ by applying the algorithm recursively, and finally color v. If v has degree 5, then let x and y be small neighbors of v with $\{x, y\} \notin E$. The nodes x and y exist by Lemma 4 and can be found in time $O(1)$. We delete v from the graph, identify x and y (this takes time $O(1)$, since v, x and y are small) and update M (cf. below). We then color G' recursively and finally extend the coloring to G. We still need to explain how to update M. When v is deleted from the graph some nodes may become small and hence some of their neighbors good. All of this can certainly be checked in time $O(1)$. Similarly, when x and y are identified, then x may become large and a common neighbor of x and y may become small. Again, all of this and the effect on M is easily determined in constant time. Thus time $O(1)$ suffices to color an additional node and hence the algorithm runs in time $O(n)$.

Theorem 1. *A 5-coloring of a planar graph can be computed in linear time.* ∎

Our next topic is a linear time planarity testing algorithm. Since a graph is planar iff its biconnected components are (cf. Section 4.6 for a linear time algorithm to compute the biconnected components of a graph) we can restrict our attention to biconnected graphs. Also we can confine ourselves to graphs with $m \le 3n - 6$ by Lemma 2. The planarity testing algorithm is an extension of depth-first-search. In the sequel we will always identify nodes with their DFS-number. A DFS on the directed version of $G = (V, E)$ partitions the darts of G into the sets T, F and B. For the planarity testing algorithm we consider the directed graph $(V, T \cup F^{-1})$ and call the edges in T tree edges and the edges in F^{-1} back edges. Also, we write B instead of F^{-1}. Note that this notation differs slightly from the one used in Section 4.5. There, reversals of tree edges were also called back edges.

We will now describe the idea underlying the planarity algorithm. Let C be any cycle starting in the root of the *dfs*-tree and consisting of tree edges followed by one back edge. Such a cycle exists since G is assumed to be biconnected. For every edge $e = (x, y)$ emanating from the cycle, i.e., x lies on C but e is not an edge of the cycle we consider the segment $S(e)$ defined as follows. If e is a back edge then $S(e)$ is the cycle formed by the tree path from y to x together with the edge e. If e is a tree edge then $S(e)$ consists of the subgraph spanned by the set $V(e) = \{w;\ y \xrightarrow[T]{*} w\}$ of nodes reachable from y by tree edges, all back edges starting in a node in $V(e)$ and ending in a node on cycle C (which is then an ancestor of x), and the tree path from the lowest attachment of $S(e)$ to cycle C to node y.

Example: In Figure 104 the cycle C consists of the tree path from node 1 to node 9 and the back edge $(9, 1)$. The four edges $(9, 10)$, $(7, 5)$, $(7, 13)$ and $(6, 4)$ emanate from the cycle. The segment $S((9, 10))$ consists of the subgraph spanned by $\{10, 11, 12\}$, the back edges $(11, 8)$, $(11, 7)$ and $(12, 5)$, and the tree path from 5 to 10. The segment $S((9, 10))$ is attached to the cycle in the nodes 9, 8, 7 and 5. ∎

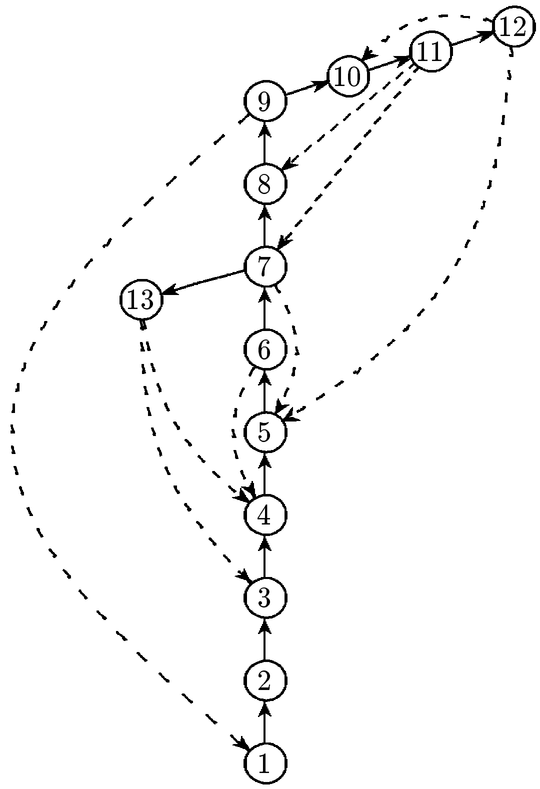


Figure 104. A *dfs*-tree of a planar graph

We test the planarity of G in a two step process. In the first step we test whether $C + S(e)$, the graph consisting of cycle C and segment $S(e)$, is planar for every edge e emanating from cycle C. This is equivalent to testing whether the segment $S(e)$ has a strongly planar embedding, i.e., an embedding where all attachments of $S(e)$ to the cycle C lie on the boundary of the outer face. In order to test the strong planarity of $S(e)$ we will use the algorithm recursively. Suppose now that the segments $S(e)$ are all strongly planar. We then try in a second step to merge the embeddings found in step one. The merging process has to decide for each segment $S(e)$ whether it should be placed inside or outside the cycle C. For this purpose, it only needs to take into account the set of attachments of the different segments emanating from C and their interaction. In our example, the segments $S((7,5))$ and $S((6,4))$ have to be embedded on different sides of C because these segments "interlace".

We will next describe the theory behind both steps in detail. With an edge $e = (x,y)$ we associate a **cycle** $C(e)$ and a **segment** $S(e)$ as follows. If e is a back edge then $C(e)$ and $S(e)$ consist of the tree path from y to x and the edge e. If e is a tree edge then let $V(e) = \{w;\ y \xrightarrow[T]{*} w\}$ be the set of tree successors of y and let $lowpt[y] = \min\{z;\ (w,z) \text{ is a back edge and } w \in V(e)\}$ be the lowest endpoint of a back edge starting in $V(e)$. The cycle $C(e)$ consists of a tree path from $lowpt[y]$ to w, where $w \in V(e)$ and $(w, lowpt[y]) \in B$ and such a back edge. The segment $S(e)$ consists of $C(e)$, the subgraph spanned by $V(e)$ and all back edges starting in a node in $V(e)$. Note that the segment $S(e)$ is uniquely defined but that there may be several choices for the cycle $C(e)$. We divide the tree path underlying the cycle $C(e)$ into two parts, its stem and its spine. The **stem** consists of the part ending in x. The **spine** is empty if e is a back edge and it is the part starting in y if e is a tree edge.

In our example, the cycle $C((9, 10))$ consists of the tree path from 5 to 12 followed by the back edge $(12, 5)$. The stem is the tree path from 5 to 9 and the spine is the tree path from 10 to 12. The cycle $C((1, 2))$ consists of the tree path from 1 to 9 and the back edge $(9, 1)$. Its stem is the node 1.

A segment $S(e)$ is called **strongly planar** if there is an embedding of $S(e)$ such that the stem of the cycle $C(e)$ borders the outer face. An embedding with this property is called a strongly planar embedding of $S(e)$. Let $w_0, w_1, \ldots, w_r$ with $e = (w_r, y)$ be the stem of $C(e)$. A strongly planar embedding of $S(e)$ is called **canonical (reversed canonical)** if for all i, $0 < i < r$, the edge $\{w_i, w_{i+1}\}$ immediately follows (precedes) the edge $\{w_i, w_{i-1}\}$ in the counterclockwise ordering of edges incident to w_i. Note that every strongly planar embedding is either canonical or reversed canonical.

In Figure 104 the embeddings of segments $S((9, 10))$ and $S((7, 13))$ are both strongly planar, the embedding of $S((7, 13))$ is canonical and the embedding of $S((9, 10))$ is reversed canonical.

Lemma 5. *Let G be a biconnected graph and let e be the unique tree edge starting in the root of the dfs-tree. Then $S(e) = G$ and G is planar iff $S(e)$ is strongly planar.*

Proof: Let $e = (1, 2)$ be the unique tree edge incident to node 1. Then $V(e) = \{2, \ldots, n\}$ and hence $S(e) = G$. Also, the stem of $C(e)$ consists only of vertex 1 and hence $S(e)$ is strongly planar iff it is planar by Lemma 1. ∎

Lemma 5 shows that we can confine ourselves to a test of strong planarity. Now let e_0 be an edge and $C = C(e_0)$ be the cycle associated with e_0. An edge $e = (x, y)$ is said to **emanate** from C if x lies on the spine of C but e does not belong to C. Clearly, if e emanates from $C(e_0)$ then the stem of $C(e)$ is part of the tree path underlying $C(e_0)$ and $S(e)$ is a subgraph of $S(e_0)$. Also, $S(e_0)$ is the union of $C(e_0)$ and the segments $S(e)$, where e emanates from $C(e_0)$. The basis of step 1 of the planarity algorithm is the following

Lemma 6. *Let $C = C(e_0)$ be a cycle and let e emanate from C. Then $C + S(e)$ is planar iff $S(e)$ is strongly planar.*

Proof: "$\Rightarrow$": Consider any embedding of $C + S(e)$. The cycle C divides the plane into a bounded and an unbounded region. We may assume w.l.o.g. that the edge $e = (x, y)$ lies in the bounded region. Hence all nodes in $V(e)$ must lie in the bounded region since every node in $V(e)$ is reachable from y without passing through a node of C. If we remove the part of cycle C between x and $lowpt[y]$ then we have the desired strongly planar embedding of $S(e)$.

"$\Leftarrow$": Given a strongly planar embedding of $S(e)$ we can clearly add the missing part of C to obtain an embedding of $C + S(e)$. ∎

For step 2 of the algorithm we need the concepts of attachments and interlacing. Let $C = C(e_0)$ and let $e = (x, y)$ emanate from C. The set $A(e)$ of **attachments** of segment $S(e)$ to cycle C is defined to be the set $\{x, y\}$ if e is a back edge and the set $\{x\} \cup \{z;\ (w, z) \text{ is a back edge, } w \in V(e) \text{ and } z \notin V(e)\}$ if e is a tree edge. Two segments $S(e)$ and $S(e')$ where e and e' emanate from C are said to **interlace** if either there are nodes $x < y < z < u$ on cycle C such that $x, z \in A(e)$ and $y, u \in A(e')$ or $A(e)$ and $A(e')$ have three points in common (cf. Fig. 105; note that the segments shown may have further attachments).

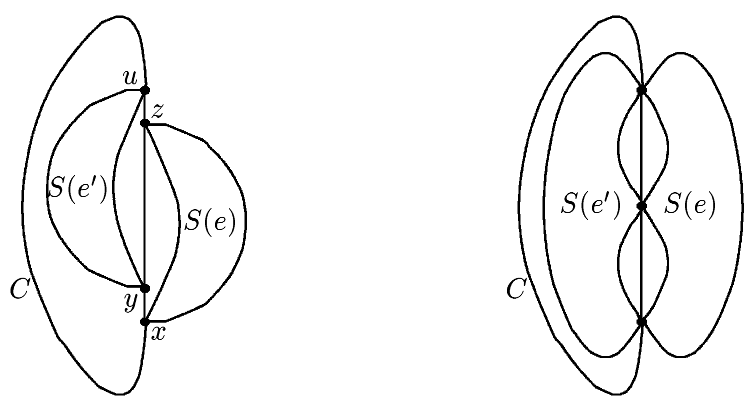


Figure 105. Interlacing segments

Clearly, interlacing segments cannot be embedded on the same side of C. The **interlacing graph** $IG(C)$ with respect to cycle C is defined as follows: The nodes of $IG(C)$ are the segments $S(e)$ where e emanates from C. Also, $S(e)$ and $S(e')$ are connected by an edge iff $S(e)$ and $S(e')$ interlace. The interlacing graph for the cycle $C((1,2))$ of Figure 104 is shown in Figure 106. This graph is bipartite with segments S_1 and S_3 forming one of the sides of the bipartite graph. Note also that the planar embedding of the graph of Figure 104 has S_1 and S_3 on one side of C and S_2 and S_4 on the other side of C.

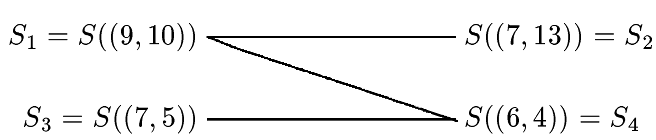


Figure 106. Interlacing graph

Lemma 7. *Let e_0 be a tree edge, let $C = C(e_0) = w_0 \underset{T}{\rightarrow} w_1 \underset{T}{\rightarrow} \cdots \underset{T}{\rightarrow} w_k \underset{B}{\rightarrow} w_0$ and let $e_0 = (w_r, w_{r+1})$. Let $e_1, \ldots, e_m$ be the edges leaving the spine of C, i.e.,* *they leave the cycle in nodes w_j, $r < j \le k$. Then $S(e_0)$ is planar iff $S(e_i)$ is strongly planar for every i, $1 \le i \le m$, and $IG(C)$ is bipartite, i.e., there is a partition L, R of $\{S(e_1), \ldots, S(e_m)\}$ such that no two segments in L resp. R interlace. Moreover, segment $S(e_0)$ is strongly planar iff in addition for every connected component B of $IG(C)$: either $\{w_1, \ldots, w_{r-1}\} \cap \bigcup_{S(e) \in B \cap L} A(e) = \emptyset$ or $\{w_1, \ldots, w_{r-1}\} \cap \bigcup_{S(e) \in B \cap R} A(e) = \emptyset$.*

Proof: "$\Rightarrow$": Note first that $S(e_0) = C + S(e_1) + \cdots + S(e_m)$. Hence, if $S(e_0)$ is planar then $C + S(e_i)$, $1 \le i \le m$, is planar and hence $S(e_i)$ is strongly planar by Lemma 6. Consider any planar embedding of $S(e_0)$. Let $L = \{S(e_i);\ S(e_i)$ is embedded inside cycle C, $1 \le i \le m\}$ and let R be the remaining segments. Then no two segments in L resp. R interlace because interlacing segments have to be embedded on different sides of C. Hence $IG(C)$ is bipartite. Finally, assume that $S(e_0)$ is strongly planar. Consider any strongly planar embedding of $S(e_0)$, i.e., tree path $w_0 \to w_1 \to w_2 \to \cdots \to w_r$ borders the outer face. Then no segment $S(e_i)$, $1 \le i \le m$, which is embedded outside C can have an attachment in $\{w_1, \ldots, w_{r-1}\}$ and hence $\{w_1, \ldots, w_{r-1}\} \cap \bigcup_{S(e) \in R} A(e) = \emptyset$.

"$\Leftarrow$": The proof of this direction is postponed. It will be given in Lemma 9. ▮

Lemma 7 suggests an algorithm for testing strong planarity. In order to test strong planarity of a segment $S(e_0)$, test strong planarity of the segments $S(e_i)$, $1 \le i \le m$, construct the interlacing graph and test for the conditions stated in Lemma 7. Unfortunately, the size of the interlacing graph might be quadratic and therefore we cannot afford to construct the interlacing graph explicitly. Rather, we compute the connected components (and their partition into left and right side) of $IG(C)$ and an embedding of $S(e_0) = C + S(e_1) + \cdots + S(e_m)$ by considering segment by segment. We start with cycle C and then try to add segment by segment. We will consider the segments $S(e_1), \ldots, S(e_m)$ in an order such that adding a canonical embedding of $S(e_{i+1})$ to an embedding of $C + S(e_1) + \cdots + S(e_i)$ can always be achieved (if at all) in a particularly simple way, namely by moving some of the $S(e_l)$, $l \le i$, to the other side of C and then adding $S(e_{i+1})$ inside C and close to the tree path underlying C, cf. Figure 111. In that figure the segment $S(e_{i+1})$ emanates from w_j, $e_{i+1} = (w_j, y)$ and $z = \min A(e_{i+1})$ is the lowest attachment of $S(e_{i+1})$. Also, there is a face F inside C such that the tree path from z to w_j is on the boundary of F. Clearly, a canonical embedding of $S(e_{i+1})$ can be added inside F to the embedding of $C + S(e_1) + \cdots + S(e_i)$ in this case.

In order to follow this embedding strategy we should first consider all segments emanating from w_k, then all segments emanating from w_{k-1}, … . For any node w_j we consider the segments emanating from w_j in the order of lowest attachment, considering the segments with lower attachment first. Among the segments emanating from w_j and having the same lowest attachment, say w_i with $i < j$, we first consider the segments having only w_i and w_j as attachments and then all the others (there can be at most two segments of the latter kind because any two such segments interlace). We will now show how to compute this ordering on the edges emanating from C. We do so by showing how to reorder the adjacency list of each node such that the order of the adjacency list corresponds to the order defined above. For every node v let

$$lowpt[v] = \min(\{v\} \cup \{z;\ v \xrightarrow[T]{*} w \xrightarrow[B]{} z \text{ for some } w \in V\}), \quad \text{and}$$

$$lowpt2[v] = \min(\{v\} \cup \{z;\ v \xrightarrow[T]{*} w \xrightarrow[B]{} z \text{ for some } w \in V \text{ and } z \neq lowpt[v]\}).$$

$lowpt[v]$ is the lowest node reachable from v by a sequence of tree edges followed by one back edge. Since G is assumed to be biconnected we have $lowpt[v] < v$ for all $v \neq 1$. $lowpt2[v]$ is the second lowest node reachable from v in this way, if there is one. The default value for both functions is v. The functions *lowpt* and *lowpt2* are easily computed during *dfs* since

$$lowpt[v] = \min(\{v\} \cup \{z;\ (v,z) \in B\} \cup \{lowpt[w];\ (v,w) \in T\})$$

and

$$\begin{aligned} lowpt2[v] = \min(\{v\} \cup \{z;\ (v,z) \in B \text{ and } z \neq lowpt[v]\} \\ \cup \{lowpt[w];\ (v,w) \in T, lowpt[w] \neq lowpt[v]\} \\ \cup \{lowpt2[w];\ (v,w) \in T\}). \end{aligned}$$

These equations suggest to compute *lowpt* and *lowpt2* by two separate applications of *dfs*. In the first application of *dfs* one computes *lowpt* and in the second application one computes *lowpt2* using *lowpt*. We leave it to the reader to show that one *dfs* suffices to compute both functions. For an edge $e = (w_j, y)$ let

$$lowpt[e] = \textbf{if } e \in B \textbf{ then } y \textbf{ else } lowpt[y] \textbf{ fi}.$$

Then $lowpt[e] = \min A(e)$ and $|A(e)| \geq 3$ iff $e \in T$ and $lowpt2[y] < w_j$ for any edge $e = (w_j, y)$ emanating from the cycle C. We want to reorder the adjacency list of w_j such that an edge $e = (w_j, y)$ is before an edge $e' = (w_j, y')$ if either $lowpt[e] < lowpt[e']$ or $lowpt[e] = lowpt[e']$ and $|A(e)| = 2$ and $|A(e')| \geq 3$. Let $c : E \to \mathbb{N}$ be defined by

$$c((v,w)) = \begin{cases} 2 \cdot w & \text{if } (v,w) \in B; \\ 2 \cdot lowpt[w] & \text{if } (v,w) \in T \text{ and } lowpt2[w] \geq v; \\ 2 \cdot lowpt[w] + 1 & \text{if } (v,w) \in T \text{ and } lowpt2[w] < v. \end{cases}$$

Then reordering an adjacency list according to non-decreasing values of c yields the desired ordering of outgoing edges. We can do the reordering in linear time by bucket sort. Have $2n$ initially empty buckets. Step through the edges of G one by one and throw edge (v,w) into bucket $c((v,w))$. After having done so we go through the buckets in decreasing order. When edge (v,w) is encountered we add (v,w) to the front of v's adjacency list.

In our example, the edges out of node 7 are ordered $(7,8),(7,13),(7,5)$ and the edges out of node 11 are ordered $(11,12),(11,7),(11,8)$.

From now on, we assume that adjacency lists are reordered in the way described above. The reordering has the additional property that a cycle $C(e_0)$ for a tree edge $e_0 = (x,y)$ is very easy to find. We start at node y and construct a path by always taking the first edge out of each node until a back edge is encountered. This path is a spine of $C(e_0)$, as is easily verified.

We now resume the discussion of how to deal with the interlacing graph. As in Lemma 7, $C = C(e_0)$,

$$C = w_0 \underset{T}{\rightarrow} w_1 \underset{T}{\rightarrow} \cdots \underset{T}{\rightarrow} w_k \underset{B}{\rightarrow} w_0$$

and $e_0 = (w_r, w_{r+1})$ for some r. Let $e_1, \ldots, e_m$ be the edges leaving the spine of C *in order*, i.e., the edges leaving w_k are considered first and for each w_j the edges are ordered as described above. Let $IG_i(C)$ be the subgraph of $IG(C)$ spanned by $S(e_1), \ldots, S(e_i)$. If $IG_i(C)$ is non-bipartite then so is $IG(C)$ and hence $S(e_0)$ is not strongly planar. If $IG_i(C)$ is bipartite then every connected component (= block) of $IG_i(C)$ is. If B is a block of $IG_i(C)$ then we use LB, RB to denote the partition of B induced by the bipartite graph.

Our next goal is to describe how the blocks of $IG_{i+1}(C)$ can be obtained from the blocks of $IG_i(C)$. Let $e_{i+1} = (w_j, y)$. For every block B of $IG_i(C)$ let

$$ALB = \{w_h;\ 0 \le h < j \text{ and } w_h \in A(e) \text{ for some } S(e) \in LB\}$$

be the set of attachments (below w_j) of segments in LB. ARB is defined similarly.

Lemma 8. *If $IG_i(C)$ is bipartite, then:*

a) *There is some ordering of the blocks of $IG_i(C)$, say $B_1, B_2, \ldots, B_h, B_{h+1}, \ldots$ such that*

$$\max(ALB_l \cup ARB_l) \le \min(ALB_{l+1} \cup ARB_{l+1})$$

for $1 \le l < h$ and $ALB_l = ARB_l = \emptyset$ for $l > h$.

b) *$IG_{i+1}(C)$ is bipartite iff for all l, $1 \le l \le h$, either $\max ALB_l \le \min A(e_{i+1})$ or $\max ARB_l \le \min A(e_{i+1})$.*

c) *If $IG_{i+1}(C)$ is bipartite then the blocks of $IG_{i+1}(C)$ can be obtained as follows: Assume w.l.o.g. that $\max ALB_l \le \min A(e_{i+1})$ for all l. (This can always be achieved by interchanging LB and RB for some blocks B.) Let $d = \min(\{l;\ \max ARB_l > \min A(e_{i+1})\} \cup \{h+1\})$. Then the blocks of $IG_{i+1}(C)$ are $B_1, \ldots, B_{d-1}, B_d \cup \cdots \cup B_h \cup \{S(e_{i+1})\}, B_{h+1}, \ldots$.*

d) *If $IG_{i+1}(C)$ is bipartite and $S(e_l)$, $1 \le l \le i+1$, are strongly planar then there is a planar embedding of $C + S(e_1) + \cdots + S(e_{i+1})$ such that all segments in $\bigcup_l LB_l$ are embedded inside C and all segments in $\bigcup_l RB_l$ are embedded outside C.*

Proof: We use induction on i. For $i = 0$ little remains to be shown. $IG_0(C)$ is empty and $IG_1(C)$ consists of a single node. This shows a), b) and c). For part d) we only have to observe that $S(e_1)$ can be embedded inside as well as outside C, if $S(e_1)$ is strongly planar.

So let us turn to the case $i > 0$. We will show parts b), c), a) and d) in this order.

b) "$\Rightarrow$": Note first that it suffices to show the following

Claim 1. *If* $\max ALB_l > \min A(e_{i+1})$ *for some* l *then there is a segment* $S(e) \in LB_l$ *such that* $S(e)$ *and* $S(e_{i+1})$ *interlace.*

Suppose that we have shown Claim 1. If there were l, $1 \le l \le h$, such that $\max ALB_l > \min A(e_{i+1})$ and $\max ARB_l > \min A(e_{i+1})$ then $S(e_{i+1})$ interlaces with a segment $S(e) \in LB_l$ and a segment $S(e') \in RB_l$ by Claim 1. Since $S(e)$ and $S(e')$ belong to the same block there is a path from $S(e)$ to $S(e')$ in $IG_i(C)$. Since $IG_i(C)$ is bipartite this path necessarily has odd length. Together with edges $\{S(e), S(e_{i+1})\}$ and $\{S(e_{i+1}), S(e')\}$ we obtain an odd length cycle in $IG_{i+1}(C)$. Hence $IG_{i+1}(C)$ is non-bipartite, a contradiction. We still have to show Claim 1.

Proof of Claim 1: Let $z = \min A(e_{i+1})$. Since $\max ALB_l > z$ there must be a segment $S(e) \in LB_l$ such that $w \in A(e)$ for some w with $z \underset{T}{\overset{+}{\rightarrow}} w \underset{T}{\overset{+}{\rightarrow}} w_j$. Edge e emanates from node w_p for some $p \ge j$.

Case 1: $p > j$.
Then $z \underset{T}{\overset{+}{\rightarrow}} w \underset{T}{\overset{+}{\rightarrow}} w_j \underset{T}{\overset{+}{\rightarrow}} w_p$, $z, w_j \in A(e_{i+1})$ and $w, w_p \in A(e)$. Hence segments $S(e)$ and $S(e_{i+1})$ interlace (cf. Figure 107).

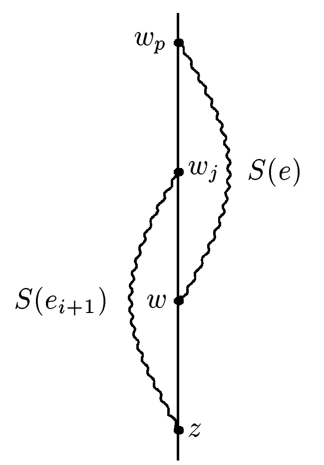


Figure 107. Case 1

Case 2: $p = j$.
Let $e = (w_j, u)$. Since e is considered before e_{i+1} and hence $\min A(e) \le z$, edge e cannot be a back edge. (If it were a back edge then $\min A(e) = u = w > z$, a contradiction.) Hence e is a tree edge and $\min A(e) = lowpt[u]$.

Case 2.1: $lowpt[u] < z$.
Then $lowpt[u] \xrightarrow[T]{+} z \xrightarrow[T]{+} w \xrightarrow[T]{+} w_j$, $lowpt[u], w \in A(e)$ and $z, w_j \in A(e_{i+1})$. Hence segments $S(e)$ and $S(e_{i+1})$ interlace (cf. Fig. 108).

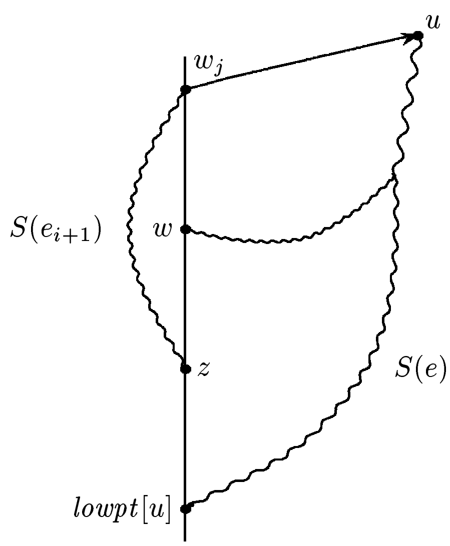


Figure 108. Case 2.1

Case 2.2: $lowpt[u] = z$.
Since $w \in A(e)$ we have $lowpt2[u] < w_j$. Since e is considered before e_{i+1} we must have $|A(e_{i+1})| \geq 3$, and hence edge e_{i+1} cannot be a back edge. Rather, it must be a tree edge and we must have $lowpt2[y] < w_j$. If $lowpt2[y] \neq lowpt2[u]$, say $lowpt2[y] \xrightarrow[T]{+} lowpt2[u]$, then we have $z \xrightarrow[T]{+} lowpt2[y] \xrightarrow[T]{+} lowpt2[u] \xrightarrow[T]{+} w_j$, $z, lowpt2[u] \in A(e)$, and $lowpt2[y], w_j \in A(e_{i+1})$. Hence $S(e_{i+1})$ and $S(e)$ interlace (cf. Fig. 109). If $lowpt2[y] = lowpt2[u]$ then $A(e)$ and $A(e_{i+1})$ have three points in common and hence $S(e_i)$ and $S(e_{i+1})$ interlace (cf. Figure 110). ∎

"$\Leftarrow$": Assume now that $\max ALB_l \leq \min A(e_{i+1})$ or $\max ARB_l \leq \min A(e_{i+1})$ for all l, $1 \leq l \leq h$. By interchanging LB_l and RB_l, if necessary, we can achieve that $\max ALB_l \leq \min A(e_{i+1})$ for all l, $1 \leq l \leq h$.

Claim 2. *Let $S(e) \in \bigcup_l LB_l$ be arbitrary. Then $S(e)$ and $S(e_{i+1})$ do not interlace.*

Proof: $A(e_{i+1}) \subseteq \{w;\ \min A(e_{i+1}) \xrightarrow[T]{*} w \xrightarrow[T]{*} w_j\}$ and $A(e) \subseteq \{w;\ w \xrightarrow[T]{*} \min A(e_{i+1})$ or $w_j \xrightarrow[T]{*} w\}$. Hence $S(e)$ and $S(e_{i+1})$ do not interlace. ∎

The bipartiteness of $IG_{i+1}(C)$ now follows from Claim 2 because it is safe to add $S(e_{i+1})$ to the "left side" of the interlacing graph.

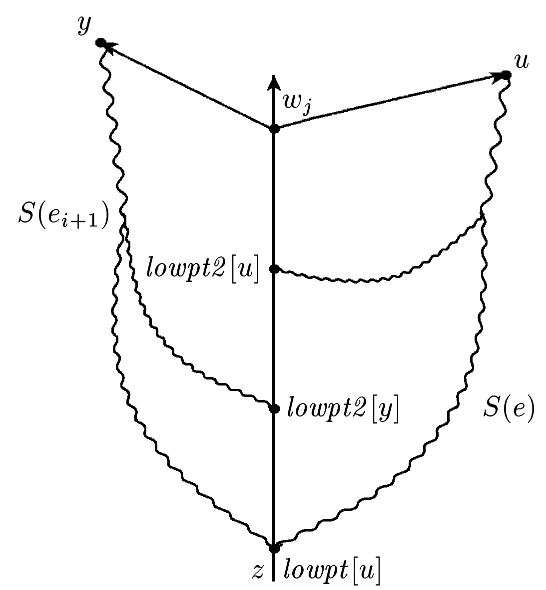


Figure 109. Case 2.2, $lowpt2[y] \neq lowpt2[u]$

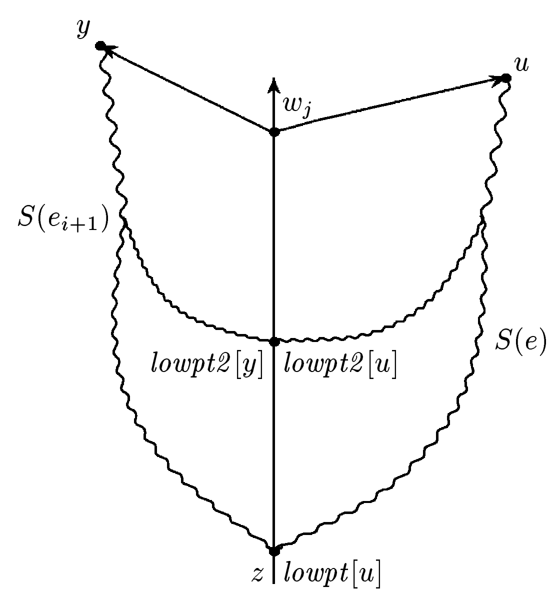


Figure 110. Case 2.2, $lowpt2[y] = lowpt2[u]$

c) Assume that $IG_{i+1}(C)$ is bipartite. Then for all l, $1 \le l \le h$, $\max ALB_l \le \min A(e_{i+1})$ or $\max ARB_l \le \min A(e_{i+1})$ by part b). By interchanging LB_l and RB_l, if necessary, we can achieve $\max ALB_l \le \min A(e_{i+1})$ for all l, $1 \le l \le h$. Let $d = \min(\{l;\ \max ARB_l > \min A(e_{i+1})\} \cup \{h+1\})$.

Claim 3. *For all l: There is a segment $S(e) \in RB_l$ such that $S(e)$ and $S(e_{i+1})$ interlace iff $d \le l \le h$.*

Proof: "$\Leftarrow$": Let $d \le l \le h$. Then

$$\begin{aligned}\min A(e_{i+1}) &< \max ARB_d && \text{[by definition of } d]\\ &\le \max ARB_l && \text{[by induction hypothesis, part a) and } d \le l]\\ &< w_j && \text{[since } l \le h]\end{aligned}$$

and hence there is a segment $S(e) \in RB_l$ such that $S(e)$ and $S(e_{i+1})$ interlace by Claim 1.

"$\Rightarrow$": (Indirect.) Let $l < d$ or $l > h$ and let $S(e) \in RB_l$. Then $A(e) \subseteq \{w;\ w_j \overset{*}{\underset{T}{\rightarrow}} w\}$ if $l > h$ and $A(e) \subseteq \{w;\ w_j \overset{*}{\underset{T}{\rightarrow}} w$ or $w \overset{*}{\underset{T}{\rightarrow}} \min A(e_{i+1})\}$ if $l < d$. The former inclusion follows from the definition of h, the latter inclusion follows from the definition of d, and part a) of the induction hypothesis. Also $A(e_{i+1}) \subseteq \{w;\ \min A(e_{i+1}) \overset{*}{\underset{T}{\rightarrow}} w \overset{*}{\underset{T}{\rightarrow}} w_j\}$ and hence $S(e)$ and $S(e_{i+1})$ do not interlace. ∎

We conclude from Claims 2 and 3 that $S(e_{i+1})$ is connected to segments in blocks $B_d, \ldots, B_h$. Hence the blocks of $IG_{i+1}(C)$ are $B_1, \ldots, B_{d-1}, B_d \cup \cdots \cup B_h \cup \{S(e_{i+1})\}, B_{h+1}, \ldots$. Let $B = B_d \cup \cdots \cup B_h \cup \{S(e_{i+1})\}$ be the new block. Then B can be partitioned into LB and RB where $LB = \bigcup_{d \le l \le h} LB_l \cup \{S(e_{i+1})\}$ and $RB = \bigcup_{d \le l \le h} RB_l$. Moreover, if $d \le h$, $\max ARB_d \le \min ARB_{d+1} \le \max ARB_{d+1} \le \cdots \le \min ARB_h \le \max ARB_h$ by part a) and $\max ALB_d \le \min ALB_{d+1} \le \max ALB_{d+1} \le \cdots \le \min ALB_h \le \max ALB_h \le \min A(e_{i+1})$ by part a) and the assumption that $\max ALB_l \le \min A(e_{i+1})$ for all l, $1 \le l \le h$.

a) Follows immediately from part c). The ordering of the blocks of $IG_{i+1}(C)$ given in part c) satisfies the conditions required in part a). This follows immediately from the discussion completing the proof of part c).

d) Assume that $IG_{i+1}(C)$ is bipartite and that $S(e_l)$, $1 \le l \le i+1$, are strongly planar. Let B'_1, B'_2, … be the blocks of $IG_{i+1}(C)$. By part c) we have $B'_1 = B_1$, …, $B'_{d-1} = B_{d-1}$, $B'_d = B_d \cup \cdots \cup B_h \cup \{S(e_{i+1})\}$, $B'_{d+1} = B_{h+1}$, …, where B_1, B_2, … are the blocks of $IG_i(C)$. Moreover, $LB'_l = LB_l$, $RB'_l = RB_l$ for $l < d$, $LB'_{d+l} = LB_{h+l}$, $RB'_{d+l} = RB_{h+l}$ for $l \ge 1$ and $LB'_d = \bigcup_{d \le l \le h} LB_l \cup \{S(e_{i+1})\}$ and $RB'_d = \bigcup_{d \le l \le h} RB_l$. By induction hypothesis there is a planar embedding of $C + S(e_1) + \cdots + S(e_i)$ such that all segments in $\bigcup_l LB_l$ are embedded inside C and all segments in $\bigcup_l RB_l$ are embedded outside C. By the proof of Claim 2 no segment $S(e) \in \bigcup_l LB_l$ has an attachment w which lies strictly between $\min A(e_{i+1})$ and w_j. Thus there is a face F inside C such that the tree path from $\min A(e_{i+1})$ to w_j is part of the boundary of F. All attachments of $S(e_{i+1})$ lie between $\min A(e_{i+1})$ and w_j inclusively. Moreover, $S(e_{i+1})$ is strongly planar and hence there is a planar embedding of $S(e_{i+1})$ where the tree path from $\min A(e_{i+1})$ to w_j borders the outer face. We can add this embedding to the embedding of $C + S(e_1) + \cdots + S(e_i)$ by putting it inside face F. In this way we obtain a planar embedding of $C + S(e_1) + \cdots + S(e_{i+1})$ (cf. Fig. 111). This completes the proof of Lemma 8. ∎

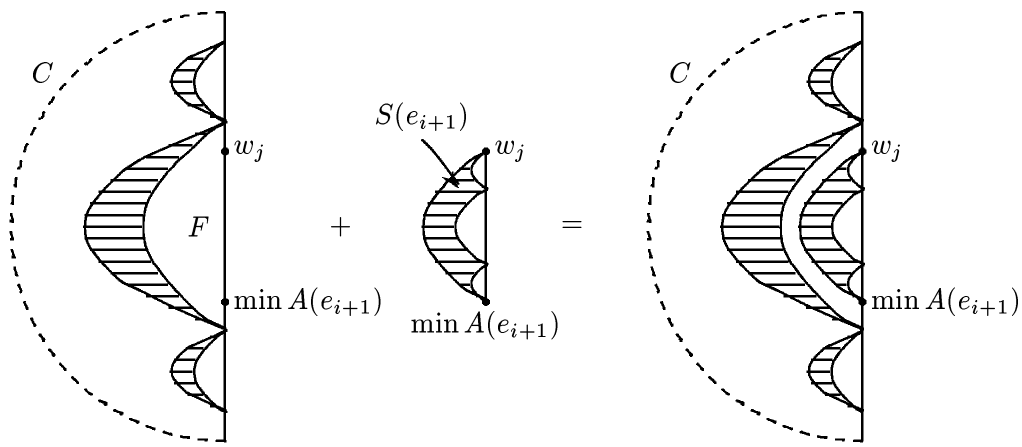


Figure 111. Addition of $S(e_{i+1})$ inside F

Lemma 9. *The if-part of Lemma 7 holds.*

Proof: If $IG(C)$ is bipartite and $S(e_i)$, $1 \le i \le m$, is strongly planar then by Lemma 8d) there is an embedding of $C + S(e_1) + \cdots + S(e_m) = S(e_0)$ such that all segments in $\bigcup_i LB_i$ are embedded inside C and all segments in $\bigcup_i RB_i$ are embedded outside C. In particular, $S(e_0)$ is planar. Assume now that in addition $ALB_l \cap \{w_1, \ldots, w_{r-1}\} = \emptyset$ or $ARB_l \cap \{w_1, \ldots, w_{r-1}\} = \emptyset$ for all l where ALB_l and ARB_l are defined with $j = r + 1$, i.e., when all edges $e_1, \ldots, e_m$ are embedded. We may assume w.l.o.g. (by interchanging L and R for some blocks) that $ARB_l \cap \{w_1, \ldots, w_{r-1}\} = \emptyset$ for all l. Thus outside C there are no attachments to nodes $w_1, \ldots, w_{r-1}$ and hence there is a face F outside C such that the stem $w_0, \ldots, w_r$ of $S(e_0)$ borders F. Lemma 1 allows us to turn F into the outer face and yields a canonical embedding of $S(e_0)$. ∎

We illustrate Lemma 9 on our example. Let C be the cycle which runs from node 1 to node 9 along tree edges and then back to node 1. There are four segments emanating from this cycle: $S_1 = S((9,10))$, $S_2 = S((7,13))$, $S_3 = S((7,5))$ and $S_4 = S((6,4))$. All four segments are strongly planar. When segment $S_2 = S((7,13))$ is considered, we have: $IG_1(C)$ has one block B_1 consisting of segment S_1. Say S_1 belongs to RB_1. Then $ALB_1 = \emptyset$ and $ARB_1 = \{5\}$. Lemma 8b) is satisfied and hence $IG_2(C)$ is bipartite. We have $d = 1$ in Lemma 8c) and hence $IG_2(C)$ has only block B_1, say $LB_1 = \{S_2\}$ and $RB_1 = \{S_1\}$. Then $ALB_1 = \{3,4\}$ and $ARB_1 = \{5\}$ when S_3 is considered. $IG_3(C)$ is bipartite and has two blocks B_1 and B_2, say $LB_1 = \{S_2\}$, $RB_1 = \{S_1\}$, $RB_2 = \{S_3\}$. Then $ALB_1 = \{3,4\}$, $ARB_1 = \{5\}$, $ARB_2 = \{5\}$, $ALB_2 = \emptyset$ when S_4 is considered. S_4 forces us to merge blocks B_1 and B_2, i.e., $d = 1$ in Lemma 8c), and hence $IG_4(C)$ has only one block B_1. Moreover $LB_1 = \{S_2, S_4\}$ and $RB_1 = \{S_1, S_3\}$.

It is now easy to derive an efficient way of dealing with the interlacing graph from Lemma 8. Suppose that we processed edges $e_1, \ldots, e_i$ already and want to process edge e_{i+1} next. At this point we keep blocks $B_1, \ldots, B_h$ in a stack S where

h is defined as in Lemma 8a). Also for each l, $1 \le l \le h$, we maintain the multi-sets ALB_l and ARB_l in a doubly linked list. The lists ALB_l and ARB_l are *ordered* according to DFS-numbers. From the stack position corresponding to B_l we have pointers to the front and back end of lists ALB_l and ARB_l. The test for bipartiteness of $IG_{i+1}(C)$ given in Lemma 8b) is now easily implemented by Program 31.

```
l ← h + 1;
while max(ALB_{l−1} ∪ ARB_{l−1}) > lowpt[e_{i+1}]
do if ALB_{l−1} is non-empty and max ALB_{l−1} > lowpt[e_{i+1}]
     then interchange LB_{l−1} and RB_{l−1} fi;
     if ALB_{l−1} is non-empty and max ALB_{l−1} > lowpt[e_{i+1}]
     then IG_{i+1}(C) is not bipartite and hence
               the graph can be declared non-planar fi;
     l ← l − 1
od;
d ← l.
```

Program 31

The running time of Program 31 is clearly $O(h - d + 2)$. Also, it correctly computes d as defined in Lemma 8c). The new blocks of $IG_{i+1}(C)$ are now easily formed by Program 32.

```
ALB ← ARB ← ∅;
for l from d to h
do ALB ← ALB concatenated with ALB_l;
     ARB ← ARB concatenated with ARB_l
od;
ALB ← ALB concatenated with (A(e_{i+1}) − {w_j});
pop B_h, ..., B_d from stack S;
add B to stack S.
```

Program 32

Again the running time of Program 32 is clearly $O(h - d + 2)$ provided we are given $(A(e_{i+1}) - \{w_j\})$. Also, it correctly computes lists ALB and ARB. Note that these lists are ordered according to the remark at the end of the proof of Lemma 9c). We can now give the complete planarity testing algorithm, see Program 33.

Lemma 10. *Program 33 tests strong planarity in linear time and space.*

Proof: Observe first that line (1) determines the spine of cycle $C(e_0)$ in time proportional to the length of the spine. The stem $w_0, \ldots, w_r$ is not explicitly constructed; we only mention it in order to keep the same notation as in Lemmas 7

```
(0)   procedure stronglyplanar(e_0 : edge);
      co tests whether segment S(e_0), e_0 = (x, y), is strongly planar.
         If so, it returns the ordered (according to dfsnum) list of
         attachments of S(e_0) excluding x oc
(1)   find the spine of cycle C(e_0) by starting in node y and always
      taking the first edge on every adjacency list until a back edge is
      encountered. This back edge leads to node w_0 = lowpt[y].
      Let w_0, ..., w_r be the tree path from w_0 to x = w_r and
      and let w_{r+1} = y, ..., w_k be the spine constructed above;
(2)   let S be an empty stack of blocks;
(3)   for j from k downto r + 1
(4)   do for all edges e' (except the first) emanating from w_j
(5)      do stronglyplanar(e');
(6)         let A(e') be the ordered list of attachments of S(e')
            as returned by the successful call stronglyplanar(e');
(7)         update stack S as described in Programs 31 and 32
(8)      od;
(9)      let B_h be the top entry in stack S;
(10)     while max(ALB_h ∪ ARB_h) = w_{j-1}
(11)     do remove node w_{j-1} from ALB_h and ARB_h;
(12)        if ALB_h and ARB_h become empty
(13)        then pop B_h from the stack; h ← h - 1 fi
(14)     od
(15)  od;
      co if control reaches this point then IG(C) is bipartite.
         We will now test for strong planarity and compute A(e_0) oc
(16)  L ← ∅;      co an empty list oc
(17)  for l from 1 to h
(18)  do if max ALB_l ≥ w_1 and max ARB_l ≥ w_1
(19)     then declare S(e_0) not strongly planar and stop fi;
(20)     if ALB_l ≠ ∅ and max ALB_l ≥ w_1
(21)     then L ← L conc ARB_l conc ALB_l
(22)     else L ← L conc ALB_l conc ARB_l fi
(23)  od;
(24)  return L
(25)  end.
```

Program 33

and 9. Next we argue that bipartiteness of $IG(C)$ is tested correctly. The correctness of loop (4)–(8) is obvious from the discussions above. Suppose now that we processed all edges emanating from w_j. In order to prepare for processing the edges emanating from w_{j-1} we only have to delete all occurrences of w_{j-1} on lists ALB_l and ARB_l. This is done in lines (9)–(14). Note that all occurrences of w_{j-1}

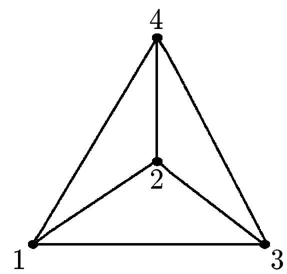
must be in the top entries of stack S by Lemma 8a). Hence lines (9)–(14) work correctly. When control reaches line (16) the interlacing graph $IG(C)$ is bipartite and hence $S(e_0)$ is planar. Moreover, for every block B in the stack S the lists ALB and ARB contain exactly the attachments below w_r of segments in the block. In line (18) we now test the condition for strong planarity given in Lemma 7. It states that for all blocks B of $IG(C)$ either $\{w_1, \ldots, w_{r-1}\} \cap \bigcup_{S(e) \in LB} A(e) = \emptyset$ or $\{w_1, \ldots, w_{r-1}\} \cap \bigcup_{S(e) \in RB} A(e) = \emptyset$. Of course, we can always interchange L and R such that the former is the case. It remains to argue that lines (20) to (22) correctly compute the ordered set $A(e_0) - \{x\}$ of attachments. Let l_0 be minimal such that $\max(ALB_{l_0} \cup ARB_{l_0}) \geq w_1$. Then $ALB_l \cup ARB_l \subseteq \{w_0\}$ for $l < l_0$ and either $ALB_{l_0} \subseteq \{w_0\}$ or $ARB_{l_0} \subseteq \{w_0\}$ by line (18). Also $\min(ALB_l \cup ARB_l) \geq \max(ALB_{l-1} \cup ARB_{l-1}) \geq \max(ARB_{l_0} \cup ALB_{l_0})$ for $l > l_0$ by Lemma 8a) and hence either $ALB_l = \emptyset$ or $ARB_l = \emptyset$ for $l > l_0$ by line (18). Thus lines (20) to (22) work correctly and the correctness proof is complete.

We still have to analyze the running time. Note first that *stronglyplanar* is called at most once for each edge. Also, each tree edge belongs to exactly one spine. Hence the total time spent in lines (1), (2), (3), (4), (5) (without counting the time spent within recursive calls), (6), (8), (9) and (16) is $O(m)$. Let us look at line (7) next. Observe that line (7) is executed at most once for each edge. Also, at most one block is pushed on stack S in one execution of line (7), and execution time of line (7) is proportional to the number of entries removed from stack S. Since only m elements are added to stacks S altogether, only m elements can be removed and hence the total time spent in line (7) is $O(m)$. The same argument shows that the total time spent in lines (17)–(23) is $O(m)$, because the time spent in these lines is proportional to the number of elements removed from stacks S in these lines. Lines (10)–(14) still remain to be considered. Only endpoints of back edges are placed on lists ALB and ARB. No back edge is placed twice on a list and each back edge is removed at most once. Hence the total cost of lines (10)–(14) is $O(m)$. ∎

Theorem 2. *Let $G = (V, E)$ be a graph. Then planarity of G can be tested in time $O(n)$.*

Proof: If $m > 3n - 6$ then G is non-planar. If $m \leq 3n - 6$ then we can divide G into its biconnected components in time $O(m) = O(n)$. For each biconnected component we can test its planarity in linear time. Also, a graph is planar iff its biconnected components are planar. ∎

At this point we have developed an $O(n)$ algorithm for testing planarity. Suppose now that $G = (V, E)$ is a planar graph. Does a successful planarity test also tell us something about a planar embedding? We show that it does; more specifically, we show how to extend the planarity test such that it computes a combinatorial embedding. Recall that a graph $G = (V, E)$ together with a cyclic ordering σ of the edges incident to any node $v \in V$ is called a **planar map** (or **combinatorial embedding**) if there is a planar embedding of G such that the cyclic ordering σ agrees with the clockwise ordering of the edges in the embedding. Figure 112 shows a planar map (think of the adjacency lists as circular lists) and a corresponding embedding.



1: (1,2), (1,3), (1,4)
2: (2,4), (2,3), (2,1)
3: (3,1), (3,2), (3,4)
4: (4,2), (4,1), (4,3)

Figure 112. Planar map and its embedding

We now show how the planarity testing algorithm can be used to turn a planar graph into a planar map. Let $G = (V, E)$ be a planar graph. Consider an application of the planarity testing algorithm to graph G. Let $C = C(e_0)$ be the cycle associated with some edge e_0 and let $e_1, \ldots, e_m$ be the edges emanating from the spine path. The planarity testing algorithm computes the blocks (and their partition into sides) of $IG(C')$. More precisely, it computes a mapping $\alpha : \{S(e_1), \ldots, S(e_m)\} \to \{L, R\}$ such that no two segments with the same label interlace and such that (cf. Lemmas 8d and 9) there is a canonical embedding of $S(e_0)$ with precisely the segments $S(e_i)$ with $\alpha(S(e_i)) = L$, embedded inside C. The mapping α can be computed as follows. Let B be the block of $IG(C)$ which contains $S(e)$. Our procedure *stronglyplanar* computes B iteratively. The construction of B is certainly completed when B is popped from stack S. Let $\alpha(S(e)) = R$ if $S(e) \in RB$ at that moment and let $\alpha(S(e)) = L$ otherwise. With this extension, algorithm *stronglyplanar* computes mapping α in linear time.

Suppose now that we have computed the mapping α and run algorithm *stronglyplanar* again. We can then avoid all flipping of sides by embedding the segments $S(e_i)$ as prescribed by α. More precisely, when $\alpha(S(e_i)) = L$ then we add a canonical embedding of $S(e_i)$ inside C and when $\alpha(S(e_i)) = R$ then we add a reversed canonical embedding of $S(e_i)$ outside C to the embedding of $C + S(e_1) + \cdots + S(e_{i-1})$. Finally, we turn the obtained embedding of $S(e_0) = C + S(e_1) + \cdots + S(e_m)$ into a canonical embedding of $S(e_0)$ by the construction of Lemma 1 (cf. the proof of Lemma 10). Note that the construction of Lemma 1 does not change the combinatorial embedding.

Similarly, a reversed canonical embedding of $S(e_0)$ can be determined by interchanging the roles of L and R, i.e., if $\alpha(S(e_i)) = L$ then a reversed canonical embedding of $S(e_i)$ is added outside C and if $\alpha(S(e_i)) = R$ then a canonical embedding of $S(e_i)$ is added inside C.

In summary, we have shown that a canonical or reversed canonical embedding of $S(e_0)$ can be constructed by adding appropriate embeddings of the segments $S(e_i)$

as directed by the mapping α. The remaining question to be addressed is whether a canonical or reversed canonical embedding of $S(e_0)$ is needed in the embedding of G. Let S be the set of edges for which *stronglyplanar*(e) is called. Define the type $t(e) \in \{canonical, reversed\_canonical\}$ as follows. If $e = (1,2)$ then $t(e) = canonical$. If $e \in S, e \neq (1,2)$, let $e_0 \in S$ be such that *stronglyplanar*(e) is called by *stronglyplanar*(e_0). Then $t(e) = canonical$ if either $t(e_0) = canonical$ and $\alpha(S(e)) = L$ or $t(e_0) = reversed\_canonical$ and $\alpha(S(e)) = R$, and $t(e) = reversed\_canonical$ otherwise. The significance of the type $t(e)$ of an edge e lies in the fact that the induced embedding of the subgraph $S(e)$ in a canonical embedding of $S((1,2)) = G$ is a $t(e)$ embedding for all edges $e \in S$. It is clear that the types $t(e)$ for $e \in S$ can be computed in linear time.

We can no extend procedure *stronglyplanar* such that it computes a planar map corresponding to a canonical embedding of $S((1,2))$ as follows. We start with an embedding of the cycle $C((1,2))$. A call *stronglyplanar*(e), $e \in S - \{(1,2)\}$ adds the spine of $C(e)$ and the back edge belonging to $C(e)$ to the embedding. Let $w_r, w_{r+1}, \ldots, w_k, w_0)$ be the spine followed by the back edge (w_k, w_0). If $t(e) = canonical$ ($reversed\_canonical$) then the dart (w_r, w_{r+1}) is inserted immediately after (before) the dart $(w_r, parent[w_r])$ and into the clockwise ordering of edges around w_r and the dart (w_0, w_k) is insert immediately before (after) the dart $(w_0, active\_edge[w_0])$ into the clockwise ordering of darts around w_0; also the nodes w_i, $r+1 \leq i \leq k$, and the two incident edges are added to the planar map. Here, *parent* is a precomputed array which contains for each node $w \neq 1$ the parent of w in the tree T, i.e., $parent[w] = v$ iff $(v,w) \in T$, and *active_edge* is an array with $active\_edge[w] = e$ if $e = (v,w) \in T$ and the edge $e' \in S$ for which *stronglyplanar*(e') is currently active starts in $V(e)$, and $active\_edge[v] = \text{nil}$ otherwise. It is clear that the array *parent* can be precomputed in linear time and the array *active_edge* can be maintained in linear time. It is also clear, that adding the path $w_r, w_{r+1}, \ldots, w_k, w_0$ to the planar map takes time proportional to the number of edges added and hence a planar map can be computed in linear time. we summarize in

Theorem 3. *Let $G = (V,E)$ be a planar graph. Then G can be turned into a planar map (G,σ) in linear time.* ∎

In our example we have $S = \{(1,2),(9,10),(12,10),(11,7),(11,8),(7,13),(13,4),(7,5),(6,4)\}$, $\alpha((6,4)) = \alpha((7,13)) = L$, $\alpha((9,10)) = \alpha((7,5)) = R$, $\alpha((12,10)) = R$, $\alpha(11,8)) = \alpha((11,7)) = L$ $\alpha(13,4)) = L$, $t((1,2)) = t((6,4)) = t((7,13)) = t((13,4)) = t((12,10)) = canonical$ and $t((9,10)) = t((7,5)) = t((11,8)) = t((11,7)) = reversed\_canonical$. When edge $(7,5)$ is added to the embedding, we have $active\_edge[5] = (5,6)$ and hence $(7,5)$ is inserted before $(7,6)$ in the order around 7 and $(5,7)$ is inserted after the dart $(5,6)$ in the order around 5. Altogether the planar map shown in Figure 113 is constructed.

A planar map is still a combinatorial object, it is not yet a drawing of a graph. We will now show how to produce drawings, in fact we show how to compute drawings where nodes are mapped into grid points and edges are mapped into

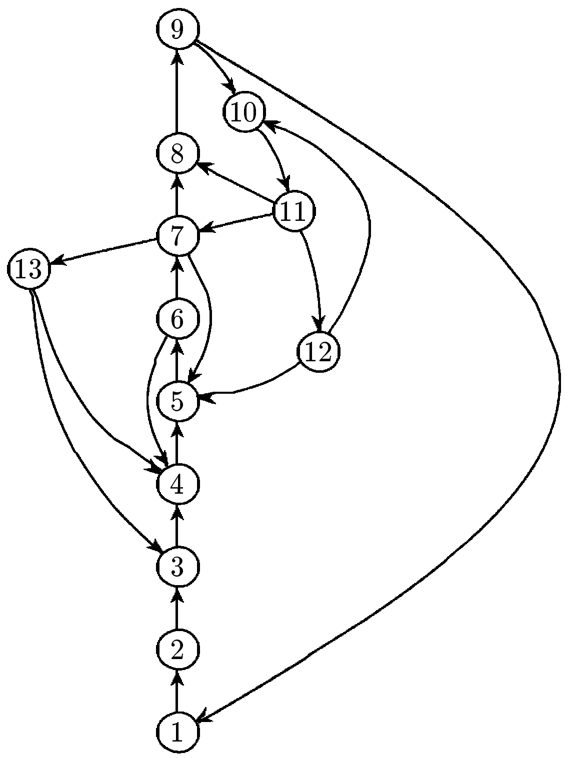


Figure 113. The planar map constructed for the graph of Figure 106

straight-line segments, (cf. Fig. 114). Such an embedding is called a straight-line or Fáry embedding.

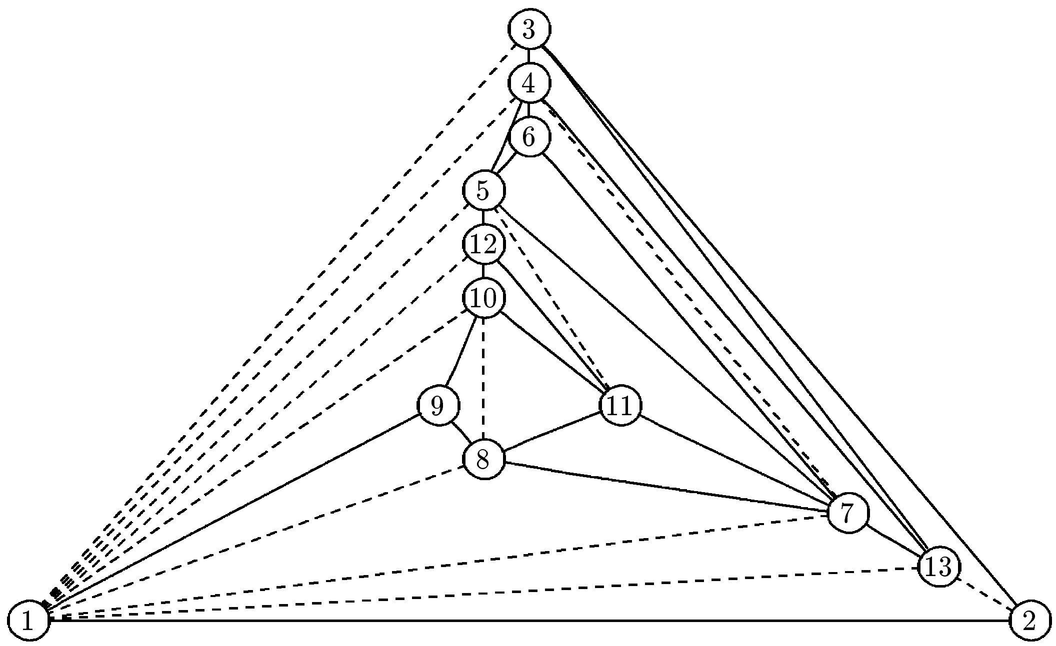


Figure 114. A straight-line embedding of the graph of Figure 115

Theorem 4. *Any planar graph with n nodes has a straight-line embedding into the $2n-4$ by $n-2$ grid, i.e., vertices are mapped into elements of $\{0,\dots,2n-4\}\times\{0,\dots,n-2\}$ and edges are mapped into straight-line segments. Also, such an embedding can be constructed in time $O(n\log n)$.*

Let G be a planar graph. We may assume w.l.o.g. that we have a combinatorial embedding of G and that G is triangulated, i.e., every face is a triangle. Note that a combinatorial embedding can be computed in linear time by Theorem 3 and that the obtained planar map can be triangulated in linear time by subdividing faces with more than three vertices (cf. Exercise 33?). Figure 115 shows a triangulation of the planar map of Figure 113. Our strategy for computing a straight-line embedding works iteratively, i.e., we start with a single triangle and then add node after node. The basis for the iteration is the following Lemma 11.

Lemma 11. *Let (G,σ) be a triangulated planar map with outer face u,v,w. Then there is a labelling $v_1=u, v_2=v, v_3, v_4, \dots, v_n=w$ of the nodes meeting the following requirements for every k, $4\le k\le n$.*

(1) *The subgraph $G_{k-1}\subseteq G$ induced by $v_1,v_2,\dots,v_{k-1}$ is biconnected, and the boundary of its outer face is a cycle C_{k-1} containing the edge $\{u,v\}$.*

(2) *The nodes of G lying inside or on the cycle C_{k-1} are exactly the vertices $v_1,\dots,v_{k-1}$.*

(3) *v_k is in the outer face of G_{k-1}, and its neighbors in G_{k-1} form an (at least 2-element) subinterval of the path $C_{k-1}-\{u,v\}$ (cf. Fig. 116).*

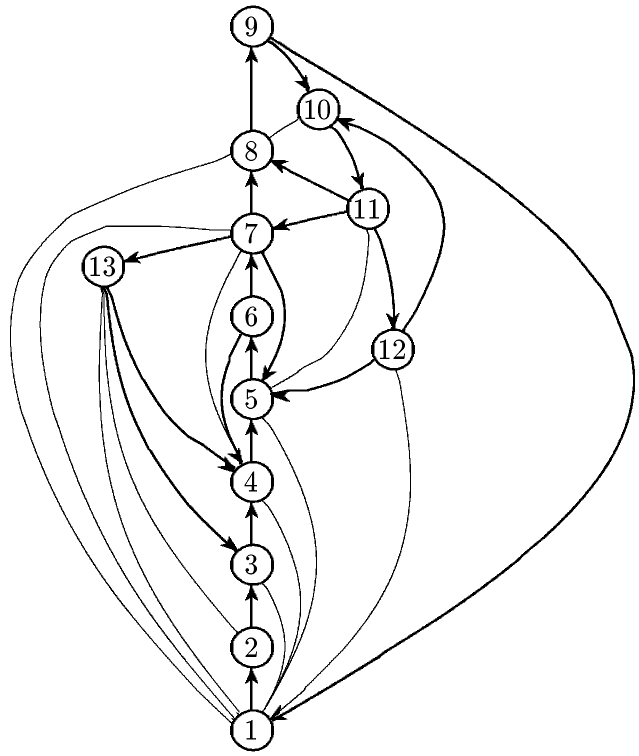


Figure 115. A triangulation of the planar map of Figure 113. The additional edges are drawn light.

Moreover the labelling can be computed in linear time.

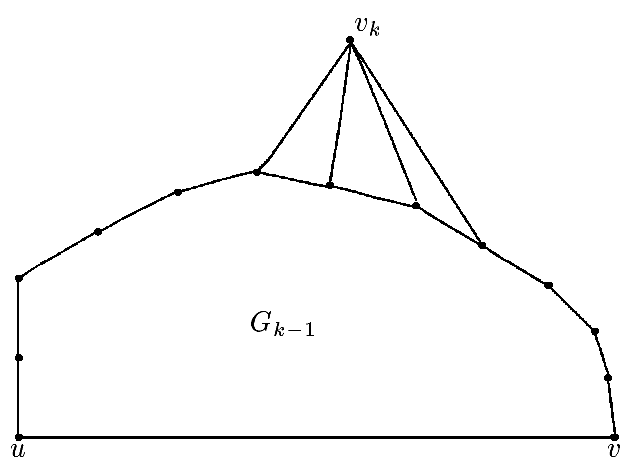


Figure 116. The vertex v_k is attached to a subinterval of the path $C_{k-1} - \{u, v\}$

Proof: Let v_3 be the unique vertex such that the triangle u, v, v_3 is a bounded face of G. Then (1) and (2) hold for $k \geq 4$. Assume inductively that $v_1, \ldots, v_{k-1}$ have been defined and that (1) and (2) hold for the subgraph G_{k-1}. Let $w_1, \dots, w_r, w_1$ with $w_1 = u$, $w_r = v$ be the cycle G_{k-1} and let $K = \{z;\ z \notin \{v_1, \dots, v_{k-1}\}$ and z is adjacent to at least two vertices on the cycle $C_{k-1}\}$. For $z \in K$ let $\min(z) = \min\{i;\ w_i$ is a neighbor of $z\}$, $\max(z) = \max\{i;\ w_i$ is a neighbor of $z\}$ and let $C(z)$ be the cycle $z, w_{\min(z)}, w_{\min(z)+1}, \dots, w_{\max(z)}, z$. Let $z_0 \in K$ be such that the cycle $C(z_0)$ contains a minimal number of nodes in its interior. We claim that withe $v_k = z_0$ requirement (3) is met and that (1) and (2) hold for the subgraph G_k.

Note first that $K \neq \emptyset$ since for every edge $e_i = \{w_i, w_{i+1}\}$, $1 \le i < r$, there is a unique node $z(e_i)$ outside C_{k-1} such that the triangle $w_i, w_{i+1}, z(e_i)$ is a face of G. The nodes $z(e_i)$ belong to K, $1 \le i < r$, and hence $K \neq \emptyset$. So z_0 exists. Note next that the cycle $C(z_0)$ contains no node in its interior because otherwise there would be a node $z \in K$ in its interior such that $C(z)$ contains even fewer nodes in its interior. This also implies $z_0 \neq w$ of $k < n$ since $\min(w) = 1$, $\max(w) = r$ and hence $C(w)$ has non-empty interior if $k < n$. Finally, since G is triangulated z_0 must be connected to all nodes w_i, $\min(z_0) \le i \le \max(z_0)$. This proves (3) and also that (1) and (2) hold for G_k. Thus the disired labelling exists.

It can be computed in linear time as follows. We maintain a partition of the unlabelled nodes $\neq w$ into classes:

A: no neighbor labelled yet

B: exactly one neighbor labelled

i: more than one neighbor labelled and the labelled neighbors form i intervals in the cyclic ordering of edges around the node, $i \ge 1$.

We start with only nodes u and v labelled and an initial partition which can certainly be computed in linear time. Assume now that we have determined nodes $v_1, \dots, v_{k-1}$ already. We then choose any node in class 1 and label it v_k. Note that class 1 is never empty by the argument given in the existence proof of the labelling. Conversely, let z be any node in class 1. Then certainly $z \in K$. Also, z is adjacent to all nodes w_i, $\min(z) \le i \le \max(z)$, since z belongs to class 1. Assume now that one of the triangles z, w_i, w_{i+1} where $\min(z) \le i \le \max(z)$ is not a face of G and hence contains a node in its interior. Any node in the interior of this triangle is unlabelled and one of the nodes must be adjacent to z since G is triangulated. Thus z dose not belong to class 1, a contradiction. After labelling v_k we consider all unlabelled neighbors of v_k and update their class membership as follows. Let z be any such neighbor. If z belongs to class A then it is moved to class B. If z belongs to class B then it is moved to either class 1 or class 2 and if v belongs to class i then it is moved to either class $i - 1$ or $i + 1$ or stays in class i. For each neighbor z this decision takes constant time end hence the time required to label v_k is proportional to the degree of v_k. Thus the entire labelling is computed in linear time. ∎

In the example of Figure 115, we may choose $u = 1$, $v = 2$, $w = 9$ and use the labelling $v_1 = 1$, $v_2 = 2$, $v_3 = 3$, $v_4 = 4$, $v_5 = 5$, $v_6 = 6$, $v_7 = 13$, $v_8 = 7$, $v_9 = 11$, $v_{10} = 8$, $v_{11} = 12$, $v_{12} = 10$ and $v_{13} = 9$.

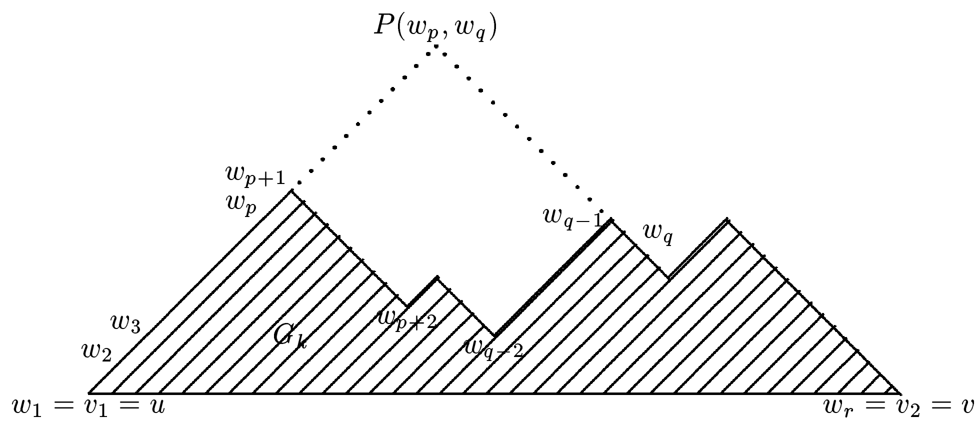


Figure 117. G_k

We can now describe the iteration in more detail. We start by placing v_1 at $(0,0)$, v_2 at $(2,0)$ and v_3 at $(1,1)$. Assume inductively, that we embedded G_k as a mountain range (cf. Fig. 118) with base $\{u, v\}$, i.e.,

(1) v_1 is at $(0,0)$, v_2 is at $(2k-4, 0)$ and all points of G_k are embedded onto lattice points of the first quadrant;

(2) If $w_1 = v_1, w_2, \dots, w_r = v_2$ denote the vertices on the outer face of G_k (in the order of their appearance), denotes the x-coordinate of w_i, then

$$x(w_1) < x(w_2) < \dots < x(w_r);$$

(3) The line segments $L(w_i, w_{i+1})$, $1 \le i < r$, all have slope $+1$ or -1.

Note that (3) implies that the Manhattan distance between any two nodes w_i and w_j of the outer face is *even*. (The Manhattan distance of (x, y) and (x', y') is $|x - x'| + |y - y'|$. Hence, if $i < j$, then the intersection of the line with slope $+1$ through w_i and the line with slope -1 through w_j is a *lattice point* $P(w_i, w_j)$.

Let $w_p, w_{p+1}, \dots, w_q$ be the neighbor of v_{k+1} in G_{k+1} $(1 \le p < q \le r)$, (cf. part (3) of Lemma 11). The idea is to place the node v_k at the lattice point $P(w_p, w_q)$. This may fail because e.g. w_q may not be visible from that lattice point; cf. Figure 117. To make sure that all nodes $w_p, w_{p+1}, \dots, w_q$ are visible from $P(w_p, w_q)$ we deform the embedding such that the slope of the line segment $L(w_p, w_{p+1}$ becomes less than 1, the slope of the line segment $L(w_{q+1}, w_q)$ becomes larger than -1, and the slopes of all other line segments on the boundary of the outer face remain the same. One way to achieve this is to first move nodes $w_{p+1}, wp + 2, \dots, w_r$ one unit to the right and then to move nodes $w_q, w_{q+1}, \dots, w_r$ one further unit to the right. However, in order to not destroy the straight-line embedding of G_{k-1} we may also have to move some other nodes of G_k. For this purpose, we maintain for each node w_i on the outer face of G_k a set $M(k, w_i) \subseteq \{v_1, \dots, v_k\}$ such that

(4) $w_j \in M(k, w_i)$ iff $j \ge i$

(5) $M(k, w_1) \supseteq M(k, w_2) \supseteq \cdots \supseteq M(k, w_r)$

(6) For any sequence $\alpha_1, \ldots, \alpha_r$ of non-negative numbers, if we sequentially translate all vertices in $M(k, w_i)$ with distance α_i to the right ($i = 1, 2, \ldots, r$), then the embedding of G_k remains a Fáry embedding. (Note that many vertices will move several times; e.g., all points in $M(k, w_i) \setminus M(k, w_{i+1})$ will be translated by $\alpha_1 + \alpha_2 + \cdots + \alpha_i$.) For $k = 3$ these conditions are met by straight-line embedding $v_1 \mapsto (0,0)$, $v_2 \mapsto (2,0)$, $v_3 \mapsto (1,1)$ and by the sets $M(3, v_1) = \{v_1, v_2, v_3\}$, $M(3, v_2) = \{v_2, v_3\}$, $M(3, v_3) = \{v_3\}$. We can now describe how to embed node v_k. We apply (6) with $\alpha_{p+1} = \alpha_q = 1$ and all other $\alpha_i = 0$ to find a new straight-line embedding of G_k. This assumes $p + 1 < q$. If $p + 1 = q$ then we apply (6) with $\alpha_q = 2$ and all other $\alpha_i = 0$. The Manhattan distance between w_p and the new location of w_q is still even, thus we can place v_{k+1} at the intersection of the lines with slope $+1$ and -1 through w_q and the new location of w_q, respectively. Conditions (1), (2) and (3) will trivially remain true for this new embedding of G_{k+1}. (cf. Fig. 118)

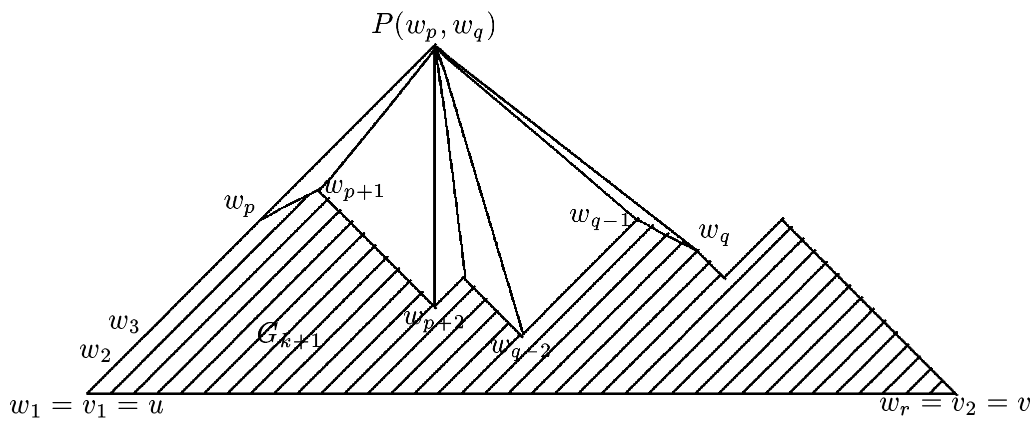


Figure 118. G_{k+1}

The nodes of the outer face of G_{k+1} are $u = w_1, w_2, \ldots, w_p, v_{k+1}, w_q, \ldots, w_r = v$. For each member z of this sequence we have to define a set $M(k+1, z) \subseteq V(G_{k+1})$. Let

$$M(k+1, w_i) = M(k, w_i) \cup \{v_{k+1}\} \text{ for } i \leq p,$$
$$M(k+1, v_{k+1}) = M(k, w_{p+1}) \cup \{v_{k+1}\},$$
$$M(k+1, w_j) = M(k, w_j) \text{ for } j \geq q.$$

It is obvious that these sets have properties (4) and (5). To check that property (6) remains true as well consider any sequence of nonnegative numbers $\alpha(w_1), \ldots, \alpha(w_p), \alpha(v_{k+1}), \alpha(w_q), \ldots, \alpha(w_m)$. For all z on the outer face of G_{k+1} translate the set $M(k+1, z)$ with distance $\alpha(z)$ to the right. Observe that after this motion the part of G_{k+1} below the polygon $w_1 w_2 \ldots w_m$ (i.e., G_k) remains straight-line embedded (by condition (6) applied to G_k with $\alpha_1 = (w_1)$, $\ldots$, $\alpha_p = 1 + \alpha(w_p)$, $\alpha_{p+1} = 1 + \alpha(v_{k+1})$, $\alpha_q = 1 + \alpha(w_q)$,

$\alpha_{q+1} = 1 + \alpha(w_{q+1})$, ..., $\alpha_m = 1 + \alpha(w_m)$ and every other $\alpha_i = 0$ if $p+1 < q$ and $\alpha_1 = \alpha(w_1)$, ..., $\alpha_p = \alpha(w_p)$, $\alpha_q = 2 + \alpha(w_{k+1}) + \alpha(w_q)$, $\alpha_{q+1} = \alpha(w_{q+1})$, ..., $\alpha(w_m)$ if $p+1 = q$). On the other hand, it is easy to see that the part of G_{k+1} above $w_1 w_2 \dots w_m$ (i.e., v_{k+1} and the upper contour of G_k) remains straight-line embedded too, since during the motion the subgraph induced by $w_{p+1}, w_{p+2}, \dots, w_{q-1}$ and v_{k+1} moves rigidly (to a distance $\alpha(w_1) + \cdots + \alpha(w_p) + \alpha(v_{k+1})$). The final output of our algorithm is a straight-line embedding $\mathcal{E}$ of $G_n = G$ satisfying conditions (1), (2) and (3) with $k = n$. This immediately implies that every point ov G is embedded in some lattice pont of the triangle determined by $\mathcal{E}(v_1) = \mathcal{E}(u) = (0,0)$, $\mathcal{E}(v_2) = \mathcal{E}(v) = (2n-4, 0)$ and $\mathcal{E}(v_n) = \mathcal{E}(w) = (n-2, n-2)$. This proves the existence of the desired embedding.

It is easy to derive an $O(n^2)$ algorithm from the constructive existence proof. A less direct implementation which avoids the explicit construction of embeddings for the intermediate graphs achieves a running time of $O(n \log n)$. We describe this implementation next.

Let $p_k(l) = (x_k(l), y_k(l))$ be the position of node v_k in the embedding of G_l, $k \leq l$. We are interested in $p_k(n)$ but in order to make the construction work we deal with the more general problem of computing $p_k(l)$. clearly, $x_k(l) = x_k(k) + (x_k(l) - x_k(k))$ and $y_k(l) = y_k(k)$, i.e., the position of v_k in G_l is given by the position $p_k(k)$ of v_k in G_k and the shift distance $\mathit{shift}_k(l) := x_k(l) - x_k(k)$ of v_k when passing from G_k to G_l.

Lemma 12. *The positions $p_k(k)$, $p_k(n)$, $k = 1, 2, \dots, n$ can be computed in time $O(n)$ plus the time to compute $O(n)$ shift distances.*

Proof: Clearly, knowing $p_k(k)$ and $\mathit{shift}_k(n)$ we can compute $p_k(n)$. So we only have to deal with the computation of $p_k(k)$, $k = 1, 2, \dots, n$. For node v_k let $v_{i_1}, v_{i_2}, \dots, v_{i_j}$ be the neighbors of v_k in G_{k-1} in counterclockwise order; this nodes
these nodes were
where called $w_p, w_{p+1}, \dots, w_q$ in the existence proof. Set $\mathit{first}_k = i_1$, $\mathit{second}_k = i_2$ and $\mathit{last}_k = i_j$ and observe that the functions *first*, *second* and *last* can be computed with no extra effort when computing the labelling; cf. Lemma 11. Then $p_k(k)$ is given by the intersection of the line with slope $+1$ through $p_{\mathit{first}_k}(k)$ and the line with slope -1 through $p_{\mathit{last}_k}(k)$. Also, $p_{\mathit{first}_k}(k)$ is determined by $p_{\mathit{first}_k}(\mathit{first}_k)$ and $\mathit{shift}_{\mathit{first}_k}(k)$ and similarly for $p_{\mathit{last}_k}(k)$. We conclude that the sequence $p_k(k)$, $k = 1, 2, \dots, n$, can be computed in linear time if the quantities $\mathit{shift}_{\mathit{first}_k}(k)$ and $\mathit{shift}_{\mathit{last}_k}(k)$, $4 \leq k \leq n$, are known. ∎

For the computation of shift distances we first derive an economical encoding of the sets $M(l, v_k)$. Define sequences $\pi_2, \pi_3, \dots, \pi_n$ as follows. Let $\pi_2 = (1, 2)$ and obtain π_{k+1} from π_k by inserting $k+1$ just to the left of second_k and $n+k+1$ just to the left of last_k where second_k and last_k are defined as in the proof of Lemma 12.

Lemma 13.

a) *Let v_k be a vertex on the outer face of G_l. Then*

$$v_i \in M(l, v_k) \text{ iff } i \le l \text{ and } k \text{ precedes } i \text{ in } \pi_n \text{ or } k = i.$$

b)
$$\mathit{shift}_k(l) = |\{j;\ k < j \le l \text{ or } k < j - n \le l \text{ and } j \text{ precedes } k \text{ in } \pi_n\}|$$

Proof: a) We use induction on $\max(k, i)$. For $\max(k, i) = 2$ the claim is obvious. So let us suppose $\max(k, i) > 2$ and $k \ne i$. Assume first that $i < k$. Then $v_i \in M(l, v_k)$ iff $v_i \in M(k-1, v_{second_k})$ by the definition of $M(k, v_k)$, and k precedes i in π_n iff $second_k$ precedes i in π_n by the definition of π_n. So the claim follows directly from the induction hypothesis. Assume next that $i > k$. Then $i \in M(i, v_k)$ iff v_k precedes v_i on the boundary of G_i iff v_k precedes v_{second_i} on the boundary of G_{i-1} iff k precedes i in π_n and the claim is shown.

b) Consider the addition of a node v_j, $k < j \le l$. Then node v_k is moved two units to the right if $v_k \in M(j-1, v_{last_j})$ and v_k is moved one unit to the right if $v_k \in M(j-1, v_{second_j}) - M(j-1, v_{last_j})$. By part a) and the construction of π_n this is equivalent to $j + n$ and j precede k in π_n and j but not $j + n$ precedes k in π_n. This proves part b). ∎

It is now easy to translate the computation of shift distances into a range query problem. Let S be the following set of points

$$S = \{(1,1), (2, 2n-3)\} \cup \{(k, \pi_n^{-1}(k)), (k, \pi_n^{-1}(n+k));\ 3 \le k \le n\}$$

and let $R(k, l)$ be the rectangle

$$R(k,l) = \{(j, y);\ k < j \le l \text{ and } y \le \pi_n^{-1}(k)\}.$$

Then

$$\begin{aligned}|R(k,l) \cap S| &= |\{j;\ k < j \le l \text{ or } k < j - n \le l \\ &\qquad \text{and } j \text{ precedes } k \text{ in } \pi_n\}| \\ &= \mathit{shift}_k(l).\end{aligned}$$

A query of the form "determine the cardinality of the intersection of a rectangle R and a set S of points" is called a rectangular range counting query. Our rectangles are 3-sided because there is no restriction on y from below in the definition of $R(k, l)$. In the section on segment trees in Chapter 8 it is shown that 3-sided rectangular range counting queries can be precessed in time $O(\log N)$ with a preprocessing time of $O(N \log N)$, where $N = |S|$. (In the first edition, only $O((\log N)^2)$ is shown). We summarize in:

Theorem 5. *A straight-line embedding of a planar graph of n nodes can be computed in time $O(n+P(n)+n\cdot Q(n))$ where $P(n)$ is the preprocessing time of 3-sided rectangular range counting and $Q(n)$ is the query time.* ∎

With the results of Chapter 8 the proof of Theorem 5 is now completed.

Theorem 6. *Let $G=(V,E)$ be a planar graph. Then G can be turned into a planar map (G,σ) in linear time.* ∎

Planar graphs have more structure than general graphs and are therefore in many respects computationally simpler than general graphs. The planar separator theorem (Theorem 3 below) makes planar graphs amenable to divide and conquer algorithms. It states that a planar graph can be split into about equal sized subgraphs by the removal of only $O(\sqrt{n})$ nodes.

Theorem 7. *Let $G=(V,E)$ be a planar graph and let $w: V \to \mathbb{R}_0^+$ be a weight function on the vertices of G. Let $W=\sum_{v\in V} w(v)$ be the total weight of G. Then there is a partition A,S,B of V such that*

1) $W(A)=\sum_{v\in A} w(v) \le 2W/3$, $W(B)\le 2W/3$;

2) $|S| \le 4\sqrt{n}$;

3) *S separates A from B, i.e., $E\cap(A\times B)=\emptyset$;*

4) *Partition A,S,B can be constructed in time $O(n)$.*

Proof: Assume first that G is connected. Let $s\in V$ be arbitrary and let $L(t)=\{v;\ v\in V$ and the shortest path from s to v has length $t\}$ for $t\ge 0$. Then $L(0)=\{s\}$. Let r be maximal such that $L(r)\ne\emptyset$. Add empty levels $L(-1)=L(r+1)=\emptyset$, for convenience. Let t_1 be such that

$$W(L(0)\cup\cdots\cup L(t_1-1)) \ \le\ W/2 \ \le\ W(L(0)\cup\cdots\cup L(t_1)),$$

let $t_0\le t_1$ be such that $|L(t_0)|+(t_1-t_0)<2\sqrt{n}$ and let $t_2>t_1$ be such that $|L(t_2)|+(t_2-t_1-1)<2\sqrt{n}$.

Claim 1. *t_0, t_1 and t_2 exist.*

Proof: The existence of t_1 is obvious. If $t_1<\sqrt{n}$ then we can choose $t_0=-1$. If $t_1\ge\sqrt{n}$ then $|L(t_1-\sqrt{n})|+\cdots+|L(t_1)|\le n$ and hence $|L(t_0)|<\sqrt{n}$ for some t_0, $t_1-\sqrt{n}\le t_0\le t_1$. Thus $|L(t_0)|+t_1-t_0<2\sqrt{n}$. In either case we have shown the existence of t_0. The existence of t_2 is shown similarly. ∎

Let us take a closer look at $W(L(t_0+1)\cup\cdots\cup L(t_2-1))$. If this weight is at most $2W/3$ then let $S=L(t_0)\cup L(t_2)$, let A be the heaviest of the three sets $L(0)\cup\cdots\text{Ł}(t_0-1)$, $L(t_0+1)\cup\cdots\cup L(t_2-1)$, $L(t_2+1)\cup\cdots\cup L(r)$ and let B be the union of the remaining two sets. Then $W(A)\le 2W/3$ and $W(B)\le 2W/3$.

Let us assume now that $W(L(t_0+1) \cup \cdots \cup L(t_2-1)) > 2W/3$. Construct planar graph G' as follows. Delete levels t_2 and above from the graph and shrink all nodes in level t_0 and below to a single node, i.e., replace all nodes in level t_0 and below by a single node and connect this node to all nodes in $L(t_0+1)$. The planarity of G' can be seen as follows. Consider a planar embedding of G and identify a tree of paths from s to all nodes in $L(t_0+1)$. Then delete all nodes in level t_0 and below, make s the new node and draw the new edges along the tree paths. Note that graph G' has a spanning tree with radius $t_2 - t_0 - 1$, i.e., the newly constructed node is the root and all other nodes have distance at most $t_2 - t_0 - 1$ from the root.

Claim 2. *Let $G = (V, E)$ be a connected planar graph having a spanning tree of radius r and let $w : V \to \mathbb{R}_0^+$ be a weight function. Then there is a partition A, S, B of V such that $W(A) \le 2W/3$, $W(B) \le 2W/3$, $|S| \le 2r+1$, S contains the root of the spanning tree, and S separates A from B. Moreover, partition A, S, B can be found in time $O(n)$.*

Suppose that we have shown Claim 2. Clearly, all steps of the proof preceding Claim 2 can be carried out in linear time, i.e., the construction of levels $L(0), L(1), \ldots, L(r)$, determination of t_0, t_1 and t_2, and construction of G'. By Claim 2 we can find a partition A', S', B' of the nodes of G' such that S' contains at most $2(t_2 - t_0 - 1) + 1$ nodes one of which is the node which replaced levels t_0 and below. Let $S = L(t_0) \cup L(t_2) \cup (S' - \{\text{new node}\})$. Then

$$\begin{aligned}
|S| &\le |L(t_0)| + |L(t_2)| + 2(t_2 - t_0) \\
&= |L(t_0)| + 2(t_1 - t_0) + |L(t_2)| + 2(t_2 - t_1 - 1) + 2 \\
&\le 2\sqrt{n} - 1 + 2\sqrt{n} - 1 + 2 \\
&= 4\sqrt{n}.
\end{aligned}$$

The removal of S from G splits G into sets $L(0) \cup \cdots \cup L(t_0 - 1)$, A', B', $L(t_2+1) \cup \cdots \cup L(r)$ none of which has weight exceeding $2W/3$. It is easy to form sets A and B from these four sets such that $W(A) \le 2W/3$ and $W(B) \le 2W/3$. Moreover, partition A', S', B' and hence partition A, S, B can be found in linear time.

Proof of Claim 2: If there is $v \in V$ with $w(v) \ge W/3$ then let S consist of v and the root of the spanning tree, let $A = \emptyset$ and let $B = V - S$. Clearly, partition A, S, B has all desired properties.

So let us assume next that $w(v) < W/3$ for all $v \in V$. Extend G to a planar map $\hat{G}$; this can be done in time $O(n)$ by Theorem 6. Add edges to $\hat{G}$ such that every face becomes a triangle, cf. Exercise 3302. Let T be a spanning tree of G of radius at most r. Every non-tree edge of G forms a simple cycle with some of the tree edges. This cycle has length at most $2r+1$ if the root belongs to the cycle and has length at most $2r-1$ otherwise. Every such cycle separates its inside from its outside. It therefore suffices to show that there is one such cycle such that neither the inside nor the outside has weight exceeding $2W/3$. More precisely, if e is a non-tree edge, let $C(e)$ by the cycle defined by e, let $WC(e)$ be the weight of cycle $C(e)$, i.e., $WC(e) = \sum_{v \in C(e)} w(v)$ and let $WI(e)$ be the weight of the nodes in the interior of $C(e)$.

In Figure 119 (tree edges are drawn solid, non-tree edges are drawn dashed) we have for $e = (2,6)$, $C(e) = (2,1,5,7,6)$, $WC(e) = w(2) + w(1) + w(5) + w(7) + w(6)$ and $WI(e) = w(3) + w(4)$.

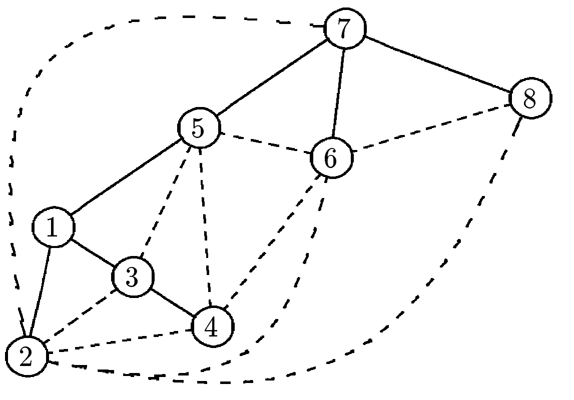


Figure 119. Example for Claim 2

We have to show that there is a non-tree edge e such that $WI(e) \leq 2W/3$ and $WI(e) + WC(e) \geq W/3$. Programs 34 and 35 compute $WI(e)$, $WC(e)$, and $C(e)$ for (all) non-tree edges e.

---

begin for all non-tree edges e **do** $WC(e) \leftarrow$undefined **od**;
  for all non-tree edges e
  do if $WC(e)$ is undefined **then** *cycle*(e) **fi od**
end

Program 34

Program 34 makes use of procedure *cycle* (see Program 35) which computes $C(e)$, $WI(e)$ and $WC(e)$ for non-tree edge e. The body of *cycle* is basically a case distinction according to the type of edges in the triangle inside $C(e)$ with edge e on its boundary. This case distinction is illustrated in Figure 120. The main program *cycle* is called at most once for every non-tree edge e.

---

procedure *cycle*(e: non-tree edge);
co computes $C(e)$ as a doubly linked list and weights $WI(e)$ and $WC(e)$;
  stops computation if $C(e)$ is desired cycle **oc**
let $e = \{x, z\}$, and let y be the third node of the triangle inside $C(e)$
which has e as an edge;

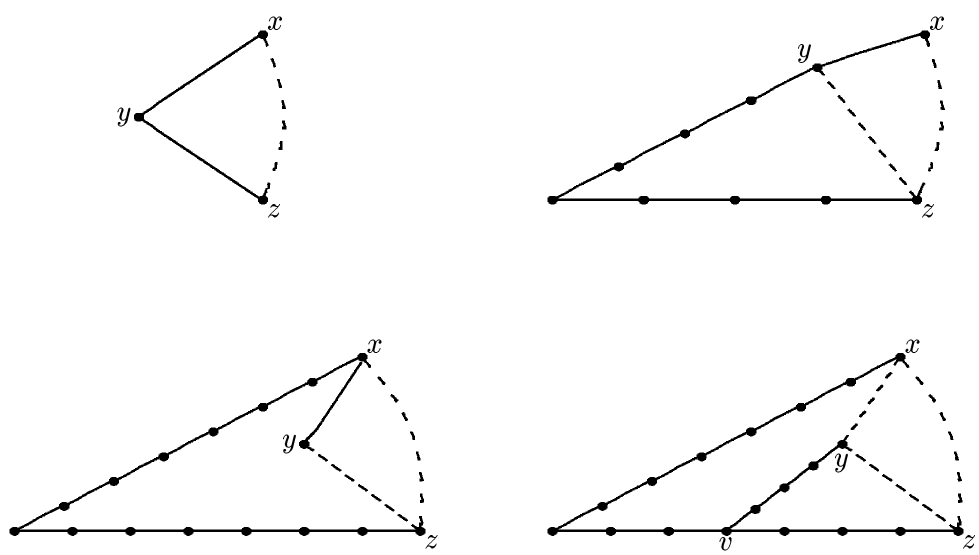


Figure 120. The 4 cases of *cycle*

Case 1: $\{x, y\}$ and $\{y, z\}$ are tree edges.

Then triangle (x, y, z) is a cycle $C(e)$ and hence $C(e) \leftarrow (x, y, z)$, $WI(e) \leftarrow 0$ and $WC(e) \leftarrow s(x) + w(y) + w(z)$.

Case 2: $\{x, y\}$ is a tree edge, and $\{y, z\}$ is not, and edge $\{x, y\}$ lies on cycle $C(e)$, i.e., y is closer to the root of the spanning tree than x.

cycle$(\{y, z\})$;
$C(e) \leftarrow x$ concatenated with $C(\{y, z\})$;
$WC(e) \leftarrow WC(\{y, z\}) + w(x)$;
$WI(e) \leftarrow WI(\{y, z\})$.

Case 3: $\{x, y\}$ is a tree edge, $\{y, z\}$ is not, and edge $\{x, y\}$ does not lie on cycle $C(e)$, i.e., y is farther away from the root of the spanning tree than x.

cycle$(\{y, z\})$;
$C(e) \leftarrow C(\{y, z\})$ minus node y;
$WC(e) \leftarrow WC(\{y, z\}) - w(y)$;
$WI(e) \leftarrow WI(\{y, z\}) + w(y)$.

Case 4: Neither $\{x, y\}$ nor $\{y, z\}$ are tree edges.

cycle$(\{x, y\})$;
cycle$(\{y, z\})$;
let p be the tree path from y to $C(e)$ including y and excluding v where v is the node on $C(e)$ where p meets $C(e)$. The node v can be found in time $O(|p|)$ where $|p|$ is the number of nodes of p as follows. The calls *cycle*$(\{x, y\})$ and *cycle*$(\{y, z\})$ return the cycles $C(\{x, y\})$ and $C(\{y, z\})$. We walk along these cycles starting in node y and away from nodes x and z respectively. Then v is the last common node and p is the path of nodes preceding v.

$C(e) \leftarrow (C(\{x,y\})$ minus $p)$ concatenated with $(C(\{y,z\})$ minus $p)$;
$WC(e) \leftarrow WC(\{x,y\}) + WC(\{y,z\}) - 2W(p) - w(v)$;
$WI(e) \leftarrow WI(\{x,y\}) + WI(\{y,z\}) + W(p)$;

end of case-distinction;

if $WI(e) \leq 2W/3$ and $WC(e) + WI(e) \geq W/3$
then stop and exhibit $C(e)$ as the desired cycle **fi**
end.

Program 35

We still have to show that some call of *cycle* finds a cycle with the desired properties and to analyze the running time. We show first that the running time is linear. Note first that *cycle* is called at most once for each non-tree edge e. Also the cost of a call of *cycle* is $O(1)$ if Case 1, 2 or 3 is taken in the body and it is $O(|p|)$ in Case 4 where $|p|$ is the number of nodes on path p. Also, $2|p|$ nodes are deleted from cycles $C(\{x,y\})$ and $C(\{y,z\})$ when cycle $C(e)$ is formed in the latter case. Since at most two tree edges are added to cycles in a single execution of Cases 1 to 3 the total number of nodes deleted in Case 4 must be $O(n)$. Thus the total cost of either case is $O(n)$ and hence the total running time is $O(n)$.

Finally, we have to show that a cycle with the desired properties is found. We will show first that there is a non-tree edge e with $WC(e) + WI(e) \geq W/3$. Since every face of $\hat{G}$ is a triangle, so is the outer face. Let e_1, e_2, e_3 be the edges bordering the outer face. At least one of them is a non-tree edge, say $e_1, \ldots, e_i$ are non-tree edges for some i, $1 \leq i \leq 3$. Then $\sum_{j=1}^{i} WC(e_j) + WI(e_j) \geq W$ since each node of G lies inside or on at least one of the cycles $C(e_j)$, $j = 1, \ldots, i$. Thus $WC(e_k) + WI(e_k) \geq W/3$ for some k, $1 \leq k \leq i$.

We can now exhibit edge e such that $C(e)$ has the desired properties. Let e be a non-tree edge such that $WC(e) + WI(e) \geq W/3$ and either Case 1 is taken for e or $WC(e') + WI(e') < W/3$ for all non-tree edges e' such that $cycle(e')$ is called by $cycle(e)$. Edge e exists since there are edges with $WC(e) + WI(e) \geq W/3$ and since Case 1 is taken for at least one edge. It remains to show that edge e has the property $WI(e) < 2W/3$. If Case 1 is taken then $WI(e) = 0$ and we are done. If Case 2 is taken then $WI(e) = WI(e') < W/3$ and we are done. If Case 3 is taken then $WI(e) + WC(e) = WI(e') + WC(e')$ which is impossible. If Case 4 is taken, let e_1 and e_2 be the two non-tree edges for which *cycle* is called. We have

$$\begin{aligned} WI(e) &= WI(e_1) + WI(e_2) + W(p) \\ &\leq WI(e_1) + W(p) + WI(e_2) + W(p) \\ &\leq WI(e_1) + WC(e_1) + WI(e_2) + WC(e_2) \\ &\leq 2W/3, \end{aligned}$$

and hence $C(e)$ is the desired cycle. ∎

We have now proved Theorem 7 for connected graphs. If G is unconnected, let $G_1, G_2, \ldots G_k$ be the connected components. If $W(G_i) \leq 2W/3$ for all i, $1 \leq i \leq k$, then a partition with $S = \emptyset$ is possible. If $W(G_i) \geq 2W/3$ for some i then split G_i as described above and proceed as in the former case. ∎

An important corollary of Theorem 3 is obtained in the unit cost case.

Corollary 1. *Let $G = (V, E)$ be a planar graph. Then there is a partition A, S, B of V such that*

1) $|A| \leq 2n/3$, $|B| \leq 2n/3$;

2) $|S| \leq 4\sqrt{n}$;

3) *S separates A from B;*

4) *Partition A, S, B can be found in time $O(n)$.*

Proof: Obvious from Theorem 3 with $w(v) = 1$ for all $v \in V$. ∎

We end this section with an application of the planar separator theorem. Let $N = (V, E, c)$ with $c : E \to \mathbb{R}$ be a directed planar network and let $s \in V$ be a designated node. As in Section 4.7 we will study the problem of computing $\mu(s, v)$, the cost of a least cost path from s to v, for any node $v \in V$. In Section 4.7.3 we saw how to solve this problem in time $O(n \cdot e) = O(n^2)$ for planar networks. A better algorithm can be obtained by applying the separator property of planar graphs.

Theorem 8. *The single source least cost path problem on planar networks can be solved in time $O(n^{1.5} \log n)$*

Proof: Let $N = (V, E, c)$ with $c : E \to \mathbb{R}$ be a planar directed network and let $s \in V$ be a designated node. We want to compute $\mu(s, v)$ for all nodes $v \in V$. The algorithm is as follows.

(1) Compute a partition V_1, S, V_2 as given by the planar separator theorem. Let $S \leftarrow S \cup \{s\}$ and let N_i be the subnetwork induced by $V_i \cup S$ for $i = 1, 2$.

(2) Compute $\mu_1(t, v)$ for all $t \in S$ and $v \in V_1 \cup S$. Here $\mu_1(t, v)$ is the cost of a least cost path from t to v in subnetwork N_1. Similarly, compute $\mu_2(t, v)$ for all $t \in S$ and $v \in V_2 \cup S$. The details of this step are spelled out below.

(3) Define network $\overline{N} = (S, S \times S, \bar{c})$ by $\bar{c}(r, t) = \min\{\infty, \mu_1(r, t), \mu_2(r, t)\}$. Compute $\bar{\mu}(s, t)$ for all $t \in S$ where $\bar{\mu}(s, t)$ is the cost of a least cost path from s to t in network $\overline{N}$.

(4) For $v \in V_i \cup S$, $i = 1, 2$, output $\mu(s, v) = \min\{\bar{\mu}(s, t) + \mu_i(t, v);\ t \in S\}$.

The correctness of this algorithm can be seen fairly easily. It follows from

Claim 1.

a) $\bar{\mu}(s,t) = \mu(s,t)$ *for all* $t \in S$.

b) $\mu(s,v) = \min\{\bar{\mu}(s,t) + \mu_i(t,v);\ t \in S\}$ *for* $v \in V_i$, $i = 1,2$.

Proof: a) Edges of $\overline{N}$ correspond to least cost paths in subnetworks N_i, $i = 1,2$. Thus $\bar{\mu}(s,t) \ge \mu(s,t)$ since every path in $\overline{N}$ gives rise to a path in N by replacing edges of $\overline{N}$ by paths in N. Also $\bar{\mu}(s,t) \le \mu(s,t)$ since a least cost path from s to t in N can be decomposed into subpaths running completely within N_i, $i = 1,2$.

b) Follows immediately from part a). ∎

It remains to describe the details of the implementation. For step (1) we use the algorithm described above in Corollary 1; it yields partition V_1, S, V_2 with $|V_1| \le 2n/3$, $|V_2| \le 2n/3$ and $|S| \le 5\sqrt{n}$. (We use 5 instead of 4 because node s is added to S). For step (3) we use the algorithm described in Section 4.7.3; it runs in time $O(|S| \cdot |S|^2) = O(n^{1.5})$. Step (4) is also easily done in time $O(n^{1.5})$. Step (2) remains to be described in detail. We do so for subnetwork N_1.

(2.1) Compute $\mu_1(s,v)$ for all $v \in V_1 \cup S$ using the algorithm recursively. This takes time $T(|V_1 \cup S|)$ where $T(n)$ is the running time of the algorithm on a n node graph.

(2.2) Use the solution of step (2.1) to make all edge costs non-negative as described in Section 4.7.4. Compute $\mu_1(t,v)$ for all $t \in S$, $v \in V_1 \cup S$ in time $O(|S| \cdot n_1 \cdot \log n_1) = O(n^{1.5} \log n)$ using the methods described in Sections 4.7.4 and 4.7.2. Here $n_1 = |V_1 \cup S|$.

We conclude that the cost of step (2) is $T(n_1) + T(n_2) + O(n^{1.5} \log n)$ where $n_i = |V_i \cup S|$, $i = 1,2$. Altogether, we have the following recurrence for $T(n)$:

$$T(n) \le \begin{cases} c \cdot n^{1.5} \log n & \text{for } n < 1500; \\ \max_{\substack{n_1+n_2 \le n+5\sqrt{n} \\ n_1,n_2 \le 4n/5}} \{T(n_1) + T(n_2) + d \cdot n^{1.5} \log n\} & \text{for } n \ge 1500. \end{cases}$$

Here c, d are appropriate constants, $n_1 = |V_1 \cup S| \le 2n/3 + 5\sqrt{n} \le 4n/5$ for $n \ge 1500$, $N_2 = |V_2 \cup S|$ and $n_1 + n_2 \le n + |S| \le n + 5\sqrt{n}$. $T(n)$ is clearly a non-decreasing function. Let $U(n) = T(n)/n$. Then

$$U(n) \le c \cdot n^{0.5} \log n \qquad \text{for } n \le 1500$$

and

$$\begin{aligned} U(n) &\le \max_{\substack{n_1+n_2 \le n+5\sqrt{n} \\ n_1,n_2 \le 4n/5}} \{(n_1/n) \cdot U(n_1) + (n_2/n) \cdot U(n_2) + d \cdot n^{0.5} \log n\} \\ &\le \max_{n_1+n_2 \le n+5\sqrt{n}} \{((n_1+n_2)/n) \cdot U(4n/5) + d \cdot n^{0.5} \log n\} \\ &\le (1 + 5/\sqrt{n}) \cdot U(4n/5) + d\sqrt{n} \log n \end{aligned}$$

for $n \geq 1500$. Let $k = k(n) = \lceil \log(n/1500)/\log(5/4) \rceil$ and $f(n) = d\sqrt{n}\log n$. Then

$$U(n) \leq \sum_{i=0}^{k} \left[\prod_{j=0}^{i-1} \left(1 + 5/\sqrt{(4/5)^j n}\right) \right] \cdot f((4/5)^i n)$$

for $n \geq 1500$.

Claim 2. *$\prod_{j=0}^{i-1}\left(1 + 5/\sqrt{(4/5)^j n}\right) \leq a$ for all $i \leq k$ and some constant a.*

Proof: We have

$$\begin{aligned}
\prod_{j=0}^{k-1}\left(1 + 5/\sqrt{(4/5)^j n}\right) &= e^{\sum_{j=0}^{k-1} \ln(1+5/\sqrt{(4/5)^j n})} \\
&\leq e^{\sum_{j=0}^{k-1} 5/\sqrt{(4/5)^j n}} && \text{(since } \ln(1+x) \leq x) \\
&\leq e^{\left(5/\sqrt{(4/5)^k n}\right) \sum_{j=1}^{k} (\sqrt{4/5})^j} \\
&\leq a
\end{aligned}$$

for some a since $\sum_{j=1}^{\infty}\left(\sqrt{4/5}\right)^j$ converges and since $1500 \geq (4/5)^k \cdot n \geq (4/5) \cdot 1500$. Constant a can be chosen as 3. ∎

Substituting into the upper bound for $U(n)$ we obtain

$$\begin{aligned}
U(n) &\leq \sum_{i=0}^{k} a \cdot f((4/5)^i n) \\
&\leq \sum_{i=0}^{k} a \cdot d\sqrt{(4/5)^i n}\log n \\
&= O(\sqrt{n}\log n)
\end{aligned}$$

This proves that $T(n) = n \cdot U(n) = O(n^{1.5}\log n)$. ∎

Other applications of the planar separator theorem can be found in Exercises 34 to 41.

4.11. Exercises

1) Let $G = (V, E)$ be a digraph. Let $G^{rev} = (V, E^{rev})$ be obtained from G by reversing all edges, i.e., $E^{rev} = \{(w, v);\ (v, w) \in E\}$. Show: Given the adjacency list representation of G one can compute the adjacency list representation of G^{rev} in time $O(n + e)$.

2) A **multi-graph** is given by a set V of nodes, a set K of edges and functions, $a, b : K \to V$. An edge $k \in K$ runs from $a(k)$ to $b(k)$. The underlying graph $G = (V, E)$ is defined by $E = \{(a(k), b(k));\ k \in K\}$, i.e., parallel edges are eliminated. Show: Given the adjacency list representation of a multi-graph, i.e., given a linear list for every i containing **multi-set** $\{b(k);\ k \in K \text{ and } a(k) = i\}$, one can compute the adjacency list representation of G in time $O(|V| + |K|)$. [Hint: Use bucket sort to sort multi-set $\{(a(k), b(k));\ k \in K\}$ in lexicographic order.]

3) Let $G = (V, E)$ be an acyclic digraph and let $\overline{G} = (V, \overline{E})$ be any acyclic digraph with the same transitive closure as G, i.e., $G^* = \overline{G}^*$. Show:

a) $E_{red} \subseteq \overline{E}$ where E_{red} is defined in Section 4.3.

b) Conclude from part a) that G_{red} is the minimal graph (with respect to set inclusion) with a fixed transitive closure.

4) Let $G = (V, E)$ be an acyclic digraph. Show that one can compute G_{red} in time $O(n \cdot e_{red})$.

5) Show that one can use procedure *explorefrom* of Section 4.4 to compute the transitive closure of an arbitrary digraph in time $O(n \cdot e)$.

6) Let G be a context-free grammar. For sentential form α let $First_1(\alpha)$ be the set of terminal symbols a such that $\alpha \xrightarrow{*} a\beta$ for some β.

a) Show how to use procedure *explorefrom* to compute $First_1(\alpha)$ if G contains no ϵ-rules.

b) Modify your solution to part a) such that ϵ-rules can also be handled.

7) Is the algorithm for strongly connected components still correct if line (24) is changed to

then $lowpt[v] \leftarrow \min(lowpt[v], lowpt[w])$?

8) Let $G = (V, E)$ be an undirected graph. $G' = (V, E')$ is a **minimal biconnected extension** of G if $E \subseteq E'$, G' is biconnected, and $|E'|$ is as small as possible. Develop an algorithm to compute minimal biconnected extensions. [Hint: Solve the problem for trees first. Extend to general graphs as follows. Let $V_1, \ldots, V_k$ be the b.c.c.'s of G. Define a graph with node set $V_1, \ldots, V_k$ and edges (V_i, V_j) iff $V_i \cap V_j \neq \emptyset$. This graph is a tree.]

9) Derive a bound $g(n)$ on the maximal number of iterations of the basic least cost path algorithm (cf. the beginning of Section 4.7) on a network of n nodes. Design networks where the algorithm might actually need approximately $g(n)$ iterations.

10) Extend all least cost path algorithms so that they not only compute the least cost of a path but also the path itself. Running time should not change. [Hint: Have array $Pred[1\,..\,n]$; whenever $cost[v]$ is changed when considering edge (u, v) set $Pred[v]$ to u. Then array $Pred$ stores a tree of least cost paths after termination.]

11) Let $lp(s, v) = \max\{c(p);\ p \text{ is a path from } s \text{ to } v\}$. Derive algorithms for computing $lp(s, v)$ for all $v \in V$ on various assumptions about the underlying network.

12) (Extension of 4.7.2, Theorem 4.) Let g_1, g_2 be estimators and let $g_1(v) \geq g_2(v)$ for all $v \in V$. Let R_i be the set of nodes removed from U when estimator g_i is used. Then $R_1 - R_2 \subseteq \{v;\ \mu(s, v) + g_1(v) = \mu(s, t)\}$ provided that g_1 is consistent.

13) Construct an instance of a least cost path problem and an estimator g such that some nodes are removed from U more than once.

14) Consider the following well-known "15-puzzle". The board consists of a 3 by 3 square with eight 1 by 1 tokens numbered 1 to 8 arranged on the board. One square of the board is empty. The goal is to arrange the tokens in ascending order.

7	3	1
2	8	
4	6	5

a) Formulate this puzzle as a path finding problem. What are the nodes and what are the edges of the graph?

b) Use the path-finding algorithm of Section 4.7.2 to find a solution. Use the following three estimators: constant zero, number of tokens out of place, total distance of tokens from their final position.

15) Show that the algorithm of Section 4.7.3 has running time $O(k_{max} \cdot e)$ where k_{max} is the length (number of edges) of the longest least cost path from s to any $v \in V$.

16) For $v \in V$ let $cost_i[v] = \min\{c(p);\ p \text{ is a path from } s \text{ to } v \text{ of length at most } i\}$, $i > 0$. Show how to compute array $cost_i[1\,..\,n]$ from array $cost_{i-1}[1\,..\,n]$ in time $O(e)$. Conclude that the single source least cost path problem can be solved in time $O(n \cdot e)$. Relate this algorithm to the algorithm described in Section 4.7.3. Relate the algorithm of this exercise to dynamic programming in general.

17) Is it a good idea to realize set U as a stack instead of a queue in the algorithm of Section 4.7.3? Is it a good idea to replace the array $count[1..n]$ of counters by a single counter $count$ which counts iterations of the loop?

18) Let $N = (V, E, c)$, $c : E \to \mathbb{R}$, be a network. Let $E_p = \{(v, w) \in E;\ c(v, w) \geq 0\}$ and let $E_n = \{(v, w) \in E;\ c(v, w) < 0\}$. If N does not have any negative cycles then (V, E_n) is acyclic. Show that one can solve a single source least cost path problem by repeatedly (at most n times) solving the problem for $N_p = (V, E_p, c)$ and $N_n = (V, E_n, c)$. Here function *cost* as computed by one algorithm is used as input for the other algorithm. Show that this idea leads to an $O(\min(n \cdot e + n^{2+1/k}, (n^2 + n \cdot e) \log n))$ algorithm for arbitrary integer k.

19) Consider the following algorithm for solving the single source least cost path problem. Let $E = \{e_1, \ldots, e_m\}$. Use the basic algorithm of the beginning of Section 4.7. Go through the elements of E in *cyclic* order and check for the triangle inequality. Prove that this algorithm runs in time $O(n \cdot e)$.

20) Design and analyze algorithms for maximum cost spanning trees.

21) Let $N = (V, E, c)$ be a network and let $s, t \in V$. Let f be a legal flow function. Show that

$$val(f) = \sum_{e \in in(t)} f(e) - \sum_{e \in out(t)} f(e).$$

22) Let f be a legal (s, t)-flow in network N. Define the augmenting network AN with respect to f by $AN = (V, E_1 \cup E_2, \bar{c})$ where E_1, E_2 are defined as in the definition of the layered network. Note that AN captures *all* augmenting paths while LN captures only the minimum length augmenting path.

a) Construct AN for the example at the beginning of Section 4.9.1.

b) Show that an analog of Lemma 2 is true with AN instead of LN.

c) Define the concept of blocking flow and depth for augmenting networks. Does Lemma 3 still hold true? [Hint: check part b) of the claim in Lemma 3 carefully.]

23) Let $N = (V, E, c)$ be a network with integral capacities, i.e., $c : V \to \mathbb{N}$. Let $vmax$ be the maximal value of any legal (s, t)-flow in N. Show that $vmax$ augmentations suffice to construct a maximal flow, where an augmentation can be carried out along any augmenting path.

24) Show that $O(\log vmax)$ augmentations suffice under the assumptions of Exercise 23 if the augmentation is always carried out along an augmenting path of maximal capacity.

25) Design efficient algorithms for each of the following versions of the max-flow problem by reducing it to the standard version:

a) The nodes, as well as the arcs, have capacities.

b) There are many sources and sinks.

c) The network is undirected.

d) There are both upper and lower bounds on the value of the flow through each arc.

26) In the $O(n^2)$ algorithm for computing a blocking flow in a layered network we first determined a node v with $PO(v) = PO^*$ and then "forwarded" and "backwarded" the flow starting at v.

a) Show that the algorithm stays correct if we only forward the flow, but start at node s.

b) Can you still prove the $O(n^2)$ time bound?

27) Describe an algorithm for procedure *simplify* in detail.

28) Adapt the $O(n^2)$ blocking flow algorithm to (0,1)-networks. Avoid the recomputation of $PO[v]$ for all $v \in V$ in line (5). Instead, compute $PO[v]$ once and update it as edges incident to v are removed in procedures *forward*, *suck* and *simplify*. Also, have an array $L[1..e]$ of linear lists. In list $L[1]$ store all nodes v with $PO[v] = 1$. Keep a pointer in this list pointing to the leftmost non-empty list. Move this pointer to the right in order to find $\min\{PO[v];\ v \in V\}$ in line (5), move it to the left when potentials are updated. Show that the total number of moves of the pointer is $O(e)$. Conclude that a blocking flow in a (0,1)-network can be computed in time $O(e)$.

29) A network $N = (V, E, c)$, $s, t \in V$, is (s,t)-planar if (V, E) is a planar graph and if s and t border the same face of the planar graph. Consider an embedding of (V, E) where s and t border the outer face. In this situation there is a natural order on the set of paths from s to t. (Path p_1 is above path p_2 if $p_1 = p'e_1p''$ and $p_2 = p'e_2p'''$ and e_1 is "above" e_2; cf. Figure 121.) Let $p_1, p_2, p_3, \ldots, p_m$ be the set of paths from s to t ordered according to the property of being above another path.

Figure 121. Ordering of paths from s to t

a) Construct a blocking flow by first saturating an edge of p_1, then an edge of p_2, etc. Show that the constructed flow is maximal. [Hint: Let c_1 be the capacity of p_1; show that there is a maximal flow which sends c_1 units across every edge of p_1. Assume otherwise. Let p_1 consist of edges $e_1, e_2, \ldots, e_k$. Let f be a maximal flow function such that $f(e_1), \ldots, f(e_i) \geq c_1$, $f(e_{i+1}) < c_1$ and no maximal flow f' satisfies $f'(e_1), \ldots, f'(e_i) \geq c_1$, $f'(e_{i+1}) > f(e_{i+1})$. Then $c_1 - f(e_{i+1})$ units must be transported from v to t along some path p' (see Figure 122). Let $j > i+1$ be minimal such that $f(e_j) \geq c_1$. (If j does not exist the argument becomes simpler.) Then $f(e_j) - f(e_{j-1}) > 0$ units of flow are transported from s to w along some path p''. Since the network is assumed to be planar p' and p'' must converge in some node, say x. It is now easy to divert some flow from the path $v \xrightarrow{*} x \xrightarrow{*} w$ to path p_1 thus contradicting the existence of f. This proves that there is a maximal flow which sends c_1 units along each edge of p_1. The correctness proof is now completed by induction.]

Figure 122. Exercise 29a)

b) Show how to implement the algorithm outlined in part a) in time $O(e \cdot \log n)$. [Hint: Use the blocking flow algorithm described in the text without change. Explain how edges around nodes have to be ordered. Show that lines (5) and (6) are executed at most e times. This follows from the observation that p always points into pf from below in lines (4)–(6) and hence pf' can be discarded because it will never be the case that p points into pf'. This will discard at least one edge except when pf' is trivial, i.e., $last(p) = first(pf)$. However, this can happen only if line (3) is executed immediately before.]

Figure 123. Exercise 29b)

30) A network flow problem with upper and lower bounds is given by a directed graph $G = (V, E)$, source s, sink t and two capacity functions $low : E \to \mathbb{R}$ and

$high : E \to \mathbb{R}$. A legal (s,t)-flow f must satisfy the conservation laws and the capacity constraints: $low(e) \le f(e) \le high(e)$ for all $e \in E$.

a) Show that the problem whether a legal flow exists can be reduced to an ordinary network flow problem. [Hint: Let $\overline{V} = V \cup \{\bar{s}, \bar{t}\}$, let $\overline{E} = E \cup (\{\bar{s}\} \times V) \cup (V \times \{\bar{t}\}) \cup \{(s,t),(t,s)\}$ and let $\bar{c} : E \to \mathbb{R}^+$ be defined by $\bar{c}(e) = high(e) - low(e)$ for $e \in E$, $\bar{c}(\bar{s}, v) = \sum_{e \in in(v)} low(e)$, $\bar{c}(v, \bar{t}) = \sum_{e \in out(v)} low(e)$, and $\bar{c}(s,t) = \bar{c}(t,s) = \infty$. Show that there is a legal flow iff the maximum flow in the auxiliary network $\overline{N}$ saturates all edges emanating from $\bar{s}$.]

b) Show how to compute a maximal flow in a network with upper and lower bounds [Hint: Start with a legal flow as constructed in a) and use augmentation.]

31) Let $G = (V_1 \cup V_2, E)$, $E \subseteq V_1 \times V_2$, be a bipartite graph with $|V_1| \le |V_2|$. Show: G has a complete matching M, i.e., $|M| = |V_1|$, if for all $S \subseteq V_1$ holds: $|\{w \in V_2;\ (v,w) \in E \text{ for some } v \in S\}| \ge |S|$.

32) Let $N = (V, E, cap, cost)$ be a weighted network and let f be a legal (s,t)-flow. Show how to compute a legal (s,t)-flow g from f with $val(g) = val(f)$ and minimal cost. Running time?

33) Let T be an undirected tree where every node has degree at most d. Show that there is a node v of T such that the removal of v splits T into subtrees of at most $(d-1) \cdot n/d$ nodes each.

34) Let $G = (V, E)$ be a planar graph. Show that there is a partition A, S, B of V such that $|A| \le n/2$, $|B| \le n/2$, $S = O(\sqrt{n})$, and S separates A from B. Moreover A, S, B can be found in linear time.

35) Let A be a symmetric, positive definite matrix. Show: If A is the adjacency matrix of a planar graph $G = (V, E)$, i.e., $(i,j) \in E$ iff $a_{ij} \ne 0$, then the linear system $A \cdot x = b$ can be solved in time $O(n^{3/2})$. [Hint: Let V_1, S, V_2 be a partition of V as given by the planar separator theorem; let P be a permutation matrix such that $P \cdot A \cdot P^{-1}$ has the form

$$\begin{array}{r} V_1 \{ \\ V_2 \{ \\ S \{ \\ \\ \end{array} \begin{pmatrix} A_1 & & A_3 \\ \hline & A_2 & A_4 \\ \hline \underbrace{A_3}_{V_1} & \underbrace{A_4}_{V_2} & \underbrace{A_5}_{S} \end{pmatrix}$$

Apply a similar reordering to submatrices A_1, A_2. Use Gaussian elimination on the reordered matrix. Study carefully, which entries of the matrix become non-zero during Gaussian elimination.]

36) Let $G = (V, E)$ be a directed planar graph. Show that one can construct a transitive reduction of G, i.e., a smallest graph with the same transitive closure, in time $O(n^{3/2})$. [Hint: Use the planar separator theorem.]

37) Let $G = (V, E)$ be a planar graph, let $w : V \to \mathbb{R}_0^+$ be a weight function, and let $W = \sum_{v \in V} w(v)$. Show that there is a partition A, S, B of V such that $|S| \leq 8 \cdot \sqrt{n}$, $|A| \leq 2 \cdot n/3$, $W(A) \leq 2 \cdot W/3$, $|B| \leq 2 \cdot n/3$, $W(B) \leq 2 \cdot W/3$ and S separates A from B. [Hint: Apply Theorem 3 to G, then apply Corollary 1 to the heavier part.]

38) Let $G = (V, E)$ be a planar graph, let $w : V \to \mathbb{R}_0^+$ be a weight function and let $0 < \epsilon \leq 1/2$. Show that there is a subset $S \subseteq V$ such that $|S| = O(\sqrt{n/\epsilon})$ and such that no connected component of $G - S$ has weight exceeding $\epsilon \cdot W$. [Hint: Use Exercise 37 repeatedly.]

39) Let $G = (V, E)$ be a planar graph. A subset $V' \subseteq V$ is **independent** if $(V' \times V') \cap E = \emptyset$. The problem of deciding whether there is an independent set of size m is NP-complete (cf. Chapter VI). Show how to find a nearly maximal independent set efficiently in planar graphs. [Hint: Use Exercise 38 with $w(v) = 1$ for all $v \in V$ and $\epsilon = (\log \log n)/n$. Find maximal independent sets of all components of $G - S$ by exhaustive search and output the union of these sets. Show that $(|I| - |I^*|)/|I| = O(1/\sqrt{\log \log n})$ where I is the independent set computed by the algorithm and where I^* is a maximum independent set. Observe that $|I^*| = \Omega(n)$ since a planar graph has a large number of nodes of small degree.]

40) Show that a maximum independent set of a planar graph can be found in time $2^{O(\sqrt{n})}$. [Hint: Split V into V_1, S, V_2 as given by the planar separator theorem. For every $S' \subseteq S$ find a maximal independent set I of the subgraph induced by $V_1 \cup S$ $(V_2 \cup S)$ such that $I \cap S = S'$ by recursive application of the algorithm.]

41) Show how to find the chromatic number of a planar graph in time $2^{O(\sqrt{n})}$. [Hint: Proceed as in the preceding exercise.]

4.12. Bibliographic Notes

The algorithm for topological sorting is due to Kahn (62) and Knuth (68). A detailed analysis of the representation problem can be found in Rivest/Vuillemin (75). The $O(n \cdot e_{red})$ algorithm for the computation of the transitive closure of digraphs is by Goralcikova/Koubek (79). The analysis for random acyclic digraphs and the improved closure algorithm have not appeared before; they were done jointly with K. Simon (Simon (83)). A linear expected time algorithm for random digraphs is described in Schnorr (78). Algorithms for the systematic exploration of a graph (maze) are very old and date back to the 19th century at least. Depth-first-search was made popular by Tarjan (72) and Sections 4.5 and 4.6 are adopted from his paper.

The presentation of the basic algorithm for least cost paths follows Johnson (77); Theorem 2c) is also due to him. Theorem 2a) is taken over from Dijkstra (59). The discussion on the use of estimators for solving one pair least cost path problems is based on Hart/Nilsson/Raphael. An algorithm which solves the all pairs problem on nonnegative networks in expected time $O(n^2 \cdot \log n \cdot \log^* n)$ is discussed in Bloniarz (80). The treatment of the general case follows Bellmann (58), Floyd (62) for Theorem 5 and Exercise 16, Edmonds/Karp (72) for Lemma 4, and Johnson (77) for Theorem 7 and Exercise 18.

The section on minimum spanning trees combines the work of Kruskal (56) (Theorem 1), Prim (57) and Dijkstra (59) (Theorem 2), Yao (75) (Theorems 3 and 4) and Cheriton/Tarjan (76) (Theorems 3 and 4). The paper by Cheriton/Tarjan contains even better algorithms than the ones described in the text.

Many fundamental results on network flow, in particular Theorem 3, are due to Ford/Fulkerson (62). The $O(n^3)$ algorithm is from Malhotra et al. (78) who refine an algorithm due to Karzanov (74). The $O(n^2 \cdot e)$ algorithm underlying Theorem 4 was invented by Dinic (70) and then refined to an $O(n \cdot e \cdot (\log n)^2)$ algorithm by Galil/Naamad (79). An $O(e \cdot n \log n)$ algorithm was recently described by Sleator/Tarjan (Sleator (79)). Theorem 7 is also due to Galil/Naamad (79). Section 4.9.2 on (0,1)-Networks combines work of Even/Tarjan (75) (Theorems 7, 8 and 10a)), Hopcroft/Karp (75) (Theorem 9) and Becker et al. (82) (Theorem 10b)). Weighted network flow was treated by Jewel (58), Busacker/Gowen (61) (Lemma 13) and Edmonds/Karp (72) (Lemma 14). Exercise 29 is from Itai/Shiloach (79) and Galil/Naamad (79). The linear time planarity testing algorithm is due to Hopcroft/Tarjan (72). The planar separator theorem and many of its applications (Exercises 35, 37–41) are from Lipton/Tarjan (77,77). The application to least cost path computations is taken over from Mehlhorn/Schmidt (83). An $O(n^{3/2})$ algorithm for least cost path computations in planar graphs was described by Tarjan (81). Exercise 36 was proposed by Th. Lengauer.