Chapter 1

PATHS IN GRAPHS

1.1 INTRODUCTION TO GRAPH THEORY

A *graph* $G(V, E)$ is a structure which consists of a set of *vertices* $V = \{v_1, v_2, \ldots\}$ and a set of edges $E = \{e_1, e_2, \ldots\}$; each edge e is *incident* to the elements of an unordered pair of vertices $\{u, v\}$ which are not necessarily distinct.

Unless otherwise stated, both V and E are assumed to be finite. In this case we say that G is finite.

For example, consider the graph represented in Figure 1.1. Here $V = \{v_1, v_2, v_3, v_4, v_5\}$, $E = \{e_1, e_2, e_3, e_4, e_5\}$. The edge e_2 is incident to v_1 and v_2, which are called its *endpoints*. The edges e_4 and e_5 have the same endpoints and therefore are called *parallel* edges. Both endpoints of the edge e_1 are the same; such an edge is called a *self-loop*.

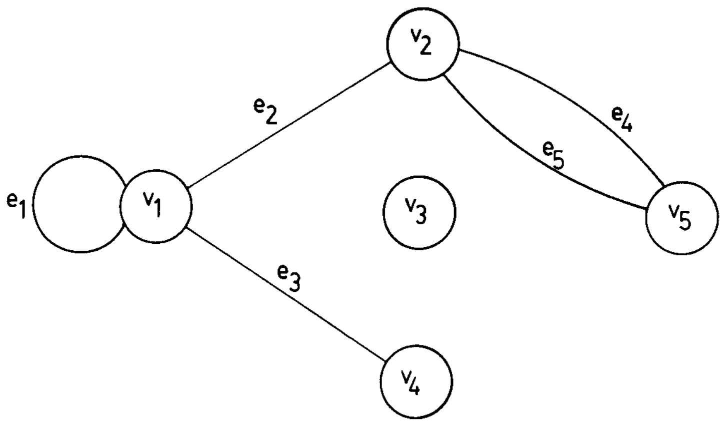


Figure 1.1

The *degree* of a vertex v, $d(v)$, is the number of times v is used as an endpoint of the edges. Clearly, a self-loop uses its endpoint twice. Thus, in our example $d(v_4) = 1$, $d(v_2) = 3$ and $d(v_1) = 4$. Also, a vertex v whose degree is zero is called *isolated*; in our example v_3 is isolated since $d(v_3) = 0$.

Lemma 1.1 *The number of vertices of odd degree in a finite graph is even.*

Proof: Let $|V|$ and $|E|$ be the number of vertices and edges, respectively. Then,

$$\sum_{i=1}^{|V|} d(v_i) = 2 \cdot |E|,$$

since each edge contributes two to the left hand side; one to the degree of each of its two endpoints, if they are different, and two to the degree of its endpoint if it is a self-loop. It follows that the number of odd degrees must be even.

Q.E.D.

The notation $u \overset{e}{—} v$ means that the edge e has u and v as endpoints. In this case we also say that e *connects* vertices u and v, and that u and v are *adjacent.*

A *path* is a sequence of edges $e_1, e_2, \ldots$ such that:

(1) e_i and e_{i+1} have a common endpoint;

(2) if e_i is not a self-loop and is not the first or last edge then it shares one of its endpoints with e_{i-1} and the other with e_{i+1}.

The exception specified in (2) is necessary to avoid the following situation: Consider the graph represented in Figure 1.2.

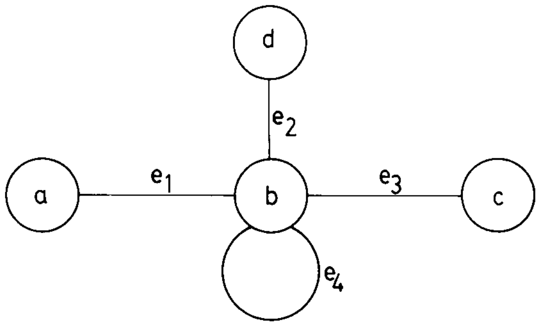


Figure 1.2

We do not like to call the sequence e_1, e_2, e_3 a path, and it is not, since the only vertex, b, which is shared by e_1 and e_2 is also the only vertex shared by e_2 and e_3. But we have no objection to calling e_1, e_4, e_3 a path. Also, the sequence e_1, e_2, e_2, e_3 is a path since e_1 and e_2 share b, e_2 and e_2 share d, e_2 and e_3 share b. It is convenient to describe a path as follows: $v_0 \overset{e_1}{—} v_1 \overset{e_2}{—} v_2 \cdots v_{l-1} \overset{e_l}{—} v_l$. Here the path is $e_1, e_2, \ldots, e_l$ and the endpoints shared are transparent; v_0 is called the *start* and v_l is called the *end* vertex. The *length* of the path is l.

A *circuit* is a path whose start and end vertices are the same.

A path is called *simple* if no vertex appears on it more than once. A circuit is called *simple* if no vertex, other than the start-end vertex, appears more than once, and the start-end vertex does not appear elsewhere in the circuit; however, $u \overset{e}{—} v \overset{e}{—} u$ is not considered a simple circuit.

If for every two vertices u and v there exists a path whose start vertex is u and whose end vertex is v then the graph is called *connected.*

A *digraph* (or *directed graph*) is defined similarly to a graph except that the pair of endpoints of an edge is now ordered; the first endpoint is called the *start-vertex* of the edge and the second (which may be the same) is called its *end-vertex.* The edge $(u \xrightarrow{e} v)$ e is said to be *directed* from u to v. Edges with the same start vertex and the same end vertex are called *parallel,* and if $u \neq v$, $u \xrightarrow{e_1} v$ and $v \xrightarrow{e_2} u$ then e_1 and e_2 are *antiparallel.* An edge $u \rightarrow u$ is called a *self-loop.*

The *outdegree*, $d_{out}(v)$, of a vertex v is the number of edges which have v as their start-vertex; *indegree*, $d_{in}(v)$, is defined similarly. Clearly, for every graph

$$\sum_{i=1}^{|V|} d_{in}(v_i) = \sum_{i=1}^{|V|} d_{out}(v_i).$$

A *directed path* is a sequence of edges $e_1, e_2, \ldots$ such that the end vertex of e_{i-1} is the start vertex of e_i. A directed path is a *directed circuit* if the start vertex of the path is the same as its end vertex. The notion of a directed path or circuit being *simple* is defined similarly to that in the undirected case. A digraph is said to be *strongly connected* if for every vertex u and every vertex v there is a directed path from u to v; namely, its *start-vertex* is u and its *end-vertex* is v.

1.2 COMPUTER REPRESENTATION OF GRAPHS

In order to understand the time and space complexities of graph algorithms one needs to know how graphs are represented in the computer memory. In this section two of the most common methods of graph representation are briefly described.

Graphs and digraphs which have no parallel edges are called *simple*. In cases of simple graphs, the specification of the two endpoints is sufficient to specify the edge; in cases of digraph the specification of the start-vertex and end-vertex is sufficient. Thus, we can represent a graph or digraph of n vertices by an $n \times n$ matrix C, where $C_{ij} = 1$ if there is an edge connecting vertex v_i to v_j and $C_{ij} = 0$, if not. Clearly, in the case of graphs $C_{ij} = 1$ implies $C_{ji} = 1$; or in other words, C is symmetric. But in the case of digraphs, any $n \times n$ matrix of zeros and ones is possible. This matrix is called the *adjacency matrix*.

Given the adjacency matrix of a graph, one can compute $d(v_i)$ by counting the number of ones in the i-th row, except that a one on the main diagonal contributes two to the count. For a digraph, the number of ones in the i row is equal to $d_{out}(v_i)$ and the number of ones in the i column is equal to $d_{in}(v_i)$.

The adjacency matrix is not an efficient representation of the graph in case the graph is *sparse*; namely, the number of edges is significantly smaller than n^2. In these cases the following representation, which also allows parallel edges, is preferred.

For each of the vertices, the edges incident to it are listed. This *incidence list* may simply be an array or may be a linked list. We may need a table which tells us the location of the list for each vertex and a table which tells us for each edge its two endpoints (or start-vertex and end-vertex, in case of a digraph).

We can now trace a path starting from a vertex, by taking the first edge on its incidence list, look up its other endpoint in the edge table, finding the incidence list of this new vertex etc. This saves the time of scanning the row of the matrix, looking for a one. However, the saving is real only if n is large and the graph is sparse, for instead of using one bit per edge, we now use edge names and auxiliary pointers necessary in our data structure. Clearly, the space required is $O(|E| + |V|)$, i.e., bounded by a constant times $|E| + |V|$. Here we assume that the basic word length of our computer is large enough to encode all edges and vertices. If this assumption is false then the space required is $O((|E| + |V|) \log(|E| + |V|))^*$.

In practice, most graphs are sparse. Namely, the ratio $(|E| + |V|)/|V|^2$ tends to zero as the size of the graphs increases. Therefore, we shall prefer the use of incidence lists to that of adjacency matrices.

The reader can find more about data structures and their uses in graph theoretic algorithms in references [1] and [2].

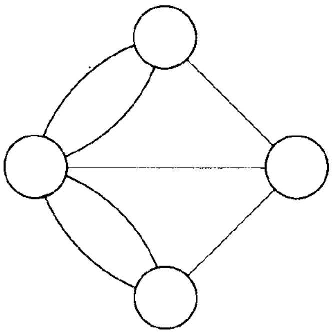
*The base of the log is unimportant (clearly greater than one), since this estimate is only up to a constant multiplier.

1.3 EULER GRAPHS

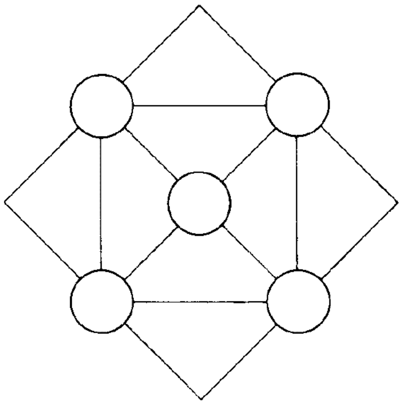
An *Euler path* of a finite undirected graph $G(V, E)$ is a path $e_1, e_2, \ldots, e_l$ such that every edge appears on it exactly once; thus, $l = |E|$. An undirected graph which has an Euler path is called an *Euler graph.*

Theorem 1.1 *A finite (undirected) connected graph is an Euler graph if and only if exactly two vertices are of odd degree or all vertices are of even degree. In the latter case, every Euler path of the graph is a circuit, and in the former case, none is.*

As an immediate conclusion of Theorem 1.1 we observe that none of the graphs in Figure 1.3 is an Euler graph, because both have four vertices of odd degree. The graph shown in Figure 1.3(a) is the famous *Königsberg bridge problem* solved by Euler in 1736. The graph shown in Figure 1.3(b) is a common misleading puzzle of the type “draw without lifting your pen from the paper”.



(a)



(b)

Figure 1.3

Proof: It is clear that if a graph has an Euler path which is not a circuit, then the start vertex and the end vertex of the path are of odd degree, while all the other vertices are of even degree. Also, if a graph has an Euler circuit, then all vertices are of even degree.

Assume now that G is a finite graph with exactly two vertices of odd degree, a and b. We shall described now an algorithm for finding an Euler path from a to b. Starting from a we choose any edge adjacent to it (an edge of which a is an endpoint) and trace it (go to its other endpoint). Upon entering a vertex we search for an unused incident edge. If the vertex is neither a nor b, each time we pass through it we use up two of its incident edges. The degree of the vertex is even. Thus, the number of unused incident edges after leaving it is even. (Here again, a self-loop is counted twice.) Therefore, upon entering it there is at least one unused incident edge to leave by. Also, by a similar argument, whenever we reenter a we have an unused edge to leave by. It follows that the only place this process can stop is in b. So far we have found a path which starts in a, ends in b, and the number of unused edges incident to any vertex is even. Since the graph is connected, there must be at least one unused edge which is incident to one of the vertices on the existing path from a to b. Starting a trail from this vertex on unused edges, the only vertex in which this process can end (because no continuation can be found) is the vertex in which it started. Thus, we have found a circuit of edges which were not used before, and in which each edge is used at most once: it starts and ends in a vertex visited in the previous path. It is easy to change our path from a to b to include this detour. We continue to add such detours to our path as long as not all edges are in it.

The case of all vertices of even degrees is similar. The only difference is that we start the initial tour at any vertex, and this tour must stop at the same vertex. This initial circuit is amended as before, until all edges are included.

Q.E.D.

In the case of digraphs, a *directed Euler path* is a directed path in which every edge appears exactly once. A *directed Euler circuit* is defined similarly. Also a *digraph* is called *Euler* if it has a directed Euler path (or circuit).

The *underlying* (undirected) *graph* of a digraph is the graph resulting from the digraph if the direction of the edges is ignored. Thus, the underlying graph of the digraph shown in Figure 1.4(a) is shown in Figure 1.4(b).

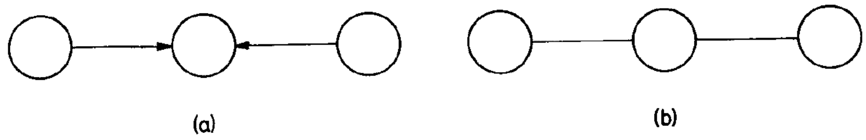


Figure 1.4

Theorem 1.2 *A finite digraph is an Euler digraph if any only if its underlying graph is connected and one of the following two conditions holds:*

1. *There is one vertex a such that $d_{out}(a) = d_{in}(a)+1$ and another vertex b such that $d_{out}(b)+1 = d_{in}(b)$, while for all other vertices v, $d_{out}(v) = d_{in}(v)$.*
2. *For all vertices v, $d_{out}(v) = d_{in}(v)$.*

If 1 holds then every directed Euler path starts in a and ends in b. If 2 holds then every directed Euler path is a directed Euler circuit.

The proof of the theorem is along the same lines as the proof of Theorem 1.1, and will not be repeated here.

Let us make now a few comments about the complexity of the algorithm for finding an Euler path, as described in the proof of Theorem 1.1. Our purpose is to show that the time complexity of the algorithm is $0(|E|)$; namely, there exists a constant K such that the time it takes to find an Euler path is bounded by $K \cdot |E|$.

In the implementation, we use the following data structures:

1. Incidence lists which describe the graph.
2. A doubly-linked list of edges P describing the path. Initially this list is empty.
3. A vertex table, specifying for each vertex v the following data:
   (a) A mark which tells whether v appears already on the path. Initially all vertices are marked "unvisited".
   (b) A pointer $N(v)$, to the next edge on the incidence list, which is the first not to have been traced from v before. Initially $N(v)$ points to the first edge on v's incidence list.
   (c) A pointer $E(v)$ to an edge on the path which has been traced from v. Initially $E(v)$ is "undefined".

4. An edge table which specified for each edge its two endpoints and whether it has been used. Initially, all edges are marked "unused".

5. A list L of vertices all of which have been visited. Each vertex enters this list at most once.

First let us describe a subroutine TRACE(d, P), where d is a vertex and P is a doubly linked list, initially empty, for storage of a traced path. The tracing starts in d and ends when the path, stored in P, cannot be extended.

TRACE(d, P):

(1) $v \leftarrow d$

(2) If v is "unvisited", put it in L and mark it "visited".

(3) If $N(v)$ is "used" but is not last on v's incidence list then have $N(v)$ point to the next edge and repeat (3).

(4) If $N(v)$ is "used" and it is the last edge on v's incidence list then stop.

(5) $e \leftarrow N(v)$

(6) Add e to the end of P.

(7) If $E(v)$ is "undefined" then $E(v)$ is made to point to the occurrence of e in P.

(8) Mark e "used".

(9) Use the edge table to find the other endpoint u of e.

(10) $v \leftarrow u$ and go to (2).

The algorithm is now as follows:

(1) $d \leftarrow a$

(2) TRACE(d, P). [Comment: The subroutine finds a path from a to b.]

(3) If L is empty, stop.

(4) Let u be in L. Remove u from L.

(5) Start a new doubly linked list of edges, P', which is initially empty. [Comment: P' is to contain the detour from u.]

(6) TRACE(u, P')

(7) Incorporate P' into P at $E(u)$. [Comment: This joins the path and the detour into one, possibly longer path. (The detour may be empty.) Since the edge $E(u)$ starts from u, the detour is incorporated in a correct place.]

(8) Go to (3).

It is not hard to see that both the time and space complexity of this algorithm is $O(|E|)$.

1.4 DE BRUIJN SEQUENCES

Let $\Sigma = \{0, 1, \ldots, \sigma - 1\}$ be an alphabet of σ letters. Clearly there are σ^n different words of length n over Σ. A *de Bruijn sequence** is a (circular) sequence $a_0 a_1 \cdots a_{L-1}$ over Σ such that for every word w of length n over Σ there exists a unique i such that

$$a_i a_{i+1} \cdots a_{i+n-1} = w,$$

where the computation of the indices is modulo L. Clearly if the sequence satisfies this condition, then $L = \sigma^n$. The most important case is that of $\sigma = 2$. Binary de Bruijn sequences are of great importance in coding theory and are implemented by shift registers. (See Golomb's book [3] on the subject.) The interested reader can find more information on de Bruijn sequences in references [4] and [5]. The only problem we shall discuss here is the existence of de Bruijn sequences for every $\sigma \geq 2$ and every n.

Let us describe a digraph $G_{\sigma,n}(V, E)$ which has the following structure:

1. V is the set of all σ^{n-1} words of length $n - 1$ over Σ.

2. E is the set of all σ^n words of length n over Σ.

3. The edge $b_1 b_2 \cdots b_n$ starts at vertex $b_1 b_2 \cdots b_{n-1}$ and ends at vertex $b_2 b_3 \cdots b_n$.

The graphs $G_{2,3}$, $G_{2,4}$, and $G_{3,2}$ are shown in Figures 1.5, 1.6 and 1.7 respectively.

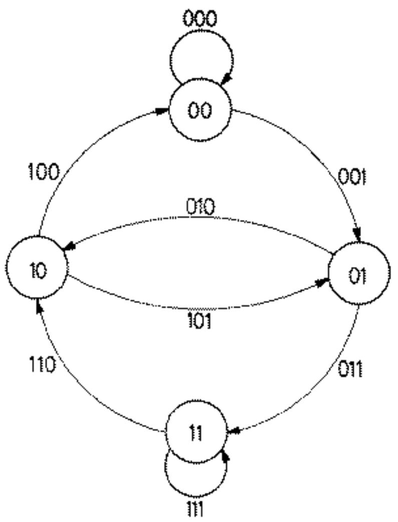


Figure 1.5

These graphs are sometimes called de Bruijn diagrams, or Good's diagrams, or shift register state diagrams. The structure of the graphs is such that the word w_2 can follow the word w_1 in a de Bruijn sequence only if the edge w_2 starts at the vertex in which w_1 ends. Also it is clear that if we find a directed Euler circuit (a directed circuit which uses each of the graph's edges exactly once) of $G_{\sigma,n}$, then we also have a de Bruijn sequence. For example, consider the directed Euler circuit of $G_{2,3}$ (Figure 1.5) consisting of the following sequence of edges:

$$000, 001, 011, 111, 110, 101, 010, 100.$$

*Sometimes they are called *maximum-length shift register sequences.*

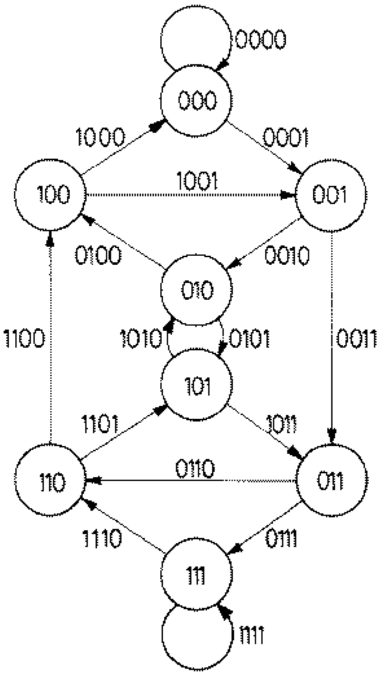


Figure 1.6

The implied de Bruijn sequence, 00011101, follows by reading the first letter of each word in the circuit. Thus, the question of existence of de Bruijn sequences is equivalent to that of the existence of directed Euler circuits in the corresponding de Bruijn diagram.

Theorem 1.3 *For every positive integers σ and n, $G_{\sigma,n}$ has a directed Euler circuit.*

Proof: We wish to use Theorem 1.2 to prove our theorem. First we have to show that the underlying undirected graph is connected. In fact, we shall show that $G_{\sigma,n}$ is strongly connected. Let $b_1b_2\cdots b_{n-1}$ and $c_1c_2\cdots c_{n-1}$ be any two vertices; the directed path $b_1b_2\cdots b_{n-1}c_1$, $b_2b_3\cdots b_{n-1}c_1c_2$, $\cdots$, $b_{n-1}c_1c_2\cdots c_{n-1}$ leads from the first to the second. Next, we have to show that $d_{out}(v) = d_{in}(v)$ for each vertex v. The vertex $b_1b_2\cdots b_{n-1}$ is entered by edges $cb_1b_2\cdots b_{n-1}$, where c can be chosen in σ ways, and is the start vertex of edges $b_1b_2\cdots b_{n-1}c$, where again c can be chosen in σ ways.

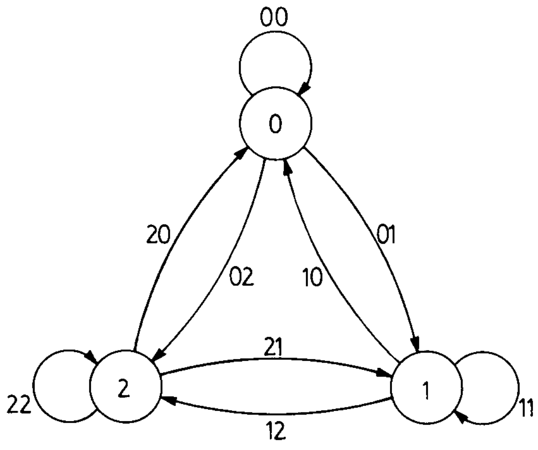


Figure 1.7

Q.E.D.

Corollary 1.1 *For every positive integers σ and n there exists a de Bruijn sequence:*

1.5 SHORTEST-PATH ALGORITHMS

In general the shortest-path problems are concerned with finding shortest paths between vertices. Many interesting problems arise, and the variety depends on the type of graph in our application and the exact question we want to answer. Some of the characteristics which may help in defining the exact problem are as follows:

1. The graph is finite or infinite.
2. The graph is undirected or directed.
3. The edges are all of length 1, or all lengths are non-negative, or negative lengths are allowed.
4. We may be interested in shortest paths from a given vertex to another, or from a given vertex to all the other vertices, or from each vertex to all the other vertices.
5. We may be interested in finding just one path, or all paths, or counting the number of shortest paths.

Clearly, this section will deal only with very few of all the possible problems. An attempt is made to describe the most important techniques.

First let us consider the case of a finite graph G in which two vertices s and t are specified. Our task is to find a path from s to t, if there are any, which uses the least number of edges. Clearly this is the case of the finite, undirected graph, with all length of edges being equal to 1, and where all we want is one path from a given vertex to another. In fact, the digraph case is just as easy and can be similarly solved.

The algorithm to be used here was suggested by Moore [6] and by now is widely used. It is well known as the *Breadth First Search (BFS)* technique.

At first no vertices of the graph are considered labeled.

1. Label vertex s with 0.
2. $i \leftarrow 0$
3. Find all unlabeled vertices adjacent to at least one vertex labeled i. If none are found, stop.
4. Label all the vertices found in (3) with $i + 1$.
5. If vertex t is labeled, stop.
6. $i \leftarrow i + 1$ and go to (3).

Clearly we can remove step 5 from the algorithm, and the algorithm is still valid for finite graphs. However, step 5 saves the work which would be wasted after t is labeled, and it permits the use of the algorithm on infinite graphs whose vertices are of finite degree and in which there is a (finite) path between s and t.

Let the *distance* between u and v be the least number of edges in a path connecting u and v, if such a path exists, and ∞ if none exists.

Theorem 1.4 *The BFS algorithm computes the distance of each vertex from s, if t is not closer.*

Proof: Let us denote the label of a vertex v, assigned by the BFS algorithm, by $\lambda(v)$.

First we show that if a vertex is labeled $\lambda(v) = k$, then there is a path of length k from s to v. Such a path can be traced as follows: There must be a vertex v_{k-1} adjacent to $v = v_k$, labeled $k-1$, and similarly, there must be a vertex v_{k-i-1} adjacent to v_{k-i} labeled $k-i-1$ for $i = 0, 1, \ldots, k-1$. Clearly $v_0 = s$, since s is the only vertex labeled 0. Thus, $v_0 - v_1 - \cdots v_{k-1} - v_k$ is a path of length k from s to v.

Now, let us prove by induction on l that if v is of distance l from s and if t is not closer to s, then $\lambda(v) = l$.

After Step 1, $\lambda(s) = 0$, and indeed the distance from s to s is zero.

Assume now that the statement holds for shorter distances, let us show that it must hold for l too. Let $s - v_1 - v_2 - \cdots v_{l-1} - v$ be a shortest path from s to v. Clearly, $s - v_1 - v_2 - \cdots v_{l-2} - v_{l-1}$ is a shortest path from s to v_{l-1}. If t is not closer to s than v then clearly it is not closer than v_{l-1} either. By the inductive hypothesis $\lambda(v_{l-1}) = l-1$. When $i = l-1$, v receives the label l. It could not have been labeled before since if it were then its label is less than l, and there is a shorter path from s to v, in contradiction to l's definition.

Q.E.D.

It is clear that each edge is traced at most twice, in this algorithm; once from each of its endpoints. That is, for each i the vertices labeled i are scanned for their incident edges in step 3. Thus, if we use the incidence lists data structures the algorithm will be of time complexity $O(|E|)$.

The directed case is even simpler because each edge is traced at most once.

A path from s to t can be traced by moving now from t to s, as described in the proof of Theorem 1.4. If we leave for each vertex the name of the edge used for labeling it, the tracing is even easier.

Let us now consider the case of a finite digraph, whose edges are assigned with non-negative lengths; thus, each edge e is assigned a length $l(e) \geq 0$. Also, there are two vertices s and t and we want to find a shortest directed path from s to t, where the length of a path is the sum of the lengths of its edges.

The following algorithm is due to Dijkstra [7]:

1. $\lambda(s) \leftarrow 0$ and for all $v \neq s$, $\lambda(v) \leftarrow \infty$.
2. $T \leftarrow V$.
3. Let u be a vertex in T for which $\lambda(u)$ is minimum.
4. If $u = t$, stop.
5. For every edge $u \xrightarrow{e} v$, if $v \in T$ and $\lambda(v) > \lambda(u) + l(e)$ then $\lambda(v) \leftarrow \lambda(u) + l(e)$.
6. $T \leftarrow T - \{u\}$ and go to step 3.

Let us denote the distance of vertex v from s by $\delta(v)$. We want to show that upon termination $\delta(t) = \lambda(t)$; that is, if $\lambda(t)$ is finite than it is equal to $\delta(t)$ and if $\lambda(t)$ is infinite then there is no path from s to t in the digraph.

Lemma 1.2 *In Dijkstra's algorithm, if $\lambda(v)$ is finite then there is a path from s to v whose length is $\lambda(v)$.*

Proof: Let u be the vertex which gave v its present label $\lambda(v)$; namely, $\lambda(u)+l(e) = \lambda(v)$, where $u \xrightarrow{e} v$. After this assignment took place, u did not change its label, since in the following step (step 6) u was removed from the set T (of temporarily assigned vertices) and its label remained fixed from there on. Next, find the vertex which gave u its final label $\lambda(u)$, and repeating this backward search, we trace a path from s to v whose length is exactly $\lambda(v)$. The backward search finds each time a vertex which has left T earlier, and therefore no vertex on this path can be repeated; it can only terminate in s which has been assigned its label in step 1.

Q.E.D.

Lemma 1.3 *In Dijkstra's algorithm, when a vertex is chosen (in Step 3), its label $\lambda(u)$ satisfies $\lambda(u) = \delta(u)$.*

Proof: By induction on the order in which vertices leave the set T. The first one to leave is s, and indeed $\lambda(s) = \delta(s) = 0$.

Assume now that the statement holds for all vertices which left T before u.

If $\lambda(u) = \infty$, let u' be the first vertex whose label $\lambda(u')$ is infinite when it is chosen. Clearly, for every v in T, at this point, $\lambda(v) = \infty$, and for all vertices $v' \in V - T$, $\lambda(v')$ is finite. Therefore, there is no edge with a start-vertex in $V - T$ and end-vertex in T. It follows that there is no path from s (which is in $V - T$) to u (which is in T).

If $\lambda(u)$ is finite, then by Lemma 1.2, $\lambda(u)$ is the length of some path from s to u. Thus, $\lambda(u) \geq \delta(u)$. We have to show that $\lambda(u) > \delta(u)$ is impossible. Let a shortest path from s to u be $s = v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \cdots v_{k-1} \xrightarrow{e_k} v_k = u$. Thus, for every $i = 0, 1, \ldots, k$

$$\delta(v_i) = \sum_{j=1}^{i} l(e_j).$$

Let v_i be the right most vertex on this path to leave T before u. By the inductive hypothesis

$$\lambda(v_i) = \delta(v_i) = \sum_{j=1}^{i} l(e_j).$$

If $v_{i+1} \neq u$, then $\lambda(v_{i+1}) \leq \lambda(v_i) + l(e_{i+1})$ after v_i has left T. Since labels can only decrease if they change at all, when u is chosen $\lambda(v_{i+1})$ still satisfies this inequality. We have:

$$\lambda(v_{i+1}) \leq \lambda(v_i) + l(e_{i+1}) = \delta(v_i) + l(e_{i+1}) = \delta(v_{i+1}) \leq \delta(u),$$

and if $\delta(u) < \lambda(u)$, u should not have been chosen. In case $v_{i+1} = u$, the same argument shows directly that $\lambda(u) \leq \delta(u)$.

Q.E.D.

It is an immediate corollary of Lemma 1.3 that $\lambda(t) = \delta(t)$ upon termination.

Let us now consider the complexity of Dijkstra's algorithm. In step 3, the minimum label of the elements of T has to be found. Clearly this can be done in $|T| - 1$ comparisons. At first $T = V$; it decreases by one each time and the search is repeated $|V|$ times. Thus, the total time spent on step 3 is $O(|V|^2)$. Step 5 can use each edge exactly once. Thus it uses, at most, $O(|E|)$ time. Since it makes no sense to have parallel edges (for all but the shortest can be dropped) or self-loops, $|E| \leq |V| \cdot (|V| - 1)$. Thus, the whole algorithm is of $O(|V|^2)$ complexity.

Clearly, for sparse graphs the BFS algorithm is better; unfortunately it does not work if not all edge lengths are equal.

Dijkstra's algorithm is applicable to undirected graphs too. Simply represent each edge of the graph by two anti-parallel directed edges with the same length. Also, it can be used on infinite graphs, if outgoing degrees are finite and there is a finite directed path from s to t. However, this algorithm is not applicable if $l(e)$ may be negative; Lemma 1.2 still holds, but Lemma 1.3 does not.

Next, an algorithm for finding the distance of all the vertices of a finite digraph from a given vertex s, is described. It allows negative edge lengths, but does not allow a directed circuit whose length (sum of the lengths of its edges) is negative. The algorithm is due to Ford [8, 9]:

1. $\lambda(s) \leftarrow 0$ and for every $v \neq s$, $\lambda(v) \leftarrow \infty$.
2. As long as there is an edge $u \xrightarrow{e} v$ such that $\lambda(v) > \lambda(u) + l(e)$ replace $\lambda(v)$ by $\lambda(u) + l(e)$.

For our purposes ∞ is not greater than $\infty + k$, even if k is negative.

It is not even obvious that the algorithm will terminate; indeed, it will not if there is a directed circuit accessible from s (namely, there is a directed path from s to one of the vertices on the circuit) whose length is negative. By going around this circuit the labels will be decreased and the process can be repeated indefinitely.

Lemma 1.4 *In the Ford algorithm, if $\lambda(v)$ is finite then there is a directed path from s to v whose length is $\lambda(v)$.*

Proof: As in the proof of the similar previous statements, this is proved by displaying a path from s to v, and its construction is backwards, from v to s. First we find the vertex u which gave v its present label $\lambda(v)$. The value of $\lambda(u)$ may have decreased since, but we shall refer to its value $\lambda'(u)$ at the time that it gave v its label. Thus, $\lambda(v) = \lambda'(u) + l(e)$, where $u \xrightarrow{e} v$. We continue from u to the vertex which gave it the value $\lambda'(u)$ etc., each time referring to an earlier time in the running of the algorithm. Therefore, this process must end, and the only place it can end is in s.

Q.E.D.

The lemma above is even true if there are negative length directed circuits. But if there are no such circuits, the path traced in the proof cannot return to a vertex visited earlier. For if it does, then by going around the directed circuit, a vertex improved its own label; this implies that the sum of the edge lengths of the circuit is negative. Therefore we have:

Lemma 1.5 *In the Ford algorithm, if the digraph has no directed circuits of negative length and if $\lambda(v)$ is finite then there is a simple directed path from s to v whose length is $\lambda(v)$.*

Since each value, $\lambda(v)$, corresponds to at least one simple path from s to v, and since the number of simple directed paths in a finite digraph is finite, the number of values possible for $\lambda(v)$ is finite. Thus, the Ford algorithm must terminate.

Lemma 1.6 *For a digraph with no negative directed circuit, upon termination of the Ford algorithm, $\lambda(v) = \delta(v)$ for every vertex v.*

Proof: By Lemma 1.5, $\lambda(v) \geq \delta(v)$. If $\lambda(v) > \delta(v)$, let $s = v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \cdots v_{k-1} \xrightarrow{e_k} v_k = v$ be a shortest path from s to v. Clearly, for every $i = 0, 1, \ldots, k$

$$\delta(v_i) = \sum_{j=1}^{i} l(e_j).$$

Let v_i be the first vertex on this path for which $\lambda(v_i) > \delta(v_i)$. Since $\lambda(v_{i-1}) = \delta(v_{i-1})$, the edge $v_{i-1} \xrightarrow{e_i} v_i$ can be used to lower $\lambda(v_i)$ to $\lambda(v_{i-1}) + l(e_i)$, (which is equal to $\delta(v_i)$). Thus, the algorithm should not have terminated.

Q.E.D.

We can use a simple device to bound the number of operation to $O(|E| \cdot |V|)$. Order the edges: $e_1, e_2, \ldots, e_{|E|}$. Now, perform step 2 by first checking e_1, then e_2, etc., and improving labels accordingly. After the first such sweep, go through additional sweeps, until an entire sweep produces no improvement. If the digraph contains no negative directed circuits then the process will terminate. Furthermore, if a shortest path from s to v consists of k edges, then by the end of the kth sweep v will have its final label; this is easily proved by induction on k. Since k is bounded by $|V|$, the whole algorithm takes at most $O(|E| \cdot |V|)$ steps. Moreover, if by the $|V|$th sweep any improvement of a label takes place then the digraph must contain a negative circuit. Thus, we can use the Ford algorithm to detect the existance of a negative circuit, if all vertices are accessible from s. If the existence of a negative circuit is indicated, we can find one by starting a backward search from a vertex whose label is improved in the $|V|$th sweep. To this end we need for each vertex v a pointer to the vertex u which gave v its last label. This is easily achieved by a simple addition to step 2.

The Ford algorithm cannot be used on undirected graphs because any negative edge has the effect of a negative circuit; one can go on it back and forth decreasing labels indefinitely. All three algorithms can be used to find the distances of all vertices from a given vertex s; the BFS and Dijkstra's algorithm have to be changed: instead of stoping when t is labeled or taken out of T, stop when all accessible vertices are labeled or when T is empty. The bounds on the number of operations remain $O(|E|)$, $0(|V|^2)$ and $O(|E| \cdot |V|)$ respectively for the BFS, Dijkstra and the Ford algorithm. If this is repeated from all vertices, in order to find the distance from every vertex to all the others, the respective complexities are $O(|E| \cdot |V|)$, $O(|V|^3)$ and $O(|E| \cdot |V|^2)$. Next, let us describe an algorithm which solves the case with negative lengths in time $O(|V|^3)$.

Let $G(V, E)$ be a finite digraph with $V = \{1, 2, \ldots, n\}$. The length of edge e is denoted by $l(e)$, as before, and it may be negative. Define

$$\delta^0(i,j) = \begin{cases} l(e) & \text{if } i \xrightarrow{e} j, \\ \infty & \text{if there is no edge from } i \text{ to } j. \end{cases}$$

Let $\delta^k(i,j)$ be the length of a shortest path from i to j among all paths which may pass through vertices $1, 2, \ldots, k$ but do not pass through vertices $k+1, k+2, \ldots, n$.

Floyd's algorithm [10] is as follows:

1. $k \leftarrow 1$

2. For every $1 \leq i, j \leq n$ compute

$$\delta^k(i,j) \leftarrow \text{Min}\{\delta^{k-1}(i,j), \delta^{k-1}(i,k) + \delta^{k-1}(k,j)\}.$$

3. If $k = n$, stop. If not, increment k and go to step 2.

The algorithm clearly yields the right answer; namely, $\delta^n(i,j)$ is the distance from i to j. The answer is only meaningful if there are no negative circuits in G. The existence of negative circuits is easily detected by $\delta^k(i,i) < 0$. Each application of step 2 requires n^2 operations, and step 2 is repeated n times. Thus, the algorithm is of complexity $O(n^3)$.

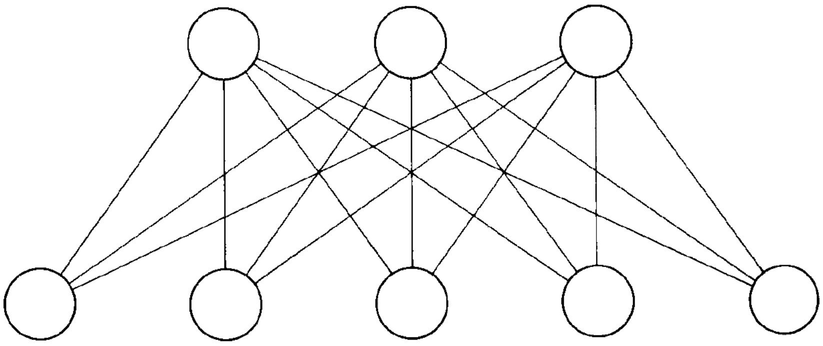
For the case of finite graphs with non-negative edge lengths, both the repeated Dijkstra algorithm and Floyd's take $O(|V|^3)$. Additional information on the shortest path problem can be found in Problems 1.9 and 1.10 and references [11] and [12].

PROBLEMS

1.1 Prove that if a connected (undirected) finite graph has exactly $2k$ vertices of odd degree then the set of edges can be partitioned into k paths such that every edge is used exactly once. Is the condition of connectivity necessary or can it be replaced by a weaker condition?

A *Hamilton path* (circuit) is a simple path (circuit) on which every vertex of the graph appears exactly once.

1.2 Prove that the following graph has no Hamilton path or circuit.

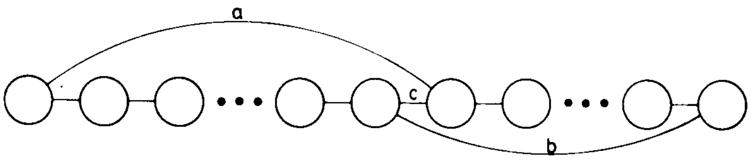


1.3 Prove that in every completely connected directed graph (a graph in which every two vertices are connected by exactly one directed edge in one of the two possible directions) there is always a directed Hamilton path. (Hint: Prove by induction on the number of vertices.)

1.4 Prove that a directed Hamilton circuit of $G_{\sigma,n}$ corresponds to a directed Euler circuit of $G_{\sigma,n-1}$. Is it true that $G_{\sigma,n}$ always has a direct Hamilton circuit?

1.5 In the following assume that $G(V,E)$ is a finite undirected graph, with no parallel edges and no self-loops.

(a) Describe an algorithm which attempts to find a Hamilton circuit in G by working with a partial simple path. If the path cannot be extended in either direction then try to close it into a simple circuit by the edge between its endpoints, if it exists, or by a switch, as suggested by the diagram, where edges a and b are added and c is deleted. Once a circuit is formed, look for an edge from one of its vertices to a new vertex, and open the circuit to a now longer path, etc.



(b) Prove that if for every two vertices u and v, $d(u)+d(v) \geq n$, where $n = |V|$, then the algorithm will never fail to produce a Hamilton circuit.

(c) Deduce Dirac's theorem [13]: If for every vertex v, $d(v) \geq n/2$, then G has a Hamilton circuit.

1.6 Describe an algorithm for finding the number of shortest paths from s to t after the BFS algorithm has been performed.

1.7 Repeat the above, after the Dijkstra algorithm has been performed. Assume $l(e) > 0$ for every edge e. Why is this asumption necessary?

1.8 Prove that a connected undirected graph G is orientable (by giving each edge some direction) into a strongly connected digraph if and only if each edge of G is in some simple circuit in G. (A path $u \overset{e}{—} v \overset{e}{—} u$ is not considered a simple circuit.)

1.9 The *transitive closure* of a digraph $G(V, E)$ is a digraph $G'(V, E)$ such that there is an edge $u \to v$ in G' if and only if there is a (non-empty) directed path from u to v in G. For the BFS, Dijkstra and Floyd's algorithms, explain how they can be used to construct G' for a given G, and compare the complexities of the resulting algorithms.

1.10 The following algorithm, due to Dantzig [14], finds all distances in a finite digraph, like Floyd's algorithm. Let $\delta^k(i,j)$ be the distance from i to j, where $1 \le i, j \le k$ and no vertices higher than k are used on the path. Let $\delta^k(i,i) = 0$ for all i and k. Also, let $l(i,j)$ be $l(e)$ if $i \xrightarrow{e} j$, and ∞ if no such edge exists.

(1) $\delta^1(1,1) \leftarrow \text{Min}\{0, l(1,1)\}$.

(2) $k \leftarrow 2$

(3) For $1 \le i < k$ do

$$\delta^k(i,k) \leftarrow \text{Min}_{1 \le j < k}\{\delta^{k-1}(i,j) + l(j,k)\}$$
$$\delta^k(k,i) \leftarrow \text{Min}_{1 \le j < k}\{l(k,j) + \delta^{k-1}(j,i)\}$$

(4) For $1 \le i, j < k$ do

$$\delta^k(i,j) \leftarrow \text{Min}\{\delta^{k-1}(i,j), \delta^k(i,k) + \delta^k(k,j)\}$$

(5) If $k = n$, stop, If not, increment k and go to step 3.

Show that Dantzig's algorithm is valid. How are negative circuits detected? What is the time complexity of this algorithm?

REFERENCES

[1] Knuth, D. E., *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison-Wesley, 1968.

[2] Aho, A. V., Hopcroft, J. E., and Ullman, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[3] Golomb, S. W., *Shift Register Sequences*, Holden-Day, 1967.

[4] Berge, C., *The Theory of Graphs and Its Applications*, Wiley, 1962, Chapter 17.

[5] Hall, M., Jr., *Combinatorial Theory*, Blaisdell, 1967, Chapter 9.

[6] Moore, E. F., "The Shortest Path Through a Maze", *Proc. Iternat. Symp. Switching Th.*, 1957, Part II, Harvard Univ. Press, 1959, pp. 285-292.

[7] Dijkstra, E. W., "A Note on Two Problems in Connection with Graphs", *Numerische Math.*, Vol. 1, 1959, pp. 269-271.

[8] Ford, L. R., Jr., "Network Flow Theory", The Rand Corp., P-923, August, 1956.

[9] Ford, L. R., Jr. and Fulkerson, D. R., *Flows in Networks*, Princeton Univ. Press, 1962, Chap. III, Sec. 5.

[10] Floyd, R. W., "Algorithm 97: Shortest Path", *Comm. ACM*, Vol. 5, 1962, p. 345.

[11] Dreyfus, S. E., "An Appraisal of Some Shortest-Path Algorithms", *Operations Research*, Vol. 17, 1969, pp. 395-412.

[12] Lawler, E. L., *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, 1976, Chapter 3.

[13] Dirac, G. A., "Connectivity Theorems for Graphs", *Quart. L Math.*, Ser. (2), Vol. 3, 1952, pp. 171-174.

[14] Dantzig, G. B., "All Shortest Routes in a Graph", Oper. Res. House, Stanford Univ. Tech. Rep. 66-3, November 1966.

Chapter 2

TREES

2.1 TREE DEFINITIONS

Let $G(V, E)$ be an (undirected), finite or infinite graph. We say that G is *circuit-free* if there are no simple circuits in G. G is called a *tree* if it is connected and circuit-free.

Theorem 2.1 *The following four conditions are equivalent:*

(a) G is a tree.

(b) G is circuit-free, but if any new edge is added to G, a circuit is formed.

(c) G contains no self-loops and for every two vertices there is a unique simple path connecting them.

(d) G is connected, but if any edge is deleted from G, the connectivity of G is interrupted.

Proof: We shall prove that conditions (a) $\Rightarrow$ (b) $\Rightarrow$ (c) $\Rightarrow$ (d) $\Rightarrow$ (a).

(a) $\Rightarrow$ (b): We assume that G is connected and circuit-free. Let e be a new edge, that is $e \notin E$; the two endpoints of e, a and b, are elements of V. If $a = b$, then e forms a self-loop and therefore a circuit exists. If $a \neq b$, there is a path in G (without e) between a and b; if we add e, this path with e forms a circuit.

(b) $\Rightarrow$ (c): We assume that G is circuit-free and that no edge can be added to G without creating a circuit. Let a and b be any two vertices of G. If there is no path between them, then we can add an edge between a and b without creating a circuit. Thus, G must be connected. Moreover, if there are two simple paths, P and P', between a and b, then there is a circuit in G. To see this, assume that $P = e_1, e_2, \ldots, e_l$ and $P' = e'_1, e'_2, \ldots, e'_m$. Since both paths are simple, one cannot be the beginning of the other. Let i be the first index for which $e_i \neq e'_i$, and let v be the first vertex on $e_i, e_{i+1}, \ldots, e_l$ which is also on $e'_i, e'_{i+1}, \ldots, e'_m$. The two disjoint subpaths between the branching off vertex and v form a simple circuit in G.

(c) $\Rightarrow$ (d): We assume the existence of a unique simple path between every pair of vertices of G. This implies that G is connected. Assume now that we delete an edge e from G. Since G has no self-loops, e is not a self-loop. Let a and b be e's endpoints. If there is now (after the deletion of e) a path between a and b, then G has more than one simple path between a and b.

(d) $\Rightarrow$ (a): We assume that G is connected and that no edge can be deleted without interrupting the connectivity. If G contains a simple circuit, any edge on this circuit can be deleted without interrupting the connectivity. Thus, G is circuit-free.

Q.E.D.

There are two more common ways to define a finite tree. These are given in the following theorem.

Theorem 2.2 *Let $G(V,E)$ be a finite graph and $n = |V|$. The following three conditions are equivalent:*

(a) G is a tree.

(b) G is circuit-free and has $n-1$ edges.

(c) G is connected and has $n-1$ edges.

Proof: For $n = 1$ the theorem is trivial. Assume $n \geq 2$. We shall prove that conditions (a) $\Rightarrow$ (b) $\Rightarrow$ (c) $\Rightarrow$ (a).

(a) $\Rightarrow$ (b): Let us prove, by induction on n, that if G is a tree, then its number of edges is $n-1$. This statement is clearly true for $n = 1$. Assume that it is true for all $n < m$, and let G be a tree with m vertices. Let us delete from G any edge e. By condition (d) of Theorem 2.1, G is not connected any more, and clearly is broken into two connected components each of which is circuit-free and therefore is a tree. By the inductive hypothesis, each component has one edge less than the number of vertices. Thus, both have $m-2$ edges. Add back e, and the number of edges is $m-1$.

(b) $\Rightarrow$ (c): We assume that G is circuit-free and has $n-1$ edges. Let us first show that G has at least two vertices of degree 1. Choose any edge e. An edge must exist since the number of edges is $n-1$ and $n \geq 2$. Extend the edge into a path by adding new edges to its ends if such exist. A new edge attached at the path's end introduces a new vertex to the path or a circuit is closed. Thus, our path remains simple. Since the graph is finite, this extension must terminate on both sides of e, yielding two vertices of degree 1.

Now, the proof that G is connected proceeds by induction on the number of vertices, n. The statement is obviously true for $n = 2$. Assume that it is true for $n = m-1$, and let G be a circuit-free graph with m vertices and $m-1$ edges. Eliminate from G a vertex v, of degree 1, and its incident edge. The resulting graph is still circuit-free and has $m-1$ vertices and $m-2$ edges; thus, by the inductive hypothesis it is connected. Therefore, G is connected too.

(c) $\Rightarrow$ (a): Assume that G is connected and has $n-1$ edges. If G contains circuits, we can eliminate edges (without eliminating vertices) and maintain the connectivity. When this process terminates, the resulting graph is a tree, and, by (a) $\Rightarrow$ (b), has $n-1$ edges. Thus, no edge can be eliminated and G is circuit-free.

Q.E.D.

Let us call a vertex whose degree is 1, a *leaf*. A corollary of Theorem 2.2 and the statement proved in the (b) $\Rightarrow$ (c) part of its proof is the following corollary:

Corollary 2.1 *A finite tree, with more than one vertex, has at least two leaves.*

2.2 MINIMUM SPANNING TREE

A graph $G'(V',E')$ is called a *subgraph* of a graph $G(V,E)$, if $V' \subseteq V$ and $E' \subseteq E$. Clearly, an arbitrary choice of $V' \subseteq V$ and $E' \subseteq E$ may not yield a subgraph, simply because it may not be a graph; that is, some of the endpoints of edges in E' may not be in V'.

Assume $G(V, E)$ is a finite, connected (undirected) graph and each edge $e \in E$ has a known length $l(e) > 0$. Assume we want to find a connected subgraph $G'(V, E')$ whose length, $\sum_{e \in E} l(e)$, is minimum; or, in other words, we want to remove from G a subset of edges whose total length is maximum, and which leaves it still connected. It is clear that such a subgraph is a tree. For G' is assumed to be connected, and since its length is minimum, none of its edges can be removed without destroying its connectivity. By Theorem 2.1 (see part (d)) G' is a tree. A subgraph of G, which contains all of its vertices and is a tree is called a *spanning tree* of G. Thus, our problem is that of finding a minimum-length spanning tree of G.

There are many known algorithms for the minimum spanning tree problem, but they all hinge on the following theorem:

Theorem 2.3 *Let $U \subset V$ and e be of minimum length among the edges with one endpoint in U and the other endpoint in $V - U$. There exists a minimum spanning tree T such that e is in T.*

Proof: Let T_0 be a minimum spanning tree. If e is not in T_0, add e to T_0. By Theorem 2.1 (part (b)) a circuit is formed. This circuit contains e and at least one more edge $u \overset{e'}{—} v$, where $u \in U$ and $v \in V - U$. Now, $l(e) \leq l(e')$, since e is of minimum length among the edges connecting U with $V - U$. We can delete e' from $T_0 + e$. The resulting subgraph is still connected and by Theorem 2.2 is a tree, since it has the right number of edges. Also, the length of this new tree, which contains e, is less than or equal to that of T_0. Thus, it is optimal.

Q.E.D.

Let $G(V, E)$ be the given graph, where $V = \{1, 2, \ldots, n\}$. We assume that there are no parallel edges, for all but the shortest can be eliminated. Thus, let $l(i, j)$ be $l(e)$ if there is an edge $i \overset{e}{—} j$, and infinity otherwise. The following algorithm is due to Prim [1]:

(1) $t \leftarrow 1, T \leftarrow \emptyset$ and $U \leftarrow \{1\}$.

(2) Let $l(t, u) = \text{Min}_{v \in V-U}\{l(t, v)\}$.

(3) $T \leftarrow T \cup \{e\}$ where e is the edge which corresponds to the length $l(t, u)$.

(4) $U \leftarrow U \cup \{u\}$.

(5) If $U = V$, stop.

(6) For every $v \in V - U$, $l(t, v) \leftarrow \text{Min}\{l(t, v), l(u, v)\}$.

(7) Go to Step (2).

(Clearly $t = 1$ throughout. We used t instead of 1 to emphasize that $l(t, v)$ may not be the original $l(1, v)$ after Step (6) has been applied.)

The algorithm follows directly the hint supplied by Theorem 2.3. The "vertex" t represents the subset U of vertices, and for $v \in V - U$ $l(t, v)$ is the length of a shortest edge from a vertex in U to v. This is affected by Step (6). Thus, in Step (2), a shortest edge connecting U and $V - U$ is chosen.

Although each choice of an edge is "plausible", it is still necessary to prove that in the end, T is a minimum spanning tree.

Let a subgraph $G'(V', E')$ be called an *induced subgraph* if E' contains all the edges of E whose endpoints are in V'; in this case we say that G ' is induced by V'.

First observe, that each time we reach Step (5), T is the edge set of a spanning tree of the subgraph induced by U. This easily proved by induction on the number of times we reach Step (5). We start with $U = \{1\}$ and $T = \emptyset$ which is clearly a spanning tree of the subgraph induced by $\{1\}$. After the first application of Steps (2), (3) and (4), we have two vertices in U and an edge in T which connects them. Each time we apply Steps (2), (3) and (4) we add an edge from a vertex of the previous U to a new vertex. Thus the new T is connected too. Also, the number of edges is one less than the number of vertices. Thus, by Theorem 2.2 (part (c)), T is a spanning tree.

Now, let us proceed by induction to prove that if the old T is a subgraph of some minimum spanning tree of G then so is the new one. The proof is similar to that of Theorem 2.3. Let T_0 be a minimum spanning tree of G which contains T as a subgraph, and assume e is the next edge chosen in Step (2) to connect between a vertex of U and $V - U$. If e is not in T_0, add it to T_0 to form $T_0 + e$. It contains a circuit in which there is one more edge, e', connecting a vertex of U with a vertex of $V - U$. By Step (2), $l(e) \leq l(e')$, and if we delete e' from $T_0 + e$, we get an minimum spanning tree which contains both T, as a subgraph, and e, proving that the new T is a subgraph of some minimum spanning tree. Thus, in the end T is a minimum spanning tree of G.

The complexity of the algorithm is $O(|V|^2)$; Step (2) requires at most $|V| - 1$ comparisons and is repeated $|V| - 1$ times, yielding $O(|V|^2)$. Step (6) requires one comparison for each edge; thus, the total time spent on it is $O(|E|)$.

It is possible to improve the algorithm and the interested reader is advised to read the Cheriton and Tarjan paper [2]. We do not pursue this here because an understanding of advanced data structures is necessary. The faster algorithms do not use any graph theory beyond the level of this section.

The analogous problem for digraphs, namely, that of finding a subset of the edges E' whose total length is minimum among those for which (V, E') is a strongly connected subgraph, is much harder. In fact, even the case where $l(e) = 1$ for all edges is hard. This will be discussed in Chapter 10.

2.3 CAYLEY'S THEOREM

In a later section we shall consider the question of the number of spanning trees in a given graph. Here we consider the more restricted, and yet interesting problem, of the number of trees one can define on a given set of vertices, $V = \{1, 2, \ldots, n\}$.

For $n = 3$, there are 3 possible trees, as shown in Figure 2.1. Clearly, for $n = 2$ there is only one tree. The reader can verify, by exhausting all the cases, that for $n = 4$ the number of trees is 16. The following theorem is due to Cayley [3]:

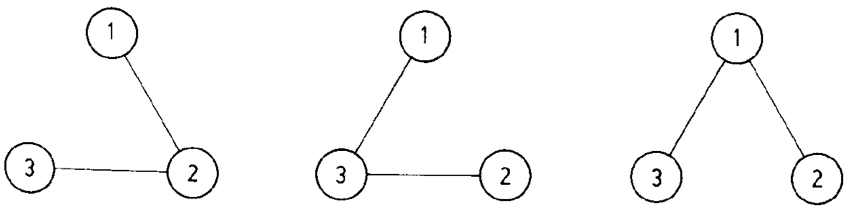


Figure 2.1

Theorem 2.4 *The number of spanning trees for n distinct vertices is n^{n-2}.*

The proof to be presented is due to Prüfer [4]. (For a survey of various proofs see Moon [5].)

Proof: Assume $V = \{1, 2, \ldots, n\}$. Let us display a one-to-one correspondence between the set of the spanning trees and the n^{n-2} words of length $n-2$ over the alphabet $\{1, 2, \ldots, n\}$. The algorithm for finding the word which corresponds to a given tree is as follows:

(1) $i \leftarrow 1$.

(2) Among all leaves of the current tree let j be the least one (i.e., its name is the least integer). Eliminate j and its incident edge e from the tree. The ith letter of the word is the other endpoint of e.

(3) If $i = n-2$, stop.

(4) Increment i and go to step 2.

For example, assume that $n = 6$ and the tree is as shown in Figure 2.2. On the first turn of Step (2), $j = 2$ and the other endpoint of its incident edge is 4. Thus, 4 is the first letter of the word. The new tree is as shown in Figure 2.3. On the second turn, $j = 3$ and the second letter is 1. On the third, $j = 1$ and the third letter is 6. On the fourth, $j = 5$ and the fourth letter is 4. Now $i = 4$ and the algorithm halts. The resulting word is 4164 (and the current tree consists of one edge connecting 4 and 6).

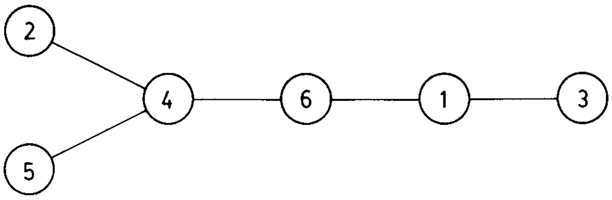


Figure 2.2

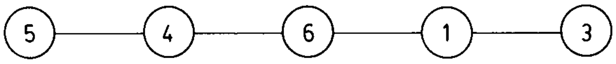


Figure 2.3

By Corollary 2.1, Step (2) can always be performed, and therefore for every tree a word of length $n-2$ is produced. It remains to be shown that no word is produced by two different trees and that every word is generated from some tree. We shall achieve both ends by showing that the mapping has an inverse; i.e., for every word there is a unique tree which produces it.

Let $w = a_1 a_2 \cdots a_{n-2}$ be a word over V. If T is a tree for which the algorithm produces w then the degree of vertex k, $d(k)$, in T, is equal to the number of times k appears in w, plus 1. This follows from the observation that when each, but the last, of the edges incident to k is deleted, k is written as a letter of w; the last edge may never be deleted, if k is one of the two vertices remaining in the tree, or if it is deleted, k is now the removed leaf, and the adjacent vertex, not k, is the written letter. Thus, if w is produced by the algorithm, for some tree, then the degrees of the vertices in the tree must be as stated.

For example, if $w = 4164$ then $d(1) = 2$, $d(2) = 1$, $d(3) = 1$, $d(4) = 3$, $d(5) = 1$ and $d(6) = 2$ in a tree which produced w.

Given this data, apply the following algorithm:

(1) $i \leftarrow 1$.

(2) Let j be the least vertex for which $d(j) = 1$. Construct an edge $j — a_i$, $d(j) \leftarrow 0$ and $d(a_i) \leftarrow d(a_i) - 1$.

(3) If $i = n - 2$, construct an edge between the two vertices whose degree is 1 and stop.

(4) Increment i and go to step 2.

It is easy to see that this algorithm picks the same vertex j as the original algorithm, and constructs a tree (the proof is by induction). Also, each step of the reconstruction is forced, therefore it is the only tree which yields w, and for every word this algorithm produces a tree.

In our example, for $i = 1$, $j = 2$ and since $a_1 = 4$ we connect $2 — 4$, as shown in Figure 2.4. Now, $d(1) = 2$, $d(2) = 0$, $d(3) = 1$, $d(4) = 2$, $d(5) = 1$ and $d(6) = 2$. For $i = 2$, $j = 3$ and since $a_2 = 1$ we connect $3 — 1$, as shown in Figure 2.5. Now $d(1) = 1$, $d(2) = 0$, $d(3) = 0$, $d(4) = 2$, $d(5) = 1$ and $d(6) = 2$. For $i = 3$, $j = 1$ and since $a_3 = 6$ we connect $1 — 6$ as shown in Figure 2.6. Now, $d(1) = d(2) = d(3) = 0$, $d(4) = 2$ and $d(5) = d(6) = 1$. Finally, $i = 4$, $j = 5$ and since $a_4 = 4$ we connect $5 — 4$, as shown in Figure 2.7. Now, $d(1) = d(2) = d(3) = d(5) = 0$ and $d(4) = d(6) = 1$. By step 3, we connect $4 — 6$ and stop. The resulting graph is as in Figure 2.2.

Q.E.D.

Figure 2.4

Figure 2.5

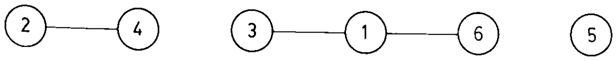


Figure 2.6

Figure 2.7

A similar problem, stated and solved by Lempel and Welch [6], is that of finding the number of ways m labeled (distinct) edges can be joined by unlabeled endpoints to form a tree. Their proof is along the lines of Prüfer's proof of Cayley's theorem and is therefore constructive, in the sense that one can use the inverse transformation to generate all the trees after the words are generated. However, a much simpler proof was pointed out to me by A. Pnueli and is the subject of Problem 2.5.

2.4 DIRECTED TREE DEFINITIONS

A digraph $G(V, E)$ is said to have a *root* r if $r \in V$ and every vertex $v \in V$ is *reachable* from r; i.e., there is a directed path which starts in r and ends in v.

A digraph (finite or infinite) is called a *directed tree* if it has a root and its underlying undirected graph is a tree.

Theorem 2.5 *Assume G is a digraph. The following five conditions are equivalent:*

(a) G is a directed tree.

(b) G has a root from which there is a unique directed path to every vertex.

(c) G has a root r for which $d_{in}(r) = 0$ and for every other vertex v, $d_{in}(v) = 1$.

(d) G has a root and the deletion of any edge (but no vertices) interrupts this condition.

(e) The underlying undirected graph of G is connected and G has one vertex r for which $d_{in}(r) = 0$, while for every other vertex v, $d_{in}(v) = 1$.

Proof: We prove that (a) $\Rightarrow$ (b) $\Rightarrow$ (c) $\Rightarrow$ (d) $\Rightarrow$ (e) $\Rightarrow$ (a).

(a) $\Rightarrow$ (b): We assume that G has a root, say r, and its underlying undirected graph G' is a tree. Thus, by Theorem 2.1, part (c), there is a unique simple path from r to every vertex in G'; also, G' is circuit-free. Thus, a directed path from r to a vertex v, in G, must be simple and unique.

(b) $\Rightarrow$ (c): Here we assume that G has a root, say r, and a unique directed path from it to every vertex v. First, let us show that $d_{in}(r) = 0$. Assume there is an edge $u \xrightarrow{e} r$. There is a directed path from r to u, and it can be continued, via e, back r. Thus, in addition to the empty path from r to itself (containing no edges), there is one more, in contradiction of the assumption of the path uniqueness. Now, we have to show that if $v \neq r$ then $d_{in}(v) = 1$. Clearly, $d_{in}(v) > 0$ for it must be reachable from r. If $d_{in}(v) > 1$, then there are at least two edges, say $v_1 \xrightarrow{e_1} v$ and $v_2 \xrightarrow{e_2} v$. Since there is a directed path P_1 from r to v_1, and a directed path P_2 from r to v_2 by adding e_1 to P_1 and e_2 to P_2 we get two different paths from r to v. (This proof is valid even if $v_1 = v_2$.)

(c) $\Rightarrow$ (d): This proof is trivial, for the deletion on any edge $u \xrightarrow{e} v$ will make v unreachable from r.

(d) $\Rightarrow$ (e): We assume that G has a root, say r, and the deletion of any edge interrupts this condition. First $d_{in}(r) = 0$, for any edge entering r could be deleted without interrupting the condition that r is a root. For every other vertex v, $d_{in}(v) > 0$, for it is reachable from r. If $d_{in}(v) > 1$, let $v_1 \xrightarrow{e_1} v$ and $v_2 \xrightarrow{e_2} v$ be two edges entering v. Let P be a simple directed path from r to v. It cannot use both e_1 and e_2. The one which is not used in P can be deleted without interrupting the fact that r is a root. Thus, $d_{in}(v) = 1$.

(e) $\Rightarrow$ (a): We assume that the underlying undirected graph of G, G', is connected, $d_{in}(r) = 0$ and for $v \neq r$, $d_{in}(v) = 1$. First let us prove that r is a root. Let P' be a simple path connecting r and v in G'. This must correspond to a directed path P from r to v in G, for if any of the edges points in the wrong direction it would either imply that $d_{in}(r) > 0$ or that for some u, $d_{in}(u) > 1$. Finally, G' must be circuit-free, for a simple circuit in G' must correspond to a simple directed circuit in G (again using $d_{in}(r) = 0$ and $d_{in}(v) = 1$ for $v \neq r$), and at least one of its vertices, u, must have $d_{in}(u) > 1$, since the vertices of the circuit are reachable from r.

Q.E.D.

In case of finite digraphs one more useful definition of a directed tree is possible:

Theorem 2.6 *A finite digraph G is a directed tree if and only if its underlying undirected graph, G', is circuit-free, one of its vertices, r, satisfies $d_{in}(r) = 0$, and for all other vertices v, $d_{in}(v) = 1$.*

Proof: The "only if" part follows directly from the definition of a directed tree and Theorem 2.5, part (c).

To prove the "if" part we first observe that the number of edges is $n-1$. Thus, by Theorem 2.2, (b) $\Rightarrow$ (c), G' is connected. Thus, by Theorem 2.5, (e) $\Rightarrow$ (a), G is a directed tree.

Q.E.D.

Let us say that a digraph is *arbitrated* (Berge [7] calls it quasi strongly connected) if for every two vertices v_1 and v_2 there is a vertex v called an *arbiter* of v_1 and v_2, such that there are directed paths from v to v_1 and from v to v_2. There are infinite digraphs which are arbitrated but do not have a root. For example, see the digraph of Figure 2.8. However, for finite digraphs the following theorem holds:

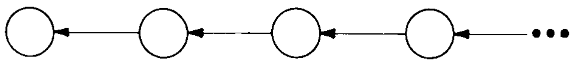


Figure 2.8

Theorem 2.7 *If a finite digraph is arbitrated then it has a root.*

Proof: Let $G(V, E)$ be a finite arbitrated digraph, where $V = \{1, 2, \ldots, n\}$. Let us prove, by induction, that every set $\{1, 2, \ldots, m\}$, where $m \leq n$, has an arbiter; i.e., a vertex a_m such that every $1 \leq i \leq m$ is reachable from a_m. By definition, a_2 exists. Assume a_{m-1} exists. Let a_m, be the arbiter of a_{m-1} and m. Since a_{m-1} is reachable from a_m, and every $1 \leq i \leq m-1$ is reachable from a_{m-1}, every $1 \leq i \leq m-1$ is also reachable from a_m.

Q.E.D.

Thus, for finite digraphs, the condition that it has a root, as in Theorem 2.5 part a, b, c and d, can be replaced by it being arbitrated.

2.5 THE INFINITY LEMMA

The following is known as König's Infinity Lemma [8]:

Theorem 2.8 *If G is an infinite digraph, with a root r and finite out-degrees for all its vertices, then G has an infinite directed path, starting in r.*

Before we present the proof let us point out the necessity of the finiteness of the out-degrees of the vertices. For if we allow a single vertex to be of infinite out-degree, the conclusion does not follow. Consider the digraph of Figure 2.9. The root is connected to vertices v_1^1, v_1^2, v_1^3, ..., where v_1^k is the second vertex on a directed path of length k. It is clear that the tree is infinite, and yet it has no infinite path. Furthermore, the replacement of the condition of finite degrees by the condition that for every k the tree has a path of length k, does not work either, as the same example shows.

Proof: First let us restrict our attention to a directed tree T which is an infinite subgraph of G. T's root is r. All vertices of distance 1 away from r in G are also of distance 1 away from r in T. In general, if a vertex v is of distance l away from r in G it is also of distance l away from r in T; all the edges entering v in G are now dropped, except one which connects a vertex of distance $l-1$ to v. It is sufficient to show that in T there is an infinite directed path from r. Clearly, since T is a subgraph of G, all its vertices are of finite outdegrees too.

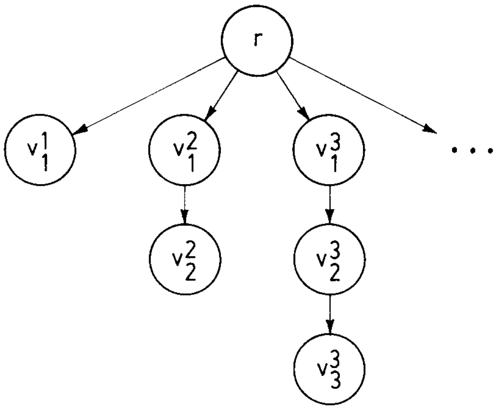


Figure 2.9

In T, r has infinitely many descendants (vertices reachable from r). Since r is of finite out-degree, at least one of its sons (the vertices reachable via one edge), say r_1, must have infinitely many descendants. One of r_1's sons has infinitely many descendants, too, and so we continue to construct an infinite directed path r, r_1, r_2, ...

Q.E.D.

In spite of the simplicity of the theorem, it is useful. For example, if we conduct a search on a directed tree of finite degrees (where a bound on the degree may not be known) for which it is known that it has no infinite directed paths, then the theorem ensures us that the tree is finite and our search will terminate.

An interesting application of Theorem 2.8 was made by Wang [9]. Consider the problem of tiling the plane with square tiles, all of the same size (Wang calls the tiles "dominoes"). There is a finite number of tile families. The sides of the tiles are labeled by letters of an alphabet, and all the tiles of one family have the same labels, thus are indistinguishable. Tiles may not be rotated or reflected, and the labels are specified for their north side, south side, and so on. There is an infinite supply of tiles of each family. The tiles may be put one next to another, the sides converging only if these two sides have the same labels. For example, if the tile families are as shown in Figure 2.10, then we can construct the "torus" shown in Figure 2.11. Now, by repeating this torus infinitely many times horizontally and vertically, we can tile the whole plane.

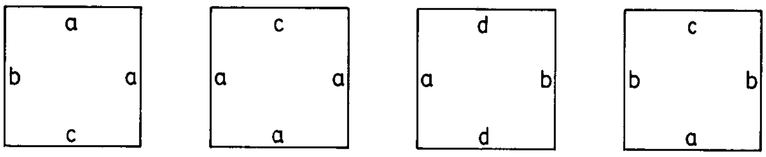


Figure 2.10

Wang proved that if it is possible to tile the upper right quadrant of the plane with a given finite set of tile families, then it is possible to tile the whole plane. The reader should realize that a southwest shift of the upper-right tiled quadrant cannot be used to cover the whole plane. In fact, if the number of tile families is not restricted to be finite, one can find sets of families for which the upper-right quadrant is tileable, while the whole plane is not.

Consider the following directed tree T: The root r is connected to vertices, each representing one of the tile families, i.e., a square 1×1 tiled with the tile of that family. For every k, each

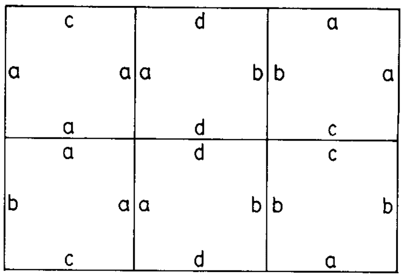


Figure 2.11

one of the legitimate ways of tiling a $(2k+1) \times (2k+1)$ square is represented by a vertex in T; its father is the vertex which represents the tiling of a $(2k-1) \times (2k-1)$ square, identical to the center part of the square represented by the son.

Now, if the upper-right quadrant is tilable, then T has infinitely many vertices. Since the number of families is finite, the out-degree of each vertex is finite (although, may not be bounded). By Theorem 2.8, there is an infinite directed path in T. Such a path describes a way to tile the whole plane.

Chapter 3

DEPTH-FIRST SEARCH

3.1 DFS OF UNDIRECTED GRAPHS

The Depths-First Search technique is a method of scanning a finite undirected graph. Since the publication of the papers of Hopcroft and Tarjan [1, 2] it is widely recognized as a powerful technique for solving various graph problems. However, the algorithm is not new; it was already known in the 19th century as a technique for threading mazes. For example, see Lucas' [3] report of Trémaux's work. Another algorithm, which was suggested later by Tarry [4], is just as good for threading mazes, and in fact DFS is a special case of it; but it is the additional structure of DFS which makes it so useful.

Assume we are given a finite connected graph $G(V, E)$. Starting in one of the vertices we want to "walk" along the edges, from vertex to vertex, visit all the vertices and halt. We seek an algorithm that will guarantee that we scan the whole graph, and recognize when we are done, without wondering too long in the "maze". We allow no preplanning, as by studying the road-map before we start our excursion; we must make our decisions, one at a time, since we discover the structure of the graph as we scan it. Clearly, we need to leave some "markers" as we go along, to recognize the fact that we have returned to a place visited before. Let us mark the *passages*, namely the connections of the edges to vertices. If the graph is presented by incidence lists then we can think of each of the two appearances of an edge in the incidence lists of its two endpoints as its two passages. It suffices to use two types of markers: F for the first passage used to enter the vertex, and E for any other passage when used to leave the vertex. No marker is ever erased or changed. As we shall prove later the following algorithm will terminate in the original starting vertex s, after scanning each edge once in each direction.

Trémaux's Algorithm:

(1) $v \leftarrow s$.

(2) If there are no unmarked passages in v, go to (4).

(3) Choose an unmarked passage, mark it E and traverse the edge to its other endpoint u. If u has any marked passages (i.e. it is not a new vertex) mark the passage, through which u has just been entered, by E, traverse the edge back to v, and go to Step (2). If u has no marked passages (i.e. it is a new vertex), mark the passage through which u has been entered by F, $v \leftarrow u$ and go to Step (2).

(4) If there is no passage marked F, halt. (We are back in s and the scanning of the graph is complete.)

(5) Use the passage marked F, traverse the edge to its other endpoint u, $v \leftarrow u$ and go to Step (2).

Let us demonstrate the algorithm on the graph shown in Figure 3.1. The initial value of v, the place "where we are" or the center of activity, is s. All passages are unlabelled. We choose one, mark it E and traverse the edge. Its other endpoint is a ($u = a$). None of its passages are marked, therefore we mark the passage through which a has been entered by F, the new center of activity is a ($v = a$), and we return to Step (2). Since a has two unmarked passages, assume we choose the one leading to b. The passage is marked E and the one at b is marked F since b is new, etc. The complete excursion is shown in Figure 3.1 by the dashed line.

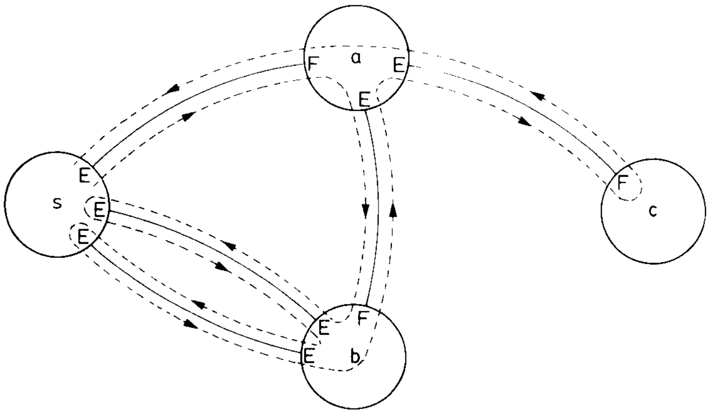


Figure 3.1

Lemma 3.1 *Trémaux's algorithm never allows an edge to be traversed twice in the same direction.*

Proof: If a passage is used as an exit (entering an edge), then either it is being marked E in the process, and thus the edge is never traversed again in this direction, or the passage is already marked F. It remains to be shown that no passage marked F is ever reused for entering the edge.

Let $u \stackrel{e}{-} v$ be the first edge to be traversed twice in the same direction, from u to v. The passage of e, at u, must be labeled F. Since s has no passages marked F, $u \neq s$. Vertex u has been left $d(u) + 1$ times; once through each of the passages marked E and twice through e. Thus, u must have been entered $d(u) + 1$ times and some edge $w \stackrel{e'}{-} u$ has been used twice to enter u, before e is used for the second time. A contradiction.

Q.E.D.

An immediate corollary of Lemma 3.1 is that the process described by Trémaux's algorithm will always terminate. Clearly it can only terminate in s, since every other visited vertex has an F passage. Therefore, all we need to prove is that upon termination the whole graph has been scanned.

Lemma 3.2 *Upon termination of Trémaux's algorithm each edge of the graph has been traversed once in each direction.*

Proof: Let us state the proposition differently: For every vertex all its incident edges have been traversed in both directions.

First, consider the start vertex s. Since the algorithm has terminated, all its incident edges have been traversed from s outward. Thus, s has been left $d(s)$ times, and since we end up in s, it has also been entered $d(s)$ times. However, by Lemma 3.1 no edge is traversed more than once in the same direction. Therefore, all the edges incident to s have been traversed once in each direction.

Assume now that S is the set of vertices for which the statement, that each of their incident edges has been traversed once in each direction, holds. Assume $V \neq S$. By the connectivity of the graph there must be edges connecting vertices of S with $V - S$. All these edges have been traversed once in each direction. Let $v \overset{e}{—} u$ be the first edge to be traversed from $v \in S$ to $u \in V - S$. Clearly, u's passage, corresponding to e, is marked F. Since this passage has been entered, all other passages must have been marked E. Thus, each of u's incident edges has been traversed outward. The search has not started in u and has not ended in u. Therefore, u has been entered $d(u)$ times, and each of its incident edges has been traversed inward. A contradiction, since u belongs in S.

Q.E.D.

The Hopcroft and Tarjan version of DFS is essentially the same as Trémaux's, except that they number the vertices from 1 to $n(= |V|)$ in the order in which they are discovered. This is not necessary, as we have seen, for scanning the graph, but the numbering is useful in applying the algorithm for more advanced tasks. Let us denote the number of vertex v by $k(v)$. Also, instead of marking passages we shall now mark edges as "used" and instead of using the F mark to indicate the edge through which we leave the vertex for the last time, let us remember for each vertex v, other than s, the vertex $f(v)$ from which v has been discovered. $f(v)$ is called the *father* of v; this name will be justified later. DFS is now in the following form:

(1) Mark all the edges "unused". For every $v \in V$, $k(v) \leftarrow 0$. Also, let $i \leftarrow 0$ and $v \leftarrow s$.

(2) $i \leftarrow i + 1$, $k(v) \leftarrow i$.

(3) If v has no unused incident edges, go to Step (5).

(4) Choose an unused incident edge $v \overset{e}{—} u$. Mark e "used". If $k(u) \neq 0$, go to Step (3). Otherwise $(k(u) = 0)$, $f(u) \leftarrow v$, $v \leftarrow u$ and go to Step (2).

(5) If $k(v) = 1$, halt.

(6) $v \leftarrow f(v)$ and go to Step (3).

Since this algorithm is just a simple variation of the previous one, our proof that the whole (connected) graph will be scanned, each edge once in each direction, still applies. Here, in Step (4), if $k(u) \neq 0$ then u is not a new vertex and we "return" to v and continue from there. Also, moving our center of activity from v to $f(v)$ (Step (6)) corresponds to traversing the edge $v — f(v)$, in this direction. Thus, the whole algorithm is of time complexity $O(|E|)$, namely, linear in the size of the graph.

After applying the DFS to a finite and connected $G(V, E)$ let us consider the set of edges E' consisting of all the edges $f(v) — v$ through which new vertices have been discovered. Also direct each such edge from $f(v)$ to v.

Lemma 3.3 *The digraph (V, E') defined above is a directed tree with root s.*

Proof: Clearly $d_{in}(s) = 0$ and $d_{in}(v) = 1$ for every $v \neq s$. To prove that s is a root consider the sequence $v = v_0, v_1, v_2, \ldots$ where $v_{i+1} = f(v_i)$ for $i \geq 0$. Clearly, this defines a directed path leading into v in (V, E'). The path must be simple, since v_{i+1} was discovered before v_i. Thus, it can only terminate in s (which has no $f(s)$). Now by Theorem 2.5 (see part (c)), (V, E') is a directed tree with root s.

Q.E.D.

Clearly, if we ignore now the edge directions, (V, E') is a spanning tree of G. The following very useful lemma is due to Hopcroft and Tarjan [1, 2]:

Lemma 3.4 *If an edge $a \overset{e}{—} b$ is not a part of (V, E') then a is either an ancestor or a descendant of b in the directed tree (V, E').*

Proof: Without loss of generality, assume that $k(a) < k(b)$. In the DFS algorithm, the center of activity (v in the algorithm) moves only along the edges of the tree (V, E'). If b is not a descendant of a, and since a is discovered before b, the center of activity must first move from a to some ancestor of a before it moves up to b. However, we backtrack from a ($v \leftarrow f(a)$) only when all a's incident edges are used, which means that e is used and therefore b is already discovered—a contradiction.

Q.E.D.

Let us call all the edges of (V, E') *tree edges* and all the other edges *back edges*. The justification for this name is in Lemma 3.4; all the non-tree edges connect a vertex back to one of its ancestors.

Consider, as an example, the graph shown in Figure 3.2. Assume we start the DFS in c ($s = c$) and discover d, e, f, g, b, a in this order. The resulting vertex numbers, tree edges and back edges are shown in Figure 3.3, where the tree edges are shown by solid lines and are directed from low to high, and the back edges are shown by dashed lines and are directed from high to low. In both cases the direction of the edge indicates the direction in which the edge has been scanned first. For tree edges this is the defined direction, and for back edges we can prove it as follows: Assume $u \overset{e}{—} v$ is a back edge and u is an ancestor of v. The edge e could not have been scanned first from u, for if v has been undiscovered at that time then e would have been a tree edge, and if v has already been discovered (after u) then the center of activity could have been in u only if we have backtracked from v, and this means that e has already been scanned from v.

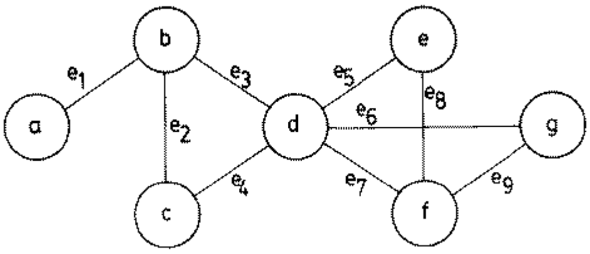


Figure 3.2

3.2 ALGORITHM FOR NONSEPARABLE COMPONENTS

A connected graph $G(V, E)$ is said to have a *separation vertex* v (sometimes also called an articulation point) if there exist vertices a and b, $a \neq v$ and $b \neq v$, such that all the paths

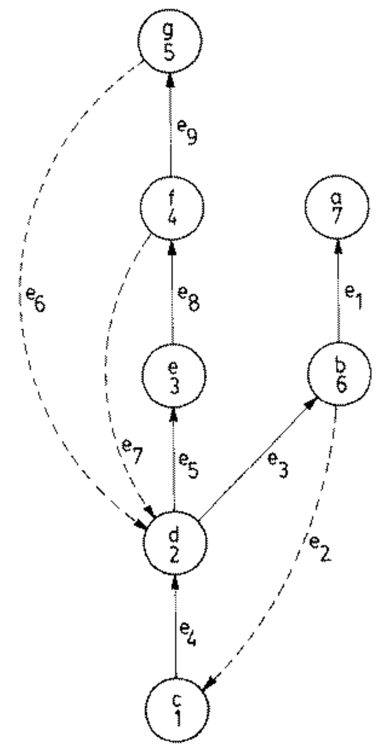


Figure 3.3

connecting a and b pass through v. In this case we also say that v *separates* a from b. A graph which has a seperation vertex is called *separable*, and one which has none is called *nonseparable*.

Let $V' \subseteq V$. The induced subgraph $G'(V', E')$ is called a *nonseparable component* if G' is nonseparable and if for every larger V'', $V' \subset V'' \subseteq V$, the induced subgraph $G''(V'', E'')$ is separable. For example, in the graph shown in Figure 3.2, the subsets $\{a, b\}$, $\{b, c, d\}$ and $\{d, e, f, g\}$ induce the nonseparable components of the graph.

If a graph $G(V, E)$ contains no separation vertex then clearly the whole G is a nonseparable component. However, if v is a separating vertex then $V - \{v\}$ can be partitioned into $V_1, V_2, \ldots, V_k$ such that $V_1 \cup V_2 \cup \ldots \cup V_k = V - \{v\}$ and if $i \neq j$ then $V_i \cap V_j = \emptyset$; two vertices a and b are in the same V_i if and only if there is a path connecting them which does not include v. Thus, no nonseparable component can contain vertices from more than one V_i. We can next consider each of the subgraphs induced by $V_i \cup \{v\}$ and continue to partition it into smaller parts if it is separable. Eventually, we end up with nonseparable parts. This shows that no two nonseparable components can share more than one vertex because each such vertex is a separating vertex. Also, every simple circuit must lie entirely in one nonseparable component.

Now, let us discuss how DFS can help to detect separating vertices.

Let the *lowpoint* of v, $L(v)$, be the least number, $k(u)$ of a vertex u which can be reached from v via a, possible empty, directed path consisting of tree edges followed by at most one back edge. Clearly $L(v) \leq k(v)$, for we can use the empty path from v to itself. Also, if a nonempty path is used then its last edge is a back edge, for a directed path of tree edges leads to vertices higher than v. For example, in the graph of Figure 3.2 with the DFS as shown in Figure 3.3 the lowpoints are as follows: $L(a) = 7$, $L(b) = L(c) = L(d) = 1$ and $L(e) = L(f) = L(g) = 2$.

Lemma 3.5 *Let G be a graph whose vertices have been numbered by DFS. If $u \to v$ is a tree edge, $k(u) > 1$ and $L(v) \geq k(u)$ then u is a separating vertex of G.*

Proof: Let S be the set of vertices on the path from the root r ($k(r) = 1$) to u, including r but not including u, and let T be the set of vertices on the subtree rooted at v, including v (that is, all the descendants of v, including v itself). By Lemma 3.4 there cannot be any edge connecting a vertex of T with any vertex of $V - (S \cup \{u\} \cup T)$. Also, if there is any edge connecting a vertex $t \in T$ with a vertex $s \in S$ then the edge $t \rightarrow s$ is a back edge and clearly $k(s) < k(u)$. Now, $L(v) \leq k(s)$, since one can take the tree edges from v to t followed by $t \rightarrow s$. Thus, $L(v) < k(u)$, contradicting the hypothesis. Thus, u is separating the S vertices from the T vertices and is therefore a separating vertex.

Q.E.D.

Lemma 3.6 *Let $G(V, E)$ be a graph whose vertices have been numbered by DFS. If u is a separating vertex and $k(u) > 1$ then there exists a tree edge $u \rightarrow v$ such that $L(v) \geq k(u)$.*

Proof: Since u is a separating vertex, there is a partition of $V - \{u\}$ into $V_1, V_2, \ldots, V_m$ such that $m \geq 2$ and if $i \neq j$ then all paths from a vertex of V_i to a vertex of V_j, pass through u. We assume that the search does not start in u. Let us assume it starts in r and $r \in V_1$. The center of activity of the DFS must pass through u. Let $u \rightarrow v$ be the first tree edge for which $v \notin V_1$. Assume $v \in V_2$. Since there are no edges connecting vertices of V_2 with $V - (V_2 \cup \{u\})$, $L(v) \geq k(u)$.

Q.E.D.

Lemma 3.7 *Let $G(V, E)$ be a graph whose vertices have been numbered by DFS, starting with r ($k(r) = 1$). The vertex r is a separating vertex if and only if there are at least two tree edges out of r.*

Proof: Assume that r is a separating vertex. Let $V_1, V_2, \ldots, V_m$ be a partition of $V - \{r\}$ such that $m \geq 2$ and if $i \neq j$ then all paths from a vertex of V_i to a vertex of V_j pass through r. Therefore, no path in the tree which starts with $r \rightarrow v$, $v \in V_i$ can lead to a vertex of V_j where $j \neq i$. Thus, there are at least two tree edges out of r.

Now, assume $r \rightarrow v_1$ and $r \rightarrow v_2$ are two tree edges out of r. Let T be the set of vertices in the subtree rooted at v_1. By Lemma 3.4, there are no edges connecting vertices of T with vertices of $V - (T \cup \{r\})$. Thus, r separates T from the rest of the graph, which is not empty since it includes at least the vertex v_2.

Q.E.D.

Let $C_1, C_2 \ldots C_m$ be the nonseparable components of the connected graph $G(V, E)$, and let $s_2, s_2, \ldots, s_p$ be its separating vertices. Let us define the *superstructure* of $G(V, E)$, $\tilde{G}(\tilde{V}, \tilde{E})$ as follows:

$$\tilde{V} = \{s_1, s_2, \ldots, s_p\} \cup (C_1, C_2, \ldots, C_m\},$$

$$\tilde{E} = \{s_i - C_j \mid s_i \text{ is a vertex of } C_j \text{ in } G\}.$$

By the observations we have made in the beginning of the section, $\tilde{G}(\tilde{V}, \tilde{E})$ is a tree. By Corollary 2.1, if $m > 1$ then there must be at least two leaf components, each containing only one separating vertex, since for every separating vertex s_i $d(s_i) \geq 2$ in $\tilde{G}$. By Lemma 3.2, the whole graph will be explored by the DFS.

Now, assume the search starts in a vertex r which is not a separating vertex. Even if it is in one of the leaf-components, eventually we will enter another leaf-component C, say via its separating vertex u and the edge $u \xrightarrow{e} v$. By Lemma 3.6, $L(v) \geq k(u)$, and if $L(v)$ is known when we backtrack from v to u, then by using Lemma 3.5, we can detect that u is a separating vertex. Also, as far as the component C is concerned, from the time C is entered until it is entirely explored, we can think of the algorithm as running on C alone with u as the starting vertex. Thus, by Lemma 3.7, there will be only one tree edge from u into C, and all the other vertices of C are descendants of v and are therefore explored after v is discovered and before we backtrack on e. This suggests the use of a stack (pushdown store) for producing the vertices of the component. We store the vertices in the stack in the order that they are discovered. If on backtracking e we discover that u is a separating vertex, we read off all the vertices from the top of the stack down to and including v. All these vertices, plus u (which is not removed at this point from the stack even if it is the next on top) constitute the component. This, in effect removes the leaf C from the tree $\tilde{G}(\tilde{V}, \tilde{E})$, and if its adjacent vertex s (a separating vertex) has now $d(s) = 1$, then we can assume that it is removed too. The new superstructure is again a tree, and the same process will repeat itself to detect and trim one leaf at a time until only one component is left when the DFS terminates.

If the search starts in a separating vertex r, then all but the components which contain r are detected and produced as before, and the ones that do contain r are detected by Lemma 3.7: Each time we backtrack into r, on $r \to v$, if r has additional unexplored incident edges then we conclude that the vertices on the stack above and including v, plus r, constitute a component.

The remaining problem is that of computing $L(v)$ in time; i.e. its value should be known by the time we backtrack from v.

If v is a leaf of the DFS tree then $L(v)$ is the least element in the following set: $\{k(u) \mid u = v \text{ or } v \to u \text{ is a back edge}\}$. Let us assign $L(v) \leftarrow k(v)$ immediately when v is discovered, and as each back edge $v \to u$ is explored, let us assign

$$L(v) \leftarrow \mathrm{Min}\{L(v), k(u)\}.$$

Clearly, by the time we backtrack from v, all the back edges have been explored, and $L(v)$ has the right value.

If v is not a leaf of the DFS tree, then $L(v)$ is the least element in the following set:

$$\{k(u) \mid u = v \text{ or } v \to u \text{ is a back edge}\} \;\cup\; \{L(u) \mid v \to u \text{ is a tree edge}\}.$$

When we backtrack from v, we have already backtracked from all its sons earlier, and therefore already know their lowpoint. Thus, all we need to add is that when we backtrack from u to v, we assign

$$L(v) \leftarrow \mathrm{Min}\{L(v), L(u)\}.$$

Let us assume that $|V| > 1$ and s is the vertex in which we start the search. The algorithm is now as follows:

(1) Mark all the edges "unused". Empty the stack S. For every $v \in V$ let $k(v) \leftarrow 0$. Let $i \leftarrow 0$ and $v \leftarrow s$.

(2) $i \leftarrow i + 1$, $k(v) \leftarrow i$, $L(v) \leftarrow i$ and put v on S.

(3) If v has no unused incident edges go to Step (5).

(4) Choose an unused incident edge $v \overset{e}{—} u$. Mark e "used". If $k(u) \neq 0$, let $L(v) \leftarrow \mathrm{Min}\{L(v), k(u)\}$ and go to Step (3). Otherwise ($k(u) = 0$) let $f(u) \leftarrow v$, $v \leftarrow u$ and go to Step (2).

(5) If $k(f(v)) = 1$, go to Step (9).

(6) ($f(v) \neq s$). If $L(v) < k(f(v))$, then $L(f(v)) \leftarrow \text{Min}\{L(f(v)), L(v)\}$ and go to Step (8).

(7) ($L(v) \geq k(f(v))$) $f(v)$ is a separating vertex. All the vertices on S down to and including v are now removed from S; this set, $with f(v)$, forms a nonseparable component.

(8) $v \leftarrow f(v)$ and go to Step (3).

(9) All vertices on S down to and including v are now removed from S; they form with s a nonseparable component.

(10) If s has no unused incident edges then halt.

(11) Vertex s is a separating vertex. Let $v \leftarrow s$ and go to Step (4).

Although this algorithm is more complicated then the scanning algorithm, its time complexity is still $O(|E|)$. This follows easily from the fact that each edge is still scanned exactly once in each direction and that the number of operations per edge is bounded by a constant.

Chapter 4

ORDERED TREES

4.2 POSITIONAL TREES AND HUFFMAN'S OPTIMIZATION PROBLEM

A *positional σ-tree* (or when σ is known, a positional tree) is a directed tree with the following property: Each edge out of a vertex v is associated with one of the letters of the alphabet $\Sigma = \{0, 1, \ldots, \sigma - 1\}$; different edges, out of v, are associated with different letters. It follows that the number of edges out of a vertex is at most σ, but may be less; in fact, a leaf has none.

We associate with each vertex v the word consisting of the sequence of letters associated with the edges on the path from the root r to v. For example, consider the binary tree (positional 2-tree) of Figure 4.1, where in each vertex the associated word is written. (λ denotes the empty word.)

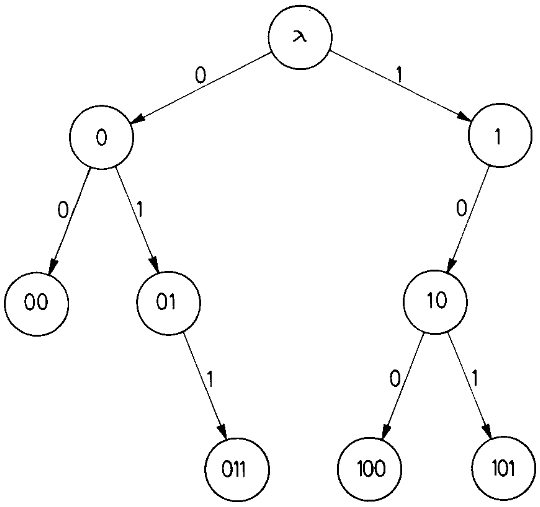


Figure 4.1

Clearly, the set of words associated with the leaves of a positional tree is a prefix code. Also, every prefix code can be described by a positional tree in this way.

The *level* of a vertex v of a tree is the length of the directed path from the root to v; it is equal to the length of the word associated with v.

Our next goal is to describe a construction of an optimum code, in a sense to be discussed shortly. It is described here as a communication problem, as it was viewed by Huffman [11], who solved it. In the next section we shall describe one more application of this optimization technique.

Assume words over a source alphabet of n letters have to be transmitted over a channel which can transfer one letter of the alphabet $\Sigma = \{0, 1, \ldots, \sigma - 1\}$ at a time, and $\sigma < n$. We want to construct a code over Σ with n code-words, and associate with each source letter a code-word. A word over the source alphabet is translated into a message over the code, by concatenating the code-words which correspond to the source letters, in the same order as they appear in the source word. This message can now be transmitted through the channel. Clearly, the code must be UD.

Assume further, that the source letters have given probabilities $p_1, p_2, \ldots p_n$ of appearance, and the choice of the next letter in the source word is independent of its previous letters. If the vector of code-word lengths is $(l_1, l_2, \ldots, l_n)$ then the average code-word length, l, is given by

$$\bar{l} = \sum_{i=1}^{n} p_i l_i \; . \tag{4.5}$$

We want to find a code for which $\bar{l}$ is minimum, in order to minimize the expected length of the message.

Since the code must be UD, by Theorem 4.1, the vector of code-word lengths must satisfy the characteristic sum condition. This implies, by Theorem 4.2, that a prefix code with the same vector of code-word lengths exists. Therefore, in seeking an optimum code, for which $\bar{l}$ is minimum, we may restrict our search to prefix codes. In fact, all we have to do is find a vector of code-word lengths for which $\bar{l}$ is minimum, among the vectors which satisfy the characteristic sum condition.

First, let us assume that $p_1 \geq p_2 \geq \cdots \geq p_n$. This is easily achieved by sorting the probabilities. We shall first demonstrate Huffman's construction for the binary case ($\sigma = 2$). Assume the probabilities are 0.6, 0.2, 0.05, 0.05, 0.03, 0.03, 0.03, 0.01. We write this list as our top row (see Fig. 4.2). We add the last (and therefore least) two numbers, and insert the sum in a proper place to maintain the non-increasing order. We repeat this operation until we get a vector with only two probabilities. Now, we assign each of them a word-length 1 and start working our way back up by assigning each of the probabilities of the previous step, its length in the present step, if it is not one of the last two, and each of the two last probabilities of the previous step is assigned a length larger by one than the length assigned to their sum in the present step.

Once the vector of code-word lengths is found, a prefix code can be assigned to it by the technique of the proof of Theorem 4.2. (An efficient implementation is discussed in Problem 4.6) Alternatively the back up procedure can produce a prefix code directly. Instead of assigning the last two probabilities with lengths, we assign the two words of length one: 0 and 1. As we back up from a present step, in which each probability is already assigned a word, to the previous step, the rule is as follows: All, but the last two probabilities of the previous step are assigned the same words as in the present step. The last two probabilities are assigned $c0$ and $c1$, where c is the word assigned to their sum in the present step.

In the general case, when $\sigma \geq 2$, we add in each step the last d probabilities of the present vector of probabilities; if n is the number of probabilities of this vector then d is given by:

$$1 < d \leq \sigma \quad \text{and} \quad n \equiv d \bmod (\sigma - 1) \tag{4.6}$$

After the first step, the length of the vector, n', satisfies $n' \equiv 1 \bmod (\sigma - 1)$, and will be equal to one, mod $(\sigma - 1)$, from there on. The reason for this rule is that we should end up with

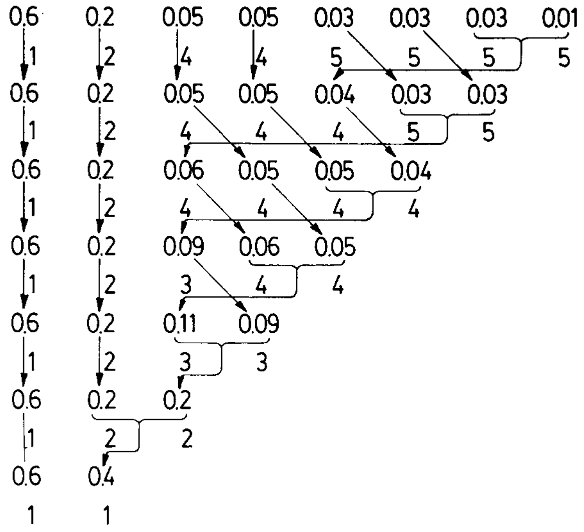


Figure 4.2

exactly σ probabilities, each to be assigned length 1. Now, $\sigma \equiv 1 \bmod (\sigma - 1)$, and since in each ordinary step the number of probabilities is reduced by $\sigma - 1$, we want $n \equiv 1 \bmod (\sigma - 1)$. In case this condition is not satisfied by the given n, we correct it in the first step as is done by our rule. Our next goal is to prove that this indeed leads to an optimum assignment of a vector of code-word lengths.

Lemma 4.2 *If $C = \{c_1, c_2, \ldots, c_n\}$ is an optimum prefix code for the probabilities $p_1, p_2, \ldots, p_n$ then $p_i > p_j$ implies that $l(c_i) \le l(c_j)$.*

Proof: Assume $l(c_i) > l(c_j)$. Make the following switch: Assign c_i to probability p_j, and c_j to p_i; all other assignments remain unchanged. Let $\tilde{l}$ denote the average code-word length of the new assignment, while $\bar{l}$ denotes the previous one. By (4.5) we have

$$\bar{l} - \tilde{l} = [p_i \cdot l(c_i) + p_j \cdot l(c_j)] - [p_i \cdot l(c_j) + p_j \cdot l(c_i)] = (p_i - p_j)(l(c_i) - l(c_j)) > 0$$

contradicting the assumption that $\bar{l}$ is minimum.

Q.E.D.

Lemma 4.3 *There exists an optimum prefix code for the probabilities $p_1 \ge p_2 \ge \cdots \ge p_n$ such that the positional tree which represents it has the following properties:*

(1) All the internal vertices of the tree, except possibly one internal vertex v, have exactly σ sons.

(2) Vertex v has $1 < \rho \le \sigma$ sons, where $n \equiv \rho$ mod$(\sigma - 1)$.

(3) Vertex v, of (1) is on the lowest level which contains internal vertices, and its sons are assigned to $p_{n-\rho+1}, p_{n-\rho+2}, \ldots, p_n$.

Proof: Let T be a positional tree which represents an optimum prefix code. If there exists an internal vertex u, which is not on the lowest level of T containing internal vertices and it has less than σ sons, then we can perform the following change in T: Remove one of the leaves of T from its lowest level and assign to the probability a new son of u. The resulting tree, and therefore, its corresponding prefix code has a smaller average code-word length. A contradiction. Thus, we conclude that no such internal vertex u exists.

If there are internal vertices, on the lowest level of internal vertices, which have less than σ sons, choose one of them, say v. Now eliminate sons from v and attach their probabilities to new sons of the others, so that their number of sons is σ. Clearly, such a change does not change the average length and the tree remains optimum. If before filling in all the missing sons, v has no more sons, we can use v as a leaf and assign to it one of the probabilities from the lowest level, thus creating a new tree which is better than T. A contradiction. Thus, we never run out of sons of v to be transferred to other lacking internal vertices on the same level. Also, when this process ends, v is the only lacking internal vertex (proving (1)) and its number of remaining sons must be greater than one, or its son can be removed and its probability attached to v. This proves that the number of sons of v, ρ, satisfies $1 < \rho \leq \sigma$.

If v's ρ sons are removed, the new tree has $n' = n - \rho + 1$ leaves and is *full* (i.e., every internal vertex has exactly σ sons). In such a tree, the number of leaves, n', satisfies $n' \equiv 1$, mod $(\sigma - 1)$. This is easily proved by induction on the number of internal vertices. Thus, $n - \rho + 1 \equiv 1 \bmod (\sigma - 1)$, and therefore $n \equiv \rho \bmod (\sigma - 1)$, proving (2).

We have already shown that v is on the lowest level of T which contains internal vertices and number of its sons is ρ. By Lemma 4.2, we know that the least ρ probabilities are assigned to leaves of the lowest level of T. If they are not sons of v, we can exchange sons of v with sons of other internal vertices on this level, to bring all the least probabilities to v, without changing the average length.

Q.E.D.

For a given alphabet size σ and probabilities $p_1 \geq p_2 \geq \ldots \geq p_n$, let $\theta_\sigma(p_1, p_2, \ldots, p_n)$ be the set of all σ-ary positional trees with n leaves, assigned with the probabilities $p_1, p_2, \ldots, p_n$ in such a way that $p_{n-d+1}, p_{n-d+2}, \ldots, p_n$ (see (4.6)) are assigned, in this order, to the first d sons of a vertex v, which has no other sons. By Lemma 4.3, $\theta_\sigma(p_1, p_2, \ldots, p_n$ contains at least one optimum tree. Thus, we may restrict our search for an optimum tree to $\theta_\sigma(p_1, p_2, \ldots, p_n)$.

Lemma 4.4 *There is a one to one correspondence between $\theta_\sigma(p_1, p_2, \ldots, p_n)$ and the set of σ-ary positional trees with $n - d + 1$ leaves assigned with $p_1, p_2, \ldots, p_{n-d}, p'$ where $p' = \sum_{i=n-d+1}^{n} p_i$. The average word-length $\bar{l}$, of the prefix code represented by a tree T of $\theta_\sigma(p_1, p_2, \ldots, p_n)$ and the average code word-length $\bar{l}'$, of the prefix code represented by the tree T', which corresponds to T, satisfy*

$$\bar{l} = \bar{l}' + p'. \tag{4.7}$$

Proof: The tree T' which corresponds to T is achieved as follows: Let v be the father of the leaves assigned $p_{n-d+1}, p_{n-d+2}, \ldots, p_n$. Remove all the sons of v and assign p' to it.

It is easy to see that two different trees T_1 and T_2 in $\theta_\sigma(p_1, p_2, \ldots, p_n)$ will yield two different trees T_1' and T_2', and that every σ-ary tree T' with $n - d + 1$ leaves assigned $p_1, p_2, \ldots, p_{n-d}, p'$, is the image of some T; establishing the correspondence.

Let l_i denote the level of the leaf assigned p_i in T. Clearly $l_{n-d+1} = l_{n-d+2} = \ldots = l_n$. Thus,

$$\bar{l} = \sum_{i=1}^{n-d} p_i \cdot l_i + l_n \cdot \sum_{i=n-d+1}^{n} p_i = \sum_{i=1}^{n-d} p_i \cdot l_i + l_n \cdot p' = \sum_{i=1}^{n-d} p_i \cdot l_i + (l_n - 1) \cdot p' + p' = \bar{l}' + p'.$$

Q.E.D.

Lemma 4.4 suggests a recursive approach to find an optimum T. For $\bar{l}$ to be minimum, $\bar{l}'$ must be minimum. Thus, let us first find an optimum T' and then find T by attaching d sons to the vertex of T' assigned p'; these d sons are assigned $p_{n-d+1}, p_{n-d+2}, \ldots, p_n$. This is exactly what is done in Huffman's procedure, thus proving its validity.

It is easy to implement Huffman's algorithm in time complexity $O(n^2)$. First we sort the probabilities, and after each addition, the resulting probability is inserted in a proper place. Each such insertion takes at most $O(n)$ steps, and the number of insertions is $\lceil (n-\sigma)/(\sigma-1) \rceil$. Thus, the whole forward process is of time complexity $O(n^2)$. The back up process is $O(n)$ if pointers are left in the forward process to indicate the probabilities of which it is composed.

However, the time complexity can be reduced to $O(n \log n)$. One way of doing it is the following: First sort the probabilities. This can be done in $O(n \log n)$ steps [14]. The sorted probabilities are put on a queue S_1 in a non-increasing order from left to right. A second queue, S_2, initially empty, is used too. In the general step, we repeatedly take the least probability of the two (or one, if one of the queues is empty) appearing at the right hand side ends of the two queues, and add up d of them. The result, p', is inserted at the left hand side end of S_2. The process ends when after adding d probabilities both queues are empty. This adding process and the back up are $O(n)$. Thus, the whole algorithm is $O(n \log n)$.

The construction of an optimum prefix code, when the cost of the letters are not equal is discussed in Reference 12; the case of alphabetic prefix codes, where the words must maintain lexicographically the order of the given probabilities, is discussed in Reference 13. These references give additional references to previous work.

4.4 CATALAN NUMBERS

The set of *well-formed sequences of parentheses* is defined by the following recursive definition:

1. The empty sequence is well formed.

2. If A and B are well-formed sequences, so is AB (the concatenation of A and B).

3. If A is well formed, so is (A).

4. There are no other well-formed sequences.

For example, $(()(()))$ is well formed; $(()))(()$ is not.

Lemma 4.5 *A sequence of (left and right) parentheses is well formed if and only if it contains an even number of parentheses, half of which are left and the other half are right, and as we read the sequence from left to right, the number of right parentheses never exceeds the number of left parentheses.*

Proof: First let us prove the "only if" part. Since the construction of every well formed sequence starts with no parentheses (the empty sequence) and each time we add on parentheses (Step 3) there is one left and one right, it is clear that there are n left parentheses and n right parentheses. Now, assume that for every well-formed sequence of m left and m right parentheses, where m ¡ n, it is true that as we read it from left to right the number of right parentheses never exceeds the number of left parentheses. If the last step in the construction of our sequence was 2, then since A is a well-formed sequence, as we read from left to right, as long as we still read A the condition is satisfied. When we are between A and B, the count of left and right parentheses equalizes. From there on the balance of left and right is safe since B is well formed and contains less than n parentheses. If the last step in the construction of our sequence was 3, then since A satisfies the condition, so does (A).

Now, we shall prove the "if" part, again by induction on the number of parentheses. (Here, as before, the basis of the induction is trivial.) Assume that the statement holds for all sequences of m left and m right parentheses, if m ¡ n, and we are given a sequence of n left and n right parentheses which satisfies the condition. Clearly, if after reading $2m$ symbols of it from left to right the number of left and right parentheses is equal and if m ¡ n, this subsequence, A, by the inductive hypothesis is well formed. Now, the remainder of our sequence, B, must satisfy the condition, too, and again by the inductive hypothesis is well formed. Thus, by Step 2, AB is well formed. If there is no such nonempty subsequence A, which leaves a nonempty B, then as we read from left to right the number of right parentheses, after reading one symbol and before reading the whole sequence, is strictly less then the number of left parentheses. Thus, if we delete the first symbol, which is a "(", and the last, which is a ")", the remainder sequence, A, still satisfies the condition, and by the inductive hypothesis is well formed. By Step 3 our sequence is well formed too.

Q.E.D.

We shall now show a one-to-one correspondence between the non-well-formed sequences of n left and n right parentheses, and all sequences of $n-1$ left parentheses and $n+1$ right parentheses.

Let $p_1p_2\cdots p_{2n}$ be a sequence of n left and n right parentheses which is not well formed. By Lemma 4.5, there is a prefix of it which contains more right parentheses then left. Let j be the least integer such that the number of right parentheses exceeds the number of left parentheses in the subsequence $p_1p_2\cdots p_j$. Clearly, the number of right parentheses is then one larger than the number of left parentheses, or j is not the least index to satisfy the condition. Now, invert all p_i's for $i > j$ from left parentheses to right parentheses, and from right parentheses to left parentheses. Clearly, the number of left parentheses is now $n-1$, and the number of right parentheses is now $n+1$.

Conversely, given any sequence $p_1p_2\cdots p_{2n}$ of $n-1$ left parentheses and $n+1$ right parentheses, let j be the first index such that $p_1p_2\cdots p_j$ contains one right parenthesis more than left parentheses. If we now invert all the parentheses in the section $p_{j+1}p_{j+2}\cdots p_{2n}$ from left to right and from right to left, we get a sequence of n left and n right parentheses which is not well formed. This transformation is the inverse of the one of the previous paragraph. Thus, the one-to-one correspondence is established.

The number of sequences of $n-1$ left and $n+1$ right parentheses is

$$\binom{2n}{n-1},$$

for we can choose the places for the left parentheses, and the remaining places will have right parentheses. Thus, the number of well-formed sequences of length n is

$$\binom{2n}{n} - \binom{2n}{n-1} = \frac{1}{1+n}\binom{2n}{n}. \tag{4.9}$$

These numbers are called *Catalan numbers*.

An *ordered tree* is a directed tree such that for each internal vertex there is a defined order of its sons. Clearly, every positional tree is ordered, but the converse does not hold: In the case of ordered trees there are no predetermined "potential" sons; only the order of the sons counts, not their position, and there is no limit on the number of sons.

An *ordered forest* is a sequence of ordered trees. We usually draw a forest with all the roots on one horizontal line. The sons of a vertex are drawn from left to right in their given order. For example, the forest shown in Fig. 4.6 consists of three ordered trees whose roots are A, B, and C.

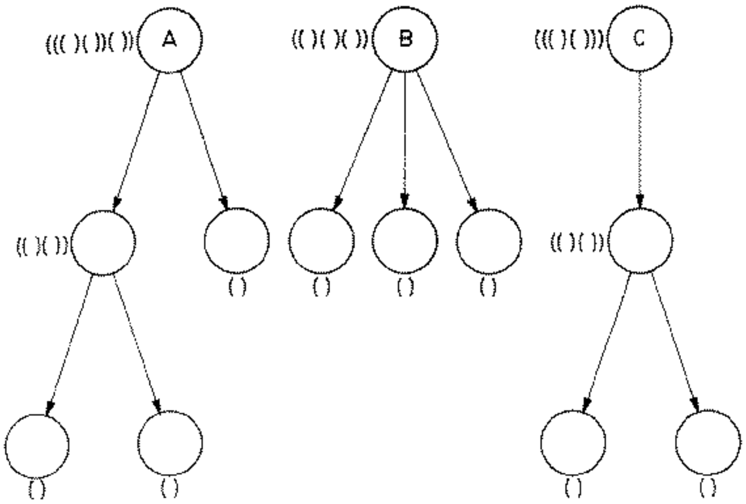


Figure 4.6

There is a natural correspondence between well-formed sequences of n pairs of parentheses and ordered forests of n vertices. Let us label each leaf with the sequence (). Every vertex whose sons are labeled $w_1, w_2, \ldots, w_s$ is labeled with the concatenation $(w_1 w_2 \cdots w_s)$; clearly, the order of the labels is in the order of the sons. Finally, once the roots are labeled $x_1, x_2, \ldots, x_r$ the sequence which corresponds to the forest is the concatenation $x_1 x_2 \cdots x_n$. For example, the sequence which corresponds to the forest of Fig. 4.6 is $((()())())(()()())((()()))$. The inverse transformation clearly exists and thus the one-to-one correspondence is established. Therefore, the number of ordered forests of n vertices is given by (4.9).

We shall now describe a one-to-one correspondence between ordered forests and positional binary trees. The leftmost root of the forest is the root of the binary tree. The leftmost son of the vertex in the forest is the left son of the vertex in the binary tree. The next brother on the right, or, in the case of a root, the next root on the right is the right son in the binary tree. For example, see Fig. 4.7, where an ordered forest and its corresponding binary tree are drawn. Again, it is clear that this is a one-to-one correspondence and therefore the number of positional binary trees with n vertices is given by (4.9).

There is yet another combinatorial enumeration which is directly related to these.

A *stack* is a storage device which can be described as follows. Suppose that n cars travel on a narrow one-way street where no passing is possible. This leads into a narrow two-way street on which the cars can park or back up to enter another narrow one-way street (see Fig. 4.8). Our problem is to find how may permutations of the cars can be realized from input to output if we assume that the cars enter in the natural order.

The order of operations in the stack is fully described by the sequence of drive-in and drive-out operations. There is no need to specify which car drives in, for it must be the first one on the leading-in present queue; also, the only one which can drive out is the top one in the stack. If we denote a drive-in operation by "(", and a drive-out operation by ")", the whole procedure

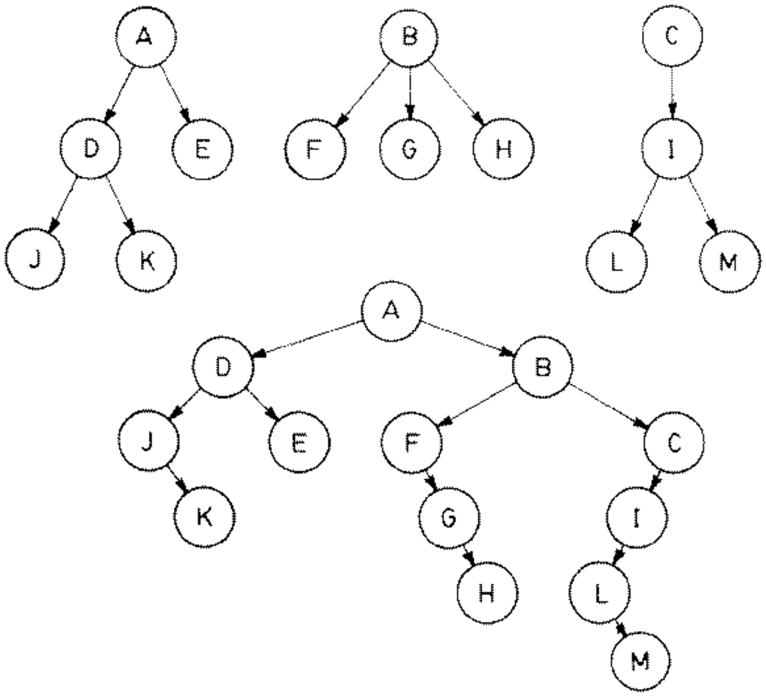


Figure 4.7

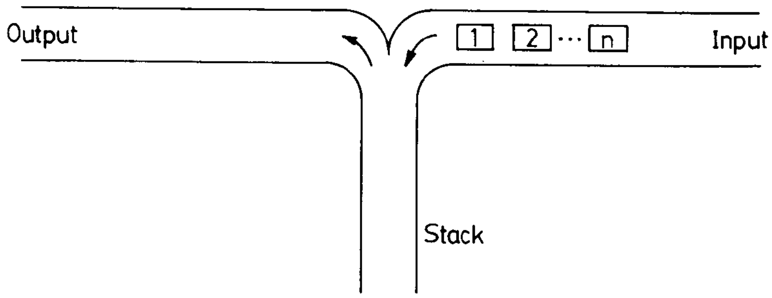


Figure 4.8

is described by a well-formed sequence of n pairs of parentheses.

The sequence must be well-formed, by Lemma 4.5, since the number of drive-out operations can never exceed the number of drive-in operations. Also, every well-formed sequence of n pairs of parentheses defines a realizable sequence of operations, since again by Lemma 4.5, a drive-out is never instructed when the stack is empty. Also, different sequences yield different permutations. Thus, the number of permutations on n cars realizable by a stack is given by (4.9).

Let us now consider the problem of finding the number of full binary trees. Denote the number of leaves of a binary tree T by $L(T)$, and the number of internal vertices by $I(T)$. It is easy to prove, by induction on the number of leaves, that $L(T) = I(T) + 1$. Also, if all leaves of T are removed, the resulting tree of $I(T)$ vertices is a positional binary tree T'. Clearly, different T-s will yield different T'-s, and one can reconstruct T from T' by attaching two leaves to each leaf of T', and one leaf (son) to each vertex which in T' has only one son. Thus, the number of full-binary trees of n vertices is equal to the number of positional binary trees of $(n-1)/2$

vertices. By (4.9) this number is

$$\frac{2}{n+1}\binom{n-1}{\frac{n-1}{2}}$$

Chapter 5

MAXIMUM FLOW IN A NETWORK

5.1 THE FORD AND FULKERSON ALGORITHM

A *network* consists of the following data:

(1) A finite digraph $G(V, E)$ with no self-loops and no parallel edges.*

(2) Two vertices s and t are specified; s is called the *source* and t, the *sink*.†

(3) Each edge $e \in E$ is assigned a non-negative number $c(e)$ called the *capacity* of e.

A *flow function* f is an assignment of a real number $f(e)$ to each edge e, such that the following two conditions hold:

(C1) For every edge $e \in E$, $0 \leq f(e) \leq c(e)$.

(C2) Let $\alpha(v)$ and $\beta(v)$ be the sets of edges incoming to vertex v and outgoing from v, respectively. For every $v \in V - \{s, t\}$

$$0 = \sum_{e \in \alpha(v)} f(e) - \sum_{e \in \beta(v)} f(e). \tag{5.1}$$

The *total flow* F of f is defined by

$$F = \sum_{e \in \alpha(t)} f(e) - \sum_{e \in \beta(t)} f(e). \tag{5.2}$$

Namely, F is the net sum of flow into the sink. Our problem is to find an f for which the total flow is maximum.

*The exclusion of self-loops and parallel edges is not essential. It will shortly become evident that no generality is lost; the flow in a self-loop gains nothing, and a set of parallel edges can be replaced by one whose capacity is the sum of their capacities. This condition ensures that $|E| \leq |V| \cdot (|V| - 1)$.

†The choice of s or t is completely arbitrary. There is no requirement that s is a graphical source; i.e. has no incoming edges, or that t is a graphical sink; i.e. has no outgoing edges. The edges entering s or leaving t are actually redundant and have no effect on our problem, but we allow them since the choice of s and t may vary, while we leave the other data unchanged.

Let S be a subset of vertices such that $s \in S$ and $t \notin S$. $\bar{S}$ is the complement of S, i.e. $\bar{S} = V - S$. Let $(S; \bar{S})$ be the set of edges of G whose start-vertex is in S and end-vertex is in $\bar{S}$. The set $(\bar{S}; S)$ is defined similarly. The set of edges connecting vertices of S with $\bar{S}$ (in both directions) is called the *cut* defined by S.

By definition, the total flow F is measured at the sink. Our purpose is to show that F can be measured at any cut.

Lemma 5.1 *For every S*

$$F = \sum_{e \in (S;\bar{S})} f(e) - \sum_{e \in (\bar{S};S)} f(e). \tag{5.3}$$

Proof: Let us sum up equation (5.2) with all the equations (5.1) for $v \in \bar{S} - \{t\}$. The resulting equation has F on the left hand side. In order to see what happens on the right hand side, consider an edge $x \xrightarrow{e} y$. If both x and y belong to S then $f(e)$ does not appear on the r.h.s. at all, in agreement with (5.3). If both x and y belong to $\bar{S}$ then $f(e)$ appears on the r.h.s. once positively, in the equation for y, and once negatively, in the equation for x. Thus, in the summation it is canceled out, again in agreement with (5.3). If $x \in S$ and $y \in \bar{S}$ then $f(e)$ appears on the r.h.s. of the equation for y, positively, and in no other equation we use, and indeed $e \in (S; \bar{S})$, and again we have agreement with (5.3). Finally, if $x \in \bar{S}$ and $y \in S$, $f(e)$ appears negatively on the r.h.s. of the equation for x, and again this agrees with (5.3) since $e \in (\bar{S}; S)$.

Q.E.D.

Let us denote by $c(S)$ the *capacity of the cut* determined by S which is defined as follows:

$$c(S) = \sum_{e \in (S;\bar{S})} c(e). \tag{5.4}$$

Lemma 5.2 *For every flow function f, with total flow F, and every S,*

$$F \le c(S). \tag{5.5}$$

Proof: By Lemma 5.1

$$F = \sum_{e \in (S;\bar{S})} f(e) - \sum_{e \in (\bar{S};S)} f(e).$$

By $C1$, $0 \le f(e) \le c(e)$ for every $e \in E$. Thus,

$$F \le \sum_{e \in (S;\bar{S})} f(e) - 0 = c(S).$$

Q.E.D.

A very important corollary of Lemma 5.2, which allows us to detect that a given total flow F is maximum, and a given cut, defined by S, is minimum is the following:

Corollary 5.1 *If F and S satisfy (5.5) by equality then F is maximum and the cut defined by S is of minimum capacity.*

Ford and Fulkerson [1] suggested the use of augmenting paths to change a given flow function in order to increase the total flow. An *augmenting path* is a simple path from s to t, which is not necessarily directed, but it can be used to advance flow from s to t. If on this path, e points in the direction from s to t, then in order to be able to push flow through it, $f(e)$ must be less than $c(e)$. If e points in the opposite direction, then in order to be able to push through it additional flow from s to t, we must be able to cancel some of its flow. Therefore, $f(e) > 0$ must hold.

In attempt to find an augmenting path for a given flow, a labeling procedure is used. We label s. Then, every vertex v, for which we can find an augmenting path from s to v, is labeled. If t is labeled then an augmenting path has been found. This path is used to increase the total flow, and the procedure is repeated.

A *forward labeling* of vertex v by the edge $u \xrightarrow{e} v$ is applicable if

(1) u is labeled and v is not;

(2) $c(e) > f(e)$.

The label that v gets is $'e'$. If e is used for forward labeling we define $\Delta(e) = c(e) - f(e)$.

A *backward labeling* of vertex v by the edge $u \xleftarrow{e} v$ is applicable if

(1) u is labeled and v is not;

(2) $f(e) > 0$.

The label that v gets is $'e'$. In this case we define $\Delta(e) = f(e)$.

The Ford and Fulkerson algorithm is as follows:

(1) Assign some legal initial flow f to the edges; an assignment $f(e) = 0$ to every edge e will do.

(2) Mark s "labeled" and all other vertices "unlabeled".

(3) Search for a vertex v which can be labeled by either a forward or backward labeling. If none exists, halt; the present flow is maximum. If such a vertex v exists, label it $'e'$, where e is the edge through which the labeling is possible. If $v = t$, go to Step (4); otherwise, repeat Step (3).

(4) Starting from t and by the use of the labels, backtrack the path through which the labeling reached t from s. Let this path be $s = v_0 \overset{e_1}{—} v_1 \overset{e_2}{—} v_2 \overset{e_3}{—} \cdots — v_{l-1} \overset{e_l}{—} v_l = t$. (The directions of the edges are not shown, since each may be in either direction.) Let $\Delta = \mathrm{Min}_{1 \le i \le l} \Delta(e_i)$. If e_i is forward, i.e. $v_{i-1} \xrightarrow{e_i} v_i$, then $f(e_i) \leftarrow f(e_i) + \Delta$. If e_i is backward, i.e. $v_{i-1} \xleftarrow{e_i} v_i$, then $f(e_i) \leftarrow f(e_i) - \Delta$.

(5) Go to Step (2)

(Note that if the initial flow on the edges entering s is zero, it will never change. This is also true for the edges leaving t.)

As an example, consider the network shown in Fig. 5.1. Next to each edge e we write $c(e), f(e)$ in this order. We assume a zero initial flow everywhere. A first wave of label propagation might be as follows: s is labeled; e_2 used to label c; e_6 used to label d; e_4 used to label a; e_3 used to label b; and finally, e_7 used to label t. The path is $s \xrightarrow{e_2} c \xrightarrow{e_6} d \xrightarrow{e_4} a \xrightarrow{e_3} b \xrightarrow{e_7} t$. $\Delta = 4$, and the new flow is shown in Fig. 5.2.

The next augmenting path may be

$$s \xrightarrow{e_1} a \xrightarrow{e_3} b \xrightarrow{e_5} c \xrightarrow{e_6} d \xrightarrow{e_8} t.$$

Now, $\Delta = 3$ and the flow is as in Fig. 5.3.

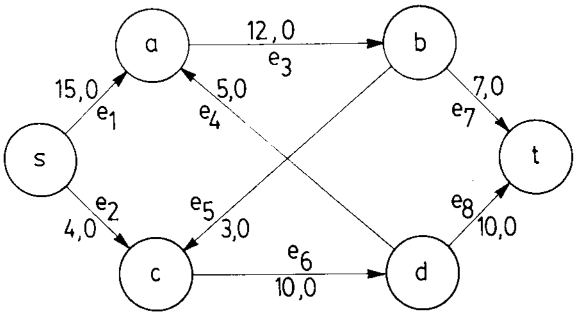


Figure 5.1

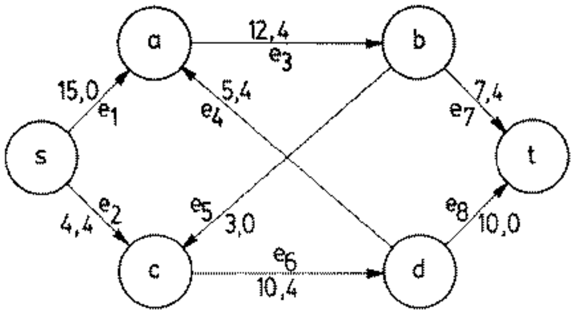


Figure 5.2

The next augmenting path may be $s \xrightarrow{e_1} a \xrightarrow{e_3} b \xrightarrow{e_7} t$. Now, $\Delta = 3$ and the new flow is as in Fig. 5.4. Now, the labeling can proceed as follows: s is labeled; e_1 is used to label a; e_3 used to label b; (so far we have not used backward labeling, but this next step is forced) e_4 is used to label d; e_8 is used to label t. The path we backtrack is $s \xrightarrow{e_1} a \xleftarrow{e_4} d \xrightarrow{e_8} t$. Now, $\Delta(e_1) = 9$, $\Delta(e_4) = 4$ and $\Delta(e_8) = 7$. Thus, $\Delta = 4$. The new flow is shown in Fig. 5.5. The next wave of label propagation is as follows: s is labeled, e_1 is used to label a, e_3 used to label b. No more labeling is possible and the algorithm halts.

It is easy to see that the flow produced by the algorithm remains legal throughout. The definition of $\Delta(e)$ and Δ guarantees that forward edges will not be overflowed, i.e., $f(e) \leq c(e)$, and that backward edges will not be underflowed, i.e., $f(e) \geq 0$. Also, since Δ is pushed from s to t on a path, the incoming flow will remain equal to the outgoing flow in every vertex $v \in V - \{s, t\}$.

Assuming the algorithm halts, the last labeling process has not reached t. Let S be the set of vertices labeled in the last wave. (In our example $S = \{s, a, b\}$,) If an edge $x \xrightarrow{e} y$ belongs

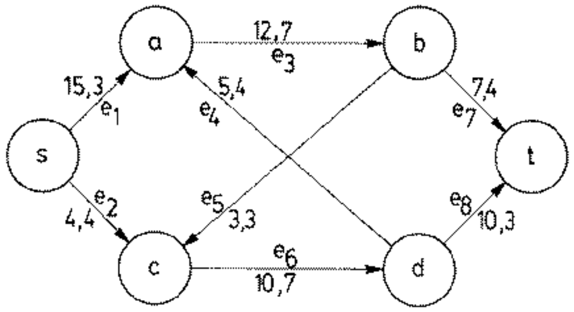


Figure 5.3

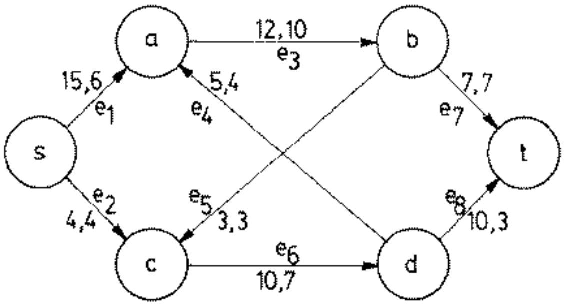


Figure 5.4

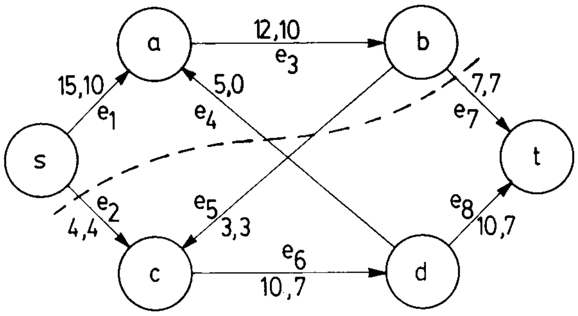


Figure 5.5

to $(S;\bar{S})$ then it must be saturated, i.e., $f(e) = c(e)$, or a forward labeling could use it. Also, if e belongs to $(\bar{S};S)$ then it follows that $f(e) = 0$, or a backward labeling could use it. By Lemma 5.1 we have

$$F = \sum_{e\in(S;\bar{S})} f(e) - \sum_{e\in(\bar{S};S)} f(e) = \sum_{e\in(S;\bar{S})} c(e) = c(S).$$

Now, by Corollary 5.1, F is a maximum total flow and S defines a minimum cut. In our example, $(S;\bar{S}) = \{e_2, e_5, e_7\}$, $(\bar{S};S) = \{e_4\}$ and the value of F is 14.

The question of whether the algorithm will always halt remains to be discussed. Note first a very important property of the Ford and Fulkerson algorithm: If the initial flow is integral, for example, zero everywhere, and if all the capacities are integers, then the algorithm never introduces fractions. The algorithm adds and subtracts, but it never divides. Also, if t is labeled, the augmenting path is used to increase the total flow by at least one unit, Since there is an upper bound on the total flow (any cut), the process must terminate.

Ford and Fulkerson showed that their algorithm may fail, if the capacities are allowed to be irrational numbers. Their counterexample (Reference 1, p. 21) displays an infinite sequence of flow augmentations. The flow converges (in infinitely many steps) to a value which is one fourth of the maximum total flow. We shall not bring their example here; it is fairly complex and as the reader will shortly discover, it is not as important any more.

One could have argued that for all practical purposes, we may assume that the algorithm is sure to halt. This follows from the fact that our computations are usually through a fixed radix (decimal, binary, and so on) number representation with a bound on the number of digits used; in other words, all figures are multiples of a fixed quantum and the termination proof works here as it does for integers. However, a simple example shows the weakness of this argument. Consider the network shown in Fig. 5.6. Assume that M is a very large integer. If the algorithm starts with $f(e) = 0$ for all e, and alternatively uses s — a — b — t and s — b — a — t as augmenting paths, it will take $2M$ augmentations before $F = 2M$ is achieved.

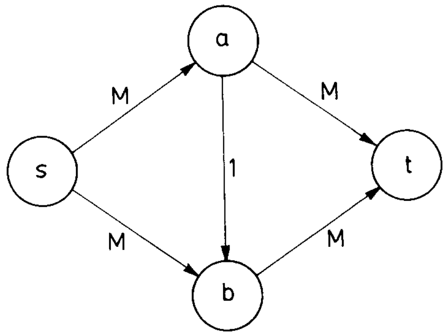


Figure 5.6

Edmonds and Karp [2] were first to overcome this problem. They showed that if one uses breadth-first search (BFS) in the labeling algorithm and always uses a shortest augmenting path, the algorithm will terminate in $O(|V|^3|E|)$ steps, regardless of the capacities. (Here, of course, we assume that our computer can handle, in one step, any real number.) In the next section we shall present the more advanced work of Dinic [3]; his algorithm has time complexity $O(|V|^2|E|)$. Karzanov [4] and Cherkassky [5] have reduced it to $O(|V|^3)$ and $O(|V|^2|E|^{1/2})$, respectively. These algorithms are fairly complex and will not be described. A recent algorithm of Malhotra, Pramodh Kumar and Maheshwari [6] has the same time complexity as Karzanov's and is much simpler; it will be described in the next section.

The existence of these algorithms assures that, if one proceeds according to a proper strategy in the labeling procedure, the algorithm is guaranteed to halt. When it does, the total flow is maximum, and the cut indicated is minimum, thus providing the *max-flow min-cut* theorem:

Theorem 5.1 *Every network has a maximum total flow which is equal to the capacity of a cut for which the capacity is minimum.*

5.2 THE DINIC ALGORITHM

As in the Ford and Fulkerson algorithm, the Dinic algorithm starts with some legal flow function f and improves it. When no improvement is possible the algorithm halts, and the total flow is maximum.

If presently an edge $u \overset{e}{—} v$ has flow $f(e)$ then we say that e is *useful* from u to v if one of the following two conditions holds:

(1) $u \xrightarrow{e} v$ and $f(e) < c(e)$.

(2) $u \xleftarrow{e} v$ and $f(e) > 0$.

The *layered network* of $G(V, E)$ with a flow f is defined by the following algorithm:

(1) $V_0 \leftarrow \{s\}$, $i \leftarrow 0$.

(2) Construct $T \leftarrow \{v \mid v \notin V_j$ for $j \leq i$ and there is a useful edge from a vertex of V_i to $v\}$.

(3) If T is empty, the present total flow F is maximum, halt.

(4) If T contains t then $l \leftarrow i + 1$, $V_l \leftarrow \{t\}$ and halt.

(5) Let $V_{i+1} \leftarrow T$, increment i and return to Step (2).

For every $1 \le i \le l$, let E_i be the set of edges useful from a vertex of V_{i-1} to a vertex of V_i. The sets V_i are called *layers*.

The construction of the layered network investigates each edge at most twice; once in each direction. Thus, the time complexity of this algorithm is $O(|E|)$.

Lemma 5.3 *If the construction of the layered network terminates in Step (3), then the present total flow, F, is indeed maximum.*

Proof: The proof here is very similar to the one in the Ford and Fulkerson algorithm: Let S be the union of $V_0, V_1, \ldots, V_i$. Every edge $u \xrightarrow{e} v$ in $(S; \bar{S})$ is saturated, i.e. $f(e) = c(e)$, or else e is useful from u to v and T is not empty. Also, every edge $u \xleftarrow{e} v$ is $(\bar{S}; S)$ has $f(e) = 0$, or again e is useful from u to v, etc. Thus, by Lemma 5.1,

$$F = \sum_{e \in (S;\bar{S})} f(e) - \sum_{e \in (\bar{S};S)} f(e) = \sum_{e \in (S;\bar{S})} c(e) - 0 = c(S).$$

By Corollary 5.1, F is maximum.

Q.E.D.

For every edge e in E_j let $\tilde{c}(e)$ be defined as follows:

(i) If $u \in V_{j-1}$, $v \in V_j$ and $u \xrightarrow{e} v$ then $\tilde{c}(e) = c(e) - f(e)$.

(ii) If $u \in V_{j-1}$, $v \in V_j$ and $u \xleftarrow{e} v$ then $\tilde{c}(e) = f(e)$.

We now consider all edges of E_j to be directed from V_{j-1} to V_j, even if in $G(V, E)$ they may have the opposite direction (in case (ii)). Also, the initial flow in the new network is $\tilde{f}(e) = 0$ everywhere. We seek a maximal flow $\tilde{f}$ in the layered network; by a *maximal flow* $\tilde{f}$ we mean that $\tilde{f}$ satisfies the condition that for every path $s \overset{e_1}{—} v_1 \overset{e_2}{—} v_2 — \cdots v_{l-1} \overset{e_l}{—} t$, where $v_j \in V_j$ and $e_j \in E_j$, there is at least one edge e_j such that $\tilde{f}(e_j) = \tilde{c}(e_j)$. Clearly, a maximal flow is not necessarily maximum as the example of Figure 5.7 shows: If for all edges $\tilde{c} = 1$ and we push one unit flow from s to t via a and d then the resulting flow is maximal in spite of the fact that the total flow is 1 while a total flow of 2 is possible.

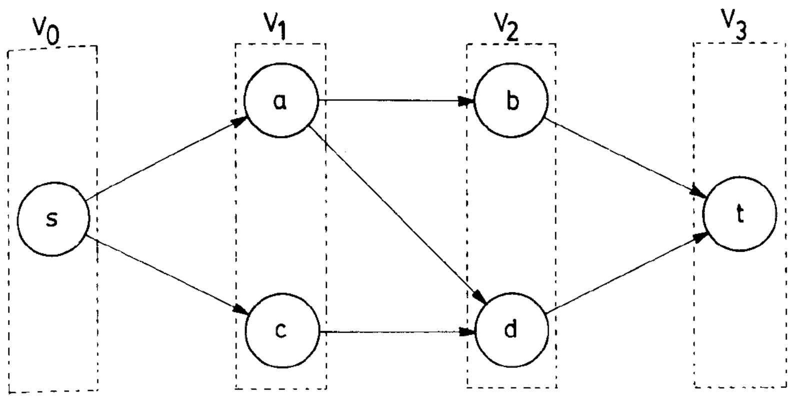


Figure 5.7

Later we shall describe how one can find a maximal flow function $\tilde{f}$ efficiently. For now, let us assume that such a flow function has been found and its total value is $\tilde{F}$. The flow f in $G(V, E)$ is changed into f' as follows:

(i) If $u \xrightarrow{e} v$, $u \in V_{j-1}$ and $v \in V_j$ then $f'(e) = f(e) + \tilde{f}(e)$.

(ii) If $u \xleftarrow{e} v$, $u \in V_{j-1}$ and $v \in V_j$ then $f'(e) = f(e) - \tilde{f}(e)$.

It is easy to see that the new flow f' satisfies both $C1$ (due to the choice of $\tilde{c}$) and $C2$ (because it is the superposition of two flows which satisfy $C2$). Clearly $F' = F + \tilde{F} > F$.

Let us call the part of the algorithm which starts with f, finds its layered network, finds a maximal flow $\tilde{f}$ in it and improves the flow in the original network to become f' — a *phase*. We want to show that the number of phases is bounded by $|V|$. For this purpose we shall prove that the length of the layered network increases from phase to phase; by *length* is meant the index of the last layer, which we called l in Step (4) of the layered network algorithm. Thus, l_k denotes the length of the layered network of the kth phase.

Lemma 5.4 *If the $(k+1)$st phase is not the last then $l_{k+1} > l_k$.*

Proof: There is a path of length l_{k+1} in the $(k+1)$st layered network which starts with s and ends with t:

$$s \overset{e_1}{—} v_1 \overset{e_2}{—} \cdots v_{l_{k+1}-1} \overset{e_{l_{k+1}}}{—} t.$$

First, let us assume that all the vertices of the path appear in the k-th layered network. Let V_j be the jth layer of the kth layered network. We claim that if $v_a \in V_b$ then $a \geq b$. This is proved by induction on a. For $a = 0$, $(v_0 = s)$ the claim is obviously true. Now, assume $v_{a+1} \in V_c$. If $c \leq b + 1$ the inductive step is trivial. But if $c > b + 1$ then the edge e_{a+1} has not been used in the kth phase since it is not even in the kth layered network, in which only edges between adjacent layers appear. If e_{a+1} has not been used and is useful from v_a to v_{a+1} in the beginning of phase $k+1$, then it was useful from v_a to v_{a+1} in the beginning of phase k. Thus, v_{a+1} cannot belong to V_c (by the algorithm). Now, in particular, $t = v_{l_{k+1}}$ and $t \in V_{l_k}$. Therefore, $l_{k+1} \geq l_k$. Also, equality cannot hold, because in this case the whole path is in the kth layered network, and if all its edges are still useful in the beginning of phase $k+1$ then the $\tilde{f}$ of phase k was not maximal.

If not all the vertices of the path appear in the kth layered network then let $v_a \overset{e_{a+1}}{—} v_{a+1}$ be the first edge such that for some b $v_a \in V_b$ but v_{a+1} is not in the kth layered network. Thus, e_{a+1} was not used in phase k. Since it is useful in the beginning of phase $k+1$, it was also useful in the beginning of phase k. The only possible reason for v_{a+1} not to belong to V_{b+1} is that $b + 1 = l_k$. By the argument of the previous paragraph $a \geq b$. Thus $a + 1 \geq l_k$, and therefore $l_{k+1} > l_k$.

Q.E.D.

Corollary 5.2 *The number of phases is less than or equal to $|V| - 1$.*

Proof: Since $l \leq |V| - 1$, Lemma 5.4 implies the corollary.

Q.E.D.

The remaining task is to describe an efficient algorithm to construct a maximal flow in a layered network.

First, let us show Dinic's method.

We assume that $\tilde{N}$ is a layered network, and for every edge e in $\tilde{N}$ $\tilde{c}(e) > 0$.

(1) For every e in $\tilde{N}$, mark e "unblocked" and let $\tilde{f}(e) \leftarrow 0$.

(2) $v \leftarrow s$ and empty the stack S.

(3) If there is no unblocked edge $v — u$, with u in the next layer, then (v is a *dead-end* and) perform the following operations:

(3.1) If $s = v$, halt; the present $\tilde{f}$ is maximal.

(3.2) Delete the top-most edge $u \overset{e}{—} v$ from S.

(3.3) Mark e "blocked" and let $v \leftarrow u$.

(3.4) Repeat Step (3).

(4) Choose an unblocked edge $v \overset{e}{—} u$, with u in the next layer. Put e in S and let $v \leftarrow u$. If $v \neq t$ then go to Step (3).

(5) The edges on S form an augmenting path: $s \overset{e_1}{—} v_1 \overset{e_2}{—} v_2 \overset{e_3}{—} \cdots v_{l-1} \overset{e_l}{—} t$. Perform the following operations:

(5.1) $\Delta \leftarrow \text{Min}_{1 \leq i \leq l}(\tilde{c}(e_i) - \tilde{f}(e_i))$.

(5.2) For every $1 \leq i \leq l$, $\tilde{f}(e_i) \leftarrow \tilde{f}(e_i) + \Delta$ and if $\tilde{f}(e_i) = \tilde{c}(e_i)$ then mark e_i "blocked".

(5.3) Go to Step (2).

It is easy to see that an edge is declared "blocked" only if no additional augmenting path (of length l) can use it. Thus, when the algorithm halts (in Step (3.1)) the resulting flow $\tilde{f}$ is maximal in $\tilde{N}$. Also, the number of edges scanned, in between two declarations of edge blocking, is at most l, and $l \leq |V| - 1$. Since the number of edges in $\tilde{N}$ is at most $|E|$ and since no blocked edge becomes unblocked, the number of edge scannings is bounded by $|V| \cdot |E|$. Thus, the algorithm for finding a maximal flow in $\tilde{N}$ is $O(|V| \cdot |E|)$, yielding abound $O(|V|^2|E|)$ for the whole algorithm.

5.3 NETWORKS WITH UPPER AND LOWER BOUNDS

In the previous sections we have assumed that the flow in the edges is bounded from above but the lower bound on all the edges is zero. The significance of this assumption is that the assignment of $f(e) = 0$, for every edge e, defines a legal flow, and the algorithm for improving the flow can be started without any difficulty.

In this section, in addition to the upper bound, $c(e)$, on the flow through e, we assume that the flow is also bounded from below by $b(e)$. Thus, f must satisfy

$$b(e) \leq f(e) \leq c(e) \tag{5.6}$$

in every edge e. Condition $C2$ remains unchanged.

Thus, our problem of finding a maximum flow is divided into two. First, we want to check whether the given network has legal flows, and if the answer is positive, we want to find one. Second, we want to increase the flow and find a maximum flow.

A simple example of a network which has no legal flow is shown in Figure 5.8. Here next to each edge e we write $b(e), c(e)$.

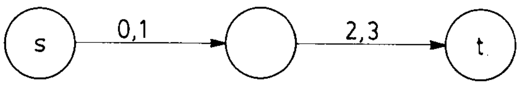


Figure 5.8

The following method for testing whether a given network has a legal flow function is due to Ford and Fulkerson [1]. In case of a positive answer, a flow function is found.

The original network with graph $G(V, E)$ and bounds $b(e)$ and $c(e)$ is modified as follows:

(1) The new set of vertices, $\bar{V}$, is defined by

$$\bar{V} = \{\bar{s}, \bar{t}\} \cup V.$$

$\bar{s}$ and $\bar{t}$ are new vertices, called the auxiliary source and sink, respectively,

(2) For every $v \in V$ construct an edge $v \xrightarrow{e} \bar{t}$ with an upper bound (capacity)

$$\bar{c}(e) = \sum_{e \in \beta(v)} b(e),$$

where $\beta(v)$ is the set of edges which emanate from v in G. The lower bound is zero.

(3) For every $v \in V$ construct an edge $\bar{s} \xrightarrow{e} v$ with an upper bound

$$\bar{c}(e) = \sum_{e \in \alpha(v)} b(e),$$

where $\alpha(v)$ is the set of edges which enter v in G. The lower bound is zero.

(4) The edges of E remain in the new graph but the bounds change: The lower bounds are all zero and the upper bound $\bar{c}(e)$ of $e \in E$ is defined by $\bar{c}(e) = c(e) - b(e)$.

(5) Construct new edges $s \xrightarrow{e} t$ and $t \xrightarrow{e'} s$ with very high upper bounds $\bar{c}(e)$ and $\bar{c}(e')$ $(= \infty)$ and zero lower bounds.

The resulting auxiliary network has a source $\bar{s}$, a sink $\bar{t}$; s and t are regarded now as regular vertices which have to conform to the conservation rule, i.e. condition $C2$.

Let us demonstrate this construction on the graph shown in Fig. 5.9(a). The auxiliary network is shown in Fig. 5.9(b). The upper bounds $\bar{c}(e)$ are shown next to the edges to which they apply.

Now we can use the Ford and Fulkerson or the Dinic (with or without the MPM improvement) algorithms to find a maximum flow in the auxiliary network.

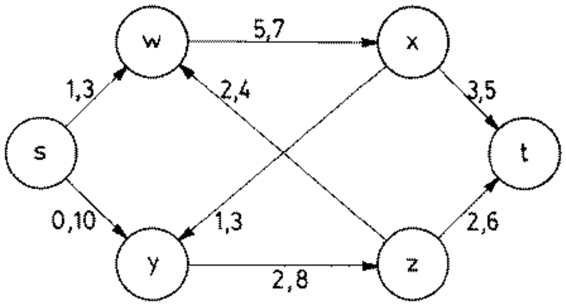
Theorem 5.2 *The original network has a legal flow if and only if the maximum flow of the auxiliary network saturates all the edges which emanate from $\bar{s}$.*

Clearly, if all the edges which emanate from $\bar{s}$ are saturated, then so are all the edges which enter $\bar{t}$ This follows from the fact that each $b(e)$, of the original graph, contributes its value to the capacity of one edge emanating from $\bar{s}$ and to the capacity of one edge entering $\bar{t}$. Thus, the sum of capacities of edges emanating from $\bar{s}$ is equal to the sum of capacities of edges entering $\bar{t}$.

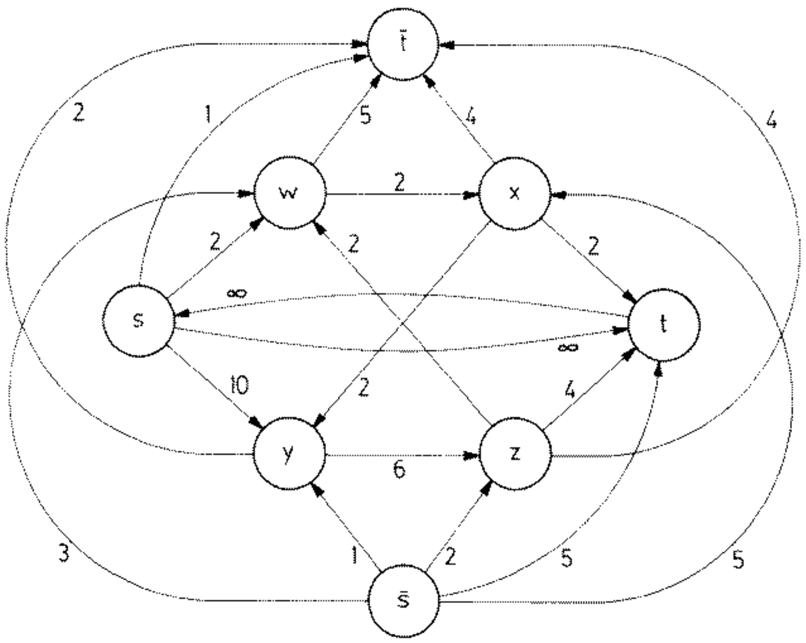
Proof: Assume a maximum flow function $\bar{f}$ of the auxiliary network saturates all the edges which emanate from $\bar{s}$. Define the following flow function, for the original network:

For every $e \in E$

$$f(e) = \bar{f}(e) + b(e). \tag{5.7}$$



(a)



(b)

Figure 5.9

Since

$$0 \leq \bar{f}(e) \leq \bar{c}(e) = c(e) - b(e),$$

we have

$$b(e) \leq f(e) \leq c(e),$$

satisfying (5.6).

Now let $v \in V - \{s, t\}$; $\alpha(v)$ is the set of edges which enter v in the original network and $\beta(v)$ is the set of edges which emanate from v in it. Let $\bar{s} \xrightarrow{\sigma} v$ and $v \xrightarrow{\tau} \bar{t}$ be the edges of the auxiliary network, as constructed in parts (3) and (2). Clearly,

$$\sum_{e \in \alpha(v)} \bar{f}(e) + \bar{f}(\sigma) = \sum_{e \in \beta(v)} \bar{f}(e) + \bar{f}(\tau). \tag{5.8}$$

By the assumption

$$\bar{f}(\sigma) = \bar{c}(\sigma) = \sum_{e \in \alpha(v)} b(e)$$

and

$$\bar{f}(\tau) = \bar{c}(\tau) = \sum_{e \in \beta(v)} b(e).$$

Thus

$$\sum_{e \in \alpha(v)} f(e) = \sum_{e \in \beta(v)} f(e). \tag{5.9}$$

This proves that $C2$ is satisfied too, and f is a legal flow function of the original network.

The steps of this proof are reversible, with minor modifications. If f is a legal flow function of the original network, we can define $\bar{f}$ for the auxiliary network by (5.7). Since f satisfies (5.6), by subtracting $b(e)$, we get that $\bar{f}(e)$ satisfies $C1$ in $e \in E$. Now, f satisfies (5.9) for every $v \in V - \{s, t\}$. Let $\bar{f}(\sigma) = \bar{c}(\sigma)$ and $\bar{f}(\tau) = \bar{c}(\tau)$. Now (5.8) is satisfied and therefore condition $C2$ is held while all the edges which emanate from $\bar{s}$ are saturated. Finally, since the net flow which emanates from s is equal to the net flow which enters t, we can make both of them satisfy $C2$ by flowing through the edges of part (5) of the construction, this amount.

Q.E.D.

Let us demonstrate the technique for establishing whether the network has a legal flow, and finding one in the case the answer is positive, on our example (Fig. 5.9). First, we apply the Dinic algorithm on the auxiliary network and end up with the flow, as in Fig. 5.10(a). The maximum flow saturates all the edges which emanate from $\bar{s}$, and we conclude that the original network has a legal flow. We use (5.7) to define a legal flow in the original network; this is shown in Fig. 5.10(b) (next to each edge e we write $b(e)$, $c(e)$, $f(e)$, in this order).

Once a legal flow has been found, we turn to the question of optimizing it. First, let us consider the question of maximizing the total flow.

One can use the Ford and Fulkerson algorithm except that the backward labeling must be redefined as follows:

A *backward labeling* of vertex v by the edge $u \overset{e}{\leftarrow} v$ is applicable if:

(1) u is labeled and v is not;

(2) $f(e) > b(e)$.

The label that v gets is $'e'$. In this ease we define $\Delta(e) = f(e) - b(e)$.

We start the algorithm with the known legal flow. With this exception, the algorithm is exactly as described in Section 5.1. The proof that when the algorithm terminates the flow is maximum is similar too. We need to redefine the *capacity of a cut* determined by S as follows:

$$c(S) = \sum_{e \in (S;\bar{S})} c(e) - \sum_{e \in (\bar{S};S)} b(e).$$

It is easy to prove that the statement analogous to Lemma 5.2, still holds; for every flow f with total flow F and every S

$$F \leq c(S). \tag{5.10}$$

Now, the set of labeled vertices S, when the algorithm terminates satisfies (5.10) by equality. Thus, the flow is maximum and the indicated cut is minimum.

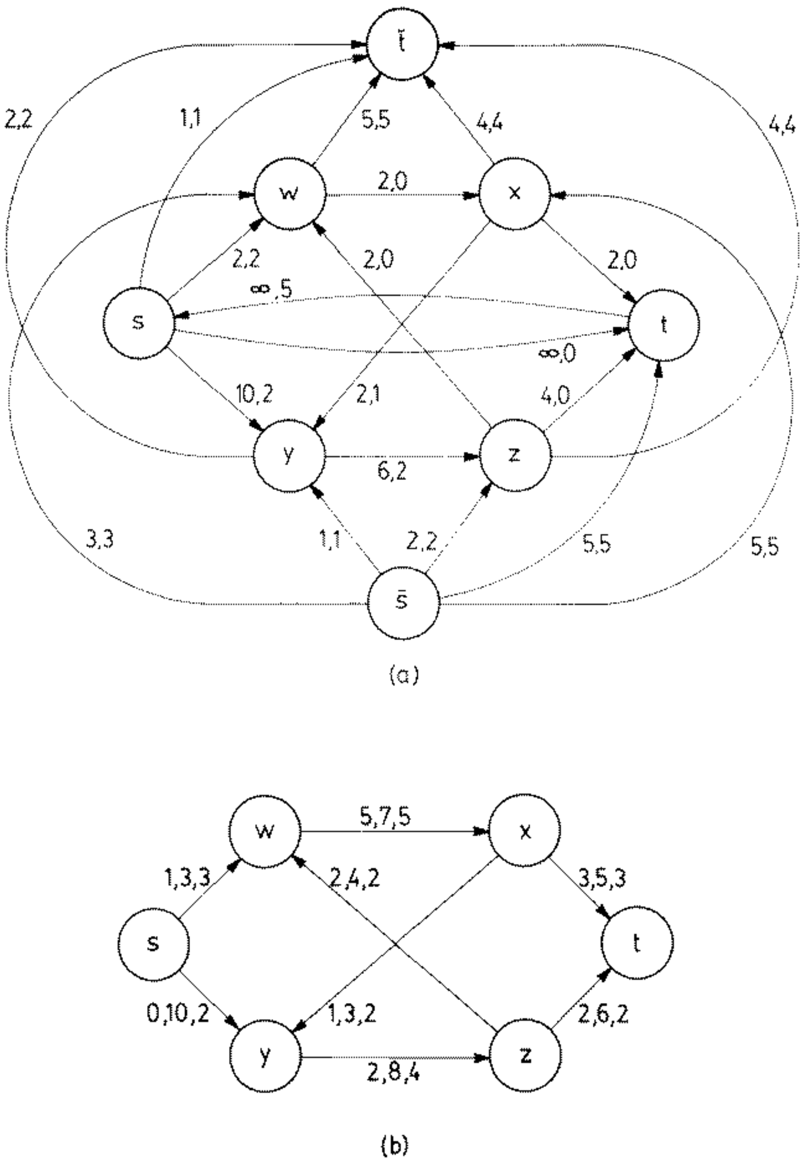


Figure 5.10

The Dinic algorithm can be used too. The only change needed is in the definition of a useful edge, part (2): $u \overset{e}{\leftarrow} v$ and $f(e) > b(e)$, instead of $f(e) > 0$. Also, in the definition of $\tilde{c}(e)$, part (ii): If $u \in V_{i-1}, v \in V_i$ and $u \overset{e}{\leftarrow} v$ then $\tilde{c}(e) = f(e) - b(e)$.

Let us demonstrate the maximizing of the flow on our example, by the Dinic algorithm. The layered network of the first phase for the network, with legal flow, of Fig. 5.10(b) is shown in Fig. 5.11(a). The pair $\tilde{c}(e)$, $\tilde{f}(e)$ is shown next to each edge. The new flow of the original network is shown in Fig. 5.11(b). The layered network of the second phase is shown in Fig. 5.11(c). The set $S = \{s, y\}$ indicates a minimum cut, and the flow is maximum.

In certain applications, what we want is a *minimum flow*, i.e. a legal flow function f for which the total flow F is minimum. Clearly, a minimum flow from s to t is a maximum flow from t to s. Thus, our techniques solve this problem too, by simply exchanging the roles of s and t. By the max-flow min-cut theorem, the max-flow from t to s, $F(t, s)$ is equal to a min-cut

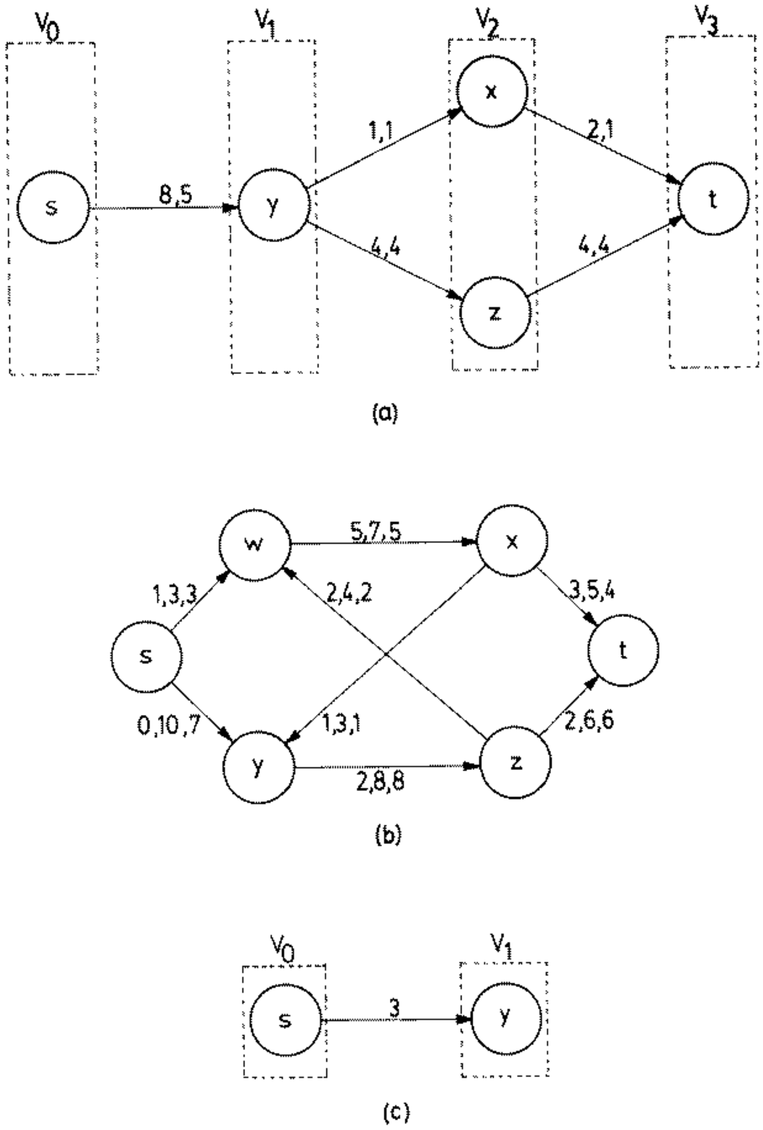


Figure 5.11

from t to s. Therefore, there exists a $T \subset V$, $t \in T$, $s \notin T$ such that

$$F(t,s) = c(T) = \sum_{e \in (T;\bar{T})} c(e) - \sum_{e \in (\bar{T};T)} b(e).$$

Now $F(t,s) = -F(s,t)$, and if $S = \bar{T}$, then

$$F(s,t) = \sum_{e \in (S;\bar{S})} b(e) - \sum_{e \in (\bar{S};S)} c(e).$$

For the min-flow problem we define the capacity of a cut determined by S, $s \in S$, $t \notin S$ by

$$c(S) = \sum_{e \in (S;\bar{S})} b(e) - \sum_{e \in (\bar{S};S)} c(e).$$

Clearly, every S yields a lower bound, $c(S)$, on the flow $F(s,t)$ and the min-flow is equal to the max-cut.

Chapter 6

APPLICATION OF NETWORK FLOW TECHNIQUES

6.4 MAXIMUM MATCHING IN BIPARTITE GRAPHS

A set of edges, M, of a graph $G(V, E)$ with no self-loops, is called a *matching* if every vertex is incident to at most one edge of M. The problem of finding a maximum matching was first solved in polynomial time by Edmonds [12]. The best known result of Even and Kariv [13] is $O(|V|^{2.5})$. These algorithms are too complicated to be included here, and they do not use network flow techniques.

An easier problem is to find a maximum matching in a *bipartite* graph, i.e., a graph in which $V = X \cup Y$, $X \cap Y = \emptyset$ and each edge has one end vertex in X and one in Y. This problem is also known as the marriage problem. We shall present here its solution via network flow and show that its complexity is $O(|V|^{1/2} \cdot |E|)$. This result was first achieved by Hopcroft and Karp [14].

Let us construct a network $N(G)$. Its digraph $\bar{G}(\bar{V}, \bar{E})$ is defined as follows:

$$\bar{V} = \{s, t\} \cup V,$$

$$\bar{E} = \{s \to x \mid x \in X\} \cup \{y \to t \mid y \in Y\} \cup \{x \to y \mid x — y \text{ in } G\}.$$

Let $c(s \to x) = c(y \to t) = 1$ for every $x \in X$ and $y \in Y$. For every edge $x \xrightarrow{e} y$ let $c(e) = \infty$. (This infinite capacity is defined in order to simplify our proof of Theorem 6.12. Actually, since there is only one edge entering x, with unit capacity, the flow in $x \to y$ is bounded by 1.) The source is s and the sink is t. For example consider the bipartite graph G shown in Fig. 6.2(a). Its corresponding network is shown in Fig. 6.2(b).

Theorem 6.11 *The number of edges in a maximum matching of a bipartite graph G is equal to the maximum flow, F, in its corresponding network, $N(G)$.*

Proof: Let M be a maximum matching. For each edge $x \to y$ of M, use the directed path $s \to x \to y \to t$ to flow one unit from s to t. Clearly, all these paths are vertex disjoint. Thus, $F \geq |M|$.

Let f be a flow function of $N(G)$ which is integral. (There is no loss of generality here, since we saw, in Chapter 5, that every network with integral capacities has a maximum integral flow.) All the directed paths connecting s and t are of the form $s \to x \to y \to t$. If such a path is used to flow (one unit) from s to t then no other edge $x \to y'$ or $x' \to y$ can carry flow, since there

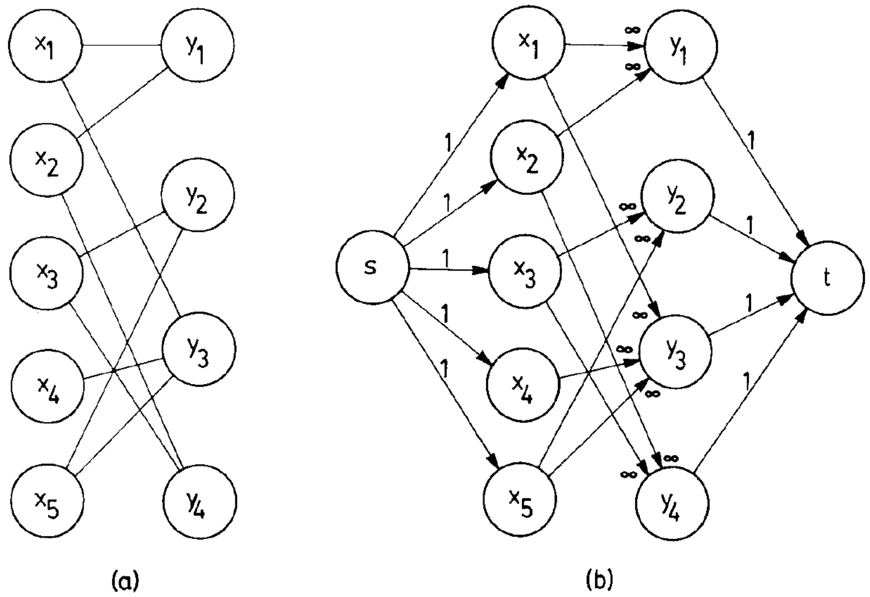


Figure 6.2

is only one edge $s \to x$ and its capacity is one, and the same is true for $y \to t$. Thus, the set of edges $x \to y$, for which $f(x \to y) = 1$, indicates a matching in G. Thus, $|M| \geq F$.

Q.E.D.

The proof indicates how the network flow solution can yield a maximum matching. For our example, a maximum flow, found by Dinic's algorithm is shown in Fig. 6.3(a) and its corresponding matching is shown in Fig. 6.3(b).

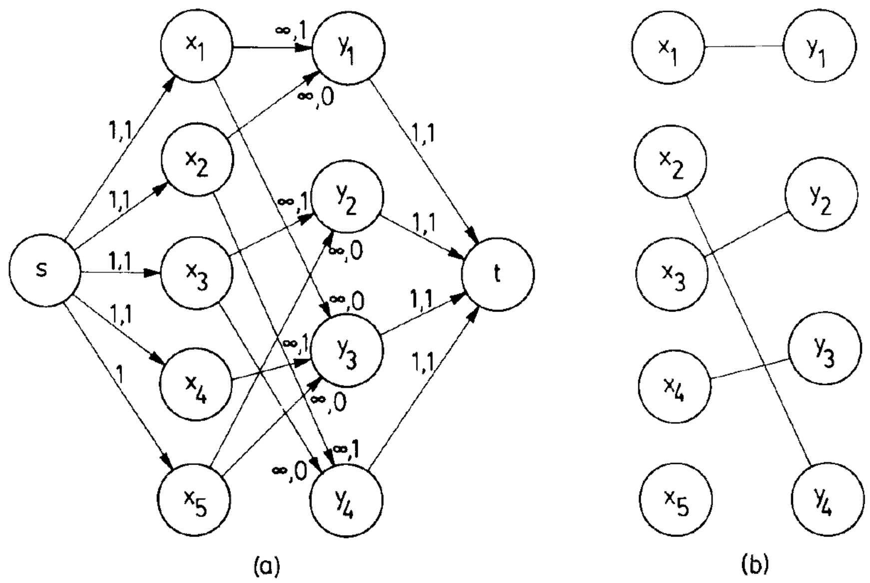


Figure 6.3

The algorithm, of the proof, is $O(|V|^{1/2} \cdot |E|)$, by Theorem 6.3, since the network is, clearly, of type 2.

Next, let us show that one can also use the max-flow min-cut theorem to prove a theorem of Hall [15]. For every $A \subset X$, let $\Gamma(A)$ denote the set of vertices (all in Y) which are connected by an edge to a vertex of A. A matching, M, is called *complete* if $|M| = |X|$.

Theorem 6.12 *A bipartite graph G has a complete matching if and only if for every $A \subset X$, $|\Gamma(A)| \geq |A|$.*

Proof: Clearly, if G has a complete matching M, then each x has a unique "mate" in Y. Thus, for every $A \subset X$, $|\Gamma(A)| \geq |A|$.

Assume now that G does not have a complete matching. Let S be the set of labeled vertices (in the Ford and Fulkerson algorithm, or Dinic's algorithm) upon termination. Clearly, the maximum total flow is equal to $|M|$, but $|M| < |X|$. Let $A = X \cap S$. Since all the edges of the type $x \to y$ are of infinite capacity, $\Gamma(A) \subset S$. Also, no vertex of $Y - \Gamma(A)$ is labeled, since there is no edge connecting it to a labeled vertex. We have

$$(S; \bar{S}) = (\{s\}; X - A) \cup (\Gamma(A); \{t\}).$$

Since $|(S; \bar{S})| = |M| < |X|$, we get

$$|X - A| + |\Gamma(A)| < |X|,$$

which implies $|\Gamma(A)| < |A|$.

Q.E.D.

6.5 TWO PROBLEMS ON PERT DIGRAPHS

A *PERT* digraph* is a finite digraph $G(V, E)$ with the following properties:

(i) There is a vertex s, called the *start vertex*, and a vertex $t (\neq s)$, called the *termination vertex*.

(ii) G has no directed circuits.

(iii) Every vertex $v \in V - \{s, t\}$ is on some directed path from s to t.

A PERT digraph has the following interpretation. Every edge represents a process. All the processes which are represented by edges of $\beta(s)$, can be started right away. For every vertex v, the processes represented by edges of $\beta(v)$ can be started when all the processes represented by edges of $\alpha(v)$ are completed.

Our first problem deals with the question of how soon can the whole project be completed; i.e., what is the shortest time, from the moment the processes represented by $\beta(s)$ are started, until all the processes represented by $\alpha(t)$ are completed. We assume that the resources for running the processes are unlimited. For this problem to be well defined let us assume that each $e \in E$ has an assigned *length* $l(e)$, which specifies the time it takes to execute the process represented by e. The minimum completion time can be found by the following algorithm:

(1) Assign s the label 0 ($\lambda(s) \leftarrow 0$). All other vertices are "unlabeled".

(2) Find a vertex, v, such that v is unlabeled and all edges of $\alpha(v)$ emanate from labeled vertices. Assign

$$\lambda(v) \leftarrow \operatorname*{Max}_{u \xrightarrow{e} v} \{\lambda(u) + l(e)\}. \tag{6.4}$$

*Program Evaluation and Review Technique

(3) If $v = t$, halt; $\lambda(t)$ is the minimum completion time. Otherwise, go to Step (2).

In Step (2), the existence of a vertex v, such that all the edges of $\alpha(v)$ emanate from labeled vertices is guaranteed by Condition (ii) and (iii): If no unlabeled vertex satisfies the condition then for every unlabeled vertex, v, there is an incoming edge which emanates from another unlabeled vertex. By repeatedly tracing back these edges, one finds a directed circuit. Thus, if no such vertex is found then we conclude that either (ii) or (iii) does not hold.

It is easy to prove, by induction on the order of labeling, that $\lambda(v)$ is the minimum time in which all processes, represented by the edges of $\alpha(v)$, can be completed.

The time complexity of the algorithm can be kept down to $O(|E|)$ as follows: For each vertex, v, we keep count of its incoming edges from unlabeled vertices; this count is initially set to $d_{in}(v)$; each time a vertex, u, gets labeled we use the list $\beta(u)$ to decrease the count for all v such that $u \to v$, accordingly; once the count of a vertex v reaches 0, it enters a queue of vertices to be labeled.

Once the algorithm terminates, by going back from t to s, via the edge which determined the label of the vertex, we can trace a longest path from s to t. Such a path is called *critical*.* Clearly, there may be more than one critical path. If one wants to shorten the completion time, $\lambda(t)$, then on each critical path at least one edge length must be shortened.

Next, we shall consider another problem concerning PERT digraphs, in which there is no reference to edge lengths. Assume that each of the processes, represented by the edges, uses one processor for its execution. The question is: How many processors do we need in order to be sure that no execution will ever be delayed because of shortage of processors? We want to avoid such a delay without relying on the values of $l(e)$'s either because they are unknown or because they vary from time to time.

Let us solve a minimum flow problem in the network whose digraph is G, source s, sink t, lower bound $b(e) = 1$ for all $e \in E$ and no upper bound (i.e. $c(e) = \infty$ for all $e \in E$). Condition (iii) assures the existence of a legal flow (see Problem 5).

For example, consider the PERT digraph of Fig. 6.4. The minimum flow (which in this case is unique) is shown in Figure 6.5(a), where a maximum cut is shown too.

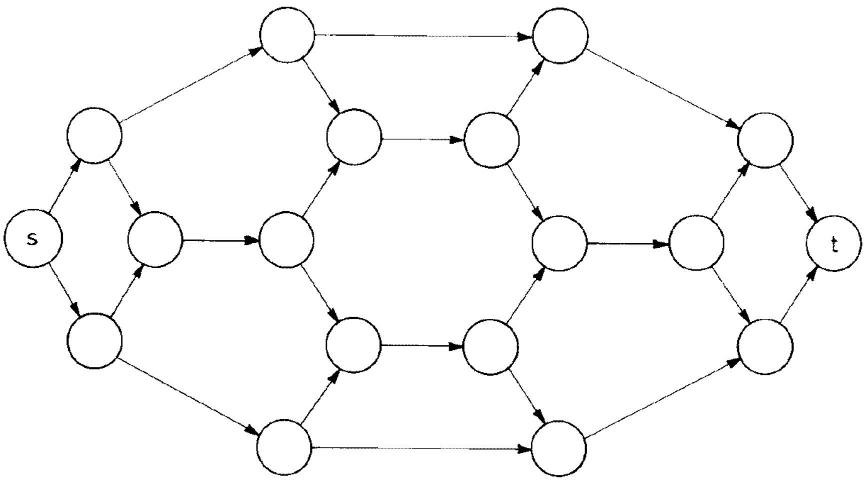
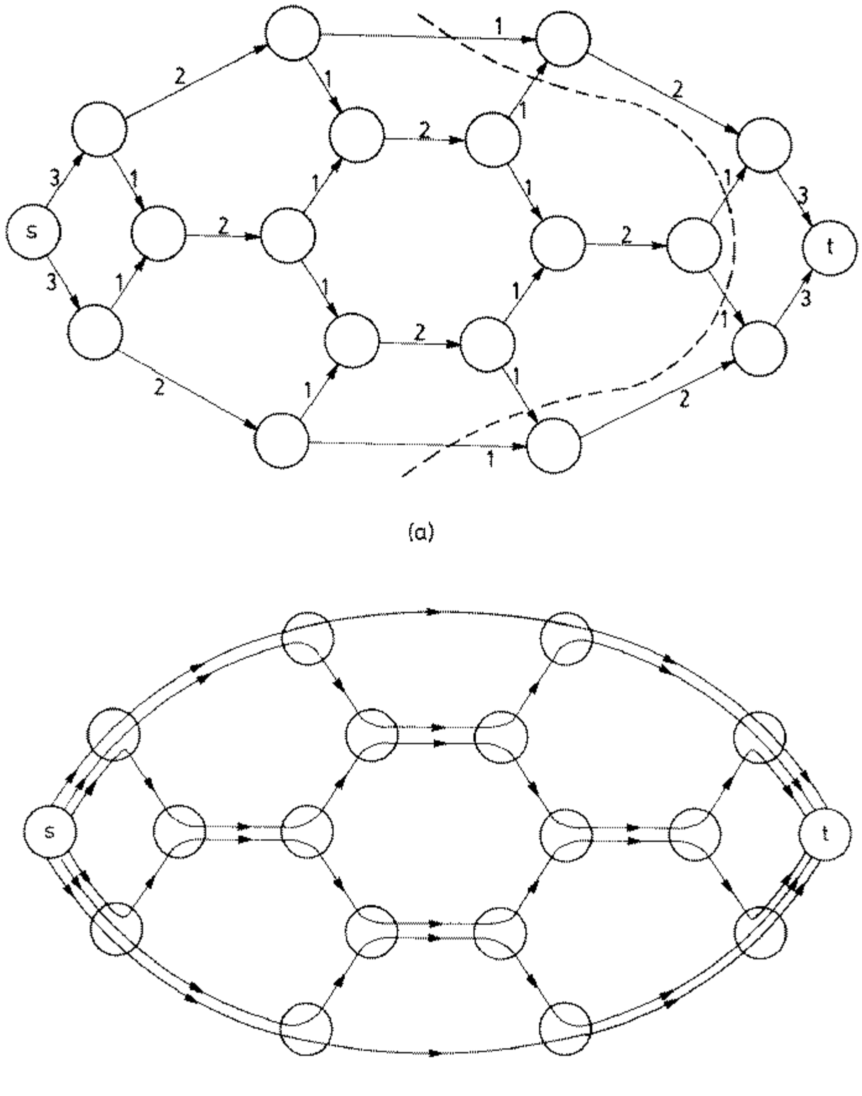


Figure 6.4

A set of edges is called *concurrent* if for no two edges in the set there is a directed path which passes through both. Now, let T be the set of vertices which are labeled in the last attempt

*The whole process is sometimes called the Critical Path Method (CPM).



(a)

(b)

Figure 6.5

to find an augmenting path from t to s. Clearly, $t \in T$ and $s \notin T$. The set of edges $(\bar{T};T)$ is a maximum cut; there are no edges in $(T;\bar{T})$, for there is no upper bound on the flow in the edges, and any such edge would enable to continue the labeling of vertices. Thus, the set $(\bar{T};T)$ is concurrent.

If S is a set of concurrent edges then the number of processors required is at least $|S|$. This can be seen by assigning the edges of S a very large length, and all the others a short length. Since no directed path leads from one edge of S to another, they all will be operative simultaneously. This implies that the number of processors required is at least $|(\bar{T};T)|$.

However, the flow can be decomposed into F directed paths from s to t, where F is the minimum total flow, such that every edge is on at least one such path (since $f(e) \geq 1$ for every $e \in E$). This is demonstrated for our example in Fig. 6.5(b). We can, now, assign to each processor all the edges of one such path. Each such processor executes the processes, represented by the edges of the path in the order in which they appear on the path. If one process is assigned to more than one processor, then one of them executes while the others are idle. It follows that whenever a process which corresponds to $u \to v$, is executable (since all the processes which correspond to $\alpha(u)$ have been executed), the processor to which this process is assigned is available for its execution. Thus, F processors are sufficient for our purpose.

Since $F = |(\bar{T}; T)|$, by the min-flow max-cut theorem, the number of processors thus assigned is minimum.

The complexity of this procedure is as follows. We can find a legal initial flow in time $O(|V| \cdot |E|)$, by tracing for each edge a directed path from s to t via this edge, and flow through it one unit. This path is found by starting form the edge, and going forward and backward from it until s and t are reached. Next, we solve a maximum flow problem, from t to s, by the algorithm of Dinic, using MPM, in time $O(|V|^3)$. Thus, the whole procedure is of complexity $O(|V|^3)$, if $|E| \leq |V|^2$.