6 Flows

What need you flow so fast?
ANONYMOUS

In this chapter, we study *flows* in networks: How much can be transported in a network from a *source* s to a *sink* t if the *capacities* of the connections are given? Such a network might model a system of pipelines, a water supply system, or a system of roads. With its many applications, the theory of flows is one of the most important parts of combinatorial optimization. In Chapter 7 we will encounter several applications of the theory of flows within combinatorics, and flows and related notions will appear again and again throughout the book. The once standard reference, *Flows in Networks* by Ford and Fulkerson [FoFu62], is still worth reading; an extensive, more recent treatment is provided in [AhMO93].

6.1 The theorems of Ford and Fulkerson

In this chapter, we study networks of the following special kind. Let $G = (V, E)$ be a digraph, and let $c\colon E \to \mathbb{R}_0^+$ be a mapping; the value $c(e)$ will be called the *capacity* of the edge e. Moreover, let s and t be two special vertices of G such that t is accessible from s.[1] Then $N = (G, c, s, t)$ is called a *flow network* with *source* s and *sink* t. An *admissible flow* or, for short, a *flow* on N is a mapping $f\colon E \to \mathbb{R}_0^+$ satisfying the following two conditions:

(F1) $0 \le f(e) \le c(e)$ for each edge e;

(F2) $\sum\limits_{e^+=v} f(e) = \sum\limits_{e^-=v} f(e)$ for each vertex $v \ne s, t$, where e^- and e^+ denote the start and end vertex of e, respectively.

Thus the *feasibility condition* (F1) requires that each edge carries a nonnegative amount of flow which may not exceed the capacity of the edge, and

[1]Some authors require in addition $d_{\text{in}}(s) = d_{\text{out}}(t) = 0$. We do not need this condition here; it would also be inconvenient for our investigation of symmetric networks and the network synthesis problem in Chapter 12.

the *flow conservation condition* (F2) means that flows are preserved: at each vertex, except for the source and the sink, the amount that flows in also flows out. It is intuitively clear that the total flow coming out of s should be the same as the total flow going into t; let us provide a formal proof.

Lemma 6.1.1. *Let $N = (G, c, s, t)$ be a flow network with flow f. Then*

$$\sum_{e^-=s} f(e) - \sum_{e^+=s} f(e) \;=\; \sum_{e^+=t} f(e) - \sum_{e^-=t} f(e). \tag{6.1}$$

Proof. Trivially,

$$\begin{aligned}\sum_{e^-=s} f(e) + \sum_{e^-=t} f(e) + \sum_{v\neq s,t}\sum_{e^-=v} f(e) \;&=\; \sum_{e} f(e) \;= \\ &= \sum_{e^+=s} f(e) + \sum_{e^+=t} f(e) + \sum_{v\neq s,t}\sum_{e^+=v} f(e).\end{aligned}$$

Now the assertion follows immediately from (F2). □

The quantity in equation (6.1) is called the *value* of f; it is denoted by $w(f)$. A flow f is said to be *maximal* if $w(f) \geq w(f')$ holds for every flow f' on N. The main problem studied in the theory of flows is the determination of a maximal flow in a given network. Note that, a priori, it is not entirely obvious that maximal flows always exist; however, we will soon see that this is indeed the case.

Let us first establish an upper bound for the value of an arbitrary flow. We need some definitions. Let $N = (G, c, s, t)$ be a flow network. A *cut* of N is a partition $V = S \mathbin{\dot\cup} T$ of the vertex set V of G into two disjoint sets S and T with $s \in S$ and $t \in T$; thus cuts in flow networks constitute a special case of the cuts of $|G|$ introduced in Section 4.3. The *capacity* of a cut (S, T) is defined as

$$c(S,T) \;=\; \sum_{e^-\in S, e^+\in T} c(e);$$

thus it is just the sum of the capacities of all those edges e in the corresponding cocycle $E(S, T)$ which are oriented from S to T. The cut (S, T) is called *minimal* if $c(S, T) \leq c(S', T')$ holds for every cut (S', T'). The following lemma shows that the capacity of a minimal cut gives the desired upper bound on the value of a flow.

Lemma 6.1.2. *Let $N = (G, c, s, t)$ be a flow network, (S, T) a cut, and f a flow. Then*

$$w(f) \;=\; \sum_{e^-\in S, e^+\in T} f(e) - \sum_{e^+\in S, e^-\in T} f(e). \tag{6.2}$$

In particular, $w(f) \leq c(S, T)$.

Proof. Summing equation (F2) over all $v \in S$ gives

$$\begin{aligned} w(f) &= \sum_{v\in S}\left(\sum_{e^-=v} f(e) - \sum_{e^+=v} f(e)\right) \\ &= \sum_{e^-\in S, e^+\in S} f(e) - \sum_{e^+\in S, e^-\in S} f(e) + \sum_{e^-\in S, e^+\in T} f(e) - \sum_{e^+\in S, e^-\in T} f(e). \end{aligned}$$

The first two terms add to 0. Now note $f(e) \le c(e)$ for all edges e with $e^- \in S$ and $e^+ \in T$, and $f(e) \ge 0$ for all edges e with $e^+ \in S$ and $e^- \in T$. □

The main result of this section states that the maximal value of a flow always equals the minimal capacity of a cut. But first we characterize the maximal flows. We need a further definition. Let f be a flow in the network $N = (G, c, s, t)$. A path W from s to t is called an *augmenting path* with respect to f if $f(e) < c(e)$ holds for every forward edge $e \in W$, whereas $f(e) > 0$ for every backward edge $e \in W$. The following three fundamental theorems are due to Ford and Fulkerson [FoFu56].

Theorem 6.1.3 (augmenting path theorem). *A flow f on a flow network $N = (G, c, s, t)$ is maximal if and only if there are no augmenting paths with respect to f.*

Proof. First let f be a maximal flow. Suppose there is an augmenting path W. Let d be the minimum of all values $c(e) - f(e)$ (taken over all forward edges e in W) and all values $f(e)$ (taken over the backward edges in W). Then $d > 0$, by definition of an augmenting path. Now we define a mapping $f' : E \to \mathbb{R}_0^+$ as follows:

$$f'(e) = \begin{cases} f(e) + d & \text{if } e \text{ is a forward edge in } W, \\ f(e) - d & \text{if } e \text{ is a backward edge in } W, \\ f(e) & \text{otherwise.} \end{cases}$$

It is easily checked that f' is a flow on N with value $w(f') = w(f) + d > w(f)$, contradicting the maximality of f.

Conversely, suppose there are no augmenting paths in N with respect to f. Let S be the set of all vertices v such that there exists an augmenting path from s to v (including s itself), and put $T = V \setminus S$. By hypothesis, (S, T) is a cut of N. Thus each edge e with $e^- \in S$ and $e^+ \in T$ must be *saturated*: $f(e) = c(e)$; and each edge e with $e^- \in T$ and $e^+ \in S$ has to be *void*: $f(e) = 0$. Then Lemma 6.1.2 gives $w(f) = c(S, T)$, so that f is maximal. □

We note that the preceding proof contains a useful characterization of maximal flows:

Corollary 6.1.4. *A flow f on a flow network $N = (G, c, s, t)$ is maximal if and only if the set S of all vertices accessible from s on an augmenting path with respect to f is a proper subset of V. In this case, $w(f) = c(S, T)$, where $T = V \setminus S$.* □

Theorem 6.1.5 (integral flow theorem). *Let $N = (G, c, s, t)$ be a flow network where all capacities $c(e)$ are integers. Then there is a maximal flow on N such that all values $f(e)$ are integral.*

Proof. By setting $f_0(e) = 0$ for all e, we obtain an integral flow f_0 on N with value 0. If this trivial flow is not maximal, then there exists an augmenting path with respect to f_0. In that case the number d appearing in the proof of Theorem 6.1.3 is a positive integer, and we can construct an integral flow f_1 of value d as in the proof of Theorem 6.1.3. We continue in the same manner. As the value of the flow is increased in each step by a positive integer and as the capacity of any cut is an upper bound on the value of the flow (by Lemma 6.1.2), after a finite number of steps we reach an integral flow f for which no augmenting path exists. By Theorem 6.1.3, this flow f is maximal. □

Theorem 6.1.6 (max-flow min-cut theorem). *The maximal value of a flow on a flow network N is equal to the minimal capacity of a cut for N.*

Proof. If all capacities are integers, the assertion follows from Theorem 6.1.5 and Corollary 6.1.4. The case where all capacities are rational can be reduced to the integral case by multiplying all numbers by their common denominator. Then real-valued capacities may be treated using a continuity argument, since the set of flows is a compact subset of $\mathbb{R}^{|E|}$ and since $w(f)$ is a continuous function of f. A different, constructive proof for the real case is provided by the theorem of Edmonds and Karp [EdKa72], which we will treat in the next section. □

Theorem 6.1.6 was obtained in [FoFu56] and, independently, in [ElFS56]. In practice, real capacities do not occur, as a computer can only represent (a finite number of) rational numbers anyway. From now on, we mostly restrict ourselves to integral flows. Sometimes we also allow networks on directed multigraphs; this is not really more general, because parallel edges can be replaced by a single edge whose capacity is the sum of the corresponding capacities of the parallel edges.

The remainder of this chapter deals with several algorithms for finding a maximal flow. The proof of Theorem 6.1.5 suggests the following rough outline of such an algorithm:

```
(1) $f(e) \leftarrow 0$ for all edges $e$;
(2) while there exists an augmenting path with respect to $f$ do
(3)         let $W = (e_1, \dots, e_n)$ be an augmenting path from $s$ to $t$;
(4)         $d \leftarrow \min\big(\{c(e_i) - f(e_i) : e_i$ is a forward edge in $W\}$
                          $\cup\ \{f(e_i) : e_i$ is a backward edge in $W\}\big)$;
(5)         $f(e_i) \leftarrow f(e_i) + d$ for each forward edge $e_i$;
(6)         $f(e_i) \leftarrow f(e_i) - d$ for each backward edge $e_i$;
(7) od
```

Of course, we still have to specify a technique for finding augmenting paths. We could use a modified breadth first search (BFS), where edges may be used regardless of their orientation as long as they satisfy the necessary condition $f(e) < c(e)$ or $f(e) > 0$. Note that we not only have to decide whether t is accessible from s by an augmenting path, but we also need to find the value for d and change the values $f(e)$ accordingly. In view of these additional requirements, it makes more sense to apply a labelling technique; moreover, this will also allow us to find a minimal cut.

Algorithm 6.1.7 (labelling algorithm of Ford and Fulkerson). Let $N = (G, c, s, t)$ be a flow network.

Procedure FORDFULK($N; f, S, T$)

```
(1) for $e \in E$ do $f(e) \leftarrow 0$ od;
(2) label $s$ with $(-, \infty)$;
(3) for $v \in V$ do $u(v) \leftarrow$ false; $d(v) \leftarrow \infty$ od;
(4) repeat
(5)         choose a vertex $v$ which is labelled and satisfies $u(v) =$ false;
(6)         for $e \in \{e \in E : e^- = v\}$ do
(7)             if $w = e^+$ is not labelled and $f(e) < c(e)$ then
(8)             $d(w) \leftarrow \min\{c(e) - f(e), d(v)\}$; label $w$ with $(v, +, d(w))$ fi
(9)         od;
(10)        for $e \in \{e \in E : e^+ = v\}$ do
(11)            if $w = e^-$ is not labelled and $f(e) > 0$ then
(12)            $d(w) \leftarrow \min\{f(e), d(v)\}$; label $w$ with $(v, -, d(w))$ fi
(13)        od;
(14)        $u(v) \leftarrow$ true;
(15)        if $t$ is labelled
(16)        then let $d$ be the last component of the label of $t$;
(17)            $w \leftarrow t$;
(18)            while $w \neq s$ do
(19)                 find the first component $v$ of the label of $w$;
(20)                 if the second component of the label of $w$ is $+$
(21)                 then set $f(e) \leftarrow f(e) + d$ for $e = vw$
(22)                 else set $f(e) \leftarrow f(e) - d$ for $e = wv$
(23)                 fi;
(24)                 $w \leftarrow v$
(25)            od;
(26)            delete all labels except for the label of $s$;
(27)            for $v \in V$ do $d(v) \leftarrow \infty$; $u(v) \leftarrow$ false od
(28)        fi
(29) until $u(v) =$ true for all vertices $v$ which are labelled;
(30) let $S$ be the set of vertices which are labelled and put $T \leftarrow V \setminus S$
```

Using the proofs we gave for Theorems 6.1.3 and 6.1.5, we immediately get the following theorem due to Ford and Fulkerson [FoFu57].

Theorem 6.1.8. *Let N be a network whose capacity function c takes only integral (or rational) values. Then Algorithm 6.1.7 determines a maximal flow f and a minimal cut (S,T), so that $w(f) = c(S,T)$ holds.* □

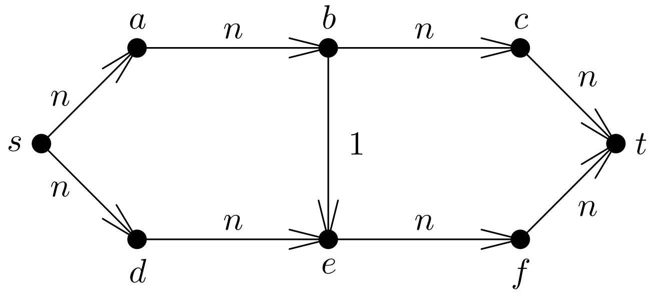

Fig. 6.1. A flow network

Algorithm 6.1.7 may fail for irrational capacities, if the vertex v in step (5) is chosen in an unfortunate way; an example for this can be found in [FoFu62], p. 21. In that example the algorithm not only does not terminate, but it even converges to a value which is only 1/4 of the maximal possible flow value. Moreover, Algorithm 6.1.7 is not polynomial even for integer capacities, because the number of necessary changes of the flow f does not only depend on $|V|$ and $|E|$, but might also depend on c. For example, if we use the paths

$$s — a — b — e — f — t \qquad \text{and} \qquad s — d — e — b — c — t$$

alternately as augmenting paths for the network in Figure 6.1 (which the algorithm will do if vertex v in step (5) is chosen suitably), the value of the flow will only be increased by 1 in each step, so that we need $2n$ iterations. Of course, this can be avoided by choosing the paths appropriately; with

$$s — a — b — c — t \qquad \text{and} \qquad s — d — e — f — t,$$

we need only two iterations. In the next section, we show how the augmenting paths can be chosen efficiently. Then we shall also apply the resulting algorithm to an example and show the computations in detail. We close this section with a few exercises.

Exercise 6.1.9. Let $N = (G, c, s, t)$ be a flow network for which the *capacities* of the vertices are likewise restricted: there is a further mapping $d\colon V \to \mathbb{R}_0^+$, and the flows f have to satisfy the additional restriction

(F3) $\quad \sum\limits_{e^+=v} f(e) \le d(v) \quad$ for $v \ne s, t$.

For instance, we might consider an irrigation network where the vertices are pumping stations with limited capacity. Reduce this problem to a problem for an appropriate ordinary flow network and generalize Theorem 6.1.6 to this situation; see [FoFu62], §1.11.

Exercise 6.1.10. How can the case of several sources and several sinks be treated?

Exercise 6.1.11. Let $N = (G, c, s, t)$ be a flow network, and assume that N admits flows of value $\neq 0$. Show that there exists at least one edge e in N whose removal decreases the value of a maximal flow on N. An edge e is called *most vital* if the removal of e decreases the maximal flow value as much as possible. Is an edge of maximal capacity in a minimal cut necessarily most vital?

Exercise 6.1.12. By Theorem 6.1.5, a flow network with integer capacities always admits an integral maximal flow. Is it true that *every* maximal flow has to be integral?

Exercise 6.1.13. Let f be a flow in a flow network N. The *support* of f is $\operatorname{supp} f = \{e \in E : f(e) \neq 0\}$. A flow f is called *elementary* if its support is a path. The proof of Theorem 6.1.6 and the algorithm of Ford and Fulkerson show that there exists a maximal flow which is the sum of elementary flows. Can every maximal flow can be represented by such a sum?

Exercise 6.1.14. Modify the process of labelling the vertices in Algorithm 6.1.7 in such a way that the augmenting path chosen always has maximal possible capacity (so that the value of the flow is always increased as much as possible). Hint: Use an appropriate variation of the algorithm of Dijkstra.

6.2 The algorithm of Edmonds and Karp

As we have seen in the previous section, the labelling algorithm of Ford and Fulkerson is, in general, not polynomial. We now consider a modification of this algorithm due to Edmonds and Karp [EdKa72] for which we can prove a polynomial complexity, namely $O(|V||E|^2)$. As we will see, it suffices if we always use an augmenting path of shortest length – that is, having as few edges as possible – for increasing the flow. To find such a path, we just make step (5) in Algorithm 6.1.7 more specific: we require that the vertex v with $u(v)$ = false which was labelled first is chosen. Then the labelling process proceeds as for a BFS; compare Algorithm 3.3.1. This principle for selecting the vertex v is also easy to implement: we simply collect all labelled vertices in a queue – that is, some vertex w is appended to the queue when it is labelled in step (8) or (12). This simple modification is enough to prove the following result.

Theorem 6.2.1. *Replace step (5) in Algorithm 6.1.7 as follows:*

(5′) among all vertices with $u(v)$ = false, let v be the vertex which was labelled first.

Then the resulting algorithm has complexity $O(|V||E|^2)$.

Proof. We have already noted that the flow f is always increased using an augmenting path of shortest length, provided that we replace step (5) by (5′). Let f_0 be the flow of value 0 defined in step (1), and let $f_1, f_2, f_3, \dots$ be the sequence of flows constructed subsequently. Denote the shortest length of an augmenting path from s to v with respect to f_k by $x_v(k)$. We begin by proving the inequality

$$x_v(k+1) \geq x_v(k) \quad \text{for all } k \text{ and } v. \tag{6.3}$$

By way of contradiction, suppose that (6.3) is violated for some pair (v, k); we may assume that $x_v(k+1)$ is minimal among the $x_w(k+1)$ for which (6.3) does not hold. Consider the last edge e on a shortest augmenting path from s to v with respect to f_{k+1}. Suppose first that e is a forward edge, so that $e = uv$ for some vertex u; note that this requires $f_{k+1}(e) < c(e)$. Now $x_v(k+1) = x_u(k+1) + 1$, so that $x_u(k+1) \geq x_u(k)$ by our choice of v. Hence $x_v(k+1) \geq x_u(k) + 1$. On the other hand, $f_k(e) = c(e)$, as otherwise $x_v(k) \leq x_u(k) + 1$ and $x_v(k+1) \geq x_v(k)$, contradicting the choice of v. Therefore e was as a backward edge when f_k was changed to f_{k+1}. As we have used an augmenting path of shortest length for this change, we conclude $x_u(k) = x_v(k) + 1$ and hence $x_v(k+1) \geq x_v(k) + 2$, a contradiction. The case where e is a backward edge can be treated in the same manner. Moreover, similar arguments also yield the inequality

$$y_v(k+1) \geq y_v(k) \quad \text{for all } k \text{ and } v, \tag{6.4}$$

where $y_v(k)$ denotes the length of a shortest augmenting path from v to t with respect to f_k.

When increasing the flow, the augmenting path always contains at least one *critical* edge: the flow through this edge is either increased up to its capacity or decreased to 0. Let $e = uv$ be a critical edge in the augmenting path with respect to f_k; this path consists of $x_v(k) + y_v(k) = x_u(k) + y_u(k)$ edges. If e is used the next time in some augmenting path (with respect to f_h, say), it has to be used in the opposite direction: if e was a forward edge for f_k, it has to be a backward edge for f_h, and vice versa.

Suppose that e was a forward edge for f_k. Then $x_v(k) = x_u(k) + 1$ and $x_u(h) = x_v(h) + 1$. By (6.3) and (6.4), $x_v(h) \geq x_v(k)$ and $y_u(h) \geq y_u(k)$. Hence we obtain

$$x_u(h) + y_u(h) = x_v(h) + 1 + y_u(h) \geq x_v(k) + 1 + y_u(k) = x_u(k) + y_u(k) + 2.$$

Thus the augmenting path with respect to f_h is at least two edges longer than the augmenting path with respect to f_k. This also holds for the case where e was a backward edge for f_h; to see this, exchange the roles of u and v in the preceding argument. Trivially, no augmenting path can contain more than $|V| - 1$ edges. Hence each edge can be critical at most $(|V| - 1)/2$ times, and thus the flow can be changed at most $O(|V||E|)$ times. (In particular, this establishes that the algorithm has to terminate even if the capacities are non-rational.) Each iteration – finding an augmenting path and updating the flow – takes only $O(|E|)$ steps, since each edge is treated at most three times: twice during the labelling process and once when the flow is changed. This implies the desired complexity bound of $O(|V||E|^2)$. □

Remark 6.2.2. As the cardinality of E is between $O(|V|)$ and $O(|V|^2)$, the complexity of the algorithm of Edmonds and Karp lies between $O(|V|^3)$ for sparse graphs (hence, in particular, for planar graphs) and $O(|V|^5)$ for dense graphs.

Examples for networks with n vertices and $O(n^2)$ edges for which the algorithm of Edmonds and Karp actually needs $O(n^3)$ iterations may be found in [Zad72, Zad73b]; thus the estimates used in the proof of Theorem 6.2.1 are best possible. Of course, this by no means precludes the existence of more efficient algorithms. One possible approach is to look for algorithms which are not based on the use of augmenting paths; we will see examples for this approach in Sections 6.4 and 6.6 as well as in Chapter 11. Another idea is to combine the iterations in a clever way into larger phases; for instance, it turns out to be useful to consider all augmenting paths of a constant length in one block; see Sections 6.3 and 6.4. Not surprisingly, such techniques are not only better but also more involved.

Example 6.2.3. We use the algorithm of Edmonds and Karp to determine a maximal flow and a minimal cut in the network N given in Figure 6.2. The capacities are given there in parentheses; the numbers without parentheses in the following figures always give the respective values of the flow. We also state the labels which are assigned at the respective stage of the algorithm; when examining the possible labellings coming from some vertex v on forward edges (steps (6) through (9)) and on backward edges (steps (10) through (13)), we consider the adjacent vertices in alphabetical order, so that the course of the algorithm is uniquely determined. The augmenting path used for the construction of the next flow is drawn bold.

We start with the zero flow f_0, that is, $w(f_0) = 0$. The vertices are labelled in the order a, b, f, c, d, t as shown in Figure 6.3; e is not labelled because t is reached before e is considered. Figures 6.3 to 6.12 show how the algorithm works. Note that the last augmenting path uses a backward edge, see Figure 6.11. In Figure 6.12, we have also indicated the minimal cut resulting from the algorithm.

The reader will note that many labels are not changed from one iteration to the next. As all the labels are deleted in step (26) after each change of the flow, this means we do a lot of unnecessary calculations. It is possible to obtain algorithms of better complexity by combining the changes of the flow into bigger *phases*. To do this, a *blocking flow* is constructed in some appropriate auxiliary network. This subject is treated in Sections 6.3 and 6.4.

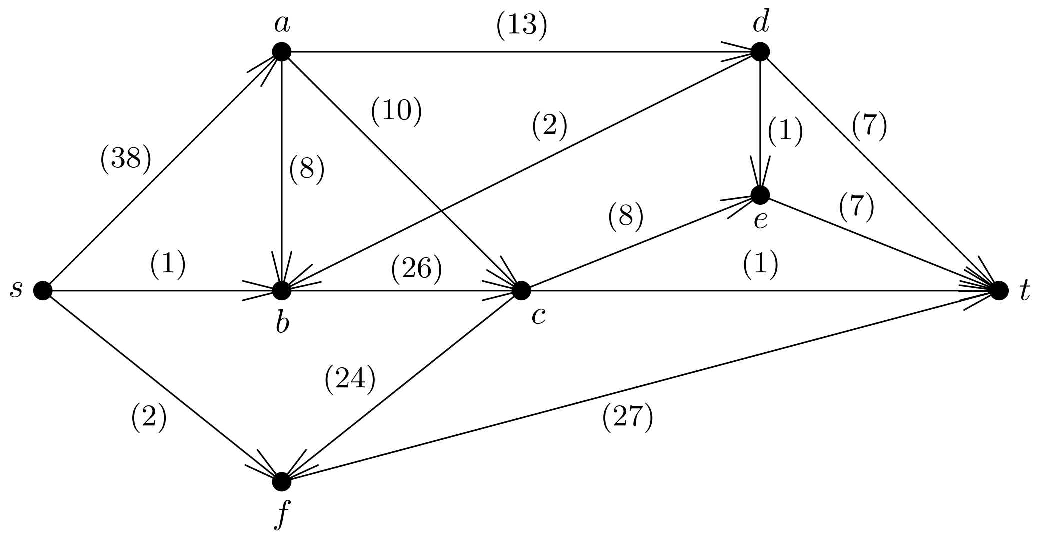

Fig. 6.2. A flow network

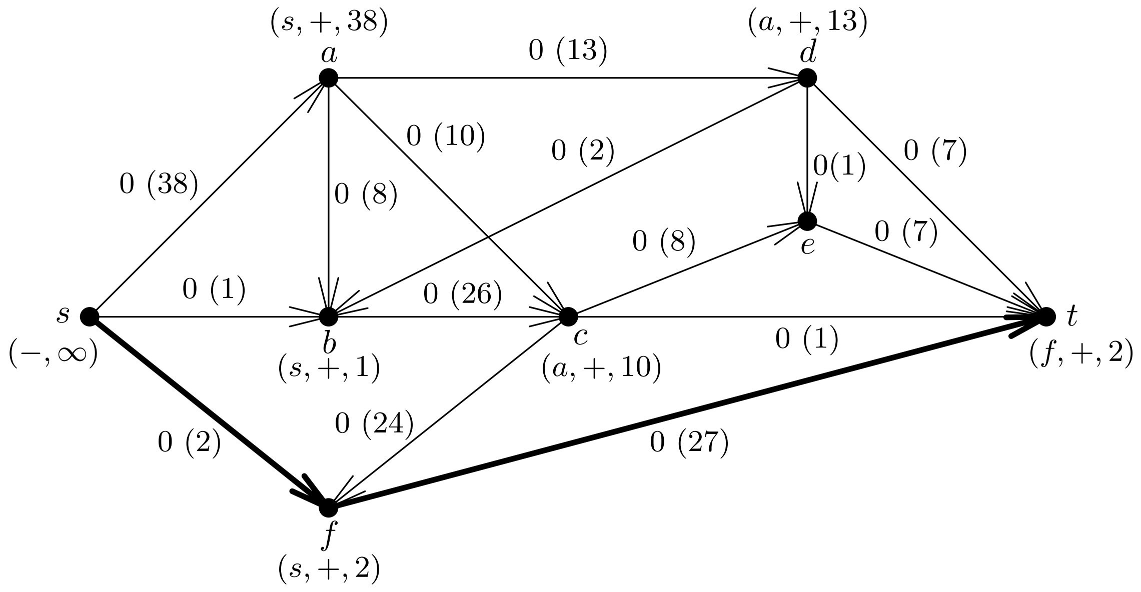

Fig. 6.3. $w(f_0) = 0$

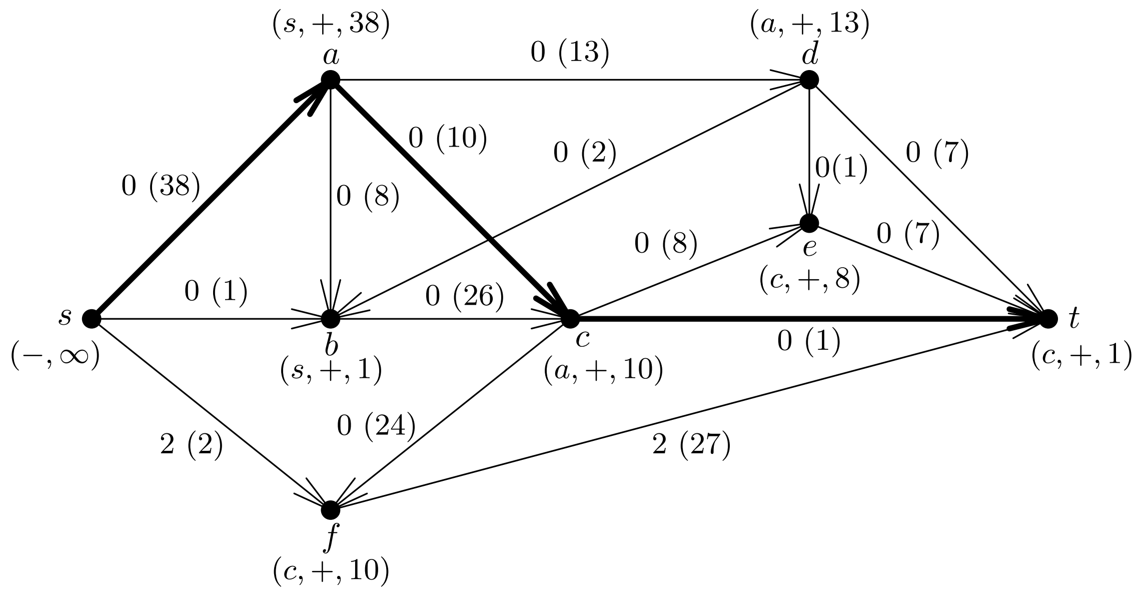

Fig. 6.4. $w(f_1) = 2$

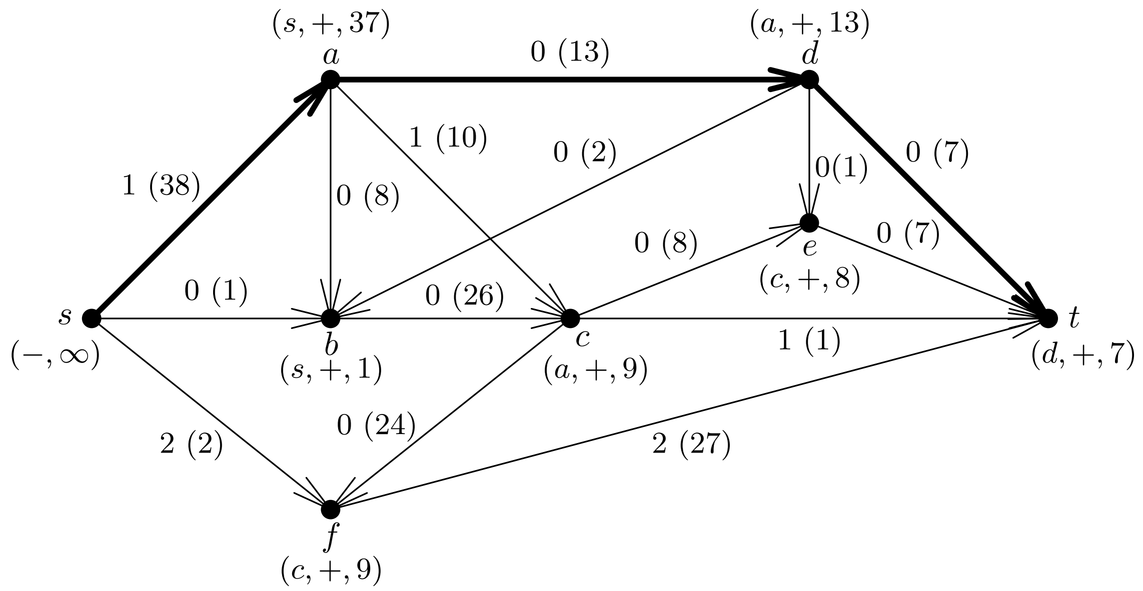

Fig. 6.5. $w(f_2) = 3$

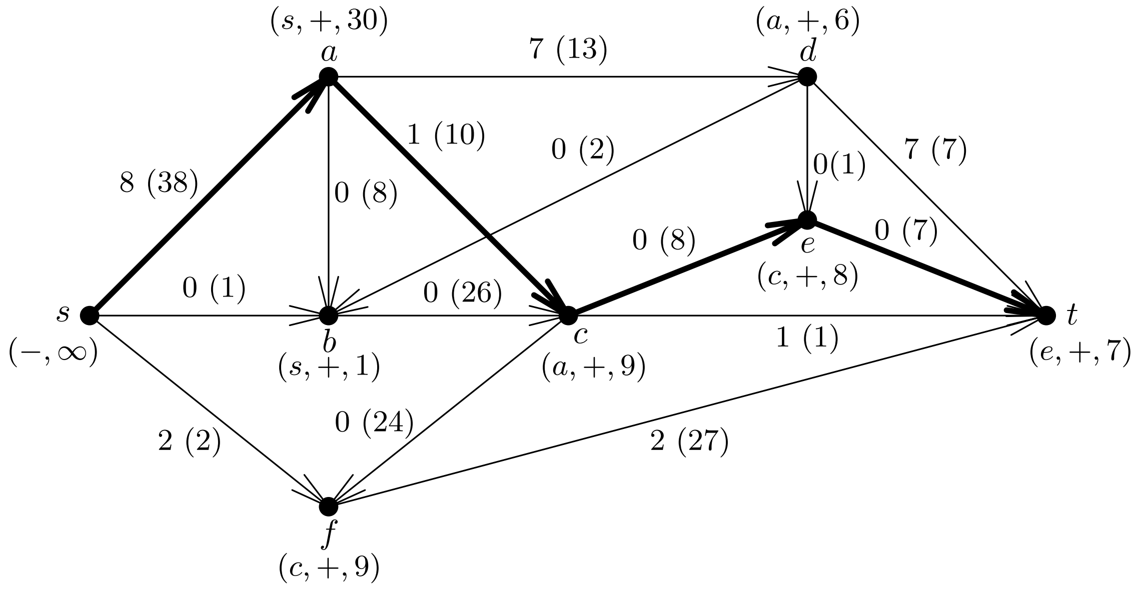

Fig. 6.6. $w(f_3) = 10$

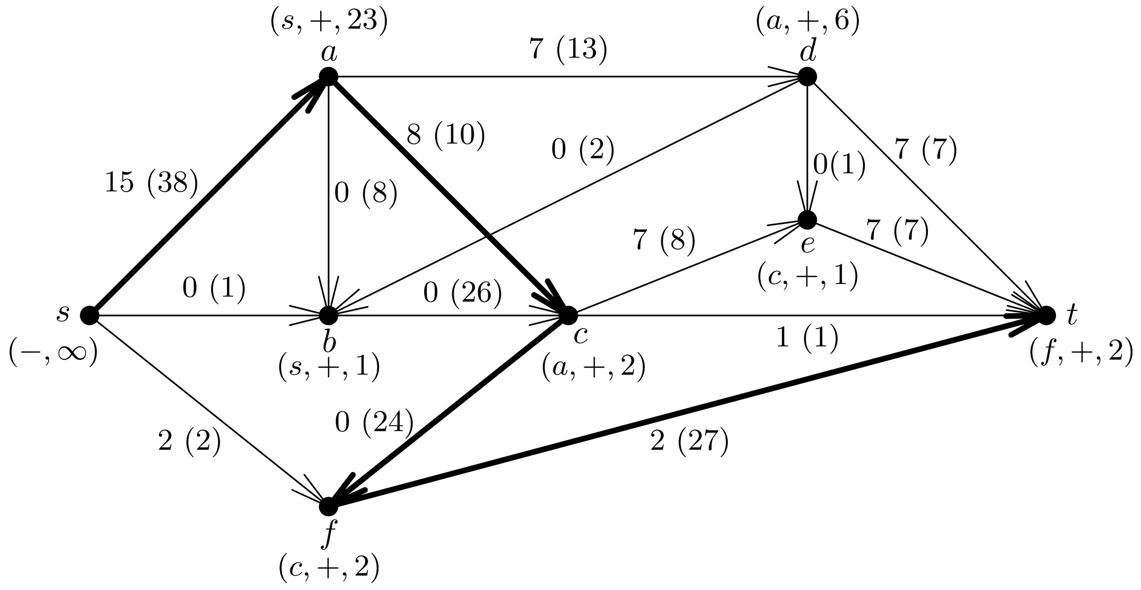

Fig. 6.7. $w(f_4) = 17$

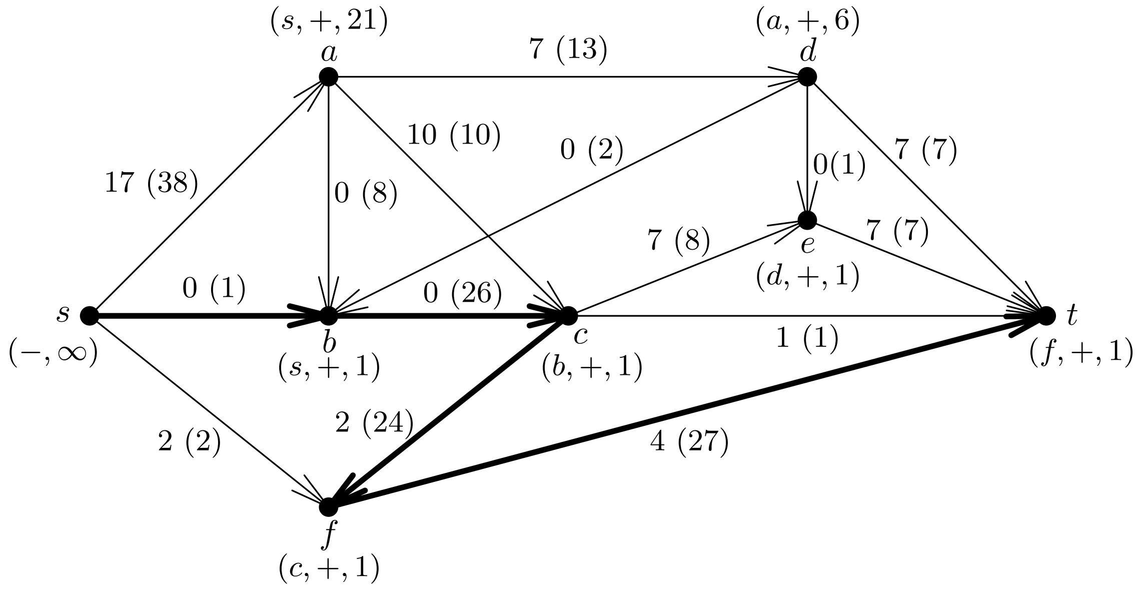

Fig. 6.8. $w(f_5) = 19$

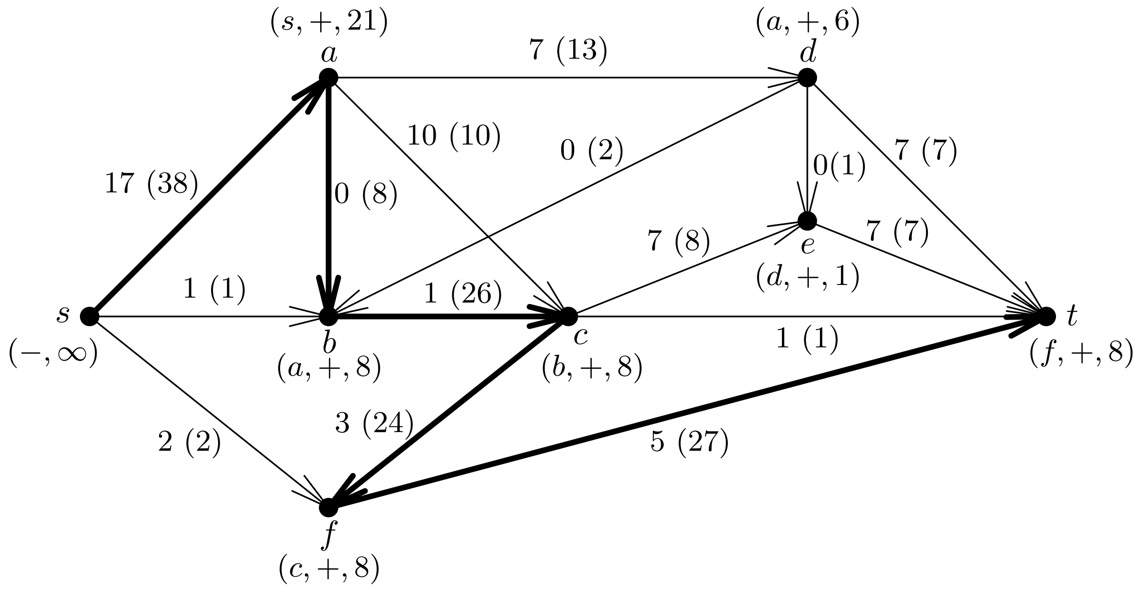

Fig. 6.9. $w(f_6) = 20$

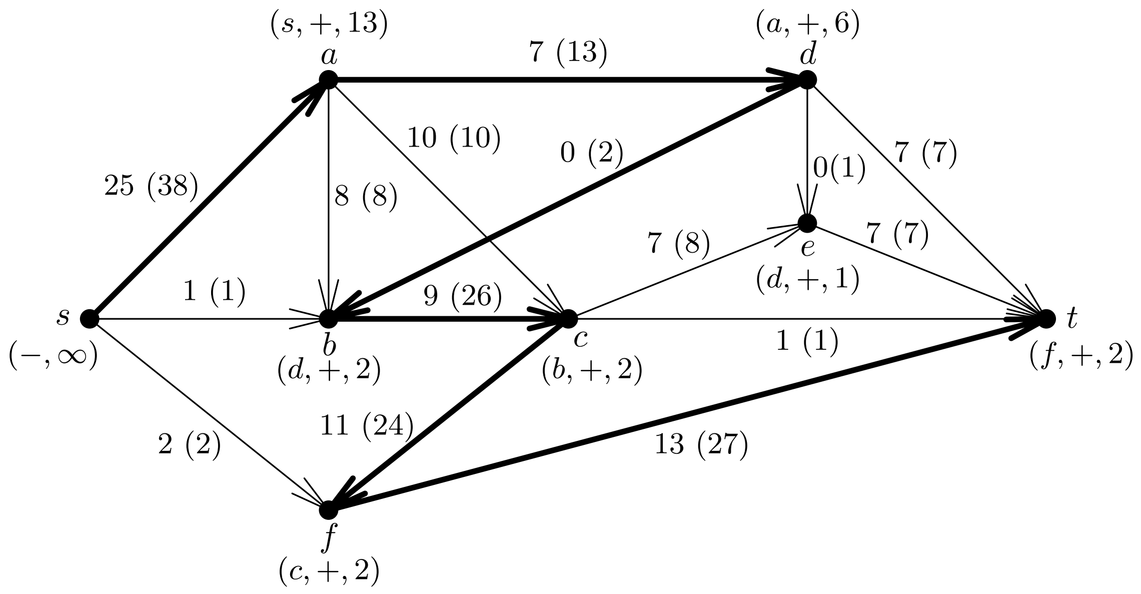

Fig. 6.10. $w(f_7) = 28$

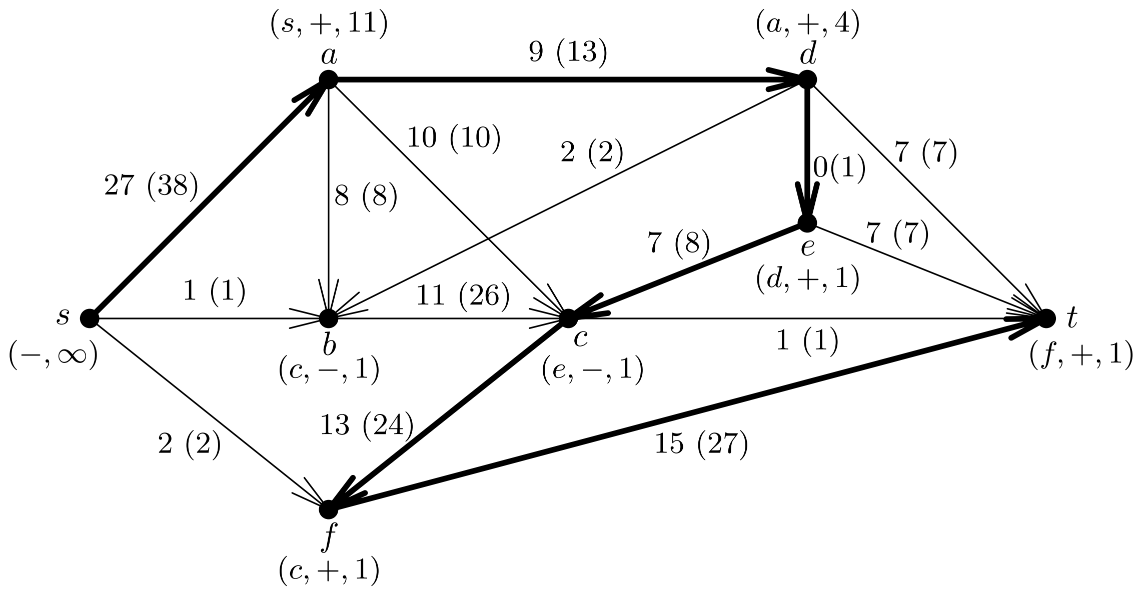

Fig. 6.11. $w(f_8) = 30$

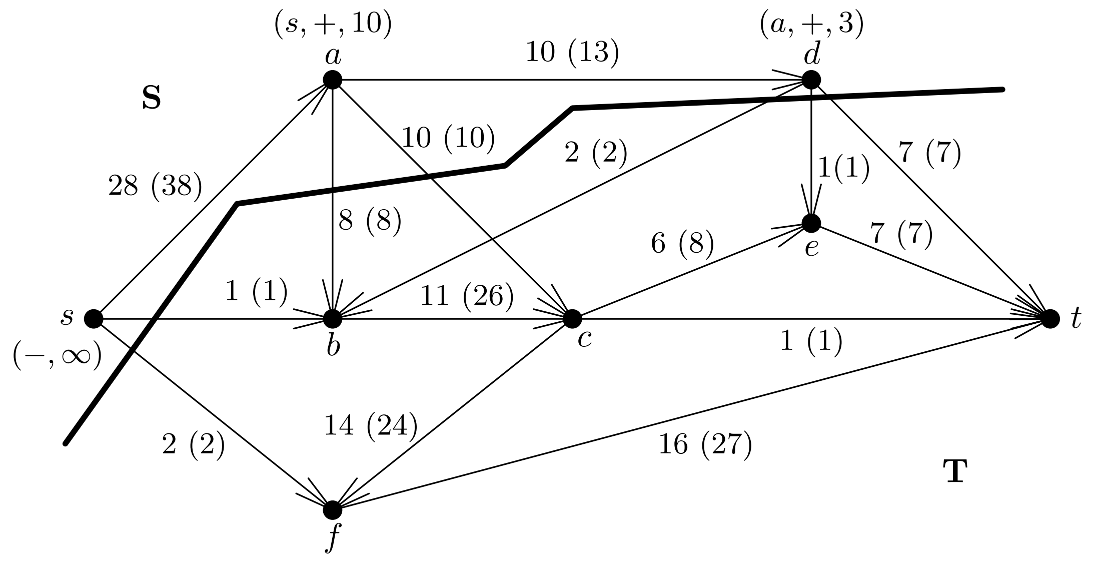

Fig. 6.12. $w(f_9) = 31 = c(S, T)$

It can be shown that it is theoretically possible to obtain a maximal flow in a given network in at most $|E|$ iterations, and that this may even be achieved using augmenting paths which consist of forward edges only; see [Law76], p. 119. However, this result is of no practical interest, because it is not known how one would actually find such paths.

We mention that Edmonds and Karp have also shown that the flow has to be changed at most $O(\log w)$ times, where w is the maximal value of a flow on N, if we always choose an augmenting path of maximal capacity. Even though we do not know w a priori, the number of steps necessary for this method is easy to estimate, as w is obviously bounded by

$$W = \min \{\sum_{e^-=s} c(e), \sum_{e^+=t} c(e)\}.$$

Note that this approach does not yield a polynomial algorithm, since the bound depends also on the capacities. Nevertheless, it can still be better for concrete examples where W is small, as illustrated by the following exercise.

Exercise 6.2.4. Determine a maximal flow for the network of Figure 6.2 by always choosing an augmenting path of maximal capacity.

Exercise 6.2.5. Apply the algorithm of Edmonds and Karp to the network shown in Figure 6.13 (which is taken from [PaSt82]).

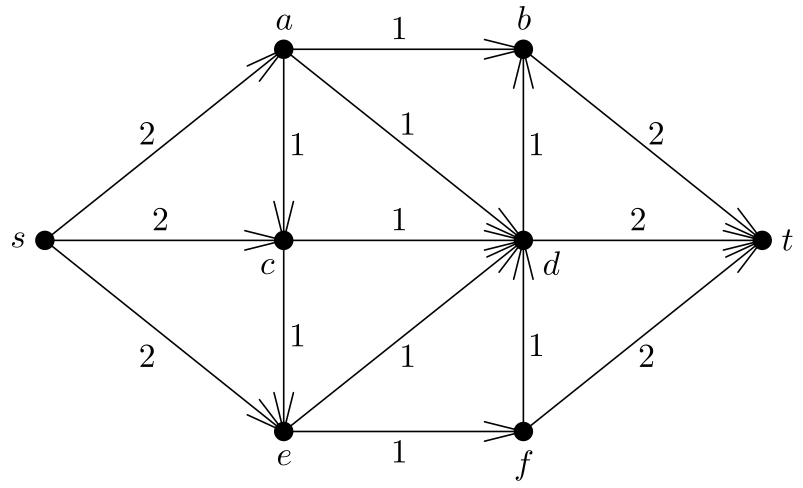

Fig. 6.13. A network

We conclude this section with three exercises showing that it is possible to change several capacities in a given network and find solutions for the corresponding new problem without too much effort, if we know a solution of the original problem.

Exercise 6.2.6. Suppose we have determined a maximal flow for a flow network N using the algorithm of Edmonds and Karp, and realize afterwards that we used an incorrect capacity for some edge e. Discuss how we may use the solution of the original problem to solve the corrected problem.

Exercise 6.2.7. Change the capacity of the edge $e = ac$ in the network of Figure 6.2 to $c(e) = 8$, and then to $c(e) = 12$. How do these modifications change the value of a maximal flow? Give a maximal flow for each of these two cases.

Exercise 6.2.8. Change the network of Figure 6.2 as follows. The capacities of the edges ac and ad are increased to 12 and 16, respectively, and the edges de and ct are removed. Determine a maximal flow for the new network.

6.3 Auxiliary networks and phases

Let $N = (G, c, s, t)$ be a flow network with a f. We define a further flow network (G', c', s, t) as follows. G' has the same vertex set as G. For each edge $e = uv$ of G with $f(e) < c(e)$, there is an edge $e' = uv$ in G' with $c'(e') = c(e) - f(e)$; for each edge $e = uv$ with $f(e) \neq 0$, G' contains an edge $e'' = vu$ with $c'(e'') = f(e)$.

The labelling process in the algorithm of Ford and Fulkerson – as given in steps (6) to (9) for forward edges and in steps (10) to (13) for backward edges – uses only those edges e of G for which G' contains the edge e' or e''; an augmenting path with respect to f in G corresponds to a directed path from s to t in G'. Thus we may use G' to decide whether f is maximal and, if this is not the case, to find an augmenting path. One calls $N' = (G', c', s, t)$ the *auxiliary network* with respect to f. The next lemma should now be clear.

Lemma 6.3.1. *Let $N = (G, c, s, t)$ be a flow network with a flow f, and let N' be the corresponding auxiliary network. Then f is maximal if and only if t is not accessible from s in G'.* □

Example 6.3.2. Consider the flow $f = f_3$ of Example 6.2.3; see Figure 6.6. The corresponding auxiliary network is given in Figure 6.14.

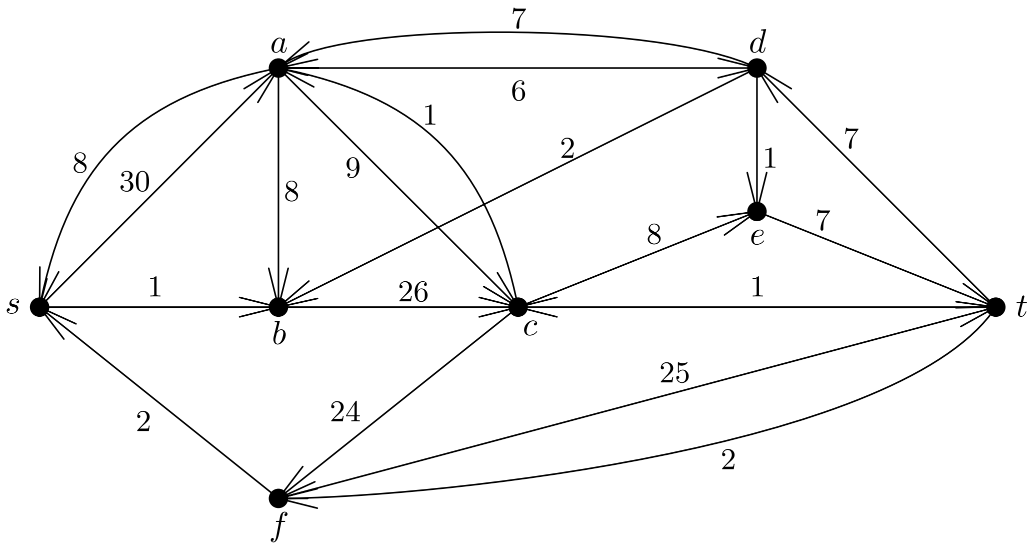

Fig. 6.14. Auxiliary network for Example 6.3.2

It is intuitively clear that a flow in N' can be used to augment f when constructing a maximal flow on N. The following two results make this idea more precise.

Lemma 6.3.3. *Let $N = (G, c, s, t)$ be a flow network with a flow f, and let N' be the corresponding auxiliary network. Moreover, let f' be a flow on N'. Then there exists a flow f'' of value $w(f'') = w(f) + w(f')$ on N.*

Proof. For each edge $e = uv$ of G, let $e' = uv$ and $e'' = vu$. If e' or e'' is not contained in N', we set $f'(e') = 0$ or $f'(e'') = 0$, respectively. We put $f'(e) = f'(e') - f'(e'')$;[2] then we may interpret f' as a (possibly non-admissible) *flow* on N: f' satisfies condition (F2), but not necessarily (F1). Obviously, the mapping f'' defined by

$$f''(e) \;=\; f(e) + f'(e') - f'(e'')$$

[2]Note that the minus sign in front of $f'(e'')$ is motivated by the fact that e' and e'' have opposite orientation.

also satisfies condition (F2). Now the definition of N' shows that the conditions $0 \leq f'(e') \leq c(e) - f(e)$ and $0 \leq f'(e'') \leq f(e)$ hold for each edge e, so that f'' satisfies (F1) as well. Thus f'' is a flow, and clearly $w(f'') = w(f) + w(f')$. □

Theorem 6.3.4. *Let $N = (G, c, s, t)$ be a flow network with a flow f, and let N' be the corresponding auxiliary network. Denote the value of a maximal flow on N and on N' by w and w', respectively. Then $w = w' + w(f)$.*

Proof. By Lemma 6.3.3, $w \geq w' + w(f)$. Now let g be a maximal flow on N and define a flow g' on N' as follows: for each edge e of G, set

$$\begin{aligned} g'(e') &= g(e) - f(e) \quad \text{if } g(e) > f(e); \\ g'(e'') &= f(e) - g(e) \quad \text{if } g(e) < f(e). \end{aligned}$$

Note that e' and e'' really are edges of N' under the conditions given above and that their capacities are large enough to ensure the validity of (F1). For every other edge e^* in N', put $g'(e^*) = 0$. It is easy to check that g' is a flow of value $w(g') = w(g) - w(f)$ on N'. This shows $w' + w(f) \geq w$. □

Exercise 6.3.5. Give an alternative proof for Theorem 6.3.4 by proving that the capacity $c'(S, T)$ of a cut (S, T) in N' is equal to $c(S, T) - w(f)$.

Remark 6.3.6. Note that the graph G' may contain parallel edges even if G itself – as we always assume – does not. This phenomenon occurs when G contains antiparallel edges, say $d = uv$ and $e = vu$. Then G' contains the parallel edges d' and e'' with capacities $c'(d') = c(d) - f(d)$ and $c'(e'') = f(e)$, respectively. For the validity of the preceding proofs and the subsequent algorithms, it is important that parallel edges of G' are *not* identified (and their capacities not added). Indeed, if we identified the edges d' and e'' above into a new edge e^* with capacity $c'(e^*) = c(d) - f(d) + f(e)$, it would no longer be obvious how to distribute a flow value $f'(e^*)$ when defining f'' in the proof of Lemma 6.3.3: we would have to decide which part of $f'(e^*)$ should contribute to $f''(d)$ (with a plus sign) and which part to $f''(e)$ (with a minus sign). Of course, it would always be possible to arrange this in such a manner that a flow f'' satisfying the feasibility condition (F1) arises, but this would require some unpleasant case distinctions. For this reason, we allow G' to contain parallel edges.[3] However, when actually programming an algorithm using auxiliary networks, it might be worthwhile to identify parallel edges of G' and add the necessary case distinctions for distributing the flow on N' during the augmentation step. In addition, one should also simplify things then by *cancelling* flow on pairs of antiparallel edges in such a way that only one edge of such a pair carries a non-zero flow.

[3]Alternatively, we could forbid G to contain antiparallel edges; this might be achieved, for instance, by always subdividing one edge of an antiparallel pair.

We have seen that it is possible to find a maximal flow for our original network N by finding appropriate flows in a series of auxiliary networks $N_1 = N'(f_0)$, $N_2 = N'(f_1), \dots$ Note that the labelling process in the algorithm of Ford and Fulkerson amounts to constructing a new auxiliary network after each augmentation of the flow. Thus constructing the auxiliary networks explicitly cannot by itself result in a better algorithm; in order to achieve an improvement, we need to construct several augmenting paths within the same auxiliary network. We require a further definition. A flow f is called a *blocking flow* if every augmenting path with respect to f has to contain a backward edge. Trivially, any maximal flow is blocking as well. But the converse is false: for example, the flow f_8 of Example 6.2.3 displayed in Figure 6.11 is blocking, but not maximal.

There is yet another problem that needs to be addressed: the auxiliary networks constructed so far are still too big and complex. Indeed, the auxiliary network in Example 6.3.2 looks rather crowded. Hence we shall work with appropriate sub-networks instead. The main idea of the algorithm of Dinic [Din70] is to use not only an augmenting path of shortest length, but also to keep an appropriate small network $N''(f)$ basically unchanged – with just minor modifications – until every further augmenting path has to have larger length.

For better motivation, we return once more to the algorithm of Edmonds and Karp. Making step (5) of Algorithm 6.1.7 more precise in step (5′) of Theorem 6.2.1 ensures that the labelling process on the auxiliary network $N' = N'(f)$ runs as a BFS on G'; thus the labelling process divides G' into *levels* or *layers* of vertices having the same distance to s; see Section 3.3. As we are only interested in finding augmenting paths of shortest length, N' usually contains a lot of superfluous information: we may omit

- all vertices $v \neq t$ with $d(s,v) \geq d(s,t)$ together with all edges incident with these vertices;
- all edges leading from some vertex in layer j to some vertex in a layer $i \leq j$.

The resulting network $N'' = N''(f) = (G'', c'', s, t)$ is called the *layered auxiliary network* with respect to f.[4] The name *layered* network comes from the fact that G'' is a *layered digraph*: the vertex set V of G'' is the disjoint union of subsets $V_0, \dots, V_k$ and all edges of G'' have the form uv with $u \in V_i$ and $v \in V_{i+1}$ for some index i.

Example 6.3.7. Consider the flow $f = f_3$ in Example 6.2.3 and the corresponding auxiliary network N' displayed in Figure 6.14. The associated layered auxiliary network N'' is shown in Figure 6.15. Here the capacities are

[4]Strictly speaking, both N' and N'' should probably only be called *networks* if t is accessible from s, that is, if f is not yet maximal – as this is part of the definition of flow networks. But it is more convenient to be a bit lax here.

written in parentheses; the other numbers are the values of a blocking flow g on N'' which arises from the three augmenting paths displayed in Figures 6.6, 6.7, and 6.8. Note that all three paths of length four in Example 6.2.3 can now be seen in the considerably clearer network N''. Note that g is blocking but not maximal: the sequence (s, a, d, e, c, f, t) determines an augmenting path containing the backward edge ce.

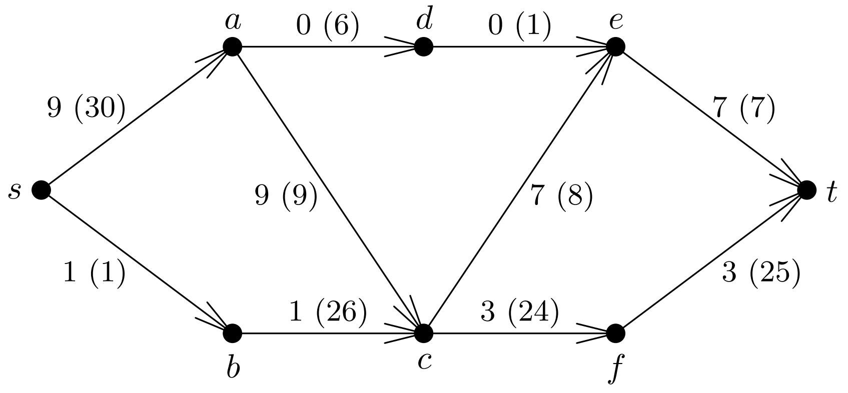

Fig. 6.15. Layered auxiliary network

We remark that even N'' might still contain superfluous elements, for example vertices from which t is not accessible. But as such vertices cannot be determined during the BFS used for constructing N'', we will not bother to find and remove them.

Exercise 6.3.8. How could vertices v in N'' from which t is not accessible be removed?

Exercise 6.3.9. Draw N' and N'' for the flow f_7 displayed in Figure 6.10 and determine a blocking flow for N''.

We will treat two algorithms for determining maximal flows. Both algorithms can take a given flow f, construct a blocking flow g in the corresponding layered auxiliary network $N''(f)$, and then use g to augment f. Note that a flow f' of value $w(f')$ on $N''(f)$ may indeed be used to augment the given flow f to a flow of value $w(f) + w(f')$, as N'' is a sub-network of N'; hence we may apply Lemma 6.3.3 and the construction given in its proof.

Exercise 6.3.10. Show that Theorem 6.3.4 does not carry over to $N''(f)$.

Thus we begin with some starting flow f_0, usually the zero flow, construct a blocking flow g_0 in $N''(f_0)$, use this flow to augment f_0 to a flow f_1 of value $w(g_0)$, construct a blocking flow g_1 in $N''(f_1)$, and so on. The algorithm terminates when we reach a flow f_k for which the sink t is not accessible from s in $N''(f_k)$. Then t is not accessible from s in $N'(f_k)$ either; hence f_k is maximal, by Lemma 6.3.1. Each construction of a blocking flow g_i, together with the subsequent augmentation of f_i to f_{i+1}, is called a *phase* of the algorithm. We postpone the problem of determining blocking flows to the next section. Now we derive an estimate for the number of phases needed and write down an algorithm for constructing the layered auxiliary network.

Lemma 6.3.11. *Let $N = (G, c, s, t)$ be a flow network with a flow f, and let $N''(f)$ be the corresponding layered auxiliary network. Moreover, let g be a blocking flow on $N''(f)$, h the flow on N of value $w(f) + w(g)$ constructed from f and g as in Lemma 6.3.3, and $N''(h)$ the layered auxiliary network with respect to h. Then the distance from s to t in $N''(h)$ is larger than in $N''(f)$.*

Proof. It is easy to see that $N''(h)$ is the layered auxiliary network for $N' = N'(f)$ with respect to g.[5] We may also view g as a flow on $N'' = N''(f)$, by assigning value 0 to all edges contained in N', but not in N''. As g is a blocking flow on N'', there is no augmenting path from s to t in N'' consisting of forward edges only. Hence each augmenting path in N' with respect to g has to contain a backward edge or one of those edges which were omitted during the construction of N''. In both cases, the length of this path must be larger than the distance from s to t in N''. Thus the distance from s to t in the layered auxiliary network for N' with respect to g – that is, in $N''(h)$ – is indeed larger than the corresponding distance in N''. □

Corollary 6.3.12. *Let N be a flow network. Then the construction of a maximal flow on N needs at most $|V| - 1$ phases.*

Proof. Let $f_0, f_1, \ldots, f_k$ be the sequence of flows constructed during the algorithm. Lemma 6.3.11 implies that the distance from s to t in $N''(f_k)$ is at least k larger than that in $N''(f_0)$. Thus the number of phases can be at most $|V| - 1$. □

Exercise 6.3.13. Let f be the flow f_3 in Example 6.2.3 and g the blocking flow on $N''(f)$ in Example 6.3.7. Draw the layered auxiliary networks with respect to g on $N'(f)$ and on $N''(f)$. What does the flow h determined by f and g on N look like? Convince yourself that $N''(h)$ is indeed equal to the layered auxiliary network with respect to g on $N'(f)$.

The following procedure for constructing the layered auxiliary network $N''(f)$ corresponds to the labelling process in the algorithm of Ford and Fulkerson with step (5) replaced by (5′) – as in Theorem 6.2.1. During the execution of the BFS, the procedure orders the vertices in layers and omits superfluous vertices and edges, as described in the definition of N''. The Boolean variable max is assigned the value *true* when f becomes maximal (that is, when t is no longer accessible from s); otherwise, it has value *false*. The variable $d + 1$ gives the number of layers of N''.

[5]The analogous claim for $N'' = N''(f)$ instead of $N'(f)$ does not hold, as Exercise 6.3.13 will show.

Algorithm 6.3.14. Let $N = (G, c, s, t)$ be a flow network with a flow f.

Procedure AUXNET($N, f; N'', \mathrm{max}, d$)

```
(1) $i \leftarrow 0$, $V_0 \leftarrow \{s\}$, $E'' \leftarrow \emptyset$, $V'' \leftarrow V_0$;
(2) repeat
(3)         $i \leftarrow i + 1$, $V_i \leftarrow \emptyset$;
(4)         for $v \in V_{i-1}$ do
(5)                 for $e \in \{e \in E : e^- = v\}$ do
(6)                         if $u = e^+ \notin V''$ and $f(e) < c(e)$
(7)                         then $e' \leftarrow vu$, $E'' \leftarrow E \cup \{e'\}$, $V_i \leftarrow V_i \cup \{u\}$;
                            $c''(e') \leftarrow c(e) - f(e)$ fi
(8)                 od;
(9)                 for $e \in \{e \in E : e^+ = v\}$ do
(10)                        if $u = e^- \notin V''$ and $f(e) \neq 0$
(11)                        then $e'' \leftarrow vu$, $E'' \leftarrow E \cup \{e''\}$, $V_i \leftarrow V_i \cup \{u\}$;
                            $c''(e'') \leftarrow f(e)$ fi
(12)                od
(13)        od;
(14)        if $t \in V_i$ then remove all vertices $v \neq t$ together with all
            edges $e$ satisfying $e^+ = v$ from $V_i$ fi;
(15)        $V'' \leftarrow V'' \cup V_i$
(16) until $t \in V''$ or $V_i = \emptyset$;
(17) if $t \in V''$ then max $\leftarrow$ false; $d \leftarrow i$ else max $\leftarrow$ true fi
```

We leave it to the reader to give a formal proof for the following lemma.

Lemma 6.3.15. *Algorithm 6.3.14 constructs the layered auxiliary network $N'' = N''(f) = (G'', c'', s, t)$ on $G'' = (V'', E'')$ with complexity $O(|E|)$.* □

In the next section, we will provide two methods for constructing a blocking flow g on N''. Let us assume for the moment that we already know such a procedure BLOCKFLOW($N''; g$). Then we want to use g for augmenting f. The following procedure performs this task; it uses the construction given in the proof of Lemma 6.3.3. Note that N'' never contains both e' and e''.

Algorithm 6.3.16. Let $N = (G, c, s, t)$ be a given flow network with a flow f, and suppose that we have already constructed $N'' = N''(f)$ and a blocking flow g.

Procedure AUGMENT($f, g; f$)

```
(1) for $e \in E$ do
(2)         if $e' \in E''$ then $f(e) \leftarrow f(e) + g(e')$ fi;
(3)         if $e'' \in E''$ then $f(e) \leftarrow f(e) - g(e'')$ fi
(4) od
```

We can now write down an algorithm for determining a maximal flow:

Algorithm 6.3.17. Let $N = (G, c, s, t)$ be a flow network.

Procedure MAXFLOW($N; f$)

```
(1) for $e \in E$ do $f(e) \leftarrow 0$ od;
(2) repeat
(3)         AUXNET($N, f; N'', \max, d$);
(4)         if max = false
(5)         then BLOCKFLOW($N''; g$); AUGMENT($f, g; f$) fi
(6) until max = true
```

Remark 6.3.18. The only part which is still missing in Algorithm 6.3.17 is a specific procedure BLOCKFLOW for determining a blocking flow g on N''. Note that each phase of Algorithm 6.3.17 has complexity at least $O(|E|)$, because AUGMENT has this complexity. It is quite obvious that BLOCKFLOW will also have complexity at least $O(|E|)$; in fact, the known algorithms have even larger complexity. Let us denote the complexity of BLOCKFLOW by $k(N)$. Then MAXFLOW has a complexity of $O(|V|k(N))$, since there are at most $O(|V|)$ phases, by Corollary 6.3.12.

Exercise 6.3.19. Modify Algorithm 6.3.17 in such a way that it finds a minimal cut (S, T) as well.

6.4 Constructing blocking flows

In this section, we fill in the gap left in Algorithm 6.3.17 by presenting two algorithms for constructing blocking flows. The first of these is due to Dinic [Din70]. The Dinic algorithm constructs, starting with the zero flow, augmenting paths of length d in the layered auxiliary network N'' (where $d+1$ denotes the number of layers) and uses them to change the flow g until t is no longer accessible from s; then g is a blocking flow. Compared to the algorithm of Edmonds and Karp, it has two advantages. First, using $N'' = N''(f)$ means that we consider only augmenting paths without any backward edges, since a path containing a backward edge has length at least $d + 2$. Second, when we update the input data after an augmentation of the current flow g on N'', we only have to decrease the capacities of the edges contained in the respective augmenting path and omit vertices and edges that are no longer needed. In particular, we do *not* have to do the entire labelling process again.

Algorithm 6.4.1. Let $N = (G, c, s, t)$ be a layered flow network with layers $V_0, \dots, V_d$, where all capacities are positive.

Procedure BLOCKFLOW($N; g$)

```
(1) for $e \in E$ do $g(e) \leftarrow 0$ od;
(2) repeat
(3)         $v \leftarrow t$, $a \leftarrow \infty$;
```

```
(4)         for $i = d$ downto 1 do
(5)             choose some edge $e_i = uv$;
(6)             $a \leftarrow \min\{c(e_i), a\}$, $v \leftarrow u$
(7)         od;
(8)         for $i = 1$ to $d$ do
(9)             $g(e_i) \leftarrow g(e_i) + a$, $c(e_i) \leftarrow c(e_i) - a$;
(10)            if $c(e_i) = 0$ then omit $e_i$ from $E$ fi
(11)        od;
(12)        for $i = 1$ to $d$ do
(13)            for $v \in V_i$ do
(14)                if $d_{\text{in}}(v) = 0$
(15)                then omit $v$ and all edges $e$ with $e^- = v$ fi
(16)            od
(17)        od
(18) until $t \notin V_d$
```

Theorem 6.4.2. *Algorithm 6.4.1 determines a blocking flow on N with complexity $O(|V||E|)$.*

Proof. By definition of a layered auxiliary network, each vertex is accessible from s at the beginning of the algorithm. Thus there always exists an edge e_i with end vertex v which can be chosen in step (5), no matter which edges $e_d, \dots, e_{i+1}$ were chosen before. Hence the algorithm constructs a directed path $P = (e_1, \dots, e_d)$ from s to t. At the end of the loop (4) to (7), the variable a contains the capacity a of P, namely $a = \min\{c(e_i) : i = 1, \dots, d\}$. In steps (8) to (11), the flow constructed so far (in the first iteration, the zero flow) is increased by a units along P, and the capacities of the edges $e_1, \dots, e_d$ are decreased accordingly. Edges whose capacity is decreased to 0 cannot appear on any further augmenting path and are therefore discarded. At the end of the loop (8) to (11), we have reached t and augmented the flow g. Before executing a further iteration of (4) to (11), we have to check whether t is still accessible from s. Even more, we need to ensure that every vertex is still accessible from s in the modified layered network. This task is performed by the loop (12) to (17). Using induction on i, one may show that this loop removes exactly those vertices which are not accessible from s as well as all edges beginning in these vertices. If t is still contained in N after this loop has ended, we may augment g again so that we repeat the entire process. Finally, the algorithm terminates after at most $|E|$ iterations, since at least one edge is removed during each augmentation; at the very latest, t can no longer be in V_d when all the edges have been removed. Obviously, each iteration (4) to (17) has complexity $O(|V|)$; this gives the desired overall complexity of $O(|V||E|)$. □

Using Remark 6.3.18, we immediately obtain the following result due to Dinic.

Corollary 6.4.3. *Assume that we use the procedure BLOCKFLOW of Algorithm 6.4.1 in Algorithm 6.3.17. Then the resulting algorithm calculates a maximal flow on a given flow network N with complexity $O(|V|^2|E|)$.* □

Note that the algorithm of Dinic has a complexity of $O(|V|^4)$ for dense graphs, whereas the algorithm of Edmonds and Karp needs $O(|V|^5)$ steps in this case. Using another, more involved, method for constructing blocking flows, we may reduce the complexity to $O(|V|^3)$ for arbitrary graphs. But first, let us work out an example for the algorithm of Dinic.

Example 6.4.4. Consider again the flow $f = f_3$ in Example 6.2.3. The corresponding layered auxiliary network N'' was displayed in Figure 6.15. We apply Algorithm 6.4.1 to N''. In step (5), let us always choose the edge uv for which u is first in alphabetical order, so that the algorithm becomes deterministic. Initially, it constructs the path s — a — c — e — t with capacity 7. The corresponding flow g_1 is shown in Figure 6.16; the numbers in parentheses give the new capacities (which were changed when the flow was changed). The edge et, which is drawn broken, is removed during this first iteration.

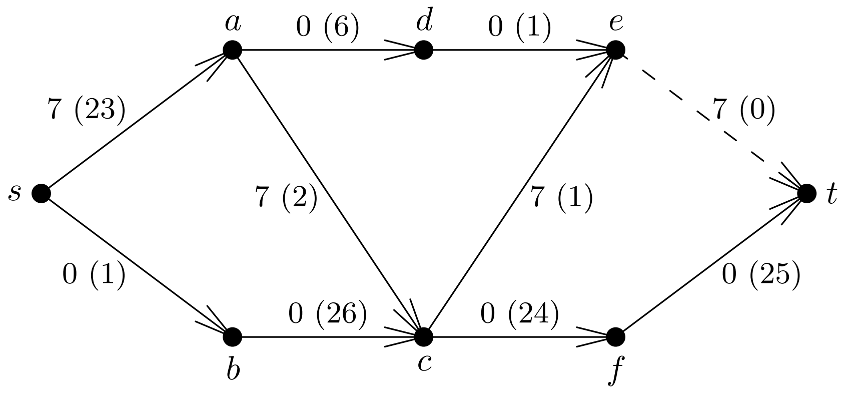

Fig. 6.16. $w(g_1) = 7$

In the second iteration, we obtain the path s — a — c — f — t in the network of Figure 6.16; it has capacity two. Figure 6.17 shows the new network with the new flow g_2. Note that the edge ac has been removed.

Finally, using the network in Figure 6.17, the third iteration constructs the path s — b — c — f — t with capacity one, and we obtain the flow g_3 displayed in Figure 6.18. During this iteration, the algorithm removes first the edge sb, the vertex b, and the edge bc; then the vertex c, and the edges ce and cf; the vertex f, and the edge ft; and finally t itself; see Figure 6.18. Hence g_3 is a blocking flow – actually, the blocking flow previously displayed in Figure 6.15.

Exercise 6.4.5. Use Algorithm 6.4.1 to determine a blocking flow on the layered network shown in Figure 6.19; this is taken from [SyDK83].

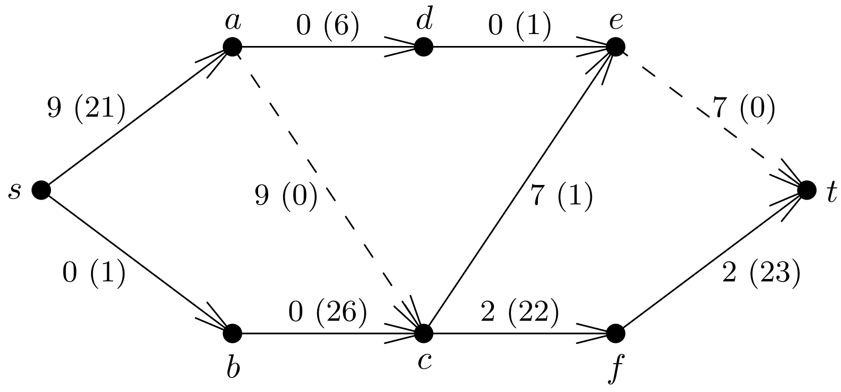

Fig. 6.17. $w(g_2) = 9$

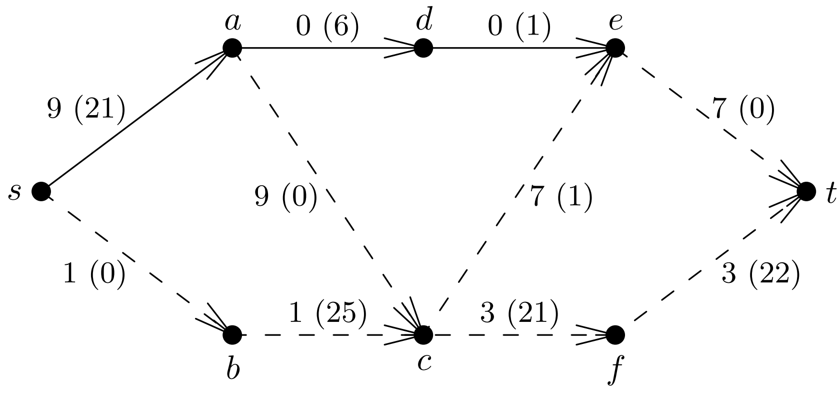

Fig. 6.18. Blocking flow g_3 with $w(g_3) = 10$

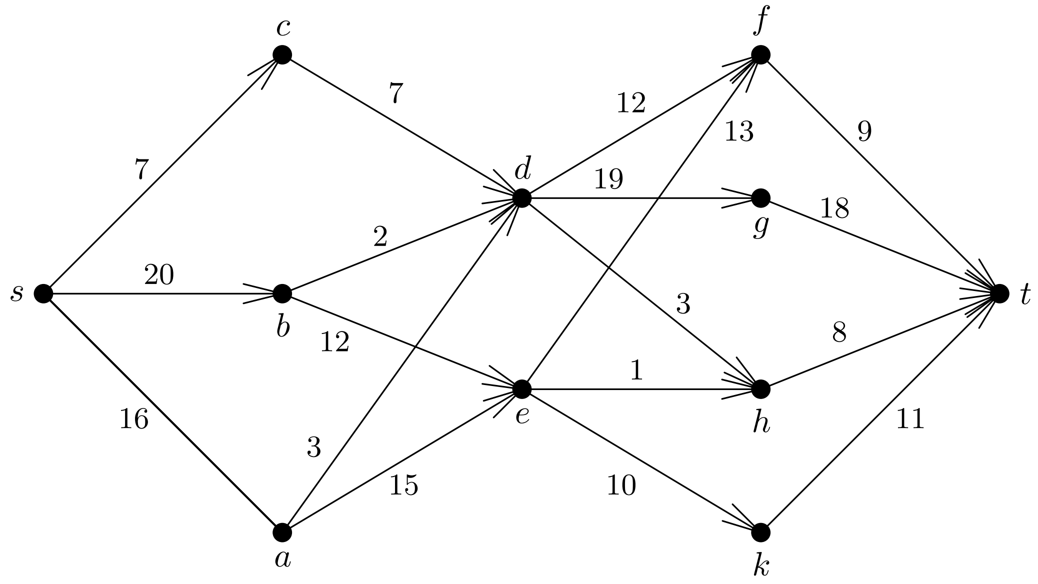

Fig. 6.19. A layered network

We now turn to a completely different method for constructing blocking flows, which is due to Malhotra, Kumar and Mahashwari [MaKM78] and has complexity $O(|V|^2)$. This algorithm does not use augmenting paths and tries instead to push as big a flow as possible through the network. We need some notation. Let $N = (G, c, s, t)$ be a layered flow network. For each vertex v, the *flow potential* $p(v)$ is defined by

$$p(v) \;=\; \min\left\{\sum_{e^-=v} c(e),\; \sum_{e^+=v} c(e)\right\};$$

thus $p(v)$ is the maximal amount of flow which could possibly pass through v. A vertex u is called a *minimal vertex* – and its flow potential the *minimal potential* – if $p(u) \le p(v)$ holds for all vertices v. Intuitively, it should be possible to construct a flow g of value $w(g) = p(u)$ by pushing the flow from u forward to t and pulling it back from u to s. This is the main idea of the following algorithm for constructing a blocking flow on N.

Algorithm 6.4.6. Let $N = (G, c, s, t)$ be a layered flow network with layers $V_1, \dots V_d$, where all capacities are positive.

Procedure BLOCKMKM($N; g$)

```
(1) for $e \in E$ do $g(e) \leftarrow 0$ od;
(2) for $v \in V$ do
(3)         if $v = t$ then $p^-(v) \leftarrow \infty$ else $p^-(v) \leftarrow \sum_{e^-=v} c(e)$ fi
(4)         if $v = s$ then $p^+(v) \leftarrow \infty$ else $p^+(v) \leftarrow \sum_{e^+=v} c(e)$ fi
(5) od;
(6) repeat
(7)         for $v \in V$ do $p(v) \leftarrow \min\{p^+(v), p^-(v)\}$ od;
(8)         choose a minimal vertex $w$;
(9)         PUSH($w, p(w)$);
(10)        PULL($w, p(w)$);
(11)        while there exists $v$ with $p^+(v) = 0$ or $p^-(v) = 0$ do
(12)                for $e \in \{e \in E : e^- = v\}$ do
(13)                        $u \leftarrow e^+$, $p^+(u) \leftarrow p^+(u) - c(e)$;
(14)                        remove $e$ from $E$
(15)                od;
(16)                for $e \in \{e \in E : e^+ = v\}$ do
(17)                        $u \leftarrow e^-$, $p^-(u) \leftarrow p^-(u) - c(e)$;
(18)                        remove $e$ from $E$
(19)                od;
(20)                remove $v$ from $V$
(21)        od
(22) until $s \notin V$ or $t \notin V$
```

Here, PUSH is the following procedure for *pushing a flow of value $p(w)$ to t*:

Procedure PUSH(y, k)

```
(1) let $Q$ be a queue with single element $y$;
(2) **for** $u \in V$ **do** $b(u) \leftarrow 0$ **od**;
(3) $b(y) \leftarrow k$;
(4) **repeat**
(5)         remove the first element $v$ from $Q$;
(6)         **while** $v \neq t$ **and** $b(v) \neq 0$ **do**
(7)                 choose an edge $e = vu$;
(8)                 $m \leftarrow \min \{c(e), b(v)\}$;
(9)                 $c(e) \leftarrow c(e) - m$, $g(e) \leftarrow g(e) + m$;
(10)                $p^+(u) \leftarrow p^+(u) - m$, $b(u) \leftarrow b(u) + m$;
(11)                $p^-(v) \leftarrow p^-(v) - m$, $b(v) \leftarrow b(v) - m$;
(12)                append $u$ to $Q$;
(13)                **if** $c(e) = 0$ **then** remove $e$ from $E$ **fi**
(14)        **od**
(15) **until** $Q = \emptyset$
```

The procedure PULL for *pulling a flow of value $p(w)$ to s* is defined in an analogous manner; we leave this task to the reader.

Theorem 6.4.7. *Algorithm 6.4.6 constructs a blocking flow g on N with complexity $O(|V|^2)$.*

Proof. We claim first that an edge e is removed from E only if there exists no augmenting path containing e and consisting of forward edges only. This is clear if e is removed in step (14) or (18): then either $p^+(v) = 0$ or $p^-(v) = 0$ (where $e^- = v$ or $e^+ = v$, respectively) so that no augmenting path containing v and consisting of forward edges only can exist. If e is removed in step (13) during a call of the procedure PUSH, we have $c(e) = 0$ at this point; because of step (9) in PUSH, this means that $g(e)$ has reached its original capacity $c(e)$ so that e cannot be used any longer as a forward edge. A similar argument applies if e is removed during a call of PULL. As each iteration of BLOCKMKM removes edges and decreases capacities, an edge which can no longer be used as a forward edge with respect to g when it is removed cannot be used as a forward edge at a later point either. Hence, there cannot exist any augmenting path consisting of forward edges only at the end of BLOCKMKM, when s or t have been removed. This shows that g is blocking; of course, it still remains to check that g is a flow in the first place.

We now show that g is indeed a flow, by using induction on the number of iterations of the **repeat**-loop (6) to (22). Initially, g is the zero flow. Now suppose that g is a flow at a certain point of the algorithm (after the i-th iteration, say). All vertices v which cannot be used any more – that is, vertices into which no flow can enter or from which no flow can emerge any more – are removed during the **while**-loop (11) to (21), together with all edges incident with these vertices. During the next iteration – that is, after the flow potentials have been brought up to date in step (7) – the algorithm chooses a vertex w with minimal potential $p(w)$; here $p(w) \neq 0$, since otherwise w would have been removed before during the **while**-loop. Next, we have to check that the procedure PUSH$(w, p(w))$ really generates a flow of value $p(w)$ from the source w to the sink t. As Q is a queue, the vertices u in PUSH are treated as in a BFS on the layers $V_k, V_{k+1}, \ldots, V_d$, where $w \in V_k$. During the first iteration of the **repeat**-loop of PUSH, we have $v = w$ and $b(v) = p(w)$; here $b(v)$ contains the value of the flow which has to flow out of v. During the **while**-loop, the flow of value $b(v)$ is distributed among the edges vu with tail v. Note that the capacity of an edge vu is always used entirely, unless $b(v) < c(e)$. In step (9), the capacity of vu is reduced – in most cases, to 0, so that vu will be removed in step (13) – and the value of the flow is increased accordingly. Then we decrease the value $b(v)$ of the flow which still has to leave v via other edges accordingly in step (11), and increase $b(u)$ accordingly in step (10); also the flow potentials are updated by the appropriate amount. In this way the required value of the flow $b(v)$ is distributed among the vertices of the next layer; as we chose w to be a vertex of minimal potential, we always have $b(v) \leq p(w) \leq p(v)$, and hence it is indeed possible to distribute the flow. At the end of procedure PUSH, the flow of value $p(w)$ has reached t, since $V_d = \{t\}$. An analogous argument shows that the subsequent call of the procedure PULL$(w, p(w))$ yields a flow of value $p(w)$ from the source s to the sink w; of course, PULL performs the actual construction in the opposite direction. We leave the details to the reader. Hence g will indeed be a flow from s to t after both procedures have been called.

Each iteration of the **repeat**-loop of BLOCKMKM removes at least one vertex, since the flow potential of the minimal vertex w is decreased to 0 during PUSH and PULL; hence the algorithm terminates after at most $|V|-1$ iterations. We now need an estimate for the number of operations involving edges. Initializing p^+ and p^- in (3) and (4) takes $O(|E|)$ steps altogether. As an edge e can be removed only once, e appears at most once during the **for**-loops (12) to (19) or in step (13) of PUSH or PULL. For each vertex v treated during PUSH or PULL, there is at most one edge starting in v which still has a capacity $\neq 0$ left after it has been processed – that is, which has not been removed. As PUSH and PULL are called at most $|V| - 1$ times each, we need at most $O(|V|^2)$ steps for treating these special edges. But $O(|V|^2)$ dominates $O(|E|)$; hence the overall number of operations needed for treating the edges is $O(|V|^2)$. It is easy to see that all other operations of the algorithm need at most $O(|V|^2)$ steps as well, so that we obtain the desired overall complexity of $O(|V|^2)$. □

The algorithm arising from Algorithm 6.3.17 by replacing BLOCKFLOW with BLOCKMKM is called the *MKM-algorithm.* As explained in Remark 6.3.18, Theorem 6.4.7 implies the following result.

Theorem 6.4.8. *The MKM-algorithm constructs with complexity $O(|V|^3)$ a maximal flow for a given flow network N .* □

Example 6.4.9. Consider again the layered auxiliary network of Example 6.3.7. Here the flow potentials are as follows: $p(s) = 31$, $p(a) = 15$, $p(b) = 1$, $p(c) = 32$, $p(d) = 1$, $p(e) = 7$, $p(f) = 24$, $p(t) = 32$. Let us choose b as minimal vertex in step (8). After the first iteration, we have the flow g_1 shown in Figure 6.20; the vertex b as well as the edges sb and bc have been removed.

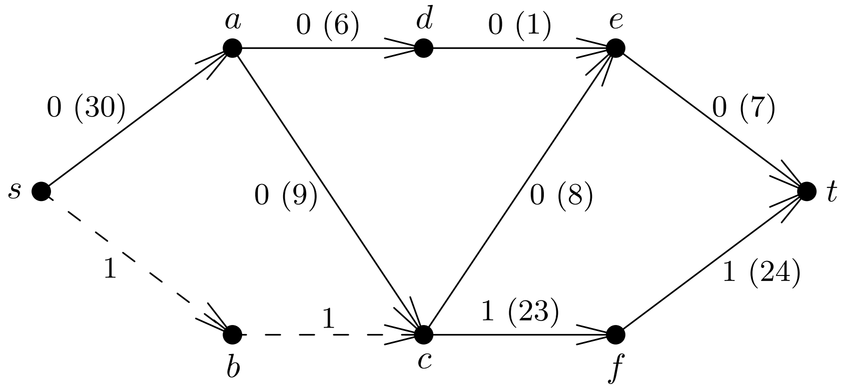

Fig. 6.20. $w(g_1) = 1$

Next, we have flow potentials $p(s) = 30$, $p(a) = 15$, $p(c) = 9$, $p(d) = 1$, $p(e) = 7$, $p(f) = 23$, $p(t) = 31$, so that d is the unique minimal vertex. After the second iteration, we have constructed the flow g_2 in Figure 6.21; also, d, ad, and de have been removed.

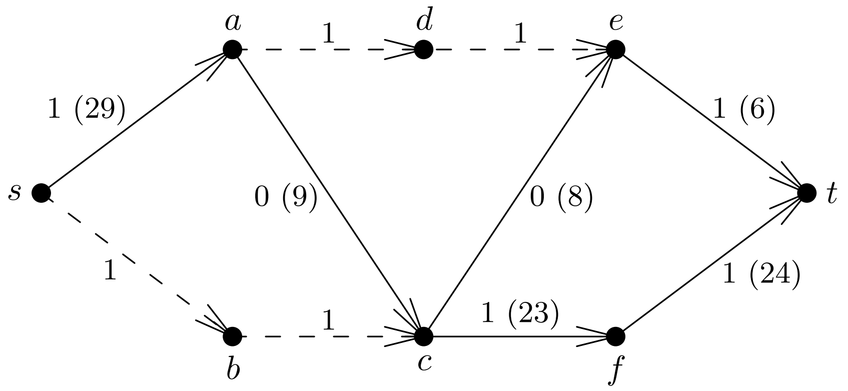

Fig. 6.21. $w(g_2) = 2$

In the following iteration, $p(s) = 29$, $p(a) = 9$, $p(c) = 9$, $p(e) = 6$, $p(f) = 23$ and $p(t) = 30$. Hence the vertex e is minimal and we obtain the flow g_3 shown in Figure 6.22; note that e, ce, and et have been removed.

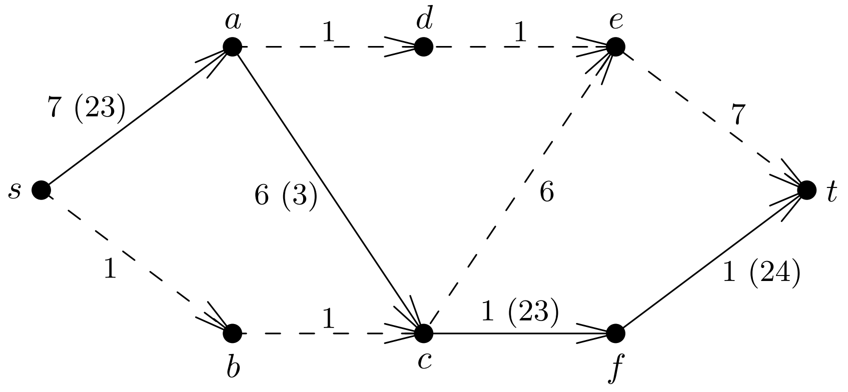

Fig. 6.22. $w(g_3) = 8$

Now the flow potentials are $p(s) = 23, p(a) = 3, p(c) = 3, p(f) = 23, p(t) = 24$. We select the minimal vertex a and construct the flow g_4 in Figure 6.23, a blocking flow with value $w(g_4) = 11$; all remaining elements of the network have been removed. Note that g_4 differs from the blocking flow constructed in Example 6.4.4.

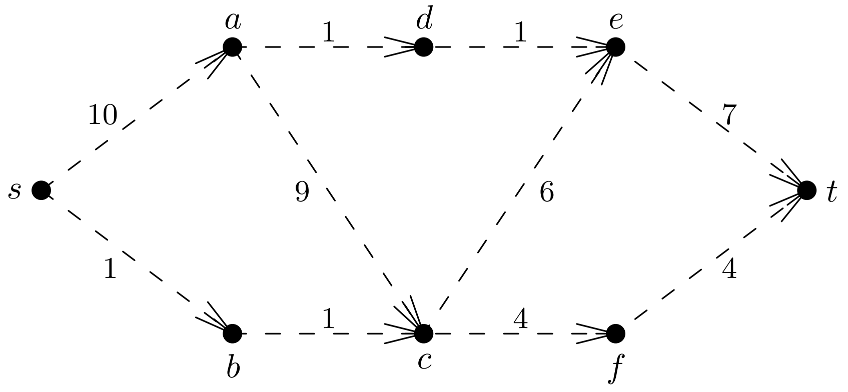

Fig. 6.23. $w(g_4) = 11$

Exercise 6.4.10. Use Algorithm 6.4.6 to find a blocking flow for the layered auxiliary network of Exercise 6.4.5.

We have now seen three classical algorithms for constructing maximal flows. Note that these algorithms have quite different complexities for the case of dense graphs, that is, for $|E| = O(|V|^2)$. As this case shows, the MKM-algorithm is superior to the other two algorithms; however, it is clearly also considerably more involved.

Further algorithms with complexity $O(|V|^3)$ are given in [Kar74], [Tar84], and [GoTa88]; the last of these papers also contains an algorithm of complexity $O(|V||E| \log(|V|^2/|E|))$. We shall present the algorithm of Goldberg and Tarjan in Section 6.6. The approach of Goldberg and Tarjan can also be used to solve the *parametrized flow problem*, where the capacities of the edges incident with s and t depend monotonically on some real parameter; interestingly, the complexity will only change by a constant factor; see [GaGT89]. A modification of the algorithm of Goldberg and Tarjan suggested in [ChMa89] results in a complexity $O(|V|^2|E|^{1/2}$; we shall include this result in Section 6.6 as well.

An algorithm with complexity $O(|V||E|\log|V|)$ is given in [Sle80]; see also [SlTa83]. In the paper [AhOr89], there is an algorithm of complexity $O(|V||E| + |V|^2 \log U)$, where U denotes the maximum of all capacities $c(e)$ occurring in the problem. This result may be improved somewhat: the term $\log U$ can be replaced by $(\log U)^{1/2}$; see [AhOT89].

A probabilistic algorithm was proposed by [ChHa89]. Later, several authors gave deterministic variants of this algorithm; see [Alo90], [KiRT94], and [ChHa95]. For instance, for graphs satisfying $|V|\log|V|)^2 \leq |E| \leq |V|^{5/3}\log|V|$, one obtains a complexity of only $O(|V||E|)$. An algorithm with complexity $O(|V|^3/\log|V|)$ can be found in [ChHM96].

Let us also mention three good surveys dealing with some of the more recent algorithms: [GoTT90] and [AhMO89, AhMO91]. For an extensive treatment of flow theory and related topics, we refer to the monograph [AhMO93].

A further new idea for solving the max-flow problem emerged in a paper by Karger [Kar99], who proceeds by computing approximately minimal cuts and uses these to compute a maximum flow, thus reversing the usual approach. He shows that his algorithm has an improved complexity with high probability; however, this seems not to be of practical interest (yet).

A completely different approach to the problem of finding a maximal flow is to use the well-known simplex algorithm from linear programming and specialize it to treat flow networks; the resulting algorithm actually works for a more general problem. It is called the *network simplex algorithm* and is of eminent practical interest; we will devote the entire Chapter 11 to this topic.

For planar graphs, one may construct a maximal flow with complexity $O(|V|^{\frac{3}{2}}\log|V|)$; see [JoVe82]. If s and t are in the same face of G, even a complexity of $O(|V|\log|V|)$ suffices; see [ItSh79].

In the undirected case – that is, in *symmetric flow networks*, see Section 12.1 – the max flow problem can be solved with complexity $O(|V|\log^2|V|)$; see [HaJo85]. For flow networks on bipartite graphs, fast algorithms can be found in [GuMF87] and in [AhOST94]; these algorithms are particularly interesting if one of the two components of the bipartition is very small compared to the other component.

Finally, let us mention some references discussing the practical efficiency of various flow algorithms: [Che80], [Gal81], [Ima83], [GoGr88], [DeMe89], and [AhKMO92]. There are also several relevant papers in the collection [JoMcG93].

6.5 Zero-one flows

In this section, we consider a special case occurring in many combinatorial applications of flow theory[6], namely integral flows which take the values 0 and 1 only. In this special case, the complexity estimates we have obtained so far can be improved considerably; it will suffice to use the algorithm of Dinic for this purpose. We need some terminology. A 0-1-*network* is a flow network $N = (G, c, s, t)$ for which all capacities $c(e)$ are restricted to the values 0 and 1. A flow f on a network N is called a 0-1-*flow* if f takes values 0 and 1 only. We begin with the following important lemma taken from [EvTa75].

Lemma 6.5.1. *Let $N = (G, c, s, t)$ be a* 0-1-*network. Then*

$$d(s,t) \;\le\; \frac{2|V|}{\sqrt{M}}, \tag{6.5}$$

where M is the maximal value of a flow on N. If, in addition, each vertex v of N except for s and t satisfies at least one of the two conditions $d_{\text{in}}(v) \le 1$ and $d_{\text{out}}(v) \le 1$, then even

$$d(s,t) \;\le\; 1 + \frac{|V|}{M}. \tag{6.6}$$

Proof. Denote the maximal distance from s in N by D, and let V_i be the set of all vertices $v \in V$ with $d(s,v) = i$ (for $i = 0, \dots, D$). Obviously,

$$(S_i, T_i) \;=\; (V_0 \cup V_1 \cup \dots \cup V_i,\; V_{i+1} \cup \dots \cup V_D)$$

is a cut, for each $i < d(s,t)$. As every edge e with $e^- \in S_i$ and $e^+ \in T_i$ satisfies $e^- \in V_i$ and $e^+ \in V_{i+1}$ and as N is a 0-1-network, Lemma 6.1.2 implies

$$M \;\le\; c(S_i, T_i) \;\le\; |V_i| \times |V_{i+1}| \quad \text{for } i = 0, \dots, d(s,t) - 1.$$

Thus at least one of the two values $|V_i|$ or $|V_{i+1}|$ cannot be smaller than $\sqrt{M}$. Hence at least half of the layers V_i with $i \le d(s,t)$ contain $\sqrt{M}$ or more vertices. This yields

$$d(s,t)\frac{\sqrt{M}}{2} \;\le\; |V_0| + \dots + |V_{d(s,t)}| \;\le\; |V|,$$

and hence (6.5) holds. Now assume that N satisfies the additional condition stated in the assertion. Then the flow through any given vertex cannot exceed one, and we get the stronger inequality

$$M \;\le\; |V_i| \quad \text{for } i = 1, \dots, d(s,t) - 1.$$

[6]We will discuss a wealth of such applications in Chapter 7.

Now

$$M(d(s,t)-1) \;\le\; |V_1| + \ldots + |V_{d(s,t)-1}| \;\le\; |V|,$$

proving (6.6). □

Using estimates which are a little more accurate, (6.5) can be improved to $d(s,t) \le |V|/\sqrt{M}$. We will not need this improvement for the complexity statements which we will establish later in this section; the reader might derive the stronger bound as an exercise.

Lemma 6.5.2. *Let $N = (G,c,s,t)$ be a layered 0-1-network. Then the algorithm of Dinic can be used to determine a blocking flow g on N with complexity $O(|E|)$.*

Proof. The reader may easily check that the following modification of Algorithm 6.4.1 determines a blocking flow g on a given 0-1-network N.

Procedure BLOCK01FLOW(N, g)

```
(1)  $L \leftarrow \emptyset$;
(2)  for $v \in V$ do $ind(v) \leftarrow 0$ od;
(3)  for $e \in E$ do $g(e) \leftarrow 0$; $ind(e^+) \leftarrow ind(e^+) + 1$ od;
(4)  repeat
(5)      $v \leftarrow t$;
(6)      for $i = d$ downto 1 do
(7)          choose an edge $e = uv$ and remove $e$ from $E$;
(8)          $ind(v) \leftarrow ind(v) - 1$; $g(e) \leftarrow 1$;
(9)          if $ind(v) = 0$
(10)         then append $v$ to $L$;
(11)              while $L \neq \emptyset$ do
(12)                  remove the first vertex $w$ from $L$;
(13)                  for $\{e \in E : e^- = w\}$ do
(14)                      remove $e$ from $E$; $ind(e^+) \leftarrow ind(e^+) - 1$;
(15)                      if $ind(e^+) = 0$ then append $e^+$ to $L$ fi;
(16)                  od;
(17)              od;
(18)         fi;
(19)         $v \leftarrow u$;
(20)     od;
(21) until $ind(t) = 0$
```

Obviously, each edge e is treated – and then removed – at most once during the **repeat**-loop in this procedure, so that the complexity of BLOCK01FLOW is $O(|E|)$. □

Theorem 6.5.3. *Let $N = (G,c,s,t)$ be a 0-1-network. Then the algorithm of Dinic can be used to compute a maximal 0-1-flow on N with complexity $O(|V|^{2/3}|E|)$.*

Proof. In view of Lemma 6.5.2, it suffices to show that the algorithm of Dinic needs only $O(|V|^{2/3})$ phases when it runs on a 0-1-network. Let us denote the maximal value of a flow on N by M. As the value of the flow is increased by at least 1 during each phase of the algorithm, the assertion is trivial whenever $M \le |V|^{2/3}$; thus we may suppose $M > |V|^{2/3}$. Consider the uniquely determined phase where the value of the flow is increased to a value exceeding $M - |V|^{2/3}$, and let f be the 0-1-flow on N which the algorithm had constructed in the immediately preceding phase. Then $w(f) \le M - |V|^{2/3}$, and therefore the value M' of a maximal flow on $N'(f)$ is given by $M' = M - w(f) \ge |V|^{2/3}$, by Theorem 6.3.4. Obviously, N' is likewise a 0-1-network, so that the distance $d(s,t)$ from s to t in $N'(f)$ satisfies the inequality

$$d(s,t) \le \frac{2|V|}{\sqrt{M'}} \le 2|V|^{2/3},$$

by Lemma 6.5.1. Now Lemma 6.3.11 guarantees that the distance between s and t in the corresponding auxiliary network increases in each phase, and hence the construction of f can have taken at most $2|V|^{2/3}$ phases. By our choice of f, we reach a flow value exceeding $M - |V|^{2/3}$ in the next phase, so that at most $|V|^{2/3}$ phases are necessary to increase the value of the flow step by step until it reaches M. Hence the number of phases is indeed at most $O(|V|^{2/3})$. □

In exactly the same manner, we get a further improvement of the complexity provided that the 0-1-network N satisfies the additional condition of Lemma 6.5.1 and hence the stronger inequality (6.6). Of course, this time the threshold M used in the argument should be chosen as $|V|^{1/2}$; also note that the 0-1-network $N'(f)$ likewise satisfies the additional hypothesis in Lemma 6.5.1. We leave the details to the reader and just state the final result.

Theorem 6.5.4. *Let $N = (G,c,s,t)$ be a 0-1-network. If each vertex $v \ne s,t$ of N satisfies at least one of the two conditions $d_{\rm in}(v) \le 1$ and $d_{\rm out}(v) \le 1$, then the algorithm of Dinic can be used to determine a maximal 0-1-flow on N with complexity $O(|V|^{1/2}|E|)$.* □

We close this section with a few exercises outlining some applications of 0-1-flows; they touch some very interesting questions which we will study in considerably more detail later. We shall also present several further applications of 0-1-flows in Chapter 7.

Exercise 6.5.5. A prom is attended by m girls and n boys. We want to arrange a dance where as many couples as possible should participate, but only couples who have known each other before are allowed. Formulate this task as a graph theoretical problem.

Exercise 6.5.6. Given a bipartite graph $G = (S \,\dot\cup\, T, E)$, we seek a matching of maximal cardinality in G; see Exercise 5.1.5. Show that this problem is equivalent to finding a maximal 0-1-flow on an appropriate flow network. Moreover, use the algorithms and results of this chapter to design an algorithm for this problem having complexity at most $O(|V|^{5/2})$.

The method for finding a maximal matching hinted at in Exercise 6.5.6 is basically due to Hopcroft and Karp [HoKa73], who used a rather different presentation; later, it was noticed by Evan and Tarjan [EvTa75] that this method may be viewed as a special case of the MAXFLOW-algorithm. We will meet maximal matchings quite often in this book: the bipartite case will be treated in Section 7.2, and the general case will be studied in Chapters 13 and 14.

Exercise 6.5.7. Let $G = (S \dot\cup T, E)$ be a bipartite graph. Show that the set system $(S, \mathbf{S})$ defined by

$$\mathbf{S} = \big\{X \subset S\colon \text{ there exists a matching } M \text{ with } X = \{e^- : e \in M\}\big\}$$

is a matroid; here e^- denotes that vertex incident with e which lies in S. Hint: Use the interpretation via a network given in Exercise 6.5.6 for a constructive proof of condition (3) in Theorem 5.2.1.

The result in Exercise 6.5.7 becomes even more interesting when seen in contrast to the fact that the set $\mathbf{M}$ of all matchings does not form a matroid on E; see Exercise 5.1.5.

6.6 The algorithm of Goldberg and Tarjan

This final section of Chapter 6 is devoted to a more recent algorithm for finding maximal flows which is due to Goldberg and Tarjan [GoTa88]. The algorithms we have presented so far construct a maximal flow – usually starting with the zero flow – by augmenting the flow iteratively, either along a single augmenting path or in phases where blocking flows in appropriate auxiliary networks are determined.

The algorithm of Goldberg and Tarjan is based on a completely different concept: it uses *preflows*. These are mappings for which *flow excess* is allowed: the amount of flow entering a vertex may be larger than the amount of flow leaving it. This preflow property is maintained throughout the algorithm; it is only at the very end of the algorithm that the preflow becomes a flow – which is then already maximal.

The main idea of the algorithm is to push flow from vertices with excess flow toward the sink t, using paths which are not necessarily shortest paths from s to t, but merely current estimates for such paths. Of course, it might occur that excess flow cannot be pushed forward from some vertex v; in this case, it has to be sent back to the source on a suitable path. The choice of all these paths is controlled by a certain labelling function on the vertex set. We will soon make all this precise. Altogether, the algorithm will be quite intuitive and comparatively simple to analyze. Moreover, it needs a complexity of only $O(|V|^3)$, without using any special tricks. By applying more complicated data structures, it can even be made considerably faster, as we have already noted at the end of Section 6.4.

Following [GoTa88], we define flows in this section in a formally different – although, of course, equivalent – way; this notation from [Sle80] will simplify the presentation of the algorithm. First, it is convenient to consider c and f also as functions from $V \times V$ to $\mathbb{R}$. Thus we do not distinguish between $f(e)$ and $f(u,v)$, where $e = uv$ is an edge of G; we put $f(u,v) = 0$ whenever uv is not an edge of G; and similarly for c. Then we drop the condition that flows have to be nonnegative, and define a flow $f : V \times V \to \mathbb{R}$ by the following requirements:

(1) $f(v,w) \leq c(v,w)$ for all $(v,w) \in V \times V$
(2) $f(v,w) = -f(w,v)$ for all $(v,w) \in V \times V$
(3) $\sum\limits_{u \in V} f(u,v) = 0$ for all $v \in V \setminus \{s,t\}$.

The anti-symmetry condition (2) makes sure that only one of the two edges in a pair vw and wv of antiparallel edges in G may carry a positive amount of flow.[7] Condition (2) also simplifies the formal description in one important respect: we will not have to make a distinction between forward and backward edges anymore. Moreover, the formulation of the flow conservation condition (3) is easier. The definition of the value of a flow becomes a little easier, too:

$$w(f) = \sum_{v \in V} f(v,t).$$

For an intuitive interpretation of flows in the new sense, the reader should consider only the nonnegative part of the flow function: this part is a flow as originally defined in Section 6.1. As an exercise, the reader is asked to use the antisymmetry of f to check that condition (3) is equivalent to the earlier condition (F2).

Now we define a *preflow* as a mapping $f : V \times V \to \mathbb{R}$ satisfying conditions (1) and (2) above and the following weaker version of condition (3):

(3′) $\sum\limits_{u \in V} f(u,v) \geq 0$ for all $v \in V \setminus \{s\}$.

Using the intuitive interpretation of flows, condition (3′) means that the amount of flow entering a vertex $v \neq s$ no longer has to equal the amount leaving v; it suffices if the in-flow is always at least as large as the out-flow. The value

[7]In view of condition (2) we have to assume that G is a *symmetric* digraph: if vw is an edge, wv must also be an edge of G. As noted earlier, there is no need for positive amounts of flow on two antiparallel edges: we could simply cancel flow whenever such a situation occurs.

$$e(v) \;=\; \sum_{u\in V} f(u,v)$$

is called the *flow excess* of the preflow f in v.

As mentioned before, the algorithm of Goldberg and Tarjan tries to push flow excess from some vertex v with $e(v) > 0$ forward towards t. We first need to specify which edges may be used for pushing flow. This amounts to defining an auxiliary network, similar to the one used in the classical algorithms; however, the algorithm itself does not involve an explicit construction of this network. Given a preflow f, let us define the *residual capacity* $r_f\colon V \times V \to \mathbb{R}$ as follows:

$$r_f(v,w) \;=\; c(v,w) - f(v,w).$$

If an edge vw satisfies $r_f(v,w) > 0$, we may move some flow through this edge; such an edge is called a *residual edge*. In our intuitive interpretation, this corresponds to two possible cases. Either the edge vw is a forward edge which is not yet saturated: $0 \le f(v,w) < c(v,w)$; or it is a backward edge, that is, the antiparallel edge wv is a non-void: $0 < f(w,v) \le c(w,v)$, and hence $f(v,w) = -f(w,v) < 0 \le c(v,w)$. The *residual graph* with respect to f is defined as

$$G_f = (V, E_f), \quad \text{where} \quad E_f = \{vw \in E\colon\, r_f(v,w) > 0\}.$$

As the intuitive interpretation shows, G_f really corresponds to the auxiliary network $N'(f)$ used in the classical algorithms. Now we may also introduce the labelling function mentioned before. A mapping $d : V \to \mathbb{N}_0 \cup \{\infty\}$ is called a *valid labelling* with respect to a given preflow f if the following two conditions hold:

(4) $d(s) = |V|$ and $d(t) = 0$;

(5) $d(v) \le d(w) + 1$ for all $vw \in E_f$.

The algorithm of [GoTa88] starts with some suitable preflow and a corresponding valid labelling. Usually, one saturates all edges emanating from s, and puts $d(s) = |V|$ and $d(v) = 0$ for all $v \in V \setminus \{s\}$. More precisely, the initial preflow is given by $f(s,v) = -f(v,s) = c(s,v)$ for all $v \ne s$ and $f(v,w) = 0$ for all $v, w \ne s$.

Then the algorithm executes a series of operations which we will specify later. These operations change either the preflow f (by pushing the largest possible amount of flow along a suitable residual edge) or the labelling d (by raising the label of a suitable vertex); in both cases, the labelling will always remain valid. As mentioned before, d is used to estimate shortest paths in the corresponding residual graph. More precisely, $d(v)$ is always a lower bound for the distance from v to t in G_f provided that $d(v) < |V|$; and if $d(v) > |V|$, then t is not accessible from v, and $d(v) - |V|$ is a lower bound for the distance from v to s in G_f. The algorithm terminates as soon as the preflow has become a flow (which is then actually a maximal flow).

We need one more notion to be able to write down the algorithm in its generic form. A vertex v is called *active* provided that $v \neq s, t$; $e(v) > 0$; and $d(v) < \infty$.

Algorithm 6.6.1. Let $N = (G, c, s, t)$ be a flow network on a symmetric digraph, where $c\colon V \times V \to \mathbb{R}_0^+$; that is, for $(v, w) \notin E$ we have $c(v, w) = 0$.

Procedure GOLDBERG($N; f$)

```
(1) for $(v,w) \in (V \setminus \{s\}) \times (V \setminus \{s\})$ do $f(v,w) \leftarrow 0$; $r_f(v,w) \leftarrow c(v,w)$ od;
(2) $d(s) \leftarrow |V|$;
(3) for $v \in V \setminus \{s\}$ do
(4)         $f(s,v) \leftarrow c(s,v)$; $r_f(s,v) \leftarrow 0$;
(5)         $f(v,s) \leftarrow -c(s,v)$; $r_f(v,s) \leftarrow c(v,s) + c(s,v)$;
(6)         $d(v) \leftarrow 0$;
(7)         $e(v) \leftarrow c(s,v)$
(8) od
(9) while there exists an active vertex $v$ do
(10)        choose an active vertex $v$ and execute an admissible operation
(11) od
```

In (10), one of the following operations may be used, provided that it is admissible:

Procedure PUSH($N, f, v, w; f$)

```
(1) $\delta \leftarrow \min\,(e(v), r_f(v,w))$;
(2) $f(v,w) \leftarrow f(v,w) + \delta$; $f(w,v) \leftarrow f(w,v) - \delta$;
(3) $r_f(v,w) \leftarrow r_f(v,w) - \delta$; $r_f(w,v) \leftarrow r_f(w,v) + \delta$;
(4) $e(v) \leftarrow e(v) - \delta$; $e(w) \leftarrow e(w) + \delta$.
```

The procedure PUSH($N, f, v, w; f$) is *admissible* provided that v is active, $r_f(v, w) > 0$, and $d(v) = d(w) + 1$.

Procedure RELABEL($N, f, v, d; d$)

```
(1) $d(v) \leftarrow \min\,\{d(w) + 1 \colon r_f(v,w) > 0\}$;
```

The procedure RELABEL($N, f, v, d; d$) is *admissible* provided that v is active and $r_f(v, w) > 0$ always implies $d(v) \leq d(w)$.[8]

Let us look more closely at the conditions for admissibility. If we want to push some flow along an edge vw, three conditions are required. Two of these requirements are clear: the start vertex v has to be active, so that there is positive flow excess $e(v)$ available which we might move; and vw has to be a residual edge, so that it has capacity left for additional flow. It is also not surprising that we then push along vw as much flow as possible, namely the smaller of the two amounts $e(v)$ and $r_f(v, w)$.

[8]The minimum in (1) is defined to be ∞ if there does not exist any w with $r_f(v, w) > 0$. However, we will see that this case cannot occur.

The crucial requirement is the third one, namely $d(v) = d(w) + 1$. Thus we are only allowed to push along residual edges vw for which $d(v)$ is exactly one unit larger than $d(w)$, that is, for which $d(v)$ takes its maximum permissible value; see (5) above. We may visualize this rule by thinking of water cascading down a series of terraces of different height, with the height corresponding to the labels. Obviously, water will flow down, and condition (5) has the effect of restricting the layout of the terraces so that the water may flow down only one level in each step.

Now assume that we are in an active vertex v – so that some water is left which wants to flow out – and that none of the residual edges leaving v satisfies the third requirement. In our watery analogy, v would be a sort of local sink: v is locally on the lowest possible level, and thus the water is trapped in v. It is precisely in such a situation that the RELABEL-operation becomes admissible: we miraculously raise v to a level which is just one unit higher than that of the lowest neighbor w of v in G_f; then a PUSH becomes permissible, that is, (some of) the water previously trapped in v can flow down to w. Of course, these remarks in no way constitute a proof of correctness; nevertheless, they might help to obtain a feeling for the strategy behind the Goldberg-Tarjan algorithm.

Now we turn to the formal proof which proceeds via a series of auxiliary results. This will allow us to show that Algorithm 6.6.1 constructs a maximal flow on N in finitely many steps, no matter in which order we select the active vertices and the admissible operations. This is in remarkable contrast to the situation for the algorithm of Ford and Fulkerson; recall the discussion in Section 6.1. To get better estimates for the complexity, however, we will have to specify appropriate strategies for the choices to be made.

We begin by showing that the algorithm is correct under the assumption that it terminates at all. Afterwards, we will estimate the maximal number of admissible operations executed during the **while**-loop and use this result to show that the algorithm really is finite. Our first lemma is just a simple but important observation; it states a result which we have already emphasized in our informal discussion.

Lemma 6.6.2. *Let f be a preflow on N, d a valid labelling on V with respect to f, and v an active vertex. Then either a* PUSH*-operation or a* RELABEL*-operation is admissible for v.*

Proof. As d is valid, we have $d(v) \le d(w) + 1$ for all w with $r_f(v, w) > 0$. If PUSH(v, w) is not admissible for any w, we must even have $d(v) \le d(w)$ for all w with $r_f(v, w) > 0$, as d takes only integer values. But then RELABEL is admissible. □

Lemma 6.6.3. *During the execution of Algorithm 6.6.1, f always is a preflow and d always is a valid labelling (with respect to f).*

Proof. We use induction on the number k of admissible operations already executed. The assertion holds for the induction basis $k = 0$: obviously, f is initialized as a preflow in steps (4) and (5); and the labelling d defined in (2) and (6) is valid for f, since $d(v) = 0$ for $v \neq s$ and since all edges sv have been saturated in step (4); also, the residual capacities and the flow excesses are clearly initialized correctly in steps (4), (5), and (7).

For the induction step, suppose that the assertion holds after k operations have been executed. Assume first that the next operation is a PUSH(v,w). It is easy to check that f remains a preflow, and that the residual capacities and the flow excesses are updated correctly. Note that the labels are kept unchanged, and that vw and wv are the only edges for which f has changed. Hence we only need to worry about these two edges in order to show that d is still valid. By definition, $vw \in E_f$ before the PUSH. Now vw might be removed from the residual graph G_f (which happens if it is saturated by the PUSH); but then the labelling stays valid trivially. Now consider the antiparallel edge wv. If this edge already is in G_f, there is nothing to show. Thus assume that wv is added to G_f by the PUSH; again, d stays valid, since the admissibility conditions for the PUSH(v,w) require $d(w) = d(v) - 1$.

It remains to consider the case where the next operation is a RELABEL(v). Then the admissibility requirement is $d(v) \leq d(w)$ for all vertices w with $r_f(v,w) > 0$. As $d(v)$ is increased to the minimum of all the $d(w)+1$, the condition $d(v) \leq d(w)+1$ holds for all w with $r_f(v,w) > 0$ after this change; all other labels remain unchanged, so that the new labelling d is still valid for f. □

Lemma 6.6.4. *Let f be a preflow on N and d a valid labelling with respect to f. Then t is not accessible from s in the residual graph G_f.*

Proof. Suppose there exists a path

$$W: \quad s = v_0 \longrightarrow v_1 \longrightarrow \dots \longrightarrow v_m = t$$

in G_f. As d is a valid labelling, $d(v_i) \leq d(v_{i+1}) + 1$ for $i = 0, \dots, m-1$. Hence $d(s) \leq d(t) + m < |V|$, since $d(t) = 0$ and since the path can have length at most $|V| - 1$. But $d(s) = |V|$, by the validity of d, and we have reached a contradiction. □

Theorem 6.6.5. *If Algorithm 6.6.1 terminates with all labels finite, then the preflow f constructed is in fact a maximal flow on N.*

Proof. By 6.6.2, the algorithm can only terminate when there are no more active vertices. As all labels are finite by hypothesis, $e(v) = 0$ has to hold for each vertex $v \neq s,t$; hence the preflow constructed by the final operation is indeed a flow on N. By Lemma 6.6.4, there is no path from s to t in G_f, so that there is no augmenting path from s to t with respect to f. Now the assertion follows from Theorem 6.1.3. □

It remains to show that the algorithm indeed terminates and that the labels stay finite throughout. We need two further lemmas.

Lemma 6.6.6. *Let f be a preflow on N. If v is a vertex with positive flow excess $e(v)$, then s is accessible from v in G_f.*

Proof. We denote the set of vertices accessible from v in G_f (via a directed path) by S , and put $T := V \setminus S$. Then $f(u,w) \le 0$ for all vertices u,w with $u \in T$ and $w \in S$, since

$$0 \;=\; r_f(w,u) \;=\; c(w,u) - f(w,u) \;\ge\; 0 + f(u,w).$$

Using the antisymmetry of f, we get

$$\begin{aligned}\sum_{w\in S} e(w) \;&=\; \sum_{u\in V, w\in S} f(u,w)\\ &=\; \sum_{u\in T, w\in S} f(u,w) + \sum_{u,w\in S} f(u,w)\\ &=\; \sum_{u\in T, w\in S} f(u,w) \;\le\; 0.\end{aligned}$$

Now the definition of a preflow requires $e(w) \ge 0$ for all $w \ne s$. But $e(v) > 0$, and hence $\sum_{w\in S} e(w) \le s$ implies $s \in S$. □

Lemma 6.6.7. *Throughout Algorithm 6.6.1, $d(v) \le 2|V| - 1$ for all $v \in V$.*

Proof. Obviously, the assertion holds after the initialization phase in steps (1) to (8). The label $d(v)$ of a vertex v can only be changed by an operation RELABEL(v), and such an operation is admissible only if v is active. In particular, $v \ne s,t$, so that the claim is trivial for s and t; moreover, $e(v) > 0$. By Lemma 6.6.6, there exists a directed path

$$W\colon\;\; v = v_0 \;—\; v_1 \;—\; \dots \;—\; v_m = s$$

in the residual graph G_f. Since d is a valid labelling, $d(v_i) \le d(v_{i+1}) + 1$ for $i = 0,\dots,m-1$. Now W has length at most $|V| - 1$, and we conclude

$$d(v) \;\le\; d(s) + m \;\le\; d(s) + |V| - 1 \;=\; 2|V| - 1. \qquad \square$$

Lemma 6.6.8. *During the execution of Algorithm 6.6.1, at most* $2|V| - 1$ RELABEL-*operations occur for any given vertex $v \ne s,t$. Hence the total number of* RELABEL-*operations is at most* $(2|V| - 1)(|V| - 2) < 2|V|^2$.

Proof. Each RELABEL(v) increases $d(v)$. Since $d(v)$ is bounded by $2|V| - 1$ throughout the entire algorithm (see Lemma 6.6.7), the assertion follows. □

It is more difficult to estimate the number of PUSH-operations. We need to distinguish two cases: a PUSH(v,w) will be called a *saturating* PUSH if $r_f(v,w) = 0$ holds afterwards (that is, for $\delta = r_f(v,w)$ in step (1) of the PUSH), and a *non-saturating* PUSH otherwise.

Lemma 6.6.9. *During the execution of Algorithm 6.6.1, fewer than $|V||E|$ saturating* PUSH-*operations occur.*

Proof. By definition, any PUSH(v, w) requires $vw \in E_f$ and $d(v) = d(w) + 1$. If the PUSH is saturating, a further PUSH(v, w) can only occur after an intermediate PUSH(w, v), since we have $r_f(v, w) = 0$ after the saturating PUSH(v, w). Note that no PUSH(w, v) is admissible before the labels have been changed in such a way that $d(w) = d(v) + 1$ holds; hence $d(w)$ must have been increased by at least 2 units before the PUSH(w, v). Similarly, no further PUSH(v, w) can become admissible before $d(v)$ has also been increased by at least 2 units. In particular, $d(v) + d(w)$ has to increase by at least 4 units between any two consecutive saturating PUSH(v, w)-operations.

On the other hand, $d(v) + d(w) \geq 1$ holds as soon as the first PUSH from v to w or from w to v is executed. Moreover, $d(v), d(w) \leq 2|V| - 1$ throughout the algorithm, by Lemma 6.6.7; hence $d(v) + d(w) \leq 4|V| - 2$ holds when the last PUSH-operation involving v and w occurs. Therefore there are at most $|V| - 1$ saturating PUSH(v, w)-operations, so that the the total number of saturating PUSH-operations cannot exceed $(|V| - 1)|E|$. □

Lemma 6.6.10. *During the execution of Algorithm 6.6.1, there are at most $2|V|^2|E|$ non-saturating* PUSH-*operations.*

Proof. Let us introduce the *potential*

$$\Phi = \sum_{v \text{ active}} d(v)$$

and investigate its development during the course of Algorithm 6.6.1. After the initialization phase, $\Phi = 0$; and at the end of the algorithm, we have $\Phi = 0$ again.

Note that any non-saturating PUSH(v, w) decreases Φ by at least one unit: because $r_f(v, w) > e(v)$, the vertex v becomes inactive so that Φ is decreased by $d(v)$ units; and even if the vertex w has become active due to the PUSH, Φ is increased again by only $d(w) = d(v) - 1$ units, as the PUSH must have been admissible. Similarly, any saturating PUSH(v, w) increases Φ by at most $2|V| - 1$, since the label of the vertex w – which might again have become active due to this PUSH – satisfies $d(w) \leq 2|V| - 1$, by Lemma 6.6.7.

Let us put together what these observations imply for the entire algorithm. The saturating PUSH-operations increase Φ by at most $(2|V| - 1)|V||E|$ units altogether, by Lemma 6.6.9; and the RELABEL-operations increase Φ by at most $(2|V| - 1)(|V| - 2)$ units, by Lemma 6.6.7. Clearly, the value by which Φ is increased over the entire algorithm must be the same as the value by which it is decreased again. As this happens for the non-saturating PUSH-operations, we obtain an upper bound of $(2|V| - 1)(|V||E| + |V| - 2)$ for the total number of non-saturating PUSH-operations. Now the bound in the assertion follows easily, using that G is connected. □

The preceding lemmas combine to give the desired result:

Theorem 6.6.11. *Algorithm 6.6.1 terminates after at most $O(|V|^2|E|)$ admissible operations (with a maximal flow).* □

The precise complexity of Algorithm 6.6.1 depends both on the way the admissible operations are implemented and on the order in which they are applied in the **while**-loop. In any case, the running time will be polynomial. We shall treat two variants which lead to particularly good results; they differ only in the manner in which they select the active vertex in step (10). Both variants use the obvious strategy not to change the active vertex v unnecessarily, but to stick with v until

- either $e(v) = 0$;
- or all edges incident with v have already been used for a PUSH(v, w), as far as this is possible, and a RELABEL(v) has occurred afterwards.

To implement this strategy, we use incidence lists. For each vertex v, there always is a distinguished *current edge* in its incidence list A_v (which may be implemented via a pointer). Initially, this edge is just the first edge of A_v; thus we assume A_v to have a fixed order. In the following algorithm, the active vertices are selected according to the rule *first in first out* – which explains its name.

Algorithm 6.6.12 (FIFO preflow push algorithm). Let $N = (G, c, s, t)$ be a flow network, where G is a symmetric digraph given by incidence lists A_v. Moreover, Q denotes a queue and *rel* a Boolean variable.

Procedure FIFOFLOW$(N; f)$

```
(1) for $(v,w) \in (V \setminus \{s\}) \times (V \setminus \{s\})$ do $f(v,w) \leftarrow 0$; $r_f(v,w) \leftarrow c(v,w)$ od;
(2) $d(s) \leftarrow |V|$; $Q \leftarrow \emptyset$;
(3) for $v \in V \setminus \{s\}$ do
(4)         $f(s,v) \leftarrow c(s,v)$; $r_f(s,v) \leftarrow 0$;
(5)         $f(v,s) \leftarrow -c(s,v)$; $r_f(v,s) \leftarrow c(v,s) + c(s,v)$;
(6)         $d(v) \leftarrow 0$; $e(v) \leftarrow c(s,v)$;
(7)         make the first edge in $A_v$ the current edge;
(8)         if $e(v) > 0$ and $v \neq t$ then append $v$ to $Q$ fi
(9) od
(10) while $Q \neq \emptyset$ do
(11)        remove the first vertex $v$ from $Q$; rel $\leftarrow$ false;
(12)        repeat
(13)            let $vw$ be the current edge in $A_v$;
(14)            if $r_f(v,w) > 0$ and $d(v) = d(w) + 1$
(15)            then PUSH$(N, f, v, w; f)$;
(16)                if $w \notin Q$ and $w \neq s, t$ then append $w$ to $Q$ fi
(17)            fi
(18)            if $e(v) > 0$ then
(19)                if $vw$ is not the last edge in $A_v$
(20)                then choose the next edge in $A_v$ as current edge
```

```
(21)                 else RELABEL($N, f, v, d; d$); rel ← true;
(22)                      make the first edge in $A_v$ the current edge
(23)                 fi
(24)            fi
(25)       until $e(v) = 0$ or rel = true;
(26)       if $e(v) > 0$ then append $v$ to $Q$ fi
(27) od
```

The reader may show that Algorithm 6.6.12 is indeed a special case of Algorithm 6.6.1; this amounts to checking that RELABEL(v) is called only when no PUSH along an edge starting in v is admissible. By Theorem 6.6.11, the algorithm terminates with a maximal flow on N. The following result giving its complexity is due to Goldberg and Tarjan [GoTa88].

Theorem 6.6.13. *Algorithm 6.6.12 determines with complexity $O(|V|^3)$ a maximal flow on N.*

Proof. Obviously, the initialization phase in steps (1) to (9) has complexity $O(|E|)$. In order to analyze the complexity of the **while**-loop, we divide the course of the algorithm into *phases*.[9] Phase 1 consists of the execution of the **repeat**-loop for those vertices which were originally appended to Q, that is, when Q was initialized in step (8). If phase i is already defined, phase $i + 1$ consists of the execution of the **repeat**-loop for those vertices which were appended to Q during phase i. We claim that there are at most $O(|V|^2)$ phases.

By Lemma 6.6.8, there are at most $O(|V|^2)$ phases involving a RELABEL. It remains to establish the same bound for phases without a RELABEL. For this purpose, we take the same approach as in the proof of Lemma 6.6.10: we define a potential and investigate its development during the course of the algorithm. This time, we let Φ be the maximum value of the labels $d(v)$, taken over all active vertices v. Let us consider how Φ changes during a phase not involving any RELABEL-operations. Then, for each active vertex v, excess flow is moved to vertices w with label $d(v) - 1$ until we finally reach $e(v) = 0$, so that v ceases to be active. Of course, Φ cannot be increased by these operations; and at the end of such a phase – when all originally active vertices v have become inactive – Φ has actually decreased by at least one unit. Hence, if Φ remains unchanged or increases during a phase, at least one RELABEL-operation must occur during this phase; we already noted that there are at most $O(|V|^2)$ phases of this type. As $\Phi = 0$ holds at the beginning as well as at the end of the algorithm, at most $O(|V|^2)$ decreases of Φ can occur. Hence there are indeed at most $O(|V|^2)$ phases not involving a RELABEL.

We can now estimate the number of steps required for all PUSH-operations; note that an individual PUSH needs only $O(1)$ steps. Hence we want to show

[9]In the original literature, the phases are called *passes over* Q, which seems somewhat misleading.

that there are only $O(|V|^3)$ PUSH-operations. In view of Lemma 6.6.9, it suffices to consider non-saturating PUSH-operations. Note that the **repeat**-loop for a vertex v is aborted as soon as a non-saturating PUSH(v, w) occurs; see step (25). Clearly, at most $O(|V|)$ vertices v are investigated during a phase, so that there are at most $O(|V|)$ non-saturating PUSH-operations during each phase. Now our result on the number of phases gives the assertion.

It remains to estimate how often each edge is examined during the **while**-loop. Consider the edges starting in a specified vertex v. During a **repeat**-loop involving v, the current edge e runs through (part of) the incidence list A_v of v. More precisely, the pointer is moved to the next edge whenever treating e leaves v with flow excess $e(v) > 0$; and the pointer is returned to the first edge only when a RELABEL(v) occurs. By Lemma 6.6.8, each vertex v is relabeled at most $2|V| - 1$ times, so that the incidence list A_v of v is examined only $O(|V|)$ times during the entire algorithm. (Note that this estimate also includes the complexity of the RELABEL-operations: each RELABEL(v) also amounts to looking through all edges in A_v.) Hence we obtain altogether $O(|V||A_v|)$ examinations of the edges starting in v; summing this over all vertices shows that the edge examinations and the RELABEL-operations only contribute $O(|V||E|)$ to the complexity of the algorithm. □

Examples which show that the FIFO-algorithm might indeed need $O(|V|^3)$ steps are provided in [ChMa89].

We now turn to our second variation of Algorithm 6.6.1. This time, we always choose an active vertex which has the maximal label among all the active vertices. To implement this strategy, we use a priority queue with priority function d instead of an ordinary queue. This variant was likewise suggested by Goldberg and Tarjan [GoTa88].

Algorithm 6.6.14 (highest label preflow push algorithm). Let $N = (G, c, s, t)$ be a flow network, where G is a symmetric digraph given by incidence lists A_v. Moreover, let Q be a priority queue with priority function d, and *rel* a Boolean variable.

Procedure HLFLOW$(N; f)$

```
 (1) for $(v,w) \in (V \setminus \{s\}) \times (V \setminus \{s\})$ do $f(v,w) \leftarrow 0$; $r_f(v,w) \leftarrow c(v,w)$ od;
 (2) $d(s) \leftarrow |V|$; $Q \leftarrow \emptyset$;
 (3) for $v \in V \setminus \{s\}$ do
 (4)         $f(s,v) \leftarrow c(s,v)$; $r_f(s,v) \leftarrow 0$;
 (5)         $f(v,s) \leftarrow -c(s,v)$; $r_f(v,s) \leftarrow c(v,s) + c(s,v)$;
 (6)         $d(v) \leftarrow 0$; $e(v) \leftarrow c(s,v)$;
 (7)         make the first edge in $A_v$ the current edge;
 (8)         if $e(v) > 0$ and $v \neq t$ then insert $v$ into $Q$ with priority $d(v)$ fi;
 (9) od
(10) while $Q \neq \emptyset$ do
(11)         remove a vertex $v$ of highest priority $d(v)$ from $Q$; rel $\leftarrow$ false;
(12)         repeat
```

```
(13)                let $vw$ be the current edge in $A_v$;
(14)                if $r_f(v, w) > 0$ and $d(v) = d(w) + 1$ then
(15)                      PUSH$(N, f, v, w; f)$;
(16)                      if $w \notin Q$ and $w \neq s, t$ then insert $w$ into $Q$
                          with priority $d(w)$ fi
(17)                fi
(18)                if $e(v) > 0$ then
(19)                      if $vw$ is not the last edge in $A_v$
(20)                      then choose the next edge in $A_v$ as current edge;
(21)                      else RELABEL$(N, f, v, d; d)$; rel $\leftarrow$ true;
(22)                           make the first edge in $A_v$ the current edge;
(23)                      fi
(24)                fi
(25)          until $e(v) = 0$ or rel = true;
(26)          if $e(v) > 0$ then insert $v$ into $Q$ with priority $d(v)$ fi;
(27) od
```

Goldberg and Tarjan proved that Algorithm 6.6.14 has a complexity of at most $O(|V|^3)$; this estimate was improved by Cheriyan and Maheshwari [ChMa89] as follows.

Theorem 6.6.15. *Algorithm 6.6.14 determines a maximal flow on N with complexity* $O(|V|^2|E|^{1/2})$.

Proof.[10] As in the proof of Theorem 6.6.13, the main problem is to establish the necessary bound for the number of non-saturating PUSH-operations; all other estimates can be done as before. Note here that $O(|V||E|)$ – that is, the bound for the saturating PUSH-operations provided by Lemma 6.6.9 – is indeed dominated by $O(|V|^2|E|^{1/2})$.

As in the proof of Theorem 6.6.13, we divide the algorithm into phases; but this time, a phase consists of all operations occurring between two consecutive RELABEL-operations. The *length* l_i of the i-th phase is defined as the difference between the values of $d_{\max}$ at the beginning and at the end of the phase, where $d_{\max}$ denotes the maximal label $d(v)$ over all active vertices v. Note that $d_{\max}$ decreases monotonically during a phase; immediately after the end of the phase, when a RELABEL-operation is executed, $d_{\max}$ increases again.

We claim that the sum of the lengths l_i over all the phases is at most $O(|V|^2)$. To see this, it suffices to show that the increase of $d_{\max}$ during the entire algorithm is at most $O(|V|^2)$. But this is an immediate consequence of Lemma 6.6.7, since the label $d(v)$ increases monotonically for each vertex v and is always bounded by $2|V| - 1$.

[10]As the proof of Theorem 6.6.15 is rather technical, the reader might decide to skip it at first reading. However, it does involve a useful method, which we have not seen before.

The basic idea of the proof is to partition the non-saturating PUSH-operations in a clever way. For this purpose, we call a non-saturating PUSH(u, v)-operation *special*[11] if it is the first PUSH-operation on the edge uv following a RELABEL(u)-operation.

Now consider a non-saturating, nonspecial PUSH-operation PUSH(z, w). We try to construct (in reverse order) a directed path T_w with end vertex w which consists entirely of edges for which the last non-saturating PUSH-operation executed was a nonspecial one. Suppose we have reached a vertex $u \neq w$, and let the last edge constructed for T_w be uv. Thus the last PUSH(u, v) was a non-saturating nonspecial PUSH. Before this PUSH-operation was executed, we had $e(u) > 0$. We want to consider the last PUSH-operation PUSH(y, u) executed before this PUSH(u, v). It is possible that no such PUSH-operation exists;[12] then we simply end the construction of T_w at the vertex u. We also terminate the construction of T_w at u if the last PUSH(y, u) was saturating or special. Otherwise we replace u by y and continue in the same manner.

Note that our construction has to terminate provided that T_w is indeed a path, which just amounts to showing that no cycle can occur during the construction. But this is clear, as PUSH-operations may only move flow towards vertices with lower labels; hence no cycle can arise, unless a RELABEL occurred for one of the vertices that we have reached; and this is not possible by our way of construction. We call the sequence of non-saturating PUSH-operations corresponding to such a path T_w a *trajectory* with *originating edge* xy, if xy is the unique edge encountered at the end of the construction of T_w for which either a saturating or a special PUSH had been executed (so that the construction was terminated at y); in the exceptional case mentioned above, we consider the edge su to be the originating edge of T_w. By definition, the originating edge is *not* a part of T_w: the trajectory starts at the head of this edge.

We claim that the whole of the nonspecial non-saturating PUSH-operations can be partitioned into such trajectories. Actually we require a somewhat stronger statement later: two trajectories containing PUSH-operations on edges which are current edges simultaneously (in different adjacency lists) cannot have any vertices in common, with the exception of possible common end vertices. We may assume w.l.o.g. that the two trajectories correspond to paths T_w and $T_{w'}$ for which (at a certain point of the algorithm) both $e(w) > 0$ and $e(w') > 0$ hold. Let xy and $x'y'$ be the originating edges of the two trajectories. Now suppose that u is a common vertex contained in both trajectories, where $u \neq y, y', w, w'$. We may also choose u to be the last such vertex. Let uv and uv' be the edges occurring in T_w and $T_{w'}$, respectively. We may assume that PUSH(u, v) was executed before PUSH(u, v'); note that

[11] In the original paper, the term *non-zeroing* is used instead.

[12] Note that this case occurs if and only if the entire flow excess in u comes directly from s, that is, if it was assigned to u during the initialization phase.

$v \neq v'$ by our choice of u. Then PUSH(u, v') can only have been executed after some flow excess was moved to u again by some PUSH(z, u)-operation. Then the condition $d(z) = d(u) + 1$ must have been satisfied; this means that there must have been a RELABEL(z)-operation executed before, since the active vertex is always a vertex having a maximal label and u was already active before z. Thus the PUSH(z, u)-operation was a special PUSH and the construction of $T_{w'}$ should have been terminated at the vertex u with the originating edge zu, contradicting the choice of u. Hence any two trajectories are always disjoint, establishing our claim.

Let us call a trajectory *short* if it consists of at most K operations; here K is a parameter whose value we will fix later in an optimal manner. As the originating edge of any trajectory comes from a saturating or a special PUSH-operation or – in the exceptional case – from the initialization of the preflow, the number of short trajectories can be bounded by $O(|V||E|)$ as follows. By Lemma 6.6.9, there are at most $O(|V||E|)$ saturating PUSH-operations. Also, there are at most $O(|V||E|)$ special PUSH-operations, since there are at most $O(|V|)$ RELABEL-operations per vertex by Lemma 6.6.8 and since a special PUSH(u, v) has to be preceded by a RELABEL(u). Hence all the short trajectories together may contain at most $O(K|V||E|)$ non-saturating PUSH-operations.

Now we have to examine the *long* trajectories, that is, those trajectories which contain more than K operations. Recall that any two trajectories containing PUSH-operations on edges which are current simultaneously cannot contain any common vertices (excepting the end vertices). Hence, at any point during the course of the algorithm, there are at most $|V|/K$ long trajectories which contain a PUSH-operation current at this point. In particular, there are at most $|V|/K$ long trajectories meeting a given phase of the algorithm. By definition, no phase contains a RELABEL-operation, and PUSH(u, v) can only be executed for $d(u) = d(v)+1$. Hence there can be only l_i non-saturating PUSH-operations per trajectory in any given phase of length l_i, as l_i is the difference between the values of $d_{\max}$ at the beginning and at the end of phase i: if PUSH-operations have been executed on a path of length c during phase i, the maximal label must have been decreased by c at least. As we already know that the sum of all lengths l_i is $O(|V|^2)$, all the long trajectories together may contain at most $O(|V|^3/K)$ non-saturating PUSH-operations.

Altogether, the entire algorithm uses at most $O(K|V||E|) + O(|V|^3/K)$ non-saturating PUSH-operations. Now we get the optimal bound on the complexity by *balancing* these two terms, that is, by choosing K in such a way that $K|V||E| = |V|^3/K$. This gives $K = |V||E|^{-q/2}$ and leads to the complexity stated in the assertion. □

Balancing techniques as in the proof above are very useful for analyzing the complexity of algorithms. Cheriyan and Maheshwari have also shown that the bound in Theorem 6.6.15 is best possible: there exist families of networks for which Algorithm 6.6.14 indeed needs $O(|V|^2|E|^{1/2})$ steps. From a practical point of view, Algorithm 6.6.14 is one of the best methods known for determining maximal flows; see the empirical studies mentioned at the end of Section 6.4.

Example 6.6.16. Let us apply Algorithm 6.6.14 to the flow network of Figure 6.2; compare Example 6.2.3. Where a choice has to be made, we use alphabetical order, as usual. We summarize several operations in each figure, namely at least one RELABEL-operation together with all PUSH-operations following it (that is, together with the subsequent phase); sometimes we even display two or three shorter phases in one figure. We will not draw pairs of antiparallel edges: we include only those edges which carry a nonnegative flow, in accordance with the intuitive interpretation discussed at the beginning of this section; this simplifies the figures.

Moreover, we give the capacities in parentheses only in the first figure (after the initialization phase). The numbers in the subsequent figures always give the values as they are after the last operation executed. So the number written on some edge e is the value $f(e)$ of the current preflow f; here all new values coming from a saturating PUSH-operation are framed, whereas all new values coming from a non-saturating PUSH-operation are circled. Additionally, the vertices v are labelled with the pair $(d(v), e(v))$; that is, we display the valid label and the flow excess in v. For the vertices s and t, only the (never changing) valid label is given; by definition, these two vertices are never active.

Below each figure, we also list the RELABEL- and PUSH-operations which have occurred and the queue Q containing the active vertices as it looks after all the operations have been executed. Note that the maximal flow constructed by HLFLOW given in Figure 6.31 differs from the maximal flow of Figure 6.12: the edge ct does not carry any flow, and the value of the flow on the edges cf and ft is larger accordingly.

Exercise 6.6.17. Apply Algorithm 6.6.12 to the flow network of Figure 6.2 (with the usual convention about alphabetical order), and compare the number of RELABEL- and PUSH-operations necessary with the corresponding numbers for Algorithm 6.6.14; see Example 6.6.16.

For a discussion of the implementation of various PUSH- and RELABEL-algorithms, see [ChGo95].

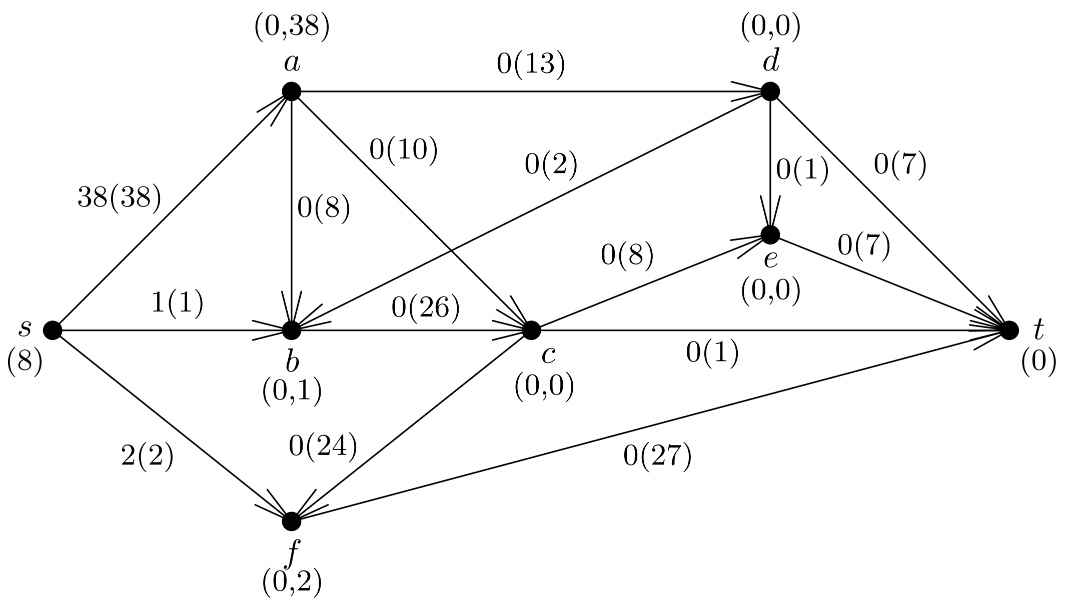

Fig. 6.24. Initialization: $Q = (a, b, f)$

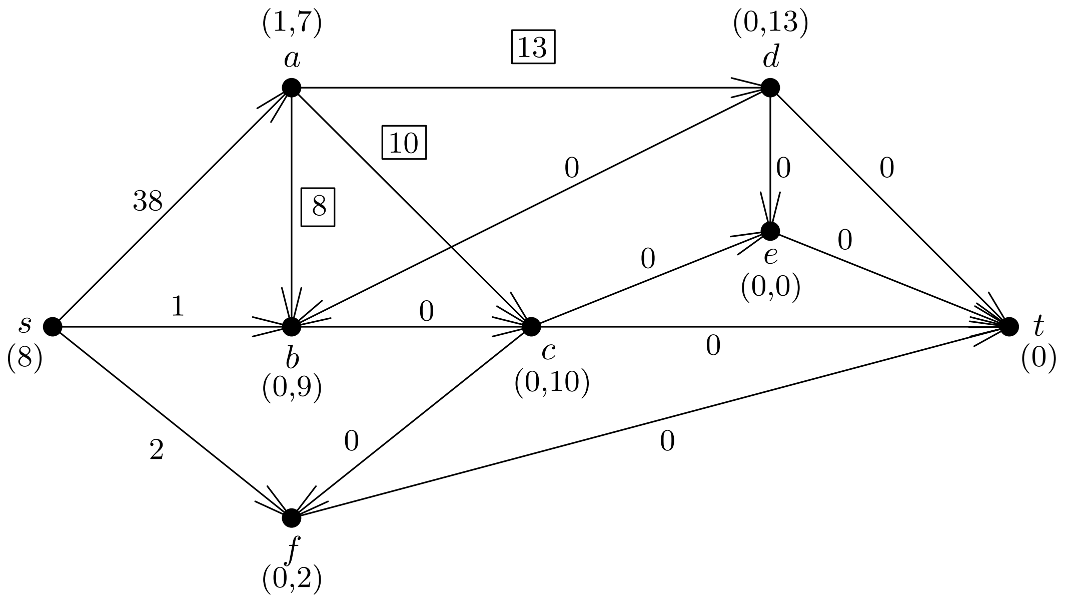

Fig. 6.25. RELABEL(a), $Q = (a, b, c, d, f)$

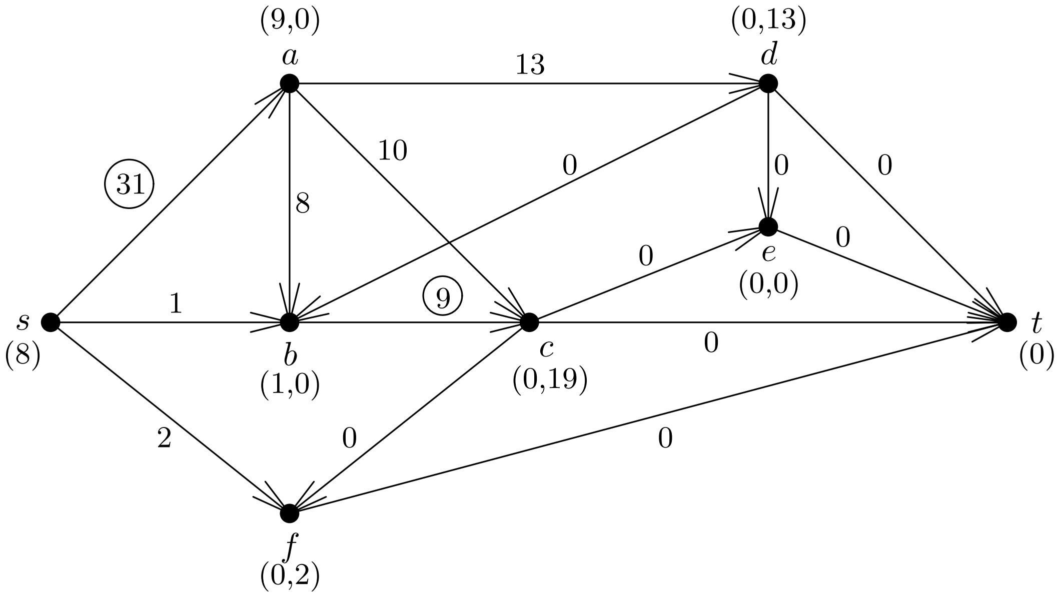

Fig. 6.26. RELABEL(a), PUSH(a, s), RELABEL(b), PUSH(b, c), $Q = (c, d, f)$

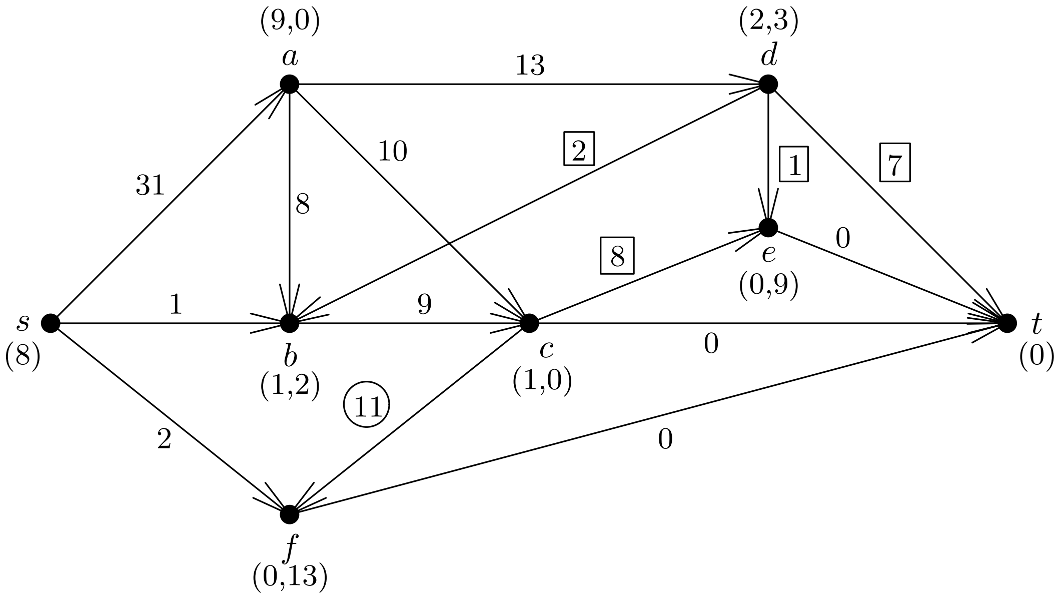

Fig. 6.27. RELABEL(c), PUSH(c, f), RELABEL(d), PUSH(d, b), PUSH(d, e), PUSH(d, t), RELABEL(d), $Q = (d, b, e, f)$

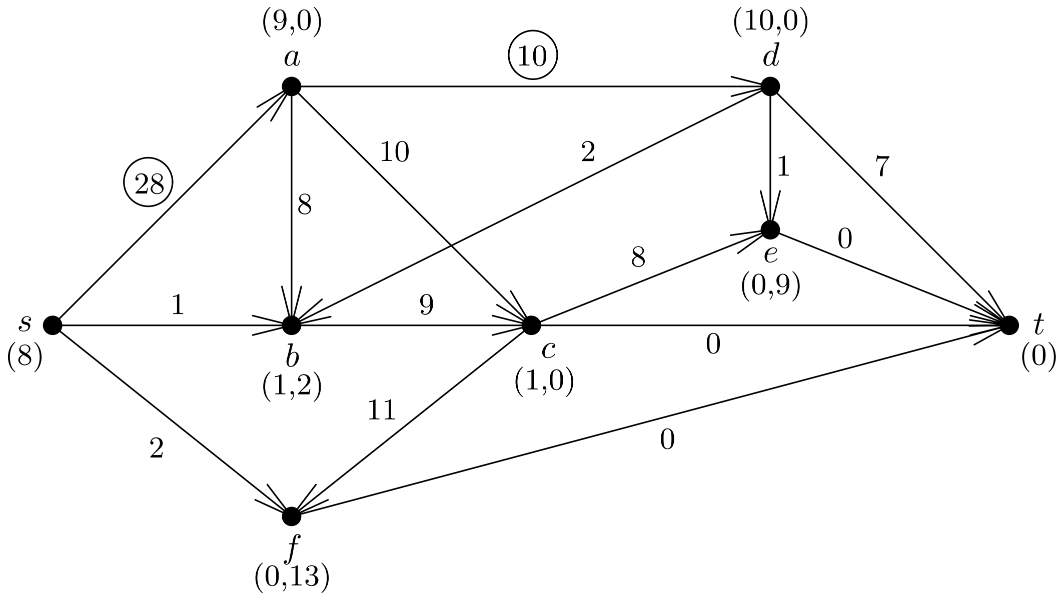

Fig. 6.28. RELABEL(d), PUSH(d, a), PUSH(a, s), $Q = (b, e, f)$

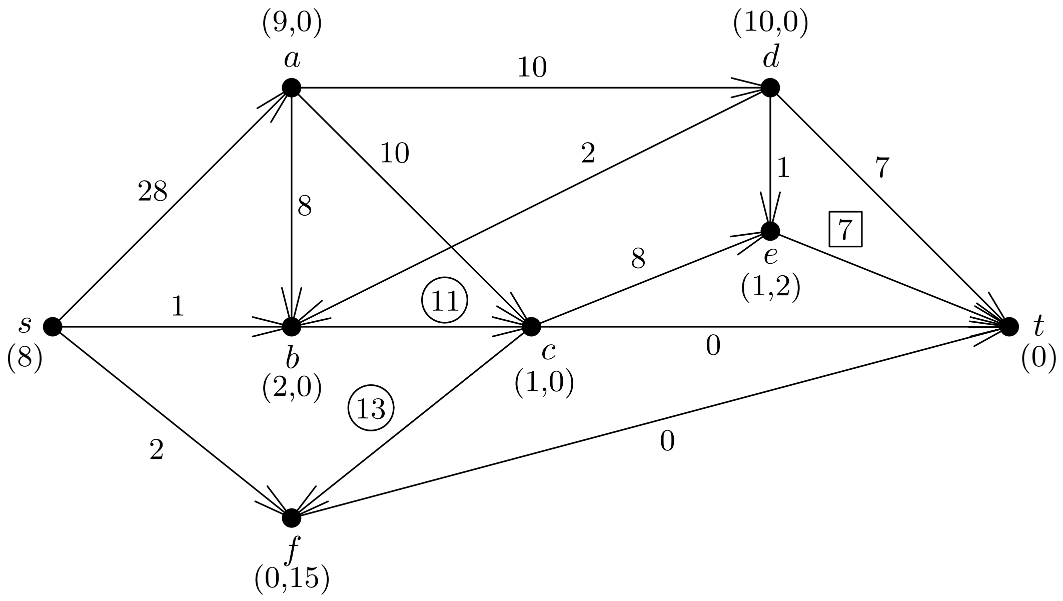

Fig. 6.29. RELABEL(b), PUSH(b, c), PUSH(c, f), RELABEL(e), PUSH(e, f), $Q = (e, f)$

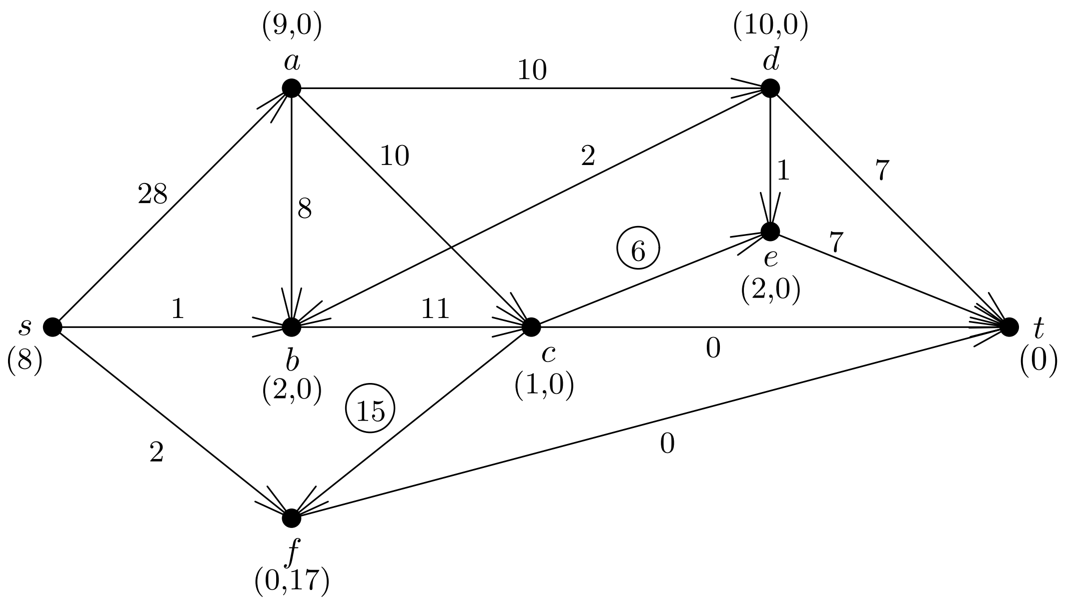

Fig. 6.30. RELABEL(e), PUSH(e, c), PUSH(c, f), $Q = (f)$

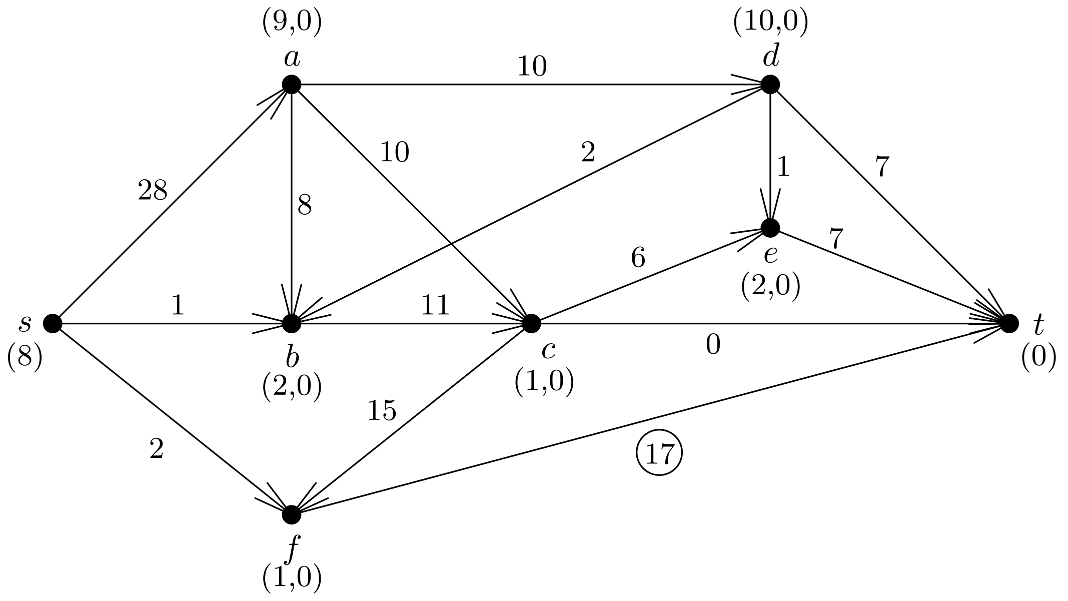

Fig. 6.31. RELABEL(f), PUSH(f, t), $Q = \emptyset$