4 Flows and Cuts

Flows and cuts allow for modeling and solving a surprisingly large set of real-world problems. Furthermore, they can be used to answer many foundational questions related to graphs, for example, whether a graph admits k arc-disjoint paths between two vertices s and t. We consider flows in directed graphs. An undirected version can easily be derived from this. Throughout this chapter, we denote by $G = (V, A)$ a directed graph with arc capacities $u\colon A \to \mathbb{Z}_{\geq 0}$. Notice that we assume that arc capacities are integral. Non-negative rational arc-capacities can be transformed into integral ones by scaling, leading to properties and results analogous to the ones we discuss in the following.

4.1 Basic notions and relations

We start by defining the notion of an s-t flow, where $s, t \in V, s \neq t$. Intuitively, one can imagine G to be a network of water pipes where one wants to continuously send water from s to t. Each arc $a = (v, w) \in A$ is a pipe that can have a maximum throughput of $u(a)$ from vertex v to w. The quantity $u(a)$ is called the *capacity* of a. Assume that one can decide how water entering a vertex v is sent out on the outgoing arcs $\delta^+(v)$, subject to respecting the arc capacities u. An s-t flow then simply captures a possible steady-state mode for continuously sending water from s to t, by indicating how much water flows on each arc per time unit.

Definition 4.1: s-t flow / flow

Let $s, t \in V$, $s \neq t$. An s-t flow in G is a function $f\colon A \to \mathbb{R}_{\geq 0}$ satisfying the following conditions.

(i) *Capacity constraints*: $f(a) \leq u(a)\ \forall a \in A$.

(ii) *Balance constraints*: for $v \in V$,

$$f(\delta^+(v)) - f(\delta^-(v)) \begin{cases} = 0 & \text{if } v \in V \setminus \{s, t\}\ , \\ \geq 0 & \text{if } v = s\ , \\ \leq 0 & \text{if } v = t\ . \end{cases}$$

The *value* of a flow f is $\nu(f) \coloneqq f(\delta^+(s)) - f(\delta^-(s))$.

The capacity constraints assure that the flow on each arc does not exceed the capacity. For a vertex $v \in V$, we call the quantity $f(\delta^+(v))$ the *outflow* of v. Similarly, $f(\delta^-(v))$ is called the *inflow* of v. Hence, the balance constraints can be rephrased as imposing that for each vertex $v \in V \setminus \{s, t\}$, the outflow of v equals the inflow into v. The balance constraints guarantee

that whatever flows into some vertex $v \in V \setminus \{s,t\}$ also has to flow out of it. Furthermore, the outflow of s is at least the inflow into s, and the outflow of t is at most the inflow into t. In other words, no flow is created or destroyed at any vertex $v \in V \setminus \{s,t\}$. Furthermore, flow can be created at vertex s, i.e., there may be more outgoing flow than incoming one, and flow may be absorbed at vertex t. Therefore, the vertex s is often called the *source* and t is called the *sink* of the flow network. Also note that the value of a flow f in the analogy based on water pipes indicates how much flow is sent from s to t per time unit. Figure 4.1 shows an example of an s-t flow.

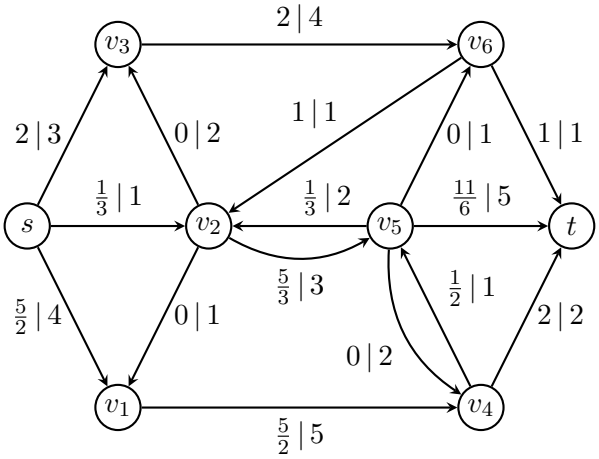


Figure 4.1: Example of an s-t flow. Next to each arc $a \in A$, one can find the value of the flow on a, i.e., $f(a)$, followed by its capacity $u(a)$, written as $f(a)\,|\,u(a)$. The value of the shown flow f is $\nu(f) = \frac{29}{6}$.

Notice that in the water pipe analogy we used, flow values correspond to how much water is sent *per time unit*, and the capacities limit the throughput, which is also measured in terms of water *per time unit*. Nevertheless, in the context of network flows, we simply talk about the *flow value* on an arc, and the *flow sent* from s to t, without repeatedly referring to a time component that may underlie the problem.

An s-t flow in G is called *maximum* if it has maximum value among all s-t flows in G. This leads to the maximum s-t flow problem, or simply the maximum flow problem, which is formally defined as follows.

Maximum flow problem, or maximum s-t flow problem

Input: A directed graph $G = (V, A)$, arc capacities $u\colon A \to \mathbb{Z}_{\geq 0}$, and $s, t \in V$, $s \neq t$.

Task: Find a maximum s-t flow in G, i.e., an s-t flow f that maximizes $\nu(f)$.

Hence, in the analogy of the water pipes, the maximum s-t flow problem asks to send water from s to t with maximum throughput. Before we present algorithms to solve this problem, we will derive some further results regarding different ways to measure the value of an s-t flow.

This will play an important role in understanding how to verify that a flow has maximum value, which is useful in the design of maximum s-t flow algorithms.

The value of a flow can be measured in several ways. We defined it as the difference between outflow of s and inflow into s, which is also called the *net outflow of* s. Since all vertices except for s and t have the same outflow as inflow, we know that the only place where the net outflow of s can be absorbed is at t. Hence, the net inflow of t must be equal to the net outflow of s, and is therefore also equal to the value of the flow $\nu(f)$. In the following, we want to formalize and generalize this reasoning, which will also play an important role in understanding how to verify that a flow has maximum value among all possible s-t flows. For this we recall the notion of s-t cut, to which we assign a *value* in the context of flow problems.

Definition 4.2: s-t cut

An s-t cut is a set $C \subseteq V$ such that $s \in C$ and $t \notin C$. Furthermore, in the context of a maximum flow problem with capacities $u\colon A \to \mathbb{Z}_{\geq 0}$, the *value* of an s-t cut C is defined as $u(\delta^+(C))$. An s-t cut C is called *minimum* if it has minimum value among all s-t cuts.

The following lemma shows that any s-t cut $C \subseteq V$ can be used to measure the value of an s-t flow f .

Lemma 4.3: Value of a flow expressed via an s-t cut

Let f be an s-t flow and $C \subseteq V$ an s-t cut. Then

$$\nu(f) = f(\delta^+(C)) - f(\delta^-(C)) \ .$$

Proof. By condition (ii) (balance constraints) in the definition of an s-t flow f, we have $f(\delta^+(v)) - f(\delta^-(v)) = 0$ for all $v \in V \setminus \{s,t\}$. Thus

$$\begin{aligned}\nu(f) &= f(\delta^+(s)) - f(\delta^-(s)) + \sum_{v \in C \setminus \{s\}} \underbrace{\Big(f(\delta^+(v)) - f(\delta^-(v))\Big)}_{=0} \\ &= \sum_{v \in C} \Big(f(\delta^+(v)) - f(\delta^-(v))\Big) \\ &= f(\delta^+(C)) - f(\delta^-(C)) \ ,\end{aligned}$$

as required. □

Exercise 4.4: Value of a flow

Show that the value of any s-t flow f is equal to the difference between inflow into t and outflow of t, i.e., $\nu(f) = f(\delta^-(t)) - f(\delta^+(t))$.

Additionally to being a means of measuring the value of an s-t flow, an s-t cut also naturally leads to an upper bound on the value of *any* s-t flow. This central result is known as the

weak max-flow min-cut theorem and can be seen to be a special case of weak duality of linear programming.

Theorem 4.5: Weak max-flow min-cut theorem

Let f be an s-t flow and let $C \subseteq V$ be an s-t cut. Then

$$\nu(f) \leq u(\delta^+(C)) \ .$$

In other words, the value of a maximum s-t flow is upper bounded by the value of a minimum s-t cut.

Proof. Using Lemma 4.3, we obtain

$$\nu(f) = \underbrace{f(\delta^+(C))}_{\leq u(\delta^+(C))} - \underbrace{f(\delta^-(C))}_{\geq 0} \leq u(\delta^+(C)) \ ,$$

where $f(\delta^+(C)) \leq u(\delta^+(C))$ and $f(\delta^-(C)) \geq 0$ follow from $0 \leq f(a) \leq u(a)\ \forall a \in A$. □

Hence, if we find an s-t flow and an s-t cut with identical values, then Theorem 4.5 implies that the s-t flow is maximum and the s-t cut is minimum. The (strong) max-flow min-cut theorem, which we will see later, guarantees that such a pair of s-t flow and s-t cut with matching values always exists.

Remark 4.6: Infinite capacities

Even though in our definition of the maximum s-t flow problem, capacities are assumed to be non-negative integers, it is common to allow that some arcs $a \in A$ have infinite capacity, i.e., $u(a) = \infty$. This case can easily be reduced to the one with finite capacities. Notice that when allowing infinite capacities, an s-t flow of infinite value may exist, which can easily be checked by using BFS only over the arcs with infinite capacity. If no infinitely large s-t flow exists, then one can for example replace the infinite capacities by a sufficiently large capacity that is at least as large as the value of a maximum s-t flow. One can use for example the value of any finite-valued s-t cut, because this is an upper bound on the maximum flow value by Theorem 4.5. All vertices reachable from s by arcs of infinite capacity form such a cut that can easily be found by BFS. Due to this simple reduction, we will sometimes use infinite capacities to model problems later on, because they express in a clean an easy-to-parse way that some arcs do not have any relevant upper bound on the flow that may traverse them.

We next discuss a first algorithm to solve maximum flow problems, namely the Ford-Fulkerson algorithm. Theorem 4.5 plays a crucial role in proving that this algorithm indeed works. More precisely, one can exhibit from the Ford-Fulkerson algorithm not only an s-t flow f , but also an s-t cut C, such that $\nu(f) = u(\delta^+(C))$. By Theorem 4.5, this will immediately imply that the s-t flow is maximum and that the s-t cut is minimum. Since the Ford-Fulkerson algorithm always terminates with such a flow/cut pair, we can then derive from Theorem 4.5 the strong max-flow min-cut theorem mentioned above.

4.2 Algorithm of Ford-Fulkerson and strong max-flow min-cut theorem

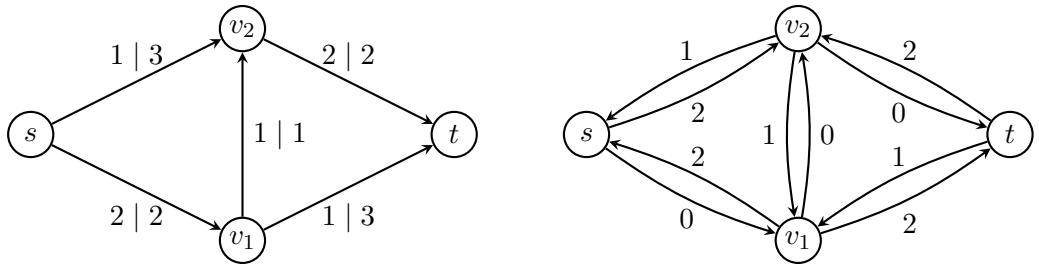
Ford and Fulkerson's max-flow algorithm starts with the *zero flow*, which is the s-t flow f defined by $f(a) = 0\ \forall a \in A$, and modifies it iteratively. In each iteration, the value of the flow strictly increases by at least one unit until a maximum flow is obtained. To find a way to augment an s-t flow f to an s-t flow with strictly larger value, Ford and Fulkerson's algorithm works on a *residual graph*, defined as follows.

Definition 4.7: f-residual graph & f-residual capacities

Let f be an s-t flow in G. The *f-residual graph* $G_f = (V, B)$ with *f-residual capacities* $u_f \colon B \to \mathbb{Z}_{\geq 0}$ is defined as follows. The set of arcs $B := A \cup A^R$ contains all original arcs A together with reverse arcs A^R , where for $a \in A$, the set A^R contains an arc a^R that is *antiparallel* to a, i.e., the head of a^R is the tail of a and vice versa. Furthermore,

$$u_f(b) := \begin{cases} u(b) - f(b) & \text{if } b \in A \ , \\ f(a) & \text{if } b = a^R \in A^R \ . \end{cases}$$

Figure 4.2 Illustrates the notion of f-residual graph and f-residual capacities.



(a) A graph $G = (V, A)$ with flow f and capacities u specified in the form $f(a) \mid u(a)$.

(b) The f-residual graph $G_f = (V, B)$ with f-residual capacities u_f.

Figure 4.2: A flow f and the corresponding f-residual graph G_f.

Notice, an arc $a \in A$ has a residual capacity of zero, i.e., $u_f(a) = 0$, if and only if $f(a) = u(a)$. Such an arc is called *f-saturated*, or simply *saturated*.

A key observation used in the Ford-Fulkerson algorithm is the fact that if there is an s-t path $P \subseteq B$ in the residual graph $G_f = (V, B)$ such that each arc of P has strictly positive residual capacity, then there is a simple operation to increase the flow f by at least one unit. Such an s-t path in G_f is called an *f-augmenting path*, or simply *augmenting path*, and is formally defined as follows.

Definition 4.8: f-augmenting path/augmenting path

Let f be an s-t flow in G. An *f-augmenting* path $P \subseteq B$ is an s-t path in $G_f = (V, B)$ with $u_f(b) > 0 \; \forall b \in P$.

Once an f-augmenting path is found, one can augment the value of the flow f as follows.

Definition 4.9: Augmentation

The *augmentation* of an s-t flow f in G along an f-augmenting path $P \subseteq B$, where $G_f = (V, B)$ is the f-residual graph, is the flow f' in G defined as

$$f'(a) = \begin{cases} f(a) + \gamma & \text{if } a \in P \ , \\ f(a) - \gamma & \text{if } a^R \in P \ , \\ f(a) & \text{if } a, a^R \notin P \ , \end{cases}$$

where $\gamma := \min\{u_f(b) \colon b \in P\} > 0$. We call the value γ the *augmentation volume* of the augmenting path P.

One can easily verify that the augmentation f' of f is indeed a flow in G of value $\nu(f') = \nu(f) + \gamma$. Furthermore, finding an augmenting path is easy. This can be done with breadth-first search, as an augmenting path is simply an s-t path in the network (V, B'), where $B' = \{b \in B \colon u_f(b) > 0\}$ are all arcs of the residual graph with strictly positive residual capacities. We thus obtain the following.

Lemma 4.10: Running time for finding f-augmenting paths

Let f be an s-t flow in G and denote the number of arcs and vertices in G by m and n, respectively. If there is an f-augmenting path, then such a path can be found in $O(m+n)$ time via breadth-first search.

Ford and Fulkerson's maximum flow algorithm starts with a zero flow and augments that flow via augmenting paths for as long as possible. Algorithm 3 describes Ford and Fulkerson's procedure. Figure 4.3 illustrates the Ford-Fulkerson algorithm on an example.

Algorithm 3: Ford and Fulkerson's algorithm to find a maximum s-t flow

Input : Directed graph $G = (V, A)$ with arc capacities $u \colon A \to \mathbb{Z}_{\geq 0}$ and $s, t \in V$, $s \neq t$.
Output: A maximum s-t flow f.

1. **Initialization:**
   $f(a) = 0 \; \forall a \in A$.
2. **while** ($\exists$ f-augmenting path P in G_f) **do:**
   Augment f along P and set f to be the augmented flow.
3. **return** f.

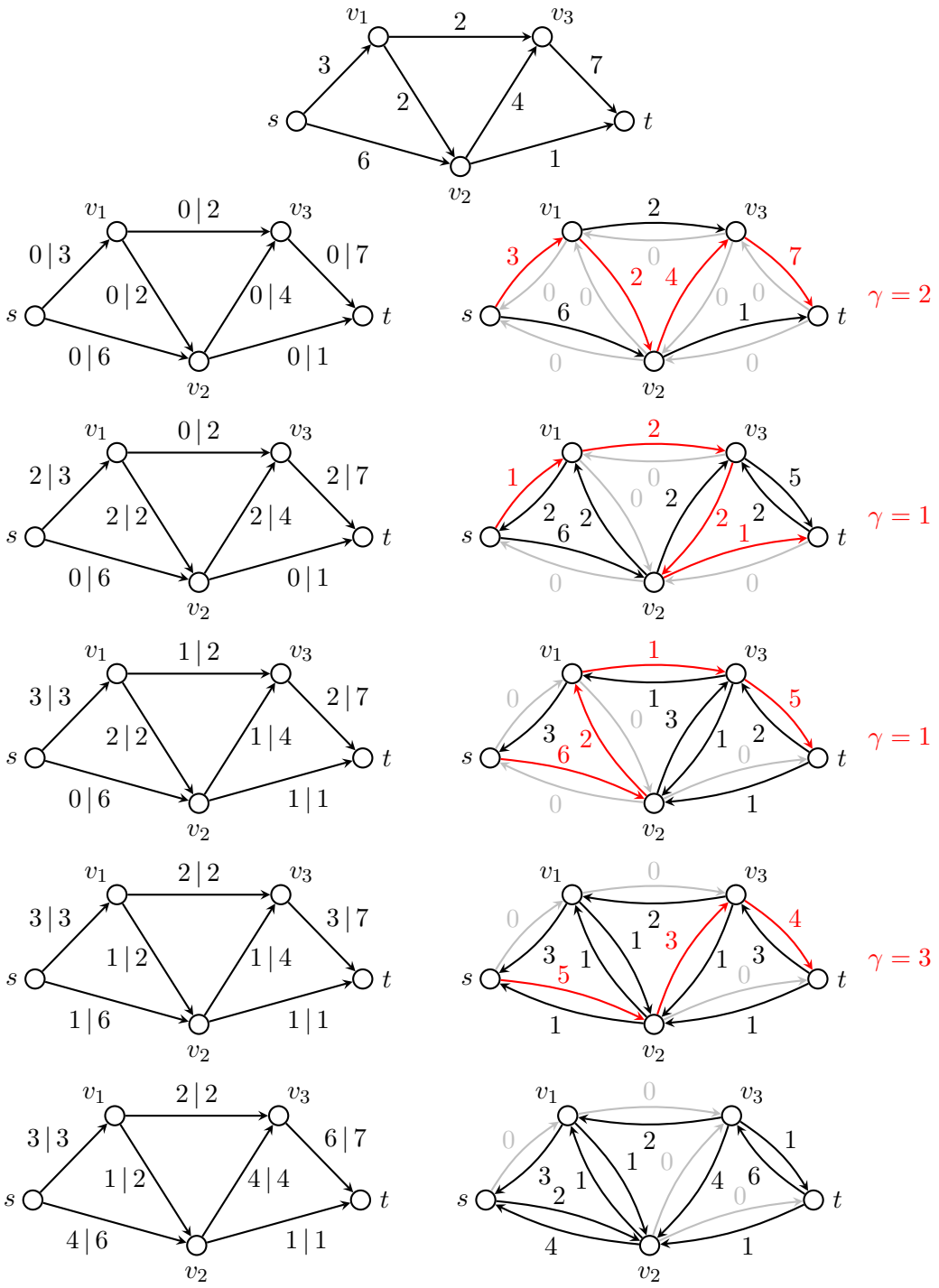


Figure 4.3: Example run of Ford and Fulkerson's algorithm. On top is the starting graph, left-hand side pictures show the current flow and right-hand side pictures the residual network with an augmenting path in red, if there is one.

There are still several points to be discussed concerning the correctness and running time of Ford and Fulkerson's algorithm. A priori, it is not even clear whether the algorithm terminates, because one may never leave the while-loop in the second step and always find new augmenting paths. It turns out that if the initial capacities are not required to be integral but can be any non-negative real numbers—in particular, irrational ones—then this can indeed happen. In such bad examples one can indefinitely find augmenting paths and augment along them. This seems somehow counter-intuitive, as the value of the flow increases each time we augment. The problem lies in the fact that, when having arbitrary real capacities, a bad example can be built such that the augmentation volumes become smaller over the iterations and the value of the s-t flow remains bounded, despite an infinite number of iterations. However, when dealing with integral (or rational) capacities $u\colon A \to \mathbb{Z}_{\geq 0}$—which is the case we are interested in and we recall that integrality of the capacities is assumed in our definition of the maximium flow problem—this cannot happen as we will show next.

Lemma 4.11: Positive integral augmentation volume

Each augmentation step of Ford and Fulkerson's algorithm has an integral augmentation volume of at least one unit.

Proof. By definition, the augmentation volume is strictly positive. Thus, it suffices to prove that it is additionally integral. We claim that each flow f encountered at the beginning and end of one iteration of step 2 is integral. This can easily be seen by induction. We start with the zero flow, which is integral. Whenever we construct the residual graph G_f for an integral flow f, the residual capacities u_f are integral, because f and u are integral. This implies that no matter which augmenting path we choose, the augmentation volume is integral. Therefore, the updated flow will be integral too.

Hence, all encountered flows f during Ford and Fulkerson's algorithm are integral, and therefore, all augmentation volumes are integral too, as already discussed above. □

From Lemma 4.11 we can easily derive bounds on the number of augmentation steps of Ford and Fulkerson's algorithm, and therefore also on its running time.

Corollary 4.12: Running time bound for Ford and Fulkerson's algorithm

The number of augmentation steps in Ford and Fulkerson's algorithm is bounded by the value $\alpha \in \mathbb{Z}_{\geq 0}$ of a maximum s-t flow in G. Hence, the running time of Ford and Fulkerson's algorithm is bounded by $O(\alpha \cdot (m+n))$, assuming $\alpha \geq 1$.

Notice that the above running time bound depends on an unknown quantity $\alpha > 0$. However, there are different ways how we can upper bound α to obtain a running time bound without unknowns. A straightforward bound is $\alpha \leq u(A) \coloneqq \sum_{a \in A} u(a)$. Often, stronger bounds can be obtained through Theorem 4.5 by using the value of a well-chosen s-t cut to bound α. A canonical candidate is often the singleton s-t cut $C = \{s\}$.

When using the trivial bound $\alpha \leq u(A)$, we therefore obtain a running time bound of $O(u(A) \cdot (m+n))$, where we assume $u(A) > 0$. It turns out that this bound can be tight

for some examples, i.e., there are examples where Ford and Fulkerson's algorithm indeed needs $\Theta(u(A) \cdot (m+n))$ elementary operations. Notice that this running time bound is not polynomial in the input size, because $u(A)$ is not polynomial in the input size. We recall that to save a positive number, like $u(a)$ for some $a \in A$, we only need $\Theta(\log u(a))$ bits. Hence, $u(a)$ can be exponential in the input size. Still, whenever the maximum capacity is small, say bounded by a polynomial in n, then the bound $O(\alpha \cdot (m+n))$ is polynomial in the input size, implying that Ford and Fulkerson's algorithm runs efficiently on such instances. This covers many interesting applications as we will see later. In the following, we will show the correctness of Ford and Fulkerson's algorithm. We will prove this together with further properties that imply the (strong) max-flow min-cut theorem.

Theorem 4.13

Let f be an s-t flow in G. Then the following are equivalent:

(i) f is a maximum s-t flow.
(ii) There does not exist an f-augmenting path in G_f.
(iii) There exists an s-t cut $C \subseteq V$ with $u(\delta^+(C)) = \nu(f)$.

Furthermore, a minimum s-t cut can be found in linear time given a maximum s-t flow.

Before proving Theorem 4.13, we discuss some of its various implications. First, Theorem 4.13 together with Theorem 4.5 implies that the value of a maximum s-t flow is always equal to the value of a minimum s-t cut. This is called the *strong max-flow min-cut theorem*, or simply the *max-flow min-cut theorem*.

Corollary 4.14: Strong max-flow min-cut theorem

The value of a maximum s-t flow in G is equal to the value of a minimum s-t cut in G:

$$\max\{\nu(f) \colon f \text{ is } s\text{-}t \text{ flow in } G\} = \min\left\{u(\delta^+(C)) \colon C \subseteq V, s \in C, t \notin C\right\} \ .$$

Consequently, Corollary 4.14 implies that the maximality of an s-t flow can always be shown by specifying a minimum s-t cut because the maximum value of an s-t flow is equal to the value of a minimum s-t cut.

Furthermore, Theorem 4.13 also implies that, as soon as one obtains an s-t flow f such that there are no f-augmenting paths, then f has maximum value. This immediately implies that Ford and Fulkerson's algorithm indeed returns a maximum s-t flow, because we already know that Ford and Fulkerson's algorithm terminates, as stated in Corollary 4.12, and we recall that the only way for the algorithm to terminate is the inexistence of further f-augmenting paths.

Corollary 4.15: Correctness of Ford and Fulkerson's algorithm

Ford and Fulkerson's algorithm returns a maximum s-t flow.

We will now provide a proof of Theorem 4.13.

Proof of Theorem 4.13. In order to prove the three equivalences in Theorem 4.13 we show the implications

$$\text{(i)} \implies \text{(ii)} \implies \text{(iii)} \implies \text{(i)} \ .$$

(i) $\Rightarrow$ (ii): We show the contraposition. The existence of an f-augmenting path P in G implies that the flow f is not maximum because augmenting f along P leads to a larger s-t flow.

(ii) $\Rightarrow$ (iii): Let $C \subseteq V$ be the set of all vertices that can be reached in G_f from s via the arcs $b \in B = A \cup A^R$ with $u_f(b) > 0$. Obviously, $s \in C$, and moreover $t \notin C$ because by assumption there are no f-augmenting paths in G_f. Hence, C is an s-t cut. By definition of C, there cannot be an arc b in G_f that is leaving C and satisfies $u_f(b) > 0$. By definition of u_f, this is equivalent to all arcs $a \in \delta_G^+(C)$ being f-saturated and all arcs $a \in \delta_G^-(C)$ satisfying $f(a) = 0$. Hence, the value of the s-t cut C satisfies

$$u(\delta_G^+(C)) = f(\delta_G^+(C)) = f(\delta_G^+(C)) - \underbrace{f(\delta_G^-(C))}_{=0} = \nu(f) \ .$$

(iii) $\Rightarrow$ (i): By the weak max-flow min-cut theorem (Theorem 4.5), point (iii) implies that f is a maximum s-t flow and C is a minimum s-t cut.

This proves the desired equivalences. Finally, the construction of a minimum s-t cut $C \subseteq V$ can be performed by following the approach used in the second part of this proof, i.e., C can be chosen to be all vertices reachable from s in G_f by using only arcs with strictly positive f-residual capacity. This can be done in $O(m + n)$ time via BFS. □

4.3 Integrality of s-t flows

The above discussion of Ford and Fulkerson's maximum s-t flow algorithm has a further important implication that is heavily exploited in Combinatorial Optimization, namely that there is a maximum s-t flow that is integral, and we can find such an s-t flow with Ford and Fulkerson's algorithm.

Theorem 4.16: Integral maximum flows

Let $G = (V, A)$ be a directed graph with capacities $u \colon A \to \mathbb{Z}_{\geq 0}$, and let $s, t \in V$, $s \neq t$. Ford and Fulkerson's algorithm finds a maximum s-t flow that is integral.

Proof. The s-t flow f returned by Ford and Fulkerson's algorithm is maximum by Corollary 4.15. Furthermore, f is integral because each augmentation increases the flow by an integral amount. □

The fact that maximum s-t flows can be chosen to be integral allows for using s-t flows to model a wide variety of combinatorial optimization problems through flow problems. A crucial advantage of having integral flows lies in the ability to interpret an integral s-t flow combinatorially. For example, if all capacities are either 0 or 1, then a maximum s-t flow can be interpreted as a subset of the arcs, namely the ones that carry one unit of flow. We will give applications of this in the next section. Finally, we want to highlight that the integrality

of s-t flows is one of the main reasons why s-t flows are studied so heavily in Combinatorial Optimization.

4.4 Applications of s-t flows

There is a very broad set of applications of flows and cuts. As we already mentioned, one reason for this is the existence of optimal integral flows, but this is by far not the only one. The existence of extremely fast algorithms to solve even very large flow problems is another key aspect. In general, state-of-the-art flow algorithms run significantly faster in practice then their worst-case guarantees, often reaching a roughly linear running time. Due to this, flows and cuts have found numerous applications in large-scale problems. Additionally, several classical theoretical results linked to graphs and graph optimization can be elegantly derived via flows, their integrality, and the strong max-flow min-cut theorem. In the following, we present a selection of several applications of flows and cuts that reflect their versatility and broad applicability.

4.4.1 Arc-connectivity

We already discussed basic connectivity properties of graphs. Often, one is not only interested in knowing whether a directed graph is strongly connected, i.e., whether every vertex can be reached from every other vertex, but also how "well" two vertices are connected. A typical way to measure this is by checking how many arc-disjoint paths there are between two vertices s and t. Using this stronger notion of connectivity leads to the notion of *k-arc-connected* graphs.

Definition 4.17: k-arc-connectivity

A directed graph $G = (V, A)$ is *k-arc-connected* if for any two vertices $s, t \in V, s \neq t$, there are at least k arc-disjoint s-t paths in G.

Notice that a 1-arc-connected graph is simply a strongly connected graph. We can check whether a graph is strongly connected via breadth-first search. A key question is how to check whether a graph is k-arc-connected. This can be reduced to the question of computing a maximum number of arc-disjoint paths from s to t for two arbitrary distinct vertices $s, t \in V$, by checking whether each pair of vertices is k-arc-connected. Interestingly, this question can be solved easily via maximum s-t flows, by introducing unit capacities on all the arcs and computing a maximum s-t flow, as highlighted in Algorithm 4. We will show that the value of this s-t flow corresponds to the number of arc-disjoint s-t paths.

We show in two steps that Algorithm 4 works correctly. First, we observe that whenever G contains k arc-disjoint s-t paths, then there is an s-t flow of value k in the flow network used in Algorithm 4.

Algorithm 4: Determining maximum number of arc-disjoint s-t paths.

Input: A directed graph $G = (V, A)$ and vertices $s, t \in V$, $s \neq t$.
Output: Maximum number of arc-disjoint s-t paths in G.

1. Define unit capacities $u\colon A \to \mathbb{Z}_{\geq 0}$, i.e., $u(a) = 1\ \forall a \in A$.
2. Compute a maximum s-t flow f in G with capacities given by u.
3. **return** $\nu(f)$, the value of f.

Observation 4.18: From arc-disjoint paths to flows

Let $G = (V, A)$ be a directed graph and $s, t \in V$ with $s \neq t$. Let $u\colon A \to \mathbb{Z}_{\geq 0}$ be unit capacities, i.e., $u(a) = 1\ \forall a \in A$. If there are k arc-disjoint s-t paths $P_1, \dots, P_k \subseteq A$ in G, then an s-t flow f of value k is obtained by setting

$$f(a) := \begin{cases} 1 & \text{if } a \in \bigcup_{i=1}^{k} P_i \\ 0 & \text{otherwise} \end{cases}.$$

Second, we show that if there is an s-t flow of value k in the flow network used in Algorithm 4, then there exist k arc-disjoint s-t paths. This is implied by the following constructive theorem, which shows how to convert an integral s-t flow of value k in a unit-capacity graph into k arc-disjoint s-t paths.

Theorem 4.19: Path decomposition

Let f be an integral s-t flow in G with unit capacities u and let $k = \nu(f)$. Then, one can find in linear time (i.e., $O(m + n)$ time), k arc-disjoint s-t paths $P_1, \dots, P_k \subseteq A$ with $\bigcup_{i=1}^{k} P_i \subseteq \{a \in A\colon f(a) = 1\}$.

Proof. Let $U = \{a \in A\colon f(a) = 1\}$. We will find k arc-disjoint s-t walks within U. The result then follows by shortcutting the walks into paths, i.e., by deleting directed cycles within a walk. We use induction on k. The case $k = 0$ is trivial; hence, we assume $k \in \mathbb{Z}_{\geq 1}$. We construct the first s-t walk $W \subseteq U$ in our family of arc-disjoint s-t walks by using Algorithm 5.

To show that Algorithm 5 indeed finds an arc-disjoint s-t walk, we have to show that whenever we are at the beginning of a new iteration of the while-loop and $v \neq t$, then $\exists (v, w) \in U \setminus W$. This then implies that the algorithm will stop with an s-t walk.

When considering a vertex $v \in V \setminus \{s, t\}$ in Algorithm 5, then, by construction, $W \subseteq U$ is an s-v path. The set W contains one more incoming arc into v than outgoing ones. Because U stems from a $\{0, 1\}$ s-t flow, the number of arcs in U that enter v is equal to the number of arcs in U that leave v. Therefore, there must exist an arc $(v, w) \in U \setminus W$, as desired.

We can follow a similar argument for $v = s$. Whenever the algorithm considers the vertex $v = s$, then $W \subseteq U$ is a closed walk containing s. Hence, W contains as many arcs entering s as there are arcs leaving s. However, because U stems from a $\{0, 1\}$ s-t flow with value $k > 0$,

Algorithm 5: Determining an s-t walk W

Input: A directed graph $G = (V, U)$, vertices $s, t \in V$, $s \neq t$, and an s-t flow $f : U \to \{0, 1\}$ with value $\nu(f) \geq 1$.
Output: An arc-disjoint s-t walk $W \subseteq U$.

1. **Initialization:**
   $v = s$.
   $W = \emptyset$.
2. **while** $v \neq t$ **do:**
   Choose an arc $(v, w) \in U \setminus W$.
   $W = W \cup \{(v, w)\}$.
   $v = w$.
3. **return** W.

the arc set U contains exactly k arcs more that leave s than arcs entering s. As $k > 0$, there exists an arc $(s, w) \in U \setminus W$.

Thus, we can find an s-t walk in $G' = (V, U)$ through Algorithm 5. The other $k-1$ required s-t walks can be found inductively among the remaining arcs $U \setminus W$ because these arcs correspond to the flow f', who is defined as follows:

$$f'(a) := \begin{cases} 1 & \text{if } a \in U \setminus W \\ 0 & \text{otherwise} \end{cases},$$

and has value $k - 1$. Hence, the result follows by induction. The bound for the running time applies because every arc in U occurs at most once as an arc (v, w) in step 2 of a call to Algorithm 5 in the course of the construction of s-t walks according to Algorithm 5. □

Combining Observation 4.18, Theorem 4.19, and the fact that there is an integral maximum s-t flow due to Theorem 4.16, we obtain the following result.

Corollary 4.20: Correctness of Algorithm 4

Algorithm 4 correctly determines the maximum number of arc-disjoint s-t paths. Furthermore, by Theorem 4.19, one can find a maximum set of arc-disjoint s-t paths in linear time once an integral maximum s-t flow is found.

Notice that we need the existence of an integral maximum s-t flow to be able to invoke Theorem 4.19. However, this is just part of the proof. If we only want to determine the maximum number of arc-disjoint s-t paths, then any maximum s-t flow algorithm can be used, even if a non-integral maximum flow is returned. However, if we want to compute a maximum number of arc-disjoint s-t paths, then we need to obtain an integral flow to invoke Theorem 4.19 constructively.

Moreover, notice that Algorithm 4 is an efficient procedure when using Ford and Fulkerson's maximum s-t flow procedure. This is despite the fact that for general maximum s-t flow problems, Ford and Fulkerson's algorithm is not efficient, as already discussed. The reason for this is that the maximum flow in a unit-capacity network with m arcs is at most m. Hence, by Corollary 4.12, the running time of Algorithm 4, when implemented with Ford and Fulkerson's maximum flow algorithm, is bounded by $O(m(m+n))$. Thus, we can answer the question whether a given directed graph is k-arc-connected in polynomial time.

Notice that Corollary 4.20 states that the arc-connectivity between s and t is equal to the value of a maximum s-t flow in the unit-capacity graph considered in Algorithm 4. Hence, by the strong max-flow min-cut theorem, this leads to the following version of Menger's Theorem.

Theorem 4.21: Menger's Theorem

Let $G = (V, A)$ be a directed graph with $s, t \in V, s \neq t$. Then the maximum number of arc-disjoint s-t paths is equal to the number of arcs in a minimum cardinality s-t cut.

In the above theorem, a minimum cardinality s-t cut refers to a minimum s-t cut with respect to unit capacities. Hence, its value is $\min\{|\delta^+(C)| \colon C \subseteq V, s \in C, t \notin C|\}$.

There are many versions of Menger's Theorem, which can be derived through variations of the above reasoning. In particular, the analogous theorems can be stated for undirected graphs and also for vertex-disjoint paths. We will get back to variations of Menger's Theorem in the problem sets.

Arc-connectivity can be generalized to undirected graphs. In this case, k-edge-connectivity is defined as follows.

Definition 4.22: k-edge-connectivity in undirected graphs

An undirected graph $G = (V, E)$ is *k-edge-connected* if for any two vertices $s, t \in V$ with $s \neq t$ there exist at least k edge-disjoint s-t paths in G.

Exercise 4.23: Checking k-edge-connectivity in undirected graphs

Reduce the problem of checking k-edge-connectivity in undirected graphs to the problem of checking k-arc-connectivity in directed graphs.

4.4.2 Maximum cardinality matchings in undirected graphs

Consider the following problem. Assume that there is a set of workers X that have to perform a set of tasks Y. Not every worker can perform every task, and each task is a one day job for a worker with the appropriate skill set. We can represent the relation of which workers can perform which tasks as a graph $G = (V, E)$, where $V = X \dot\cup Y$ and an edge $\{x, y\} \in E$ between worker $x \in X$ and task $y \in Y$ means that x has the right skill set to perform task y. Our goal is to find an assignment of tasks to workers such that a maximum number of tasks is

performed within the considered day. This problem can be solved by computing a maximum s-t flow in an appropriate auxiliary network, as we discuss next.

We start with some basic terminology to highlight some important structure of the problem and its solution. First, notice that the graph G is not a general undirected graph, but it has an additional property. Namely, each edge goes from a vertex in X to a vertex in Y, with $X \cap Y = \emptyset$. Such graphs are called *bipartite*, and one can easily check that they cannot have any odd cycles. It turns out that the non-existence of odd cycles even characterizes them.

Definition 4.24: Bipartite graph

An undirected graph $G = (V, E)$ is called *bipartite* if there is a bipartition of its vertices $V = X \dot\cup Y$ such that each edge has one endpoint in X and the other in Y.

Figure 4.4 shows an example of a bipartite graph. If G has several connected components, then there are several ways to partition V into $X \dot\cup Y$ such that all edges go between X and Y. For simplicity, we sometimes say that $G = (V, E)$ is a bipartite graph *with bipartition* $V = X \dot\cup Y$ to denote an arbitrary bipartition of the vertices such that all edges go between X and Y.

Exercise 4.25

Show that a graph is bipartite if and only if it does not contain an odd cycle. We recall that a loop is also considered to be an odd cycle.

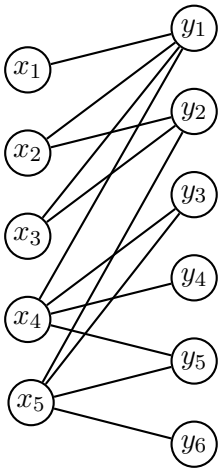


Figure 4.4: Example of a bipartite $G = (V, E)$ with bipartition $V = X \dot\cup Y$, where $X = \{x_1, \ldots, x_5\}$ and $Y = \{y_1, \ldots, y_6\}$.

Notice that a solution to our assignment problem can be represented by a subset $M \subseteq E$ of the edges that indicates which tasks are assigned to which workers. More formally, an edge $\{x, y\} \in M$ is interpreted as an assignment of worker x to task y. Notice that because every worker can be assigned to at most one task and every task can be performed by at most one

worker, a solution set M must be such that no worker or task has two edges of M incident with it, i.e., no vertex in the subgraph (V, M) has degree bigger than 1. Such an edge set M is called a *matching*.

Definition 4.26: Matching

Let $G = (V, E)$ be an undirected graph. A set $M \subseteq E$ is a *matching* in G if M does not contain loops and no two edges of M share a common endpoint.

Hence, finding a maximum assignment of workers to possible jobs can be rephrased as finding a maximum cardinality matching in the bipartite graph $G = (X \dot\cup Y, E)$. In the following, we show how to solve such problems efficiently by computing an integral maximum s-t flow. For completeness, we first give a formal description of the problem.

Maximum cardinality matching problem in bipartite graphs

Input: An undirected bipartite graph $G = (V, E)$.

Task: Find a maximum cardinality matching $M \subseteq E$ in G.

In the following discussion, where we show how to solve the maximum cardinality bipartite matching problem, we reuse the notation and terminology that we introduced for the worker-task assignment problem. Hence, our bipartite graph is $G = (V, E)$ with bipartition $V = X \dot\cup Y$. Consider the following auxiliary graph $H = (W, F)$, which is directed. It is obtained from G by directing all edges of G from X to Y. Furthermore, we introduce two new vertices s and t. For each vertex $x \in X$ we add an arc (s, x), and for each vertex $y \in Y$ we add an arc (y, t). Hence, formally, the graph H is defined as follows:

$$W = V \cup \{s, t\} \text{ , and}$$
$$F = \{(x, y) \colon x \in X, y \in Y, \{x, y\} \in E\} \cup \{(s, x) \colon x \in X\} \cup \{(y, t) \colon y \in Y\} \text{ .}$$

Figure 4.5 shows the graph H that corresponds to the example bipartite graph depicted in Figure 4.4.

The capacities $u \colon F \to \mathbb{Z}_{\geq 0}$ are set to $u(a) = 1$ for all $a \in F$. We now determine a maximum integral s-t flow f in H. Let

$$M \coloneqq \{\{x, y\} \in E \colon x \in X, y \in Y, f((x, y)) = 1\}$$

be the set of all edges in G whose corresponding arcs in H carry one unit of flow. We prove the following claim.

Claim 4.27

M is a maximum cardinality bipartite matching in G.

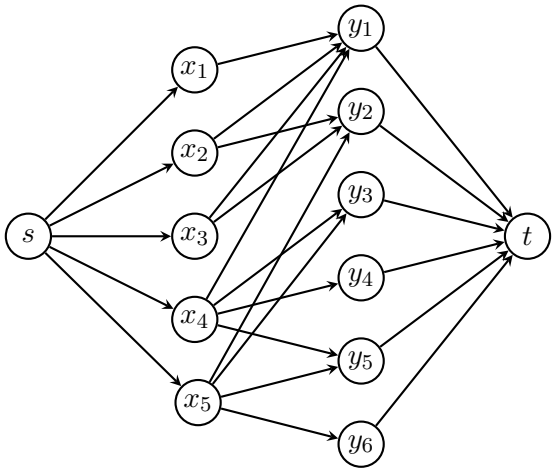


Figure 4.5: Example of the directed graph $H = (W, F)$ obtained from the bipartite graph $G = (V, E)$ shown in Figure 4.4.

Proof. First notice that M is a matching. For this we show that every vertex is incident with at most one edge of M. Indeed, consider any vertex $x \in X$ (the reasoning for $y \in Y$ is analogous). Because x has only one incoming arc in the flow network H, namely the arc (s, x), which is an arc from s with capacity one, it is impossible to have two arcs going out of x that both carry a unit of flow. This would contradict the balance constraint at x. Thus at most one of the arcs outgoing from x has flow 1., i.e., x has degree at most 1 in M. Thus, M is indeed a matching. Furthermore, we can compute the value $\nu(f)$ of f by using Lemma 4.3 with $C = \{s\} \cup X$, i.e.,

$$\nu(f) = f(\delta^+(C)) - f(\delta^-(C)) = f(\delta^+(C)) = |M| \ ,$$

where the second equality follows from $\delta^-(C) = \emptyset$, and the last one from the definition of M and the fact that f is a $\{0, 1\}$-flow due to its integrality and the unit capacities. In summary, any integral s-t flow in H of value k corresponds to a matching of cardinality k.

Conversely, every matching $M' \subseteq E$ of cardinality k can easily be transformed into an s-t flow of value k in the graph H. We start with the zero flow f, and for every arc $\{x, y\} \in M$, we set the flow values on the arcs of the path $s \to x \to y \to t$ to one unit. Hence, this implies that whenever there is a matching of cardinality k in H, then there is an s-t flow of value k.

The above showed that there is a one-to-one relation between matchings in G and integral s-t flows in H such that the cardinality of a matching is equal to the value of the corresponding s-t flow. Hence, the matching M constructed via the s-t flow problem is indeed a maximum cardinality matching in G. □

Hence, we can solve the matching problem with Ford and Fulkerson's algorithm. With regard to the running time, observe that the auxiliary graph H has size polynomially bounded in the input size. Moreover, all arcs have capacity 1, and thus, the sum of all capacities is a polynomial upper bound for the value of a maximum flow. By Corollary 4.12, it follows that the proposed algorithm for finding a maximum cardinality bipartite matching is efficient when using Ford and Fulkerson's algorithm to find an integral maximum s-t flow.

Exercise 4.28: Two jobs per day

Consider our worker-task assignment problem. Assume that every worker can perform up to two tasks within the same day. As before, only one worker can be assigned per job. Show how this problem can still be solved efficiently, by adapting the approach presented above.

When interpreting the allocation of workers to jobs, it is a natural question under which circumstances it is possible to simultaneously assign all workers in X. Formally, in a bipartite graph $G = (V, E)$ with bipartition $V = X \dot\cup Y$, we are looking for sufficient and necessary conditions for the existence of a matching $M \subseteq E$ that touches all vertices in X, i.e., matchings M such that every vertex x is incident with an edge in M. Obviously $|Y| \geq |X|$ is a necessary condition, although this condition is certainly not sufficient. It is not hard to see that the following restriction is also necessary: For each set of $X_0 \subseteq X$, the number of neighbors that X_0 has in Y must be at least $|X_0|$. If we define $N(X_0) \subseteq Y$ as the set of neighbors from X_0 in Y, i.e.,

$$N(X_0) \coloneqq \{y \in Y : \exists x \in X_0 \text{ with } \{x, y\} \in E\} \quad ,$$

then we can write the condition as $|N(X_0)| \geq |X_0|$ for all $X_0 \subseteq X$. The condition is necessary because $N(X_0)$ contains all jobs that workers in X_0 can perform. For everyone to work simultaneously, $N(X_0)$ must have at least as many jobs as there are workers in X_0. As it turns out, this condition, known as *Hall's condition*, is also sufficient. This is the statement of Hall's Theorem which, as we will see, can be considered as an implication of the (strong) max-flow min-cut theorem (Corollary 4.14).

Theorem 4.29: Hall's Theorem

Let $G = (V, E)$ be a bipartite graph with bipartition $V = X \dot\cup Y$. Then there exists a matching $M \subseteq E$ in G that touches all vertices in X if and only if

$$|N(X_0)| \geq |X_0| \qquad \text{for all } X_0 \subseteq X \ .$$

Proof. As discussed, Hall's condition is clearly necessary for the existence of a matching M that touches all vertices in X. We still have to prove sufficiency of the condition.

To this end, we consider an s-t flow problem in the same auxiliary graph $H = (W, F)$ that we constructed to solve the bipartite matching problem (see Figure 4.5). As already shown, the maximum cardinality matching in G is equal to the value of a maximum s-t flow in H. In order to prove Hall's Theorem, it suffices to conclude from Hall's condition that there exists an s-t flow of value $|X|$ in H. Note that no flow can have a value greater than $|X|$, because the total capacity of all arcs leaving s is $|X|$. The max-flow min-cut theorem (Corollary 4.14) implies that the value of a maximum s-t flow in H equals the value of a minimum s-t cut in H. In order to show the result, we thus prove that the value of a minimum s-t cut $C \subseteq W$ in H is $|X|$.

Let $C \subseteq W$ be a minimum s-t cut in H, and we define $C_X \coloneqq C \cap X$ and $C_Y \coloneqq C \cap Y$. We first show that, without loss of generality, we can assume that C fulfills

$$N(C_X) \subseteq C_Y \ . \tag{4.1}$$

If not, we can replace C by $C \cup N(C_X)$ without increasing the value of the cut. To show this, we observe how the value of the s-t cut C changes when we add $y \in N(C_X) \setminus C_Y$ to C to obtain $C' = C \cup \{y\}$. There is one new arc in the cut, namely $\delta^+(C') \setminus \delta^+(C) = \{(y,t)\}$. However, there is also at least one arc in $\delta^+(C) \setminus \delta^+(C')$, because y is a neighbor of a vertex in C_X and $y \notin C_Y$. Consequently, because all arcs have unit capacity, the value of the s-t cut C' cannot be greater than the value of the s-t cut C. By applying this reasoning iteratively to all vertices $y \in N(C_X) \setminus C_Y$, we obtain, as claimed, that the value of the s-t cut $C \cup N(C_X)$ is no greater than the value of the s-t cut C. We can thus assume that C fulfills (4.1).

The value of the s-t cut C is given by $u(\delta^+(C)) = |\delta^+(C)|$, because H has unit capacities. Consider the arcs $\delta^+(C)$. This set contains no arcs between X and Y, because C satisfies (4.1). Consequently, $\delta^+(C)$ consists of all arcs from s to $X \setminus C_X$ and all arcs from C_Y to t. Thus,

$$\begin{aligned} u(\delta^+(C)) &= |\delta^+(C)| \\ &= |X| - |C_X| + |C_Y| \\ &\geq |X| - |C_X| + |N(C_X)| && \text{(because } |C_Y| \geq |N(C_X)| \text{ by (4.1))} \\ &\geq |X| \ . && \text{(by Hall's condition } |N(C_X)| \geq |C_X|) \end{aligned}$$

In summary, the minimum s-t cut in H has a value of at least $|X|$, and thus a value of exactly $|X|$, because the s-t cut $C = \{s\}$ has value $|X|$. According to the strong max-flow min-cut theorem (Corollary 4.14), the value of a maximum s-t flow is also $|X|$, and hence, as discussed above, a maximum cardinality matching $M \subseteq E$ has size $|X|$, thus touching all vertices in X. □

4.4.3 Multiple sources and sinks with limited supply and demand

So far we have only considered flow problems with a single source s and a single sink t. We now consider multiple sources and sinks. Here, there are different natural versions one can consider, many of which can easily be reduced to the single-source single-sink case. We consider a classical setting with multiple sources and sinks with limited supply and demand. More precisely, given is a directed graph $G = (V, A)$ with arc capacities $u \colon A \to \mathbb{Z}_{\geq 0}$, sources $s_1, \ldots, s_k \in V$, and sinks $t_1, \ldots, t_\ell \in V$. Suppose that sources and sinks are all distinct. This assumption is made only to simplify the exposition and can easily be lifted. In addition, each source has limited supply and each sink has a specific demand, captured by a function $h \colon \{s_1, \ldots, s_k\} \cup \{t_1, \ldots, t_\ell\} \to \mathbb{Z}$. More precisely, positive values of h represent supplies whereas negative values capture demands, i.e., the h-value $h(s_i)$ of a source s_i is positive and represents its supply, whereas $h(t_j)$ for a sink t_j is negative and $-h(t_j)$ represents the demand of t_j. The task is to find a flow f in G that simultaneously

(i) respects the capacity constraints,
(ii) has a net outflow of no more than $h(s_i)$ at source s_i, and
(iii) has a net inflow of precisely $-h(t_j)$ into sink t_j.

In other words, the flow $f \colon A \to \mathbb{R}_{\geq 0}$ shall satisfy the capacity constraints

$$f(a) \leq u(a) \qquad \forall a \in A \ ,$$

and the balance constraints

$$\begin{cases} f(\delta^+(s_i)) - f(\delta^-(s_i)) \leq h(s_i) & \forall i \in [k] \ , \\ f(\delta^+(v)) - f(\delta^-(v)) = 0 & \forall v \in V \setminus \{s_1, \dots, s_k, t_1, \dots t_\ell\} \ , \\ f(\delta^+(t_j)) - f(\delta^-(t_j)) = h(t_j) & \forall j \in [\ell] \ . \end{cases}$$

Figure 4.6 shows an example of such a flow problem with four sources and three sinks.

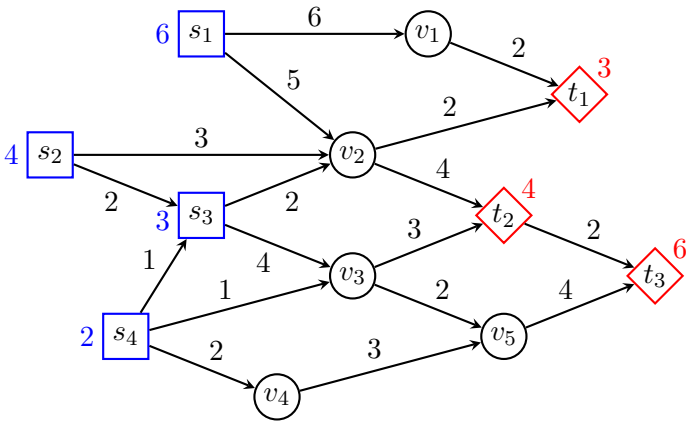


Figure 4.6: A flow problem with four sources $\{s_1, s_2, s_3, s_4\}$ and three sinks $\{t_1, t_2, t_3\}$. The supply of each source is indicated in blue, the demand of the sinks in red. The numbers on the arcs correspond to the respective capacities.

To solve the problem, we construct an auxiliary graph $H = (W, B)$, which is created from G by adding two vertices s and t and the arcs (s, s_i) for all $i \in [k]$ and (t_j, t) for all $j \in [\ell]$, i.e.,

$$W = V \cup \{s, t\} \qquad \text{and} \qquad B = A \cup \{(s, s_i) \colon i \in [k]\} \cup \{(t_j, t) \colon j \in [\ell]\} \ .$$

We define the capacities $u' \colon B \to \mathbb{Z}_{\geq 0}$ in the network H for all $b \in B$ by

$$u'(b) = \begin{cases} u(b) & \text{if } b \in A \ , \\ h(s_i) & \text{if } b = (s, s_i) \text{ for } i \in [k] \ , \\ -h(t_j) & \text{if } b = (t_j, t) \text{ for } j \in [\ell] \ . \end{cases}$$

Figure 4.7 shows the auxiliary graph H and the corresponding capacities u' that we construct to solve the flow problem from Figure 4.6.

We determine a maximum s-t flow f in the network H. The choice of capacities on the arcs (s, s_i) ensures that each source s_i has a net outflow of no more than $h(s_i)$ units over the original arcs. Similarly, the choice of capacities of the arcs (t_j, t) guarantees that each sink t_j has a net inflow of at most $-h(t_j)$ units over the original arcs. Notice that $\nu(f) \leq -\sum_{j=1}^{\ell} h(t_j)$, because the flow value is upper bounded by the value of the s-t cut $W \setminus \{t\}$. Thus, f has value $-\sum_{j=1}^{\ell} h(t_j)$ if and only if each sink t_j has a net inflow of $-h(t_j)$ on the arcs of the original graph. In summary, the original flow problem in G has a solution if and only if the maximum

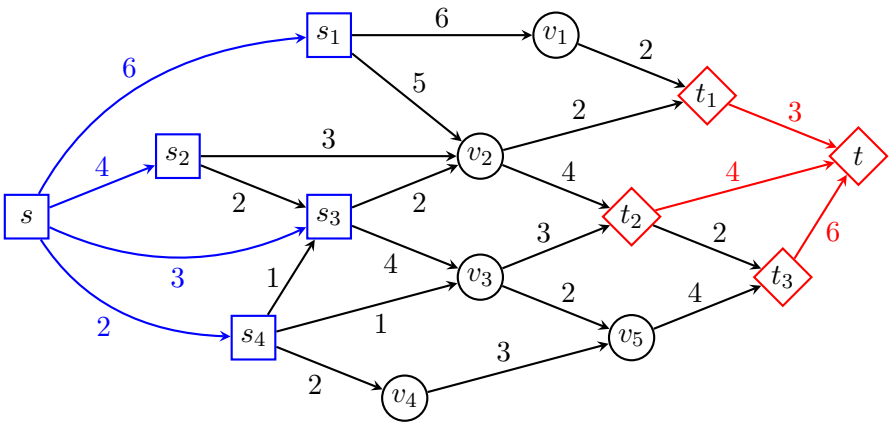


Figure 4.7: The auxiliary graph H with capacities u' used to solve the flow problem from Figure 4.6.

s-t flow in the auxiliary network H reaches the value $-\sum_{j=1}^{\ell} h(t_j)$. In this case we can read off a solution directly from a maximum s-t flow in H.

Figure 4.8 shows an optimal flow in the network from Figure 4.7, from which we can read off an optimal solution for the original problem.

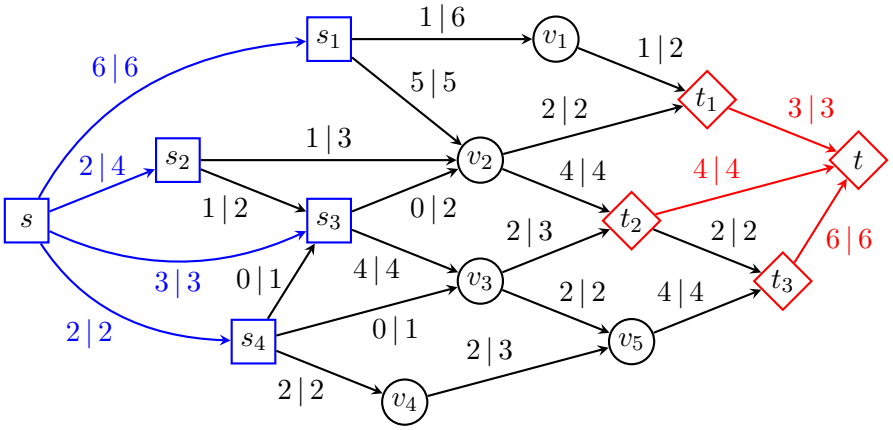


Figure 4.8: An optimal flow in the auxiliary graph H from Figure 4.7, given in the form $f(b) \mid u'(b)$ for all $b \in B$.

4.4.4 Roster planning

A construction company has a set of workers A, a set of vehicles F grouped into vehicle types, and a set of projects P. The realization of a project requires a vehicle type suitable for the project and a worker allowed to operate the vehicle. The task is to simultaneously handle as many projects as possible.

The following graph $G = (V, E)$ highlights key parts of the available information. It contains

one vertex per worker, one per vehicle type, and one per project. Let $A = \{a_1, \dots, a_n\}$ by the set of all workers, let $f_1, \dots, f_\ell$ be the different vehicle types, where $g(f_j) \in \mathbb{Z}_{\geq 0}$ for $j \in [\ell]$ denotes the number of vehicles of type f_j, and let $P = \{p_1, \dots, p_m\}$ be the set of all projects. Hence,

$$V \coloneqq A \cup \{f_1, \dots, f_\ell\} \cup P \ .$$

Moreover, the edge set E contains two types of edges:

(i) There is an edge $\{a_i, f_j\}$ between a worker a_i and vehicle type f_j if a_i can operate vehicles of type f_j.

(ii) There is an edge $\{f_j, p_k\}$ between vehicle type f_j and project p_k if a vehicle of type f_j is suitable for the project p_k.

Figure 4.9 shows a graph with 7 workers, 3 vehicle types, and 8 projects.

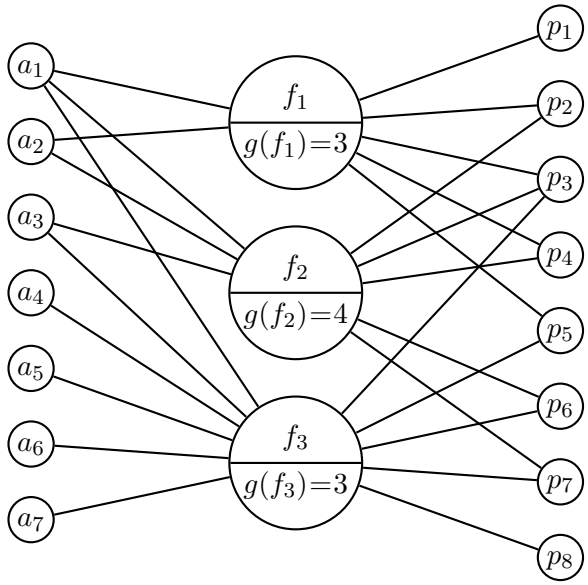


Figure 4.9: The graph $G = (V, E)$ representing a roster planning problem with 7 workers, 3 vehicle types, and 8 projects. The number of available vehicles of each type is specified in the corresponding vertex.

The assignment of a worker to a vehicle and a project in the constructed graph G corresponds to a path of length 2 from a worker a_i via a vehicle f_j to a project p_k. Hence, we are interested in the maximum number of paths such that no two paths have a worker or a project in common. Additionally, for each vehicle type f_j, no more than $g(f_j)$ paths should pass through vertex f_j. To solve this problem, we again construct a directed auxiliary network $H = (W, D)$ with capacities $u \colon D \to \mathbb{Z}_{\geq 0}$ in which we will solve a maximum flow problem. We define H by

$$\begin{aligned} W &\coloneqq A \cup \{f_j \colon j \in [\ell]\} \cup \{f'_j \colon j \in [\ell]\} \cup P \cup \{s, t\} \ , \text{ and} \\ D &\coloneqq \{(s, a_i) \colon i \in [n]\} \cup \{(a_i, f_j) \colon \{a_i, f_j\} \in E\} \cup \{(f_j, f'_j) \colon j \in [\ell]\} \\ &\quad \cup \{(f'_j, p_k) \colon \{f_j, p_k\} \in E\} \cup \{(p_k, t) \colon k \in [m]\} \ . \end{aligned}$$

Furthermore, the capacities $u\colon D \to \mathbb{Z}_{\geq 0}$ are defined as follows: For $d \in D$ we set

$$u(d) = \begin{cases} g(f_j) & \text{if } d = (f_j, f_j') \text{ for } j \in [\ell] \\ 1 & \text{otherwise} \end{cases} .$$

The directed auxiliary network H resulting from the graph in Figure 4.9 is shown in Figure 4.10.

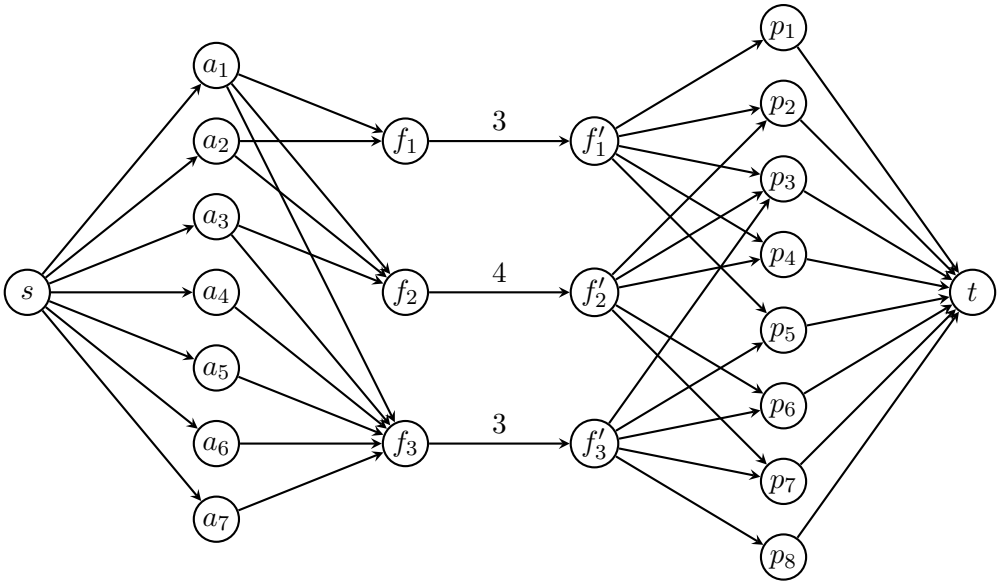


Figure 4.10: The auxiliary network H for the graph G from Figure 4.9. Arcs without explicitly specified capacity have capacity 1.

In the auxiliary graph H, the assignment of a worker a_i to a vehicle f_j and a project p_k corresponds to an s-t path $s \to a_i \to f_j \to f_j' \to p_k \to t$. Each assignment, which can be represented by such a path, is captured in the flow problem by a unit flow along the path. With this interpretation, it is easy to see that each possible roster corresponds to an integral s-t flow in H, whose value corresponds to the number of concurrently processed projects. We can simply add up the flows of the individual assignments. Thereby, no capacity is exceeded, because in a possible roster each worker and each project occur at most once—thus the unit capacities are respected—and each vehicle type f_j is used at most $g(f_j)$ times, that is, the arc (f_j, f_j') is part of at most $g(f_j)$ single flows and so the total flow over (f_j, f_j') is at most $g(f_j) = u((f_j, f_j'))$.

Conversely, given an integral s-t flow f in H, we can create a roster in which $\nu(f)$ projects can be completed simultaneously. This is done by splitting the flow into $\nu(f)$ many s-t paths, through each of which flows one unit. We can proceed in the same way as we did to determine a maximum number of arc-disjoint s-t paths in Section 4.4.1: We can iteratively select s-t paths from the arcs with non-negative flow and reduce the flow over the respective arcs by 1 for each selected s-t path.

Hence, there is a bijective assignment between the possible rosters and the integral s-t flows in H, where the value of the flow is equal to the number of simultaneously processed projects. A

maximum number of simultaneously processed projects is therefore reached by a maximum integral s-t flow in H. By Theorem 4.16, we can use Ford and Fulkerson's algorithm to determine a maximum integral s-t flow in H. After transforming this flow into a corresponding optimal roster, the rostering problem is solved. Notice that Ford and Fulkerson's algorithm is efficient for this problem, because the maximum flow value is bounded by the number of workers, as this corresponds to the value of the singleton s-t cut $\{s\}$.

4.4.5 Optimal project selection

A company has a set of projects $P = \{p_1, \dots, p_m\}$ to choose from. Each project can either be selected to be performed or discarded. Each of the projects p_i has a profit of $g(p_i)$, where $g\colon P \to \mathbb{Z}$. If $g(p_i) \geq 0$, then p_i is interpreted as a profitable project, whereas $g(p_i) < 0$ indicates that the project is generating a loss. There are precedence constraints between the available projects, i.e., for each project p_i, there is a (possibly empty) set of other projects it depends on, which have to be completed before project p_i can be started. For example, this may model an investment in infrastructure needed for various projects. The question is which projects should be realized to maximize the total profit. Figure 4.11 shows an example of projects with precedence constraints modeled as a directed graph $G = (P, A)$. An arc (p_i, p_j) indicates that project p_i is a prerequisite for project p_j. The project requirements are such that there are no directed cycles in the graph $G = (P, A)$. Such a cycle would correspond to a set of projects that are cyclically dependent on each other and therefore could never be started.

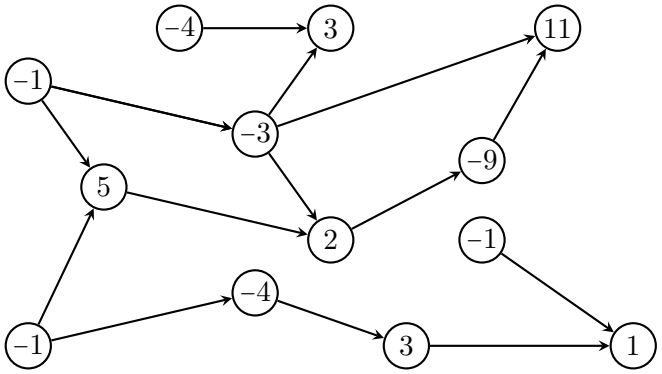


Figure 4.11: A graph G with projects and their precedence constraints. Profits (or costs) are indicated in the corresponding vertices.

To solve the problem, we construct an auxiliary graph $H = (V, B)$ with capacities $u\colon B \to \mathbb{Z}_{\geq 0}$. The graph H is obtained by starting from G and adding two vertices s, t and several arcs. In the previous examples, the auxiliary graph was built in a way that an integral flow could be interpreted in terms of the decisions that had to be taken. Here, we follow a different approach, in the sense that we build a graph where the minimum s-t cut will correspond to an optimal solution. Hence, we want to interpret certain s-t cuts as ways of how decisions can be taken. Cuts allow us to think about vertices that get selected, which we will interpret as selected projects. In short, we want to set the capacities so that a minimum s-t cut contains exactly the vertex s and an optimal selection of projects from the original graph. To do this we define the

auxiliary graph $H = (V, B)$ by setting

$$V := P \cup \{s, t\} \qquad \text{and} \qquad B := A \cup A^R \cup \{(s, p_i) \colon g(p_i) \geq 0\} \cup \{(p_i, t) \colon g(p_i) < 0\} \ ,$$

where $A^R := \{(p_j, p_i) \colon (p_i, p_j) \in A\}$ contains all arcs that are antiparallel to the arcs in A. We also define capacities $u \colon B \to \mathbb{Z}_{\geq 0} \cup \{\infty\}$ as follows: For $b \in B$ we set

$$u(b) := \begin{cases} 0 & \text{if } b \in A \ , \\ \infty & \text{if } b \in A^R \ , \\ g(p_i) & \text{if } b = (s, p_i) \text{ for } p_i \in P \ , \\ -g(p_i) & \text{if } b = (p_i, t) \text{ for } p_i \in P \ . \end{cases}$$

Figure 4.12 shows the graph resulting from the example in Figure 4.11. Notice that we set some capacities to ∞, which can easily be reduced to finite capacities as discussed in Remark 4.6.

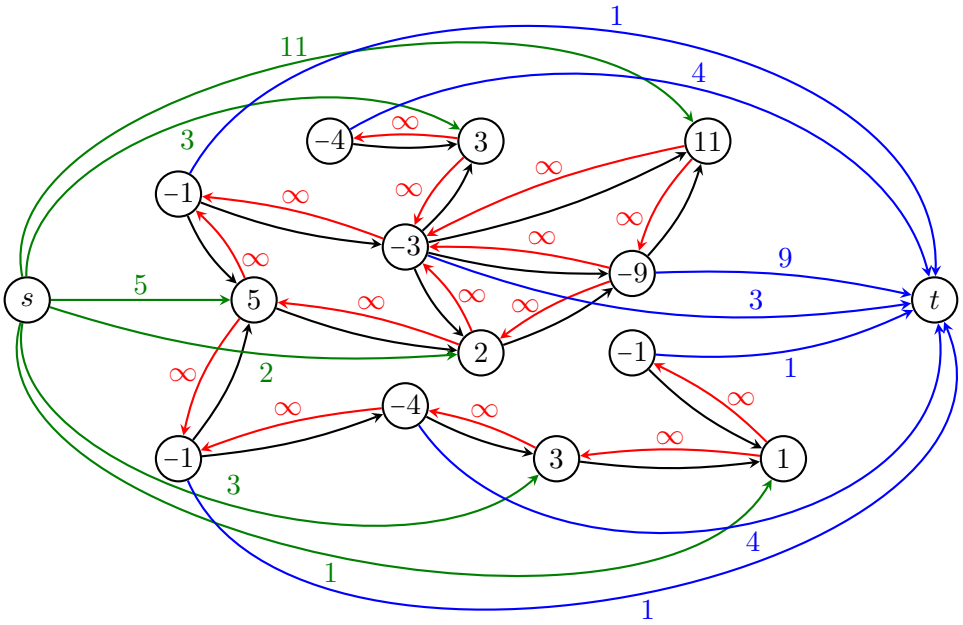


Figure 4.12: The auxiliary graph for the example in Figure 4.11. Arcs with no explicitly specified capacity have capacity 0.

The definition of this network is based on the following ideas.

- We first look at the contribution of the arcs of the form (s, p_i) and (p_i, t) for $p_i \in P$. Adding a project p_i to a given s-t cut C with $p_i \notin C$ changes its value by $u((p_i, t))$ if $g(p_i) < 0$ or by $-u((s, p_i))$ if $g(p_i) \geq 0$. By the way how we defined the capacities, the change in terms of value is $-g(p_i)$. Hence, if p_i is a project with non-negative profit $g(p_i) \geq 0$, then the cut value will decrease by $g(p_i)$ units. This is what we want because we are looking for a minimum s-t cut. Otherwise, if p_i incurs costs, then the cut value will increase.

- A minimum s-t cut C corresponds to a set of projects that fulfill the precedence constraints due to the arcs A^R, which have infinite capacities. Indeed, if C contains a project p_j but does not include some other project p_i that is a requirement for p_j, i.e., $(p_i, p_j) \in A$, then $(p_j, p_i) \in \delta^+(C)$. Hence, the value of the s-t cut C is thus $u(\delta^+(C)) \geq u((p_j, p_i)) = \infty$. Because the trivial s-t cut $\{s\}$ has finite capacity, a minimum s-t cut in H has finite capacity and, thus, fulfills all precedence constraints.

We are now formalizing these observations. Let

$$P^+ \coloneqq \{p_i \in P \colon g(p_i) \geq 0\} \ , \text{ and}$$
$$P^- \coloneqq \{p_i \in P \colon g(p_i) < 0\}$$

be the projects generating profits and losses, respectively. We consider an s-t cut C with finite cut value and first express its value $u(\delta^+(C))$ in terms of project profits and losses. Due to the second of the above observations, $\delta^+(C)$ does not contain arcs with capacity ∞. Furthermore, when calculating the value of C we can neglect the arcs with capacity 0. The only remaining arcs are those of the form (s, p_i) with capacities $g(p_i) \geq 0$ for $p_i \in P^+$ and those of the form (p_i, t) with capacities $-g(p_i) > 0$ for $p_i \in P^-$. We therefore have

$$\begin{aligned} u(\delta^+(C)) &= u(B(\{s\}, P \setminus C)) + u(B(P \cap C, \{t\})) = \\ &= g(P^+ \setminus C) - g(P^- \cap C) \\ &= g(P^+) - g(P^+ \cap C) - g(P^- \cap C) \\ &= g(P^+) - g(C \setminus \{s\}) \ , \end{aligned}$$

where $B(V_1, V_2)$, for two vertex sets $V_1, V_2 \subseteq V$, denotes the set of all arcs in B with tail in V_1 and head in V_2. The total profit of the selection $(C \setminus \{s\}) \subseteq P$ of projects is exactly $g(C \setminus \{s\})$. Hence, by minimizing the s-t cut value $u(\delta^+(C))$, we indeed maximize the total profit of the selected projects $C \setminus \{s\}$. Thus, to determine an optimal project selection, we first find a maximum s-t flow in the auxiliary graph H and then construct a minimum s-t cut C from it by Theorem 4.13. The projects $C \setminus \{s\}$ correspond to an optimal project selection. Observe that we do not need to use a maximum s-t flow algorithm that always returns an integral flow, because we are only interested in the minimum s-t cut, which can be derived in linear time from any maximum s-t flow (see Theorem 4.13).

In Figure 4.13, the vertices in a minimum s-t cut are marked in gray. The corresponding optimal selection of projects corresponds to a profit of 4 units for this example.

4.4.6 Open pit mining

In this section, we look at a planning problem inspired by open pit mining. We consider a fixed mining area and assume that, based on soil investigations, it is known—down to a certain depth—what mineral resources can be mined together with the costs incurred by mining certain areas. Figure 4.14 shows a possible soil profile, where the soil is divided into equal parts, which yield the indicated profit during mining. A negative profit indicates that the excavation costs exceed the value of the minerals contained in it. A section can only be mined if the two sections immediately above it have already been excavated. The task is to determine which parts to mine to maximize profit.

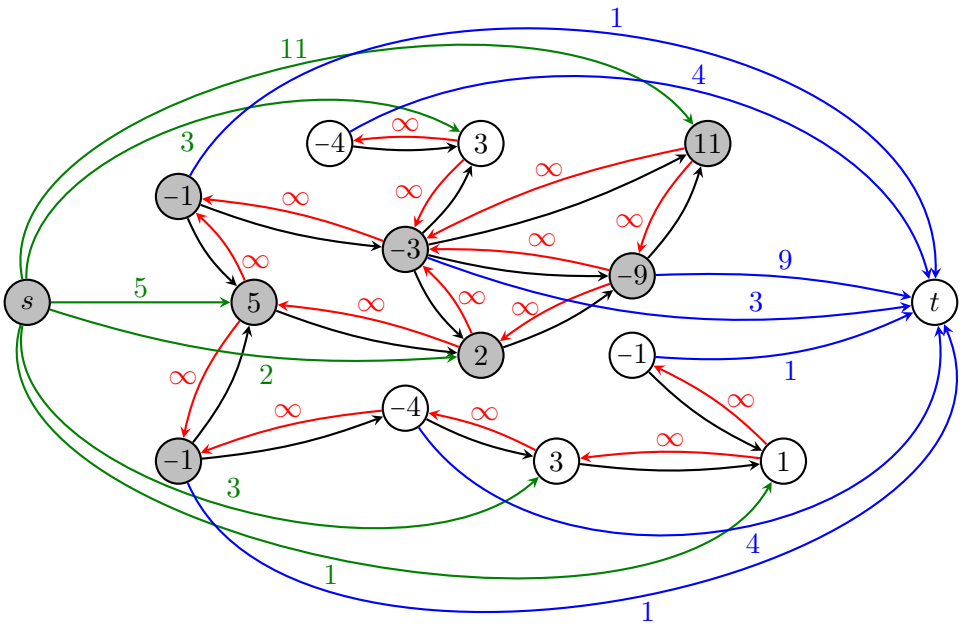


Figure 4.13: An optimal solution to the shown project selection problem. The vertices in a minimum s-t cut C are marked in gray, and the optimal set of projects to select is $C \setminus \{s\}$.

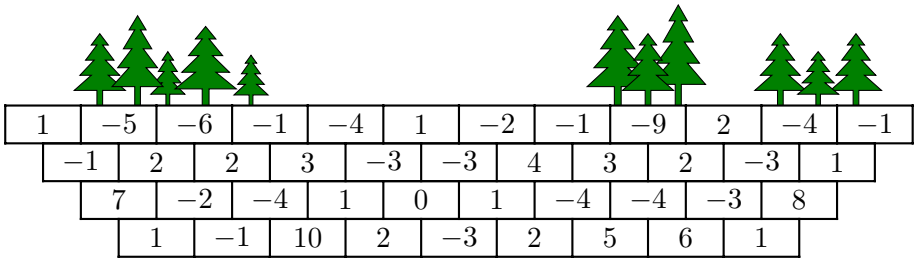


Figure 4.14: A possible soil profile with respective profits.

This problem can be rephrased as a special case of the optimal project selection problem discussed earlier, namely by considering each mining section as a project with the projects immediately above it as prerequisites. This results in the project dependency graph shown in Figure 4.15. We can thus solve these problems as described in the previous section.

4.4.7 Image segmentation

Image segmentation is a classical image processing problem. It is about splitting an image into different content-related parts, e.g., the separation of foreground and background. The version we consider is a typical problem variation faced by most image processing software. More precisely, our goal is to segment an image based on a rough manual preselection to outline the part of the image that should be separated. Figure 4.16 shows the extraction of the foreground

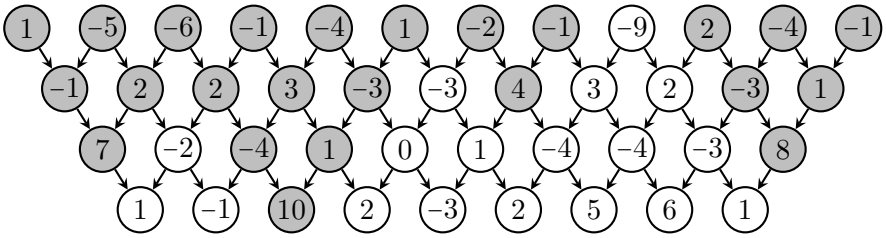


Figure 4.15: Reduction of the open pit mining problem shown in Figure 4.14 to an optimal project selection problem. The gray vertices correspond to an optimal solution.

on the example of the Mona Lisa by Leonardo da Vinci.

(a) The Mona Lisa of Leonardo da Vinci together with a manual selection.

(b) The foreground of the Mona Lisa, extracted due to color differences and manual selection.

Figure 4.16: Extraction of the foreground from an image.

We discuss an approach for image segmentation that can be modeled and solved as a flow problem. For this we represent each pixel of the image as a vertex of a graph $G = (V, A)$, add two vertices s and t as well as arcs with well-defined capacities, and look for a minimum s-t cut C. Each such cut corresponds to a part of the image. Our goal is to set the arcs and capacities so that a minimum s-t cut contains exactly the desired part of the image. In the selection of the cut C we have to find a compromise between the following two goals, which lead us to the formal definition of our auxiliary graph:

- **Not deviating too much from the manual selection.** The given manual selection is a set $W \subseteq V$ of pixels. In terms of s-t cuts, we represent this selection by the s-t cut $W \cup \{s\}$. Any deviation from $W \cup \{s\}$ should affect the value of the cut negatively, i.e., the s-t cut value should increase. We model this by fixing a penalty $x \in \mathbb{Z}_{\geq 0}$ that has to be paid for

every pixel in the returned solution that deviates from $W \cup \{s\}$. More precisely, an s-t cut $C \subseteq V$, which corresponds to the segmentation given by the pixels $C \setminus \{s\}$, would be penalized by $x \cdot |W \triangle C| = x \cdot |(W \setminus C) \cup (C \setminus W)|$. To achieve this in terms of s-t cut values, we add, for each pixel $p \in W$, an arc (s, p) with capacity $u((s, p)) = x$, and for each pixel $p \in V \setminus W$, we add an arc (p, t) with $u((p, t)) = x$. Figure 4.17 shows this construction schematically. Each vertex in the graph, except for s and t, corresponds to one pixel.

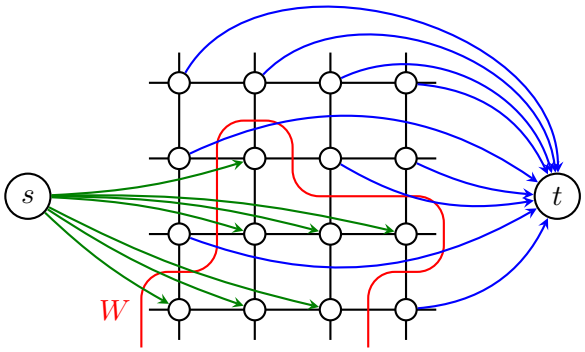


Figure 4.17: Excerpt of an image with manual segmentation W shown in red. The arcs (s, p) are shown in green and arcs (p, t) are highlighted in blue. Each of these colored arcs has equal capacity $x \in \mathbb{Z}_{\geq 0}$.

- **Cut along color differences.** The primary goal of image segmentation is to separate different objects, which typically corresponds to areas of different color or color intensity. Hence, adjacent pixels with very similar colors should typically not be separated through the segmentation. Conversely, it is normally desirable to split the image between two adjacent pixels with very different colors. To achieve this behavior for a minimum s-t cut, we introduce directed arcs (p_1, p_2) and (p_2, p_1) for any two adjacent pixels p_1 and p_2. The capacities on these arcs should be large if the colors of the two pixels are very similar. Otherwise, if the colors are very different, the capacities should be small. Hence, this punishes s-t cuts—in the sense that they have a higher value—that separate very similarly colored adjacent pixels and rewards the separation of very differently colored pixels.
  This can for example be implemented by considering the RGB color values, say with 8 bit per color. One option is to set the capacity on an arc (p_1, p_2) depending on the ℓ_1-norm of the difference of the RGB values. Hence, for (r_1, g_1, b_1) and (r_2, g_2, b_2) being the RGB color values of the neighboring pixels p_1 and p_2, the corresponding ℓ_1-norm is

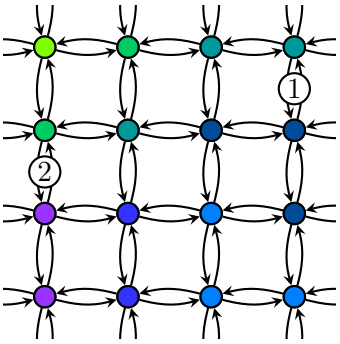
$$\|(r_1, g_1, b_1) - (r_2, g_2, b_2)\|_1 = |r_1 - r_2| + |g_1 - g_2| + |b_1 - b_2| \ .$$

  This ℓ_1-norm is small if the color values are similar and large otherwise. For the capacity $u((p_1, p_2))$, we want to achieve exactly the opposite behavior. Because each color value is in $\{0, 1, \ldots, 255\}$, we have $\|(r_1, g_1, b_1) - (r_2, g_2, b_2)\|_1 \in \{0, 1, \ldots, 3 \cdot 255 = 765\}$.

We set

$$u((p_1, p_2)) := 765 - \|(r_1, g_1, b_1) - (r_2, g_2, b_2)\|_1 \ .$$

The above choice makes sure that the capacities are non-negative (and even integral), which is required in a maximum s-t flow problem, and large capacities correspond to similar colors, as desired. Figure 4.18 illustrates the capacity calculation for arcs between neighboring pixels.



① $= (0, 153, 153)$ $= (0, 76, 153)$

$$u((\bullet, \bullet)) = 765 - |0 - 0| - |153 - 76| - |153 - 153| = 765 - 0 - 77 - 0 = 688 \ .$$

② $= (0, 204, 102)$ $= (153, 51, 255)$

$$u((\bullet, \bullet)) = 765 - |0 - 135| - |204 - 51| - |102 - 255| = 765 - 135 - 153 - 153 = 324 \ .$$

Figure 4.18: Calculation of color difference of adjacent pixels using two examples. The greater the color difference, the smaller the capacity u on the corresponding arcs.

The choice of our inter-pixel capacities was mostly to easily illustrate this modeling step. In practice, the capacities between pixels are typically not set to the ℓ_1 RGB distance, because equal RGB distances in terms of ℓ_1 norm are often not visually perceived as equal color distances.

After the implementation of these two ideas, a parameter, which we can vary, remains, namely $x \in \mathbb{Z}_{\geq 0}$. The above construction is parameterized by x, which allows for adjusting the weighing between the two penalizations introduced above, i.e., deviation from the manual selection and separation of similarly colored neighboring pixels. Through a suitable choice of x, a compromise between the two goals can be found.

Finally, the image segmentation problem can be approached via a maximum s-t flow algorithm. Again, because we are only interested in a minimum s-t cut, we do not need to find an integral maximum s-t flow.

4.4.8 Theoretical winning possibilities in sports competitions

In the course of a sports championship season it is often discussed which teams can still technically end up as (sole) leaders at the end of the season. We consider handball as an example where 2 points are awarded in every game: In case of a win, the winner gets 2 points and the loser none; in case of a draw, both get one point each. Consider a possible table of the Swiss Handball Championship shortly before the end of the season, see Table 4.1. We are interested in determining whether St. Gallen could still theoretically become sole leader of the table at the

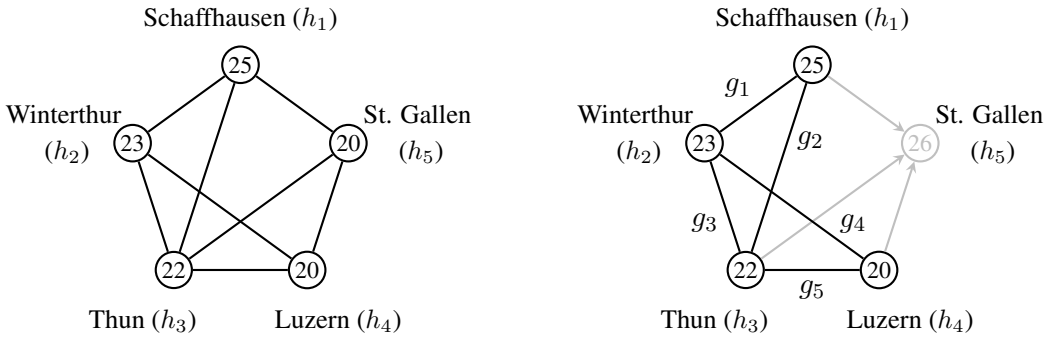
end of the season, i.e., whether it can have strictly more points at the end of the season than any other team.

rank	team	remaining games	points
1.	Schaffhausen	3	25
2.	Winterthur	3	23
3.	Thun	4	22
4.	Luzern	3	20
5.	St. Gallen	3	20
⋮	⋮	⋮	⋮

Table 4.1: A possible (partial) handball table.

To simplify the exposition, we restrict ourselves to only the first five teams as listed in Table 4.1 and assume that any other team cannot achieve strictly more than 25 points by the end of the season and, therefore, cannot exceed the currently highest score, which is held by Schaffhausen. Also, for simplicity's sake, assume that the remaining matches of the first five teams are only matches against each other. These restrictions are only assumed to simplify the exposition, and the presented technique could just as well be applied to the full table.

The remaining matches between the considered teams are given by the graph in Figure 4.19a, where an edge between two teams indicates that there is a remaining match between these teams.



(a) Remaining matches between teams: Each edge corresponds to one match.

(b) If we assume that St. Gallen wins all its games, then the 5 games $g_1, \dots, g_5$ remain.

Figure 4.19: Remaining games displayed as a graph.

To check whether St. Gallen can still become sole leader at the end of the season, we assume that it wins all remaining games. In Figure 4.19b, this is indicated by the orientation of the corresponding three arcs. In our toy example, St. Gallen then reaches 26 points, one more than

the current leader Schaffhausen has. Whereas this makes it at least plausible that St. Gallen may still be able to become sole leader, a closer look reveals that this is not possible anymore. Indeed, the three teams Schaffhausen, Winterthur, and Thun together have $25 + 23 + 22 = 70$ points, and the three remaining games between them will increase this count by another 6 points. Hence, the average number of points that these teams will have at the end of the season is $(70 + 6)/3 = 25 + 1/3 > 25$. This implies that at least one of these three teams will have 26 or more points at the end of the season, thus matching or even surpassing St. Gallen's score. This toy example illustrates that such questions are not always trivial to answer and require a careful consideration of what games still have to be played.

We now move to the general case. Assume we have n handball teams $h_1, \dots, h_n$, where team h_i has p_i points currently. Say that we want to decide whether h_n can become sole leader at the end of the season. As we did in our toy example, we assume h_n to win all remaining games. If r is the number of remaining games of h_n, then team h_n will have $p_n + 2r$ points at the end of the season. Let ℓ be the number of the remaining games $g_1, \dots, g_\ell$ not involving h_n. It remains to check whether there exist outcomes of these games such that none of the teams $h_1, \dots, h_{n-1}$ exceeds $p_n + 2r - 1$ points at the end of the season. We capture this question in terms of a flow problem on a graph $G = (V, A)$ with capacities $u \colon A \to \mathbb{Z}_{\geq 0}$ as follows:

- G has a vertex for each game $g_1, \dots, g_\ell$, a vertex for each team $h_1, \dots, h_{n-1}$, and two further vertices s and t.
- The arcs A and their capacities u are defined as follows:
  - For each game g_i, we add an arc (s, g_i) with capacity $u((s, g_i)) = 2$.
  - For each game g_i, say betweens teams h_{j_1} and h_{j_2}, we add arcs (g_i, h_{j_1}) and (g_i, h_{j_2}) with capacities $u((g_i, h_{j_1})) = u((g_i, h_{j_2})) = 2$.
  - For each team h_i with $i \in [n-1]$, we add an arc (h_i, t) with capacity $u((h_i, t)) = p_n + 2r - p_i - 1$. Here we assume without loss of generality that no team h_i for $i \in [n-1]$ has strictly more than $p_n + 2r - 1$ points; for otherwise, we can immediately observe that team h_n cannot become sole leader anymore because it cannot surpass the current score p_i of team h_i.

Figure 4.20 depicts the graph $G = (V, A)$ with capacities u for the toy example shown in Figure 4.19b.

In this network, s models the source of the game points still to be distributed. The capacities of 2 on arcs leaving s model that, for each remaining game, 2 points have to be distributed. These points can be split in any integral way among the two involved teams. This is captured by the arcs connecting games to teams. Finally, the flow on the arcs from teams to the sink model how many additional points each team will get out of the remaining games. The capacities on these arcs is set such that no team can get enough points to match or even exceed the $p_n + 2r$ points that team n will have at the end of the season. In terms of s-t flows, the problem of distributing the remaining points is the following. We need to find an integral s-t flow that saturates all arcs leaving s, because we need to assign 2 points per remaining game. This is equivalent to finding an integral s-t flow f of value $\nu(f) = 2\ell$, i.e., one that saturates all arcs leaving s. If we only want to decide whether a corresponding point assignment exists, without actually having to find the assignment, then the question reduces to checking whether an integral s-t flow of value 2ℓ

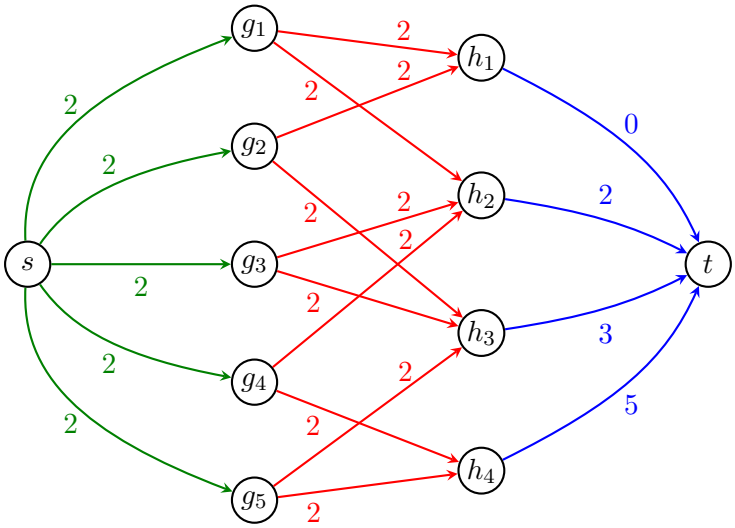


Figure 4.20: The auxiliary s-t flow network $G = (V, A)$ with corresponding capacities for the example shown in Figure 4.19b.

exists. (Note that because 2ℓ is the sum of the capacities of all arcs leaving s, there is no flow of value strictly larger than 2ℓ.) Because, by Theorem 4.16, there always exists an s-t flow of maximum value that is integral, it suffices to check whether the maximum s-t flow in G, without any integrality requirement, has value at least 2ℓ. Hence, to perform this check, we do not need to employ a maximum s-t flow algorithm that returns an integral flow. If the maximum s-t flow value is 2ℓ, then team h_n can still become sole leader by the end of the season, otherwise this is not possible anymore.

4.5 Polynomial-time variations and extensions of Ford and Fulkerson's algorithm

We come back to the fact that the maximum s-t flow algorithm of Ford and Fulkerson is not guaranteed to run in polynomial time. We recall the running time bound of $O(\alpha \cdot (m + n))$, where α is the value of a maximum s-t flow and $m := |E|$, $n := |V|$ (see Corollary 4.12). As discussed, this is not polynomial time because α may be exponential in the input size. In this section, we discuss approaches inspired by Ford and Fulkerson's algorithm that find maximum s-t flows efficiently. These are procedures that, like Ford and Fulkerson's algorithm, augment flow successively until they obtain a maximum s-t flow. We highlight that there are other classes of efficient flow algorithms, most notably so-called *preflow push* algorithms.

For simplicity, we assume that $n = O(m)$. This is not really restrictive, because whenever $m < n - 1$, then the graph is not even connected. In such a case we could first perform a BFS to check whether s and t are in the same connected component and then only work on that connected component. This leads to a graph fulfilling $n = O(m)$ because the graph is connected.

Hence, under this assumption, the running time of Ford and Fulkerson's procedure is $O(\alpha \cdot m)$, or, to avoid the dependence on the unknown quantity α, we can also bound it by $O(U \cdot m)$, where we use $U := u(A)$ as a shorthand for the sum of all capacities.

4.5.1 Capacity scaling algorithm

The capacity scaling algorithm is one of the simplest polynomial-time procedures for the maximum flow problem. It addresses the key issue of the running time bound of Ford and Fulkerson, namely the dependence on U. Scaling is an important technique with numerous applications in algorithmics. The capacity scaling algorithm for flows is a nice an instructive example of a scaling algorithm. In this context, the idea is to make augmentations along paths with large augmentation volume if possible. For this, we will consider subgraphs of the residual graph that only contain large capacities. To this end, we introduce the following notation $G_{f,\Delta}$.

Definition 4.30: $G_{f,\Delta}$

Let f be an s-t flow in the directed graph $G = (V, A)$ with capacities $u\colon A \to \mathbb{Z}_{\geq 0}$ and let $\Delta \in \mathbb{R}_{\geq 0}$. We denote by $G_{f,\Delta}$ the subgraph of the residual graph $G_f = (V, B)$ containing only the arcs with residual capacity of at least Δ.

Algorithm 6 describes the capacity scaling algorithm for maximum s-t flows. It runs in phases, defined by the outer while-loop. In each phase, it only considers augmenting paths with an augmentation volume of at least some value Δ; this is equivalent to looking for augmenting paths in $G_{f,\Delta}$. Once no more augmentations can be found during a phase, the parameter Δ is halved and the next phase begins.

Algorithm 6: Capacity scaling algorithm for maximum s-t flows

Input : Directed graph $G = (V, A)$ with arc capacities $u\colon A \to \mathbb{Z}_{\geq 0}$ and $s, t \in V$, $s \neq t$.
Output: A maximum s-t flow f.
$f(a) = 0 \ \ \forall a \in A$. `// We start with the zero flow.`
$\Delta = 2^{\lfloor \log_2(U) \rfloor}$.
while $\Delta \geq 1$ **do** `// These iterations are called phases.`
  while $\exists f$*-augmenting path* P *in* $G_{f,\Delta}$ **do**
    Augment f along P and set f to the augmented flow.
  $\Delta = \frac{\Delta}{2}$.
return f

For the running time analysis of Algorithm 6, we have to be specific on how precisely we find an f-augmenting path in $G_{f,\Delta}$. An easy way to perform this step is as follows. Instead of explicitly constructing $G_{f,\Delta}$, we construct G_f as in Ford and Fulkerson's algorithm. However, to find an augmenting path in $G_{f,\Delta}$, we then perform BFS on G_f where we slightly modify BFS to ignore arcs with a capacity strictly below Δ. This does not affect the running time of the BFS procedure.

The correctness of the algorithm follows easily from arguments identical to the ones we used

to show that Ford and Fulkerson's algorithm returns a maximum s-t flow.

Theorem 4.31

Algorithm 6 returns a maximum s-t flow.

Proof. Because the starting value of Δ is a power of two, Δ will always be integral. Consequently, the algorithm will finish after finitely many augmentations because each augmentation increases the flow value of f by at least one unit. Moreover, for the phase $\Delta = 1$, we have that $G_{f,\Delta} = G_{f,1}$ is identical to G_f, except that arcs with 0 capacity have been removed. Thus, a path is f-augmenting in G_f if and only if it is f-augmenting in $G_{f,1}$. Because the $\Delta = 1$ phase finishes with a residual graph $G_{f,1}$ without f-augmenting paths from s to t, we obtain an s-t flow f such that there is no f-augmenting path in G_f. Hence, by Theorem 4.13, the flow f is a maximum s-t flow. □

Notice that the maximum flow returned by the capacity scaling algorithm will be integral because all augmentations have an integral augmentation volume, as in the case of Ford and Fulkerson's algorithm.

We now show the crucial gain of capacity scaling in this context, namely that the running time dependence on U is only logarithmic.

Theorem 4.32

Algorithm 6 runs in $O(m^2 \log U)$ time.

Proof. The number of phases is bounded by $O(\log U)$. Hence, it suffices to show that each phase takes $O(m^2)$ time. To this end, consider a phase, which is defined by some value of Δ. Let f be the current flow at the start of the phase. Let $C \subseteq V$ be all vertices that can be reached from s in $G_{f,2\Delta}$. We claim that $t \notin C$. Indeed, if the phase we consider is the first one, i.e., $\Delta = 2^{\lfloor \log U \rfloor} > U/2$, then $2\Delta > U$; however, there is no arc with capacity strictly larger than U, implying that an augmentation volume strictly larger than U is not achievable. Otherwise, if the considered phase is not the first one, then there is no augmenting path in $G_{f,2\Delta}$ due to the termination criterion of the previous phase. Hence, the set $C \subseteq V$ of all vertices reachable from s in $G_{f,2\Delta}$ is an s-t cut, which, by definition, fulfills that for each arc $a \in \delta^+_{G_f}(C)$, we have $u_f(a) < 2\Delta$. Hence, the value of the s-t cut C in G_f is upper bounded by

$$u_f(\delta^+_{G_f}(C)) \leq 2m \cdot 2\Delta \ . \tag{4.2}$$

A crucial observation is that $u_f(\delta^+_{G_f}(C))$ is an upper bound on how much the value of f can still be augmented, i.e., the difference between the value of a maximum flow and $\nu(f)$. To see this, we recall that for each $a \in A$ we have $u_f(a) = u(a) - f(a)$, and for a reverse arc a^R of $a \in A$ we have $u_f(a^R) = f(a)$. Hence,

$$u_f(\delta^+_{G_f}(C)) = u(\delta^+(C)) - f(\delta^+(C)) + f(\delta^-(C)) = u(\delta^+(C)) - \nu(f) \ ,$$

implying, as claimed, that future augmentations will increase the value of the flow by at most $u_f(\delta^+_{G_f}(C))$, because $u(\delta^+(C))$ is an upper bound on the maximum s-t flow value by the weak max-flow min-cut theorem. Because each augmentation in the current iteration will have an augmentation volume of at least Δ, this implies, together with (4.2), that we can have at most

$$\frac{u_f(\delta^+_{G_f}(C))}{\Delta} \leq 4m$$

augmentations in the current phase, as desired. The result follows by observing that each augmentation takes $O(m)$ time. □

Notice that the proven running time bound of $O(m^2 \log U)$ is indeed polynomially bounded in the input size. For this, observe that to save the arc capacities in binary, the number of bits needed is

$$\Theta\left(m + \sum_{a \in A} \log(u(a)+1)\right) = \Theta\left(m + \log\left(\prod_{a \in A}(u(a)+1)\right)\right) = \Omega\left(m + \log U\right) \quad .$$

Hence, the input size is at least $\Omega(m + \log U)$, and the capacity scaling algorithm is thus an efficient method. However, note that it is not strongly polynomial, because the number of elementary operations depends on numbers provided in the input. We next show a method to find a maximum s-t flow in strongly polynomial time.

4.5.2 Edmonds-Karp algorithm

The Edmonds-Karp algorithm is a particular, strongly polynomial, way to realize Ford and Fulkerson's algorithm. More precisely, the Edmonds-Karp algorithm will always augment on shortest augmenting paths, i.e., augmenting paths with a minimum number of arcs. Algorithm 7 describes the procedure.

Algorithm 7: Edmonds-Karp algorithm

Input : Directed graph $G = (V, A)$ with arc capacities $u\colon A \to \mathbb{Z}_{\geq 0}$ and $s, t \in V$, $s \neq t$.
Output: A maximum s-t flow f.
$f(a) = 0 \ \ \forall a \in A$.
while $\exists f$*-augmenting path in* G_f **do**
  Find an f-augmenting path P in G_f minimizing $|P|$.
  Augment f along P and set f to augmented flow.
return f

Figure 4.21 shows an example run of the Edmonds-Karp algorithm.

As mentioned, the Edmonds-Karp algorithm is a version of Ford and Fulkerson's algorithm with a more specific rule of how to choose augmenting paths. Thus, the correctness proof of Ford and Fulkerson's method also applies to Algorithm 7. It remains to discuss the running time. Notice that finding a shortest augmenting path does not take longer than computing any augmenting path if we use BFS, because BFS automatically identifies a shortest path. Hence,

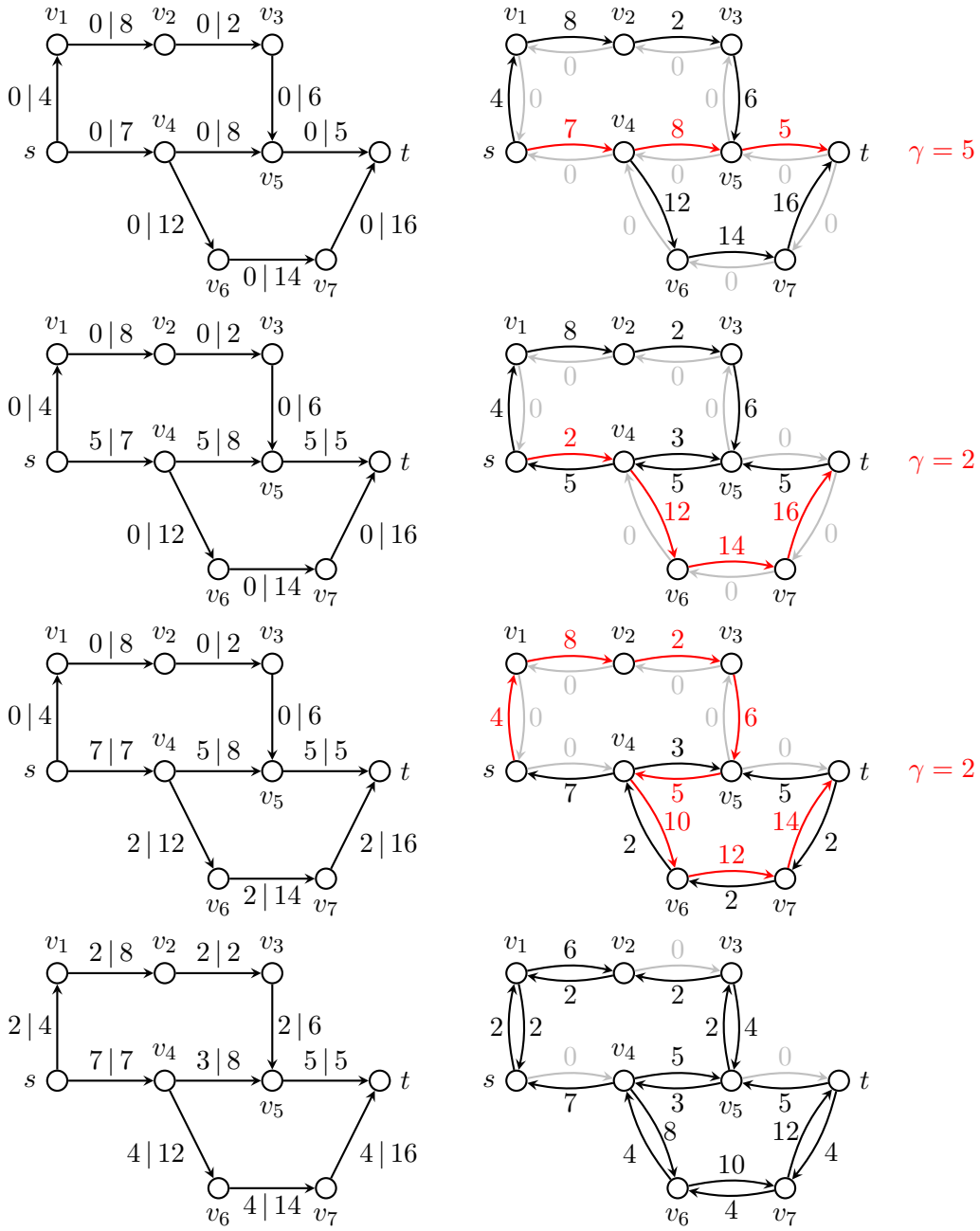


Figure 4.21: Example run of the Edmonds-Karp algorithm to find a maximum s-t flow. Each row of two figures corresponds to one iteration of the algorithm. The left-hand side graphs show the current flow values, and the right-hand side graphs are the corresponding residual graphs with a shortest augmenting path highlighted in red. The γ-values to the far right are the augmentation volumes for the highlighted augmenting paths.

also the running time bound we derived for Ford and Fulkerson's algorithm applies to Algorithm 7. We show next that the running time of Algorithm 7 is even strongly polynomial.

We want to measure the progress of Algorithm 7 in terms of the lengths of the augmenting paths found. Intuitively, we expect that augmenting paths used by the algorithm are short at the beginning and get longer later on. The following lemma implies that this is indeed the case.

Lemma 4.33

Let $G = (V, A)$ be a directed graph with arc capacities $u\colon A \to \mathbb{Z}_{\geq 0}$, and let $s, t \in V$ with $s \neq t$. Moreover, let f_1 be an s-t flow in G, and let f_2 be an s-t flow obtained by augmenting f_1 along a shortest augmenting path P in G_{f_1}. Then,

$$\begin{aligned} d_{f_1}(s,v) &\leq d_{f_2}(s,v) \quad \forall v \in V \ , \text{ and} \\ d_{f_1}(v,t) &\leq d_{f_2}(v,t) \quad \forall v \in V \ , \end{aligned}$$

where $d_f(v, w)$ denotes, for $v, w \in V$ and an s-t flow f, the length (in terms of number of arcs) of a shortest v-w path in G_f that only uses arcs with strictly positive f-residual capacity.

Proof. We only prove the first part of the statement, i.e., $d_{f_1}(s,v) \leq d_{f_2}(s,v)\ \forall v \in V$. The second part can be reduced to the first one by reversing all arc directions and flows on the arcs, and applying the first part of the lemma.

Notice that G_{f_1} and G_{f_2} have the same vertices and arcs, but they have different residual capacities u_{f_1} and u_{f_2}. Indeed the vertex set is V and the arc set is $B = A \cup A^R$, i.e., it contains the original arcs A and a reverse arc a^R for each original arc $a \in A$. We define

$$B_i \coloneqq \{b \in B\colon u_{f_i}(b) > 0\} \qquad \forall i \in [2] \ .$$

Assume with the goal of obtaining a contradiction that there is a vertex $v \in V$ such that

$$d_{f_1}(s,v) > d_{f_2}(s,v) \ . \tag{4.3}$$

Among all such vertices, we choose one where $d_{f_2}(s,v)$ is smallest. Let P_2 be a shortest s-v path in (V, B_2) and let $(w, v) \in P_2$ be the last arc in P_2. (Notice that P_2 cannot be an empty path because $s \neq v$ due to (4.3).) By our choice of v we have

$$d_{f_1}(s,w) \leq d_{f_2}(s,w) \ . \tag{4.4}$$

In particular, this implies that $(w, v) \in B_2 \setminus B_1$. For otherwise, we could extend a shortest s-w path in (V, B_1) with the arc (w, v), leading to an s-v walk in (V, B_1) of length $d_{f_1}(s, w) + 1$. Together with (4.4) this would lead to

$$d_{f_1}(s,v) \leq d_{f_1}(s,w) + 1 \leq d_{f_2}(s,w) + 1 = d_{f_2}(s,v) \ ,$$

thus contradicting (4.3). Observe that $(w, v) \in B_2 \setminus B_1$ implies $(v, w) \in P$, because only reverse arcs of arcs in P have higher u_{f_2}-capacity than u_{f_1}-capacity, which is necessary for an arc to be

in $B_2 \setminus B_1$. However, $(v, w) \in P$ is impossible because it implies $d_{f_1}(s, v) = d_{f_1}(s, w) - 1$, which leads to the following contradiction:

$$d_{f_1}(s, v) < d_{f_1}(s, w) \leq d_{f_2}(s, w) = d_{f_2}(s, v) - 1 < d_{f_1}(s, v) - 1 \ ,$$

where the second inequality is due to (4.4), the equality follows from the fact that $(w, v) \in P_2$, and the last inequality is due to (4.3). □

Theorem 4.34

Algorithm 7 runs in $O(nm^2)$ time.

Proof. We divide the different augmentation steps of Algorithm 7 into phases, where a phase corresponds to augmentations with augmenting paths of the same length. Because in a graph with n vertices the longest possible s-t path has length $n - 1$ and the shortest one has length 1, there are at most $n - 1 = O(n)$ phases. Moreover, notice that Lemma 4.33 implies that the augmenting paths found in Algorithm 7 are non-decreasing in lengths during the course of the algorithm. Hence, a phase contains augmentations done in consecutive iterations of the while-loop of Algorithm 7. We finish the proof by showing that each phase performs at most $O(m)$ augmentation, which implies the result because each augmentation takes $O(m)$ time.

To this end, consider a single phase, say the one where all augmenting paths have length $k \in [n-1]$. We call this *phase* k. For any s-t flow f and vertices $v, w \in V$, we use the notation

$$d_f(v, w) = \min \{|P| \colon P \subseteq B \text{ is } v\text{-}w \text{ path with } u_f(b) > 0 \ \forall b \in P\}$$

for the v-w distance (in terms of number of arcs) in $G_f = (V, B)$ after deleting arcs with zero f-residual capacity.

We first show the following claim.

Claim: If an arc is used in an augmenting path in phase k, then its reverse arc will never be used in an augmenting path in phase k.

To see this, assume that $(v, w) \in B$ is an arc on an augmenting path in phase k, and let f be the s-t flow right before we perform this augmentation. Because we are in phase k, the corresponding augmenting path has length k, i.e.,

$$d_f(s, v) + 1 + d_f(w, t) = k \ . \tag{4.5}$$

Moreover, because subpaths of a shortest path are shortest paths, we have

$$\begin{aligned} d_f(s, w) &= d_f(s, v) + 1 \ , \text{ and} \\ d_f(v, t) &= 1 + d_f(w, t) \ . \end{aligned} \tag{4.6}$$

For the sake of obtaining a contradiction, assume that a later augmenting path P in phase k uses the arc (w, v). Hence, P consists of:

(i) An s-w path, which has length at least $d_f(s, w)$ by Lemma 4.33,
(ii) the arc (w, v), and

(iii) a v-t path, which has length at least $d_f(v,t)$ by Lemma 4.33.

Thus, P has length

$$|P| \geq d_f(s,w) + 1 + d_f(v,t) = d_f(s,v) + d_f(w,t) + 3 = k + 2 \ ,$$

where the first equality follows from (4.6) and the second one from (4.5). However, this contradicts the assumption that P is an augmenting path in phase k, which would require $|P| = k$, and thus shows the claim.

The claim implies that whenever an arc $b \in B$ gets saturated during an augmentation in phase k, then neither the reverse arc b^R of b nor the arc b itself will be used again during the same phase. (If $b \in B$ is already a reverse arc of an arc $a \in A$, i.e., $b = a^R$, then we denote by b^R the arc a.) Indeed, b^R cannot be used due to the claim, and b cannot be used anymore because its residual capacity is zero and will not increase during the same phase again because no flow will be sent over b^R. Hence, at most m arcs can get saturated in augmentations during phase k. However, each augmentation does saturate at least one arc. Hence, there are at most $O(m)$ augmentations in each phase, as desired. □