1

Maximum flow

Network flows constitute a showcase area of theoretical computer science. The basic problems, the maximum flow problem and the minimum cut problem, are cornerstones of combinatorial optimization. They have numerous very real applications (in network connectivity, network reliability, data mining, airline scheduling, image processing, and distributed computing, to give just some of the fancier keywords), both in themselves and as subroutines in other problems. They are polynomially solvable but this is not obvious at all and requires nontrivial theory. There *is* indeed a nontrivial theory—a beautiful duality result. Finally, although the first algorithms were found in the 1950s, the playground has not been abandoned: the efficiency of maximum flow algorithms is still being pushed forward and no ultimate algorithm is in sight!

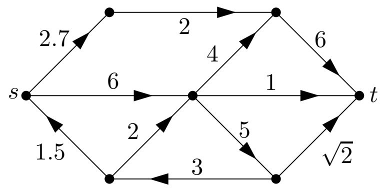
The material of this section is well-established and standard. Practically every textbook on algorithms has a chapter on network flows covering at least the basics.

1.1 Definitions and Statement of the Main Theorem

Definition 1.1.1 *A* network *is a quadruple* (G, s, t, c)*, where* $G = (V, E)$ *is a directed graph,*[1] s *and* t *are two distinct vertices of* G *(we call them the* source *and the* sink*), and the* capacity $c\colon E \to \mathbb{R}_0^+$ *is a function assigning nonnegative real values to edges.*

[1] We recall that a *directed graph* is a pair (V, E), where V is the set of vertices and $E \subseteq V \times V$ are the directed edges. An edge (u, v) is depicted by an arrow from u to v.

Here is a small example of a network:



A network can be imagined as a system of one-way water pipes, where each pipe has a given maximum throughput, with two distinguished places—the source and the sink. We let the water flow into the system at the source and at the sink it flows out. A flow is an allocation of water flows to the pipes. There cannot be more water running through a pipe than its capacity, and the system is not leaking—everything that runs into a vertex, with the possible exception of the source and sink, must go out again. By the *value* of a flow we mean the amount of the water that we have to pour into the source. Intuitively it seems obvious that this must also equal the amount pouring out from the sink, but a formal proof will take a bit of work.

Other examples of networks include telephone systems (where arcs are cables and the flow can be voice or data), electrical circuits (current flows through wires), financial networks (money flow; here some of the edges may be secret), and transportation systems (trains or cars flow, or sometimes do not flow, through roads or railroads).

Let us introduce a useful notation. Let $f\colon E \to \mathbb{R}$ be a real function on the edges of G, so far arbitrary. The *net outflow* of a vertex $v \in V$ is defined by

$$\mathrm{NetOut}_f(v) = \sum_{(v,x)\in E} f(v, x) - \sum_{(x,v)\in E} f(x, v).$$

Definition 1.1.2 *A* flow *in a network* (G, s, t, c) *is any function* $f : E \to \mathbb{R}_0^+$ *that satisfies*

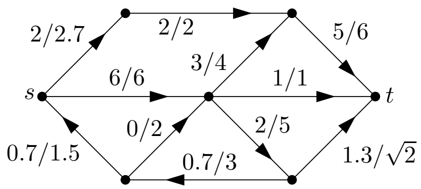
(1) $0 \le f(e) \le c(e)$ *for every edge* $e \in E$*, and*

(2) $\mathrm{NetOut}_f(v) = 0$ *for every vertex* $v \in V$ *different from* s *and* t*.*

The value *of the flow* f *is*

$$\mathrm{val}(f) = \mathrm{NetOut}_f(s).$$

In our example we can have the flow indicated below, for instance (the edge labels are flow/capacity):

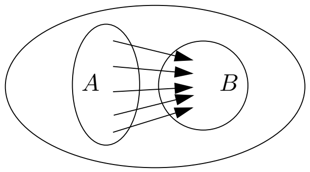


We will investigate the maximum value of a flow that can pass through a given network. The flow in our example has value 7.3 but obviously it is not maximum. First we state an existence result:

Proposition 1.1.3 *Every network has a* maximum flow*, i.e. a flow f with* $\mathrm{val}(f) \geq \mathrm{val}(f')$ *for every flow f' in the network.*

This statement is not self-evident. There are infinitely many flows and their values are generally real numbers, and so some care is needed (after all, some sets of reals, such as the open interval $(0,1)$, do not have a maximum). We will prove the existence of a maximum flow later in a roundabout way, by showing that a certain algorithm always terminates and computes a maximum flow. But a direct proof using compactness or a similar concept is not difficult.

Now we introduce a useful piece of notation and the second key notion of this section. For disjoint sets $A, B \subseteq V$ we write $E(A,B) = \{(u,v) \in E : u \in A, v \in B\}$ for the set of edges going from A to B:



(Note that the edges from B to A are *not* included.) We also write $c(A,B) = \sum_{e \in E(A,B)} c(e)$ and $f(A,B) = \sum_{e \in E(A,B)} f(e)$.

Definition 1.1.4 *An s-t-*cut *in a network (G,s,t,c) is any subset $S \subset V$ such that $s \in S$ and $t \notin S$. The* capacity *of an s-t-cut S is the number*

$$\mathrm{cap}(S) = c(S, V \setminus S).$$

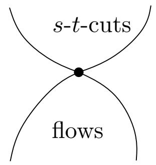
From a graph-theoretic point of view, an s-t-cut should be a special case of an *edge cut*, which is a set of edges whose removal disconnects the graph.

In our case, $E(S, V \setminus S)$ is such a set. However, for the purposes of network flows it is formally more convenient to define an s-t-cut as a special set of vertices as above.

The number of s-t-cuts, unlike the number of flows, is finite, and so clearly there is a minimum s-t-cut (an s-t-cut of minimum capacity). Now we can state the main theorem:

Theorem 1.1.5 (Main theorem about flows) *For every network the maximum value of a flow equals the minimum capacity of an s-t-cut:*

$$\max_{f \text{ a flow}} \mathrm{val}(f) = \min_{S \text{ an } s\text{-}t\text{-cut}} \mathrm{cap}(S).$$



An easy part of the theorem is that the capacity of any s-t-cut S is at least as large as the value of any flow. This is very intuitive: Whatever flows from s to t must somehow cross the boundary of S.

A harder part is the equality, which assures us that if a given flow is maximum, then there is a simple certificate for the maximality, namely, an s-t-cut with capacity equal to the value of the flow. The usual proof of the theorem also provides an algorithm for finding a maximum flow. It is not very efficient but it can be improved by additional tricks.

1.2 Proof of the Main Theorem

Lemma 1.2.1 *For every flow f and every s-t-cut S we have*

$$\mathrm{val}(f) = f(S, V \setminus S) - f(V \setminus S, S).$$

Thus the net flow over the boundary of S, written on the right-hand side, is the same for every S.

In particular, setting $S = V \setminus \{t\}$, we see that the value of a flow, which was defined as the net outflow from the source, also equals the net inflow in the sink, $\mathrm{val}(f) = -\mathrm{NetOut}_f(t)$.

Proof. We consider the sum $\sum_{v\in S}\mathrm{NetOut}_f(v)$. On the one hand, for $v \neq s$ we have $\mathrm{NetOut}_f(v) = 0$, so the sum equals $\mathrm{NetOut}_f(s) = \mathrm{val}(f)$. On the other hand, let us write out the sum explicitly and look at the contribution of an edge (u, v) to it:

$$\sum_{v\in S}\mathrm{NetOut}_f(v) = \sum_{v\in S}\Bigg(\sum_{(v,x)\in E} f(v,x) - \sum_{(x,v)\in E} f(x,v)\Bigg).$$

If both $u \in S$ and $v \in S$, then we obtain $f(u, v)$ once and $-f(u, v)$ once, so the contribution is 0. If $u \in S$ and $v \notin S$, we have only $+f(u, v)$ once, and if $u \notin S$ and $v \in S$, we have $-f(u, v)$ once. Thus we get exactly the expression in the lemma. □

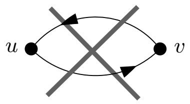
Corollary 1.2.2 *The value of every flow is bounded above by the capacity of any s-t-cut:* $\mathrm{val}(f) \leq \mathrm{cap}(S)$.

Proof.

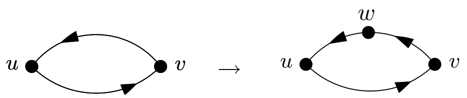
$$\mathrm{val}(f) = f(S, V \setminus S) - f(V \setminus S, S) \leq f(S, V \setminus S) \leq \mathrm{cap}(S).$$

□

The residual network. First we make an additional assumption: *The considered network never contains a pair of opposite edges (u, v) and (v, u):*



This makes many things in the sequel considerably simpler to write down. Why can we afford such an assumption? For theoretical purposes, such as the proof of the main theorem about flows, we can easily get rid of pairs of opposite edges by adding subdividing one of the edges by a new vertex:



For algorithmic purposes, where increasing the number of vertices by much could lead to loss of efficiency, we can use a little more complicated reduction, which we will not describe here, and which at most doubles the

number of vertices. If needed, one can also redo everything for networks admitting pairs of opposite edges; no new ideas are needed, only the treatment becomes less suitable for expository purposes.

Now we introduce a notion very convenient in the proof of the main theorem and also in algorithms.

Definition 1.2.3 *Let (G, s, t, c) be a network that contains no pair of opposite edges and let f be a flow in it. The* residual network (G_f, s, t, r) *is defined as follows. The vertex set of G_f is the same as the vertex set of G, and the edge set consists of directed edges specified by the following rules:*

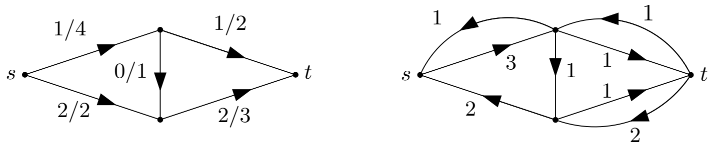
- *If e is an edge of G with $f(e) < c(e)$, then e is an edge of G_f.*
- *If $e = (u, v)$ is an edge of G with $f(e) > 0$, then the edge $e^{\mathrm{opp}} = (v, u)$ is an edge of G_f.*

The residual capacity *of an edge e of G_f is*

$$r(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E(G) \text{ and } f(e) < c(e), \\ f(e^{\mathrm{opp}}) & \text{if } e^{\mathrm{opp}} \in E(G) \text{ and } f(e) > 0. \end{cases}$$

The residual capacity of (u, v) measures how much the current flow from u to v could be increased, if needed, either by increasing the flow on the edge (u, v) as far as the capacity permits, or by decreasing the flow on the opposite edge (v, u) all the way down to 0.

The following picture shows a network with a flow (left) and the corresponding residual network (right).



Note that the residual network may contain pairs of opposite edges (even when we assume that this doesn't happen for the original network).

The following proposition, together with the existence of maximum flows (Proposition 1.1.3), immediately implies the main theorem. It is also a basis for maximum flow algorithms.

Proposition 1.2.4 *A flow f is maximum if and only if there is no directed path from s to t in G_f. For every maximum flow there is a cut S such that* $\mathrm{val}(f) = \mathrm{cap}(S)$.

Proof. First we suppose that there is a directed s-t-path P in G_f. Then the flow f can be increased along P, as we now describe. We let $\varepsilon = \varepsilon(P)$ be the minimum of the residual capacities $r(e)$ over all edges of P, and we define a new flow f' as follows: For an edge $e \in E(G)$ we set

$$f'(e) = \begin{cases} f(e) + \varepsilon & \text{if } e \text{ belongs to } P \\ f(e) - \varepsilon & \text{if } e^{\text{opp}} \text{ belongs to } P \\ f(e) & \text{otherwise.} \end{cases}$$

By the choice of ε, f' remains nonnegative and does not exceed the capacity at any edge. Since we add and subtract ε at each inner vertex of P, the net outflow of each vertex remains 0. We have a new flow with value $\text{val}(f') = \text{val}(f) + \varepsilon$. The flow f was not maximum and we have a contradiction.

Next, suppose that there is no directed s-t-path in G_f. Let $S \subset V$ be the set of vertices reachable from s by a directed path in G_f. We have $t \notin S$, and so S is an s-t-cut.

For every edge $e = (u, v) \in E(G)$ with $u \in S$ and $v \notin S$ we have $f(e) = c(e)$, for otherwise, e would be an edge of G_f and v would be reachable in G_f from s via u. Similarly, if $e = (v, u) \in E$ with $u \in S$ and $v \notin S$, then $f(e) = 0$, for otherwise, $e^{\text{opp}} = (u, v)$ would be in the residual network and v would be reachable. Thus $f(S, V \setminus S) = \text{cap}(S)$ and $f(V \setminus S, S) = 0$. By Lemma 1.2.1

$$\text{val}(f) = f(S, V \setminus S) - f(V \setminus S, S) = \text{cap}(S) - 0 = \text{cap}(S).$$

The value of f thus equals the capacity of S, and since no flow can have larger value, f is maximum. □

1.3 The Ford–Fulkerson Algorithm

Directly from the proof of the main theorem we can derive a simple algorithm for computing the maximum flow. Let us call a directed s-t-path in G_f an *augmenting path* for a flow f.

```
Ford–Fulkerson Algorithm(G, s, t, c)
  f ← the zero flow
  while an augmenting path exists
      choose an augmenting path P
      modify f along P as in the proof of the main theorem
  end while
  return the current f as a maximum flow.
```


Here are two theoretical consequences of the Ford–Fulkerson algorithm. Especially the second one is very useful.

Observation 1.3.1

(i) *If the capacities of all edges are rational, the Ford–Fulkerson algorithm is finite. For networks with rational capacities it proves the existence of a maximum flow (Proposition 1.1.3).*

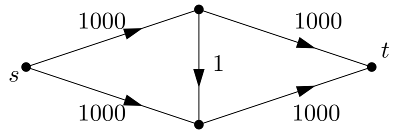
(ii) *Every network with integral capacities has an integral maximum flow.*

Proof. In (i) we may assume that the capacities are integral (if not, we first multiply them by the common denominator).

If both the capacities and the current flow are integral, then the $\varepsilon = \varepsilon(P)$ in the proof of the main theorem, by which the flow is modified along the path P, is a positive integer. The initial zero flow is integral, and so the current flow remains integral all the time.

Each step increases the value of the flow at least by 1, and hence the algorithm finishes in a number of steps bounded above by the sum of all capacities and computes an integral maximum flow. □

Unfortunately, there are networks with irrational capacities where the Ford–Fulkerson algorithm need not end, and it even need not converge to a correct solution. Even for integral capacities the algorithm may run very long if the augmenting paths are chosen in a sloppy way. For example, if we start from the zero flow and keep choosing augmenting paths of length 3 in the following network,



then the algorithm needs 2000 steps! Later we will explain better algorithms which cannot be fooled in this way.

Parameters for comparisons of algorithms. We will discuss the efficiency of the basic Ford–Fulkerson algorithm and various improved variants of it in more detail. To this end, we introduce three parameters to describe the input size of a given network:

- n is the number of vertices,
- m is the number of edges, and

- we assume that *all capacities are integral*, and we denote by U the largest edge capacity.

The running times will be expressed as functions of n, m, and sometimes also of U. We have $m \leq n^2$, and we will always assume $m \geq n-1$ (for smaller m, the network is disconnected and this case is uninteresting).

Finding an augmenting path. In order to implement the Ford–Fulkerson algorithm, and some improved versions to be discussed later, we want to compute an augmenting path, if one exists, efficiently. A directed s-t-path in the residual network can be found in $O(m+n) = O(m)$ time by a graph-search algorithm. Given such a path for the current flow, the improved flow and the updated residual network can be determined in $O(n)$ time.

We leave it to the reader to check that a maximum flow has value at most nU, and that the Ford–Fulkerson algorithm thus has running time at most $O(mnU)$. This is not so bad if the edge capacities are small integers.

1.4 Faster Augmentation: Capacity Scaling

In the example from the previous section where the Ford–Fulkerson algorithm made 2000 augmentations, the choice of the augmenting paths looks rather sloppy. In this section and in the next one we discuss rules for selecting augmenting paths that lead to much faster algorithms (although not to the best known ones).

The first idea for improving the worst-case efficiency of the Ford–Fulkerson algorithm is to use augmenting paths that increase the flow value as much as possible. For a given parameter Δ, let $G_f(\Delta)$ denote a subgraph of the residual network G_f consisting only of edges with residual capacity at least Δ.

Any directed s-t-path in $G_f(\Delta)$ corresponds to an augmenting path that can increase the value of the flow by at least Δ. The *capacity scaling algorithm* is the following variant of the Ford–Fulkerson algorithm (the capacities are assumed to be integers in range from 1 to U:


```
CapacityScaling(G, s, t, c)
  Δ ← the smallest power of 2 exceeding U
  f ← the zero flow
  repeat
      Δ ← Δ/2
      while there is a directed s-t-path P in G_f(Δ) do
          augment f along P
          update G_f(Δ) accordingly
      end while
  until Δ = 1
return f
```

So we begin with large increases in the flow and only when they are no longer possible, we go to smaller increases. Note that we do not insist on the really largest possible increase, which would complicate the computation and would not gain us much, but we are satisfied with a "reasonably large" one. This is a rather common and useful trick in the design of algorithms (be satisfied with less if the loss is not very substantial).

The algorithm correctly finds a maximum flow, since the flow remains integral all the time and so $G_f(1) = G_f$.

The algorithm goes through $O(\log U)$ values of Δ (assuming, for notational convenience, $U \geq 2$).

Lemma 1.4.1 *If there is no directed s-t-path in $G_f(\Delta)$, then* $\mathrm{val}(f)$ *differs from the value of a maximum flow by at most $m\Delta$.*

We leave a proof as an exercise. Hint: consider a suitable s-t-cut defined using $G_f(\Delta)$.

As another exercise, the reader is invited to check that for each value of Δ, the algorithm makes no more than $2m$ augmentations (control question: why the 2?). Thus the total running time is $O(m^2 \log U)$.

1.5 Shortest Augmenting Paths

Another simple idea for improving the efficiency of the Ford–Fulkerson algorithm is this: *Always choose an augmenting path with the smallest possible number of edges.* So for the current flow f we find a shortest directed s-t-path in G_f, say by breadth-first search, and use it for augmenting f.

Let $\ell(G_f)$ be the length of the shortest directed s-t-path in G_f, and let $\mathrm{SPE}(G_f)$ denote the set of all edges of G_f that lie on at least one shortest directed s-t-path in G_f. If a flow g is obtained from f by augmentation along a shortest path P, then all new edges in the residual network G_g are opposite to edges of P, and so $E(G_g) \subseteq E(G_f) \cup \mathrm{SPE}(G_f)^{\mathrm{opp}}$, where we use the shorthand $F^{\mathrm{opp}} = \{e^{\mathrm{opp}} : e \in F\}$. The next lemma describes the effect of adding edges of $\mathrm{SPE}(G_f)^{\mathrm{opp}}$ to G_f.

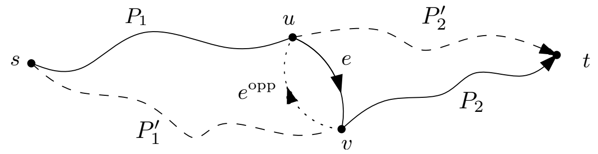
Lemma 1.5.1 *Let H be a directed graph with two distinguished vertices s and t, and let H^+ be the directed graph with $V(H^+) = V(H)$ and $E(H^+) = E(H) \cup \mathrm{SPE}(H)^{\mathrm{opp}}$. Then*

(i) $\ell(H^+) = \ell(H)$.

(ii) $\mathrm{SPE}(H^+) = \mathrm{SPE}(H)$.

Proof. Let e be an edge of $\mathrm{SPE}(H)$, and let H' be obtained by adding the edge e^{opp} to H. It suffices to prove that $\ell(H') = \ell(H)$ and $\mathrm{SPE}(H') = \mathrm{SPE}(H)$; the claim of the lemma then follows by induction (adding the edges of $\mathrm{SPE}(H)^{\mathrm{opp}}$ one by one).

Since $E(H) \subseteq E(H')$, we have $\ell(H') \le \ell = \ell(H)$. For contradiction let us suppose that H' has an s-t-path P' of length $\ell' < \ell$. Such P' has to use the edge e^{opp}, for otherwise, it would be contained in H. Let $e = (u, v)$, let P_1' be the segment of P' from s to v, of length k_1', and let P_2' be the segment of P' from u to t, of length k_2'. Then $k_1' + k_2' = \ell' - 1 \le \ell - 2$. At the same time, $e \in \mathrm{SPE}(H)$ lies on some shortest s-t-path P in H. Letting k_1 be the length of the segment P_1 of P from s to u and k_2 the length of the segment P_2 from v to t, we have $k_1 + k_2 = \ell - 1$.



Then both $P_1 \cup P_2'$ and $P_1' \cup P_2$ are s-t-paths in H, of length $k_1 + k_2'$ and $k_1' + k_2$, respectively. These lengths should be at least $\ell(H)$, and so $k_1 + k_2 + k_1' + k_2' \ge 2\ell$. But previously we have derived $k_1 + k_2 + k_1' + k_2' \le \ell - 1 + \ell - 2 = 2\ell - 3$, and this is the desired contradiction proving $\ell(H') = \ell(H)$.

Next, we want to show $\mathrm{SPE}(H') = \mathrm{SPE}(H)$. The inclusion $\mathrm{SPE}(H) \subseteq \mathrm{SPE}(H')$ is clear. If the reverse inclusion did not hold, there would be an s-t-path P' of length ℓ in H' that is not present in H, and so P' would have to use e^{opp}. But then we can repeat the previous argument, with the only change that now $k_1' + k_2' = \ell - 1$. This leads to $k_1 + k_2 + k_1' + k_2' = 2\ell - 2$, which is still a contradiction. The lemma is proved. □

Let us return to our previous setting, where g was obtained from f by a single shortest-path augmentation. The lemma and the observation $E(G_g) \subseteq E(G_f) \cup \mathrm{SPE}(G_f)^{\mathrm{opp}}$ imply that either $\ell(G_g) > \ell(G_f)$, or $\ell(G_g) = \ell(G_f)$ and $\mathrm{SPE}(G_g) \subseteq \mathrm{SPE}(G_f)$. Moreover, the inclusion has to be proper, since the edges of the minimum residual capacity on the augmenting path disappear from the residual network.

Therefore, the algorithm always using a shortest augmenting path works in phases: First it augments with paths of some length k_1, then with paths of length $k_2 > k_1$, and so on. Altogether there are at most n phases, and during each phase, the length of the augmenting paths remains fixed. Each phase has at most $2m$ augmentations, since initially $\mathrm{SPE}(G_f)$ has no more than $2m$ edges (a more detailed consideration even shows $|\mathrm{SPE}(G_f)| \le m$), and each augmentation decreases the number of edges in the current $\mathrm{SPE}(G_f)$ at least by one. We have thus proved:

Theorem 1.5.2 *The variant of the Ford–Fulkerson algorithm with shortest augmenting paths has at most n phases with $O(m)$ augmentations per phase. Thus the total running time is at most $O(m^2 n)$, independent of the capacities.*

This theorem implies the existence of a maximum flow for an arbitrary network, possibly with irrational capacities.

A further improvement: the Dinits algorithm. There are examples of networks where the discussed algorithm indeed makes about mn augmentations. To improve the bound on the running time, one has to decrease the (average) time per augmentation. There is a simple implementation, due to Dinits, that guarantees $O(mn)$ total time per phase, resulting in an $O(mn^2)$ algorithm; this is a significant improvement for large m.

First we establish simple results about the graphs $\mathrm{SPE}(G_f)$.

Lemma 1.5.3 *Let H be a directed graph with distinguished vertices s and t. Let $\vec{d}_H(x, y)$ denote the length of a shortest directed path from x to y in H (thus $\ell(H) = \vec{d}_H(s, t)$).*

(i) *An edge $(u, v) \in E(H)$ belongs to $\mathrm{SPE}(H)$ if and only if $\vec{d}_H(s, u) + \vec{d}_H(v, t) + 1 = \ell(H)$.*

(ii) *Every directed s-t-path in $\mathrm{SPE}(H)$ has length exactly $\ell(H)$.*

Proof. A shortest path from s to t that contains the edge (u, v) consists of a shortest s-u-path, the edge (u, v), and a shortest v-t-path, and thus it has length $\vec{d}_H(s, u) + 1 + \vec{d}_H(v, t)$. This implies (i).

A similar reasoning shows that $\vec{d}_H(s,v)+\vec{d}_H(v,t)=\ell(H)$ for any vertex v lying on at least one shortest s-t-path. From this one shows that if $(u,v)\in$ SPE(H), then $\vec{d}_H(v,t)=\vec{d}_H(u,t)-1$. Then (ii) follows since by traversing any edge $(u,v)\in$ SPE(H) we reduce the distance to t by one. □

Now we describe one phase of the Dinits algorithm (during which the length of the augmenting paths remains fixed). As was remarked above, given the current flow and an augmenting path, the flow can be augmented in $O(n)$ time, so we need to worry only about finding a shortest s-t-path in the current residual network.

Let f_0 be the current flow at the beginning of the considered phase. We assume that G_{f_0} has some s-t-path, since otherwise, f_0 is a maximum flow.

First we compute SPE(G_{f_0}). This can be done in $O(m)$ time using Lemma 1.5.3(i): By breadth-first search we calculate the oriented distances of all vertices of G_{f_0} from s and to t, and we identify all edges of SPE(G_{f_0}) using the condition in the lemma.

During the phase we will maintain a directed graph H on the vertex set $V(G)$, which will be used for finding the augmenting paths. Initially we set $E(H)=$ SPE(G_{f_0}), and then we modify H only by edge deletions; therefore,

$$E(H)\subseteq \mathrm{SPE}(G_{f_0}) \tag{1.1}$$

always holds.

Whenever we augment the current flow along some shortest s-t-path, we delete from H all edges of this path that are deleted from the residual network. In this way, we make sure that

$$E(H)\subseteq E(G_f) \tag{1.2}$$

always holds, where f is the current flow. By Lemma 1.5.3(ii) these two conditions, (1.1) and (1.2), imply that any s-t-path in H is a shortest s-t-path in G_f, and thus it can be used for augmentation.

To make the search for s-t-paths in H more efficient, we will also be deleting edges of H during the search, as is specified below. We will be careful not to delete any edge of SPE(G_f). Then we have

$$\mathrm{SPE}(G_f)\subseteq E(H) \tag{1.3}$$

during the whole phase, and so if there is no s-t-path in H, we know that the phase should finish, since G_f has no s-t-path of the current length.

It remains to describe how do we find s-t-paths in H. We proceed by depth-first search, starting from s and advancing forward. If we reach t directly without backtracking, we have found an s-t-path in time proportional to its length. It may happen, though, that instead of reaching t we arrive at a vertex of outdegree 0 (the graph H is acyclic, since SPE(G_{f_0}) is acyclic, and so these are the only two possibilities). Then we have to backtrack and this might take long. However, we observe that no edge of H ending in a vertex of outdegree 0 lies on any s-t-path in H. Therefore, such edges can be deleted from H without violating (1.3).

After each augmentation in the current phase, and after we have deleted from H all edges that disappeared from the residual network by the last augmentation, we perform the following "search with deletions".

Initially we set v to s and P to an empty path (no edges); throughout the procedure, P will be a path from s to v in H. In each step, we first test whether $v=t$; if yes, we return P as the next augmenting path. If not, we test whether v has any outgoing edge in H. If yes, we choose such an edge (v,w), we add it to the current path P, set v to w, and go to the next step. If not, i.e. if v has outdegree 0, we first check if $v=s$, and if yes, we finish, since no s-t-path exists in H (and the current phase of the Dinits algorithm is over). Othewrise, for $v\neq s$, we delete all incoming edges of v from H, and we retract the path P by one edge. That is, if the last edge of P is (u,v), we remove it from P, we set v to u, and we continue by the next step.

The correctness of the algorithm follows from the previous discussion, since (1.1), (1.2), (1.3) are maintained. It remains to estimate the running time. We note that each invocation of the described procedure needs time at most $O(\ell+t)$, where ℓ is the length of the returned path and t is the number of edges deleted from H, since whenever an edge is removed from the current P, it is also deleted from H. Since, as was proved earlier, each phase has at most m augmentations, the total time per phase is at most $O(m\ell+m)=O(mn)$.

Before we leave the Dinits algorithm, let us note that for a network with unit edge capacities it needs at most $O(m)$ time per phase, as opposed to $O(mn)$ for arbitrary capacities. This is because in this case *all* edges of the augmenting path are always deleted from the residual network, and thus also from the graph H. The number of augmentations in a phase is then no more than m/ℓ, and altogether they take $O(m)$ time.

1.6 A Glance of Other Maxflow Algorithms

The algorithms discussed so far maintain a flow during the whole computation and gradually increase its value. The current best maximum flow algorithms, both from the theoretical point of view (asymptotic complexity in the worst case) and in practice (speed of actual programs on benchmark inputs), are of a different type. They do not maintain a flow, only a *preflow*, which is like a flow but Kirchhoff's law is replaced by a weaker condition: the net inflow of each vertex, except possibly for s, is nonnegative. This relaxation gives the algorithms additional flexibility. These modern algorithms, and especially their analysis, are not simple, and they constitute a topic suitable for a more specialized course. (Old ideas do not get completely discarded, though: The current asymptotically fastest algorithm can be viewed as a successor of the Dinits algorithm, and it also uses a kind of capacity scaling . . .)

The following table (based on a survey by Goldberg) summarizes some of the history of maximum flow algorithms.

Year	Discoverer(s)	Method	$O(.)$ time bound
1951	Dantzig	Simplex	mn^2U
1955	Ford, Fulkerson	Augmenting path	mnU
1970	Edmonds, Karp	Shortest path	m^2n
1970	Dinits	Shortest path	mn^2
1972	Edmonds,Karp; Dinits	Capacity scaling	$m^2 \log U$
1973	Dinits; Gabow	Capacity scaling	$mn \log U$
1974	Karzanov	Preflow push	n^3
1983	Sleator, Tarjan	Dynamic trees	$mn \log n$
1986	Goldberg, Tarjan	FIFO preflow push	$mn \log(n^2/m)$
		. . . .	
1997	Goldberg, Rao		$mn^{2/3} \log \frac{n^2}{m} \log U$ $m^{3/2} \log \frac{n^2}{m} \log U$

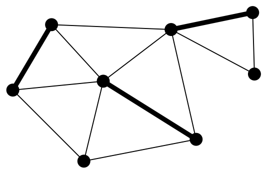
Maximum flow as a linear program. We have discussed the algorithms corresponding to the second through fifth lines of the table. The first line is based on a formulation of the maximum flow problem using linear programming and solving it by some version of the simplex algorithm. The formulation is straightforward: we introduce one variable f_e for every edge e of the network, and we maximize $\sum_{(s,x)} f_{(s,x)} - \sum_{(x,s)} f_{(x,s)}$ under the conditions $0 \leq f_e \leq c(e)$ for every e and $\sum_{(v,x)} f_{(v,x)} - \sum_{(x,v)} f_{(x,v)} = 0$ for every vertex $v \neq s, t$. Then we can use any linear programming algorithm to compute a maximum flow.

This is not a too efficient way of computing a maximum flow, but it has some conceptual advantages. Having a linear programming algorithm at disposal, we get a maximum flow algorithm practically for free. The main theorem about flows (max flow = min cut) can also easily be derived from the so-called duality theorem of linear programming; in this way, one also gets for free the existence of a maximum flow for arbitrary real capacities. When approaching a new problem, it is usually advisable to try some powerful general tools first, such as linear programming, and only if such general methods are not successful or are not fast enough, one can attempt to develop a new algorithm tailored to the particular problem.

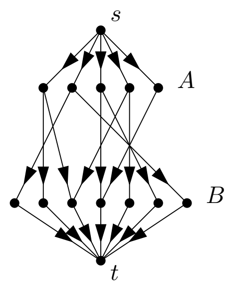
1.7 Bipartite Matchings, Hall's Theorem, and Consequences

Maximum flow algorithms yield algorithms for maximum matching in bipartite graphs by a simple reduction.

Definition 1.7.1 *A* matching *in a graph G is a set of edges $F \subseteq E(G)$ such that every vertex $v \in V(G)$ belongs to at most one edge of F.*



Given a bipartite graph $H = (V, E)$, where V is divided into two color classes A and B and edges go only between A and B, we construct a network (G, s, t, c) as in the picture:



We add s and t as new vertices, connect s to all vertices of A, all vertices of B to t, and direct the edges as indicated. All edge capacities are set to 1. We can find a maximum flow f in this network by the Ford–Fulkerson algorithm. As we know, f is integral, and thus the flow values are 0 and 1. It is easy to see that the edges between A and B with flow value 1 form a matching in H. Conversely, from any matching with k edges we can derive a flow with value k.

The Ford–Fulkerson algorithm can thus be used for finding a matching with the maximum possible number of edges in a bipartite graph in $O(mn)$ time. (The resulting maximum matching algorithm is sometimes presented without referring to flows; it is called the *Hungarian method*.) It can also be shown that the Dinits algorithm used for the maximum bipartite matching problem needs at most $O(m\sqrt{n})$ time. This can be proved in two steps: First, as we have noted in the section on the Dinits algorithm, if all capacities are 1's, a single phase runs in $O(m)$ time. Second, one shows that after a phase with augmenting paths of length k, the value of the current flow differs from the maximum value by at most n/k, and thus after phase $\sqrt{n}$, a maximum flow is reached by at most $O(\sqrt{n})$ more augmentations. (This is left as a somewhat challenging exercise.)

Hall's theorem. Next, we use network flows to derive a basic combinatorial principle. We consider a system $\mathcal{M}$ of subsets of a finite set X. For notational convenience, we assume that the sets of $\mathcal{M}$ are indexed by elements of a finite set I, so $\mathcal{M} = (M_i : i \in I)$.

Definition 1.7.2 *A* system of distinct representatives (SDR) *for a set system $\mathcal{M}$ as above is a function $f\colon I \to X$ such that*

(i) *$f(i) \in M_i$ for every $i \in I$, and*

(ii) *f is injective (one-to-one).*

In other words, a SDR is a choice of one element from each set M_i such that all the chosen elements are distinct.

SDR's can equivalently be described as matchings in suitable bipartite graphs. We define, for a given set system $\mathcal{M}$, the bipartite graph

$$B_{\mathcal{M}} = (I \cup X, \{\{i, x\} : x \in M_i\}),$$

called the *incidence graph* of the system $\mathcal{M}$.

Observation 1.7.3 *A system of distinct representatives for $\mathcal{M}$ exists if and only if there exists a matching of size $|I|$ in the incidence graph $B_{\mathcal{M}}$.*

The following important theorem is an example of a statement where the obvious necessary condition is also sufficient.

Theorem 1.7.4 (Hall) *A system of distinct representatives for $\mathcal{M}$ exists if and only if for every $J \subseteq I$*

$$\Big|\bigcup_{i\in J} M_i\Big| \geq |J|;$$

this condition is called Hall's condition.

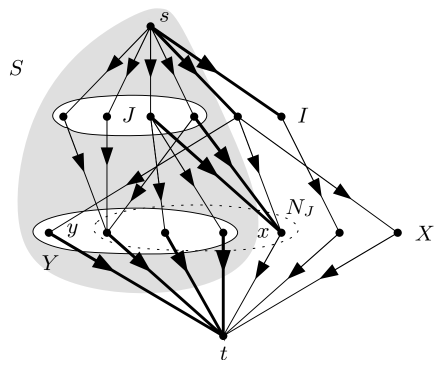
Proof. We begin with the easier implication; let there be a system of distinct representatives in $\mathcal{M}$. We choose an arbitrary $J \subseteq I$. For every index $j \in J$ there is an element $p_j \in M_j$ so that all the p_j are distinct. Therefore

$$\Big|\bigcup_{j\in J} M_j\Big| \geq |\{p_j : j \in J\}| = |J|.$$

For the other implication (the real thing) let us have a set system $\mathcal{M}$ that satisfies Hall's condition. We take the bipartite graph $B_{\mathcal{M}}$ and transform it into a network exactly as in the picture above.

As was explained above, it is enough to show that a maximum flow in this network has value $|I|$. By the main theorem about flows, it suffices to show that every s-t-cut has capacity at least $|I|$.

Let S be an s-t-cut (any subset of vertices containing s and not t) and let $J = S \cap I$ and $Y = X \cap S$ (the edges counted in the capacity of S are drawn thick):



We let N_J denote the set of all neighbors of the vertices of J in X. First, if we delete from S a vertex $y \in Y \setminus N_J$, the capacity decreases by 1, and so we may assume that $Y \subseteq N_J$. Next, if we add a vertex $x \in N_J \setminus Y$ to S, the capacity does not increase (in the picture, the arrow (x, t) becomes thick but the thick arrow going into x disappears). So we may assume $S = \{s\} \cup J \cup N_J$.

What is the capacity of such an S? The edges leaving S are those connecting s to vertices of $I \setminus J$, whose number is $|I| - |J|$, and then we have $|N_J|$ edges from N_J to t. We have $N_J = \bigcup_{i \in J} M_i$, and so by $|N_J| \geq |J|$ by Hall's condition. The capacity of any s-t-cut is thus at least $|I| - |J| + |N_J| \geq |I|$, and Hall's theorem is proved. □

Hall's theorem can also be proved directly by mathematical induction, without using flows.

We state two nice consequences of Hall's theorem (among many). The reader can try to prove them as (moderately difficult) exercises.

Proposition 1.7.5 *Let H be a bipartite graph with color classes A and B, and suppose that the minimum degree of a vertex in A is at least as large as the maximum degree of a vertex in B. Then there is a matching containing all vertices of A. In particular, if H is regular (all vertices have the same degree), then it has a* perfect matching*, i.e. a matching covering all vertices.*

An $m \times n$ Latin rectangle is a table with m rows and n columns, $n \geq m$, where each entry is one of the numbers $1, 2, \ldots, n$, and no row and no column contain any number more than once. If $m = n$, we speak about a *Latin square*. Latin squares are well-studied combinatorial objects, related to things like finite projective planes and combinatorial designs.

Proposition 1.7.6 *Every Latin $m \times n$ rectangle can be completed to an $n \times n$ Latin square.*