3 Flows

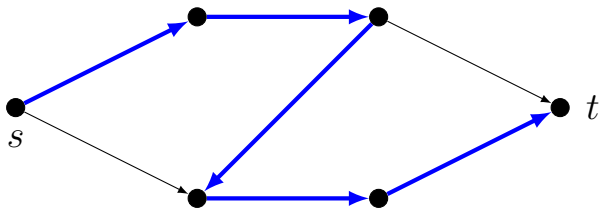
3.1 Disjoint Paths • 3.2 Flows

In this chapter, all graphs will be directed. The graphs will come with a source vertex s and a terminal vertex t. We will make the assumption that $\delta^{\text{in}}(s) = \emptyset$ and $\delta^{\text{out}}(t) = \emptyset$ (for the problems we are considering, these edges could be ignored anyway), and that the graphs have no parallel or reverse edges. All paths considered will be *directed* paths.

3.1 Disjoint Paths

EDGE-DISJOINT PATHS PROBLEM: Given a directed graph G and $s, t \in V(G)$, find the maximum number of edge-disjoint st-paths.

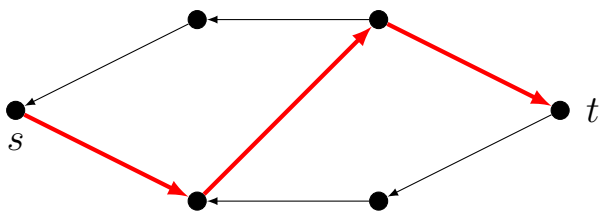
Greedy?
Let's try a greedy approach: find any st-path, remove it, find another one, etc. Here's a graph for which this will not give the maximum number of paths (2), if the first path we choose is the one in blue:



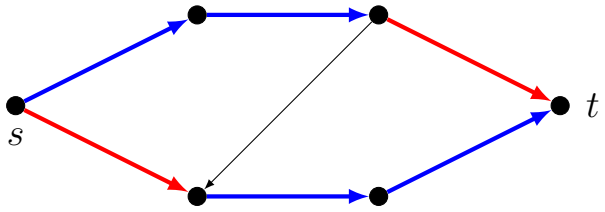
Reversal trick
We can fix the greedy approach with the following trick (although the result will no longer be greedy): Given a path, *reverse* its edges, and then look for the next path.
Write $(uv)^{-1} = vu$, for $S \subset E(G)$ write $S^{-1} = \{e^{-1} : e \in S\}$, and for any graph H write $H^{-1} = (V(H), E(H)^{-1})$.
After finding a path P, modify the graph to

$$G' = G - P + P^{-1}.$$

By this notation I just mean to remove the edges of P, then add their inverses; it does not mean that vertices of P, or adjacent edges, should be removed.
Below is the result for the graph from above, with a new path found in this graph, in red.



The good thing now is that the blue path P and the red path Q together give two edge-disjoint paths in the original graph, by removing the edge where they overlap (i.e. go against each other), and merging the remainders, like so:



This trick works more generally.

Lemma 3.1. *Given k edge-disjoint paths P_i in G, define $G' = G - \sum P_i + \sum P_i^{-1}$. If there is a path Q in G', then there are $k+1$ edge-disjoint paths R_i in G.*

Proof. Let $S = (\bigcup E(P_i)) \cap E(Q)^{-1}$, i.e. the edges of the P_i that overlap (in reverse) with Q. Take $T = ((\bigcup E(P_i)) \setminus S) \cup (E(Q) \setminus S^{-1})$, i.e. the edges of Q and the P_i that remain after all the overlapping ones are removed, and $H = (V(H), T)$.
For all vertices $v \neq s, t$ we have $\deg_H^{\text{in}}(v) = \deg_H^{\text{out}}(v)$. This is because H is a union of paths, from which we only remove edges in reverse pairs; each time we remove a reverse pair of edges, their endpoints have their in-degree and out-degree lowered by 1.
Furthermore, $\deg_H^{\text{out}}(s) = k+1$, since we have k edges out of s for the paths R_i, and one more from Q. None of these is removed, since that would mean Q has an edge into s, which is impossible. It follows that from each of s's $k+1$ outgoing edges, we can trace a path that has to end in t, since at each vertex in-degree equals out-degree. These are $k+1$ edge-disjoint paths.
(Note that H may also contain cycles, but we can just ignore those.) □

Algorithm
This trick gives us the following algorithm.

Algorithm for edge-disjoint *st*-paths in an unweighted directed graph

1. Set $S = \emptyset$;
2. Let $G' = G - \sum_{P \in S} P + \sum_{P \in S} P^{-1}$.
3. Find an st-path Q in G'; if none exists, go to 5;
4. Set $S = \{R_i\}$ where R_i are as in the lemma; go back to 2;
5. Return S.

*Step 4:*Note that step 4 is stated indirectly, just because otherwise the notation would be a bit cumbersome. Basically, it is a subalgorithm that computes H and then finds the $k+1$ paths in H (for instance using BFS).

Polynomial running time: It should be clear that the algorithm is polynomial; we won't prove it here because it follows from the polynomiality of the flow algorithm below. As usual (in this course), one could implement it much more efficiently, by not completely recomputing the G' and H every time.

Linear programming formulation

We don't know yet if the algorithm above actually finds the maximum number of disjoint paths; we merely know that it finds a set of paths that cannot be improved on with the reversal trick. Let's see if linear programming can help.

We can write an LP in a similar way to the one we saw for shortest paths:

LP for Edge-Disjoint Paths

$$\text{maximize} \sum_{sw \in \delta^{\text{out}}(s)} x_{sw} \quad \text{with } 0 \le x \le 1, \ \boxed{x \in \mathbb{Z}^{|E|}},$$

$$\sum_{e \in \delta^{\text{in}}(v)} x_e - \sum_{e \in \delta^{\text{out}}(v)} x_e = 0 \quad \text{for } v \in V \backslash \{s, t\}.$$

But this isn't ideal, because unlike for shortest paths, we really need the $x \le 1$ condition, which makes the dual a bit annoying (we'd need a variable for every vertex constraint, and another variable for every $x_e \le 1$ constraint, see the dual for the maximum flow problem).

A more convenient LP is the one below, which uses the set $\mathcal{P} = \{st\text{-paths}\}$. Its constraints simply say that every edge can be contained in at most one chosen path.

LP for Edge-Disjoint Paths

$$\text{maximize} \sum_{P \in \mathcal{P}} x_P \quad \text{with } x \ge 0, \ \boxed{x \in \mathbb{Z}^{|\mathcal{P}|}},$$

$$\sum_{P \ni e} x_P \le 1 \quad \text{for } e \in E.$$

Not all feasible solutions of the relaxation are sets of disjoint paths, but an optimal solution x is a convex combination of maximum sets of disjoint paths, i.e. $x = \sum c_i s_i$ with $c_i \ge 0$, $\sum c_i = 1$, and each s_i corresponds to a set with the maximum number of disjoint paths. Again, we won't prove this here, because it will follow from a later theorem.

This form is much easier to dualize (the relaxation of):

Dual of relaxation

$$\text{minimize} \sum_{e \in E} z_e \quad \text{with } z \ge 0,$$

$$\sum_{e \in P} z_e \ge 1 \quad \text{for } P \in \mathcal{P}.$$

Separating sets: An integral feasible dual solution corresponds to a set of edges that we call an *st-separating set*: a set of edges such that every st-path contains at least 1 of these edges. In other words, removing these edges separates s from t.

Cuts: A *minimal* separating set is called an *st-cut* (or just *cut*): a set of edges of the form $\delta^{\text{out}}(S)$ for some $S \subset V(G)$ with $s \in S$, $t \notin S$. So a cut is a feasible dual solution, but an integral feasible dual solution is only a cut if it contains no integral feasible solution with fewer edges.

(Note that not every text uses quite the same definition of cuts.)

Minimum cuts: A *minimum cut* is a cut with the minimum number of edges, i.e. an optimal dual solution. As for the primal problem, every optimal dual solution is a convex combination of integral optimal dual solutions, i.e. minimum cuts.

Theorem 3.2. *The algorithm finds a set with the maximum number of edge-disjoint paths.*

Proof. The proof is by duality: Given S that the algorithm has terminated with, we show that there is a cut with $|S|$ edges, which implies that both are optimal solutions.
Let $G' = G - \sum_{P\in S} P + \sum_{P\in S} P^{-1}$ as in the algorithm. Define

$$U = \{u \in V(G) : \exists su\text{-path in } G'\}.$$

Since the algorithm terminated with S, there is no st-path in G', so $t \notin U$, hence $\delta^{\text{out}}(U)$ is a cut (in G).
Suppose $uv \in \delta^{\text{out}}(U)$, with $u \in U$, $v \notin U$. Then $uv \notin G'$, otherwise we would have $v \in U$. Thus $uv \in P$ for some $P \in S$; we claim that this is a 1-to-1 correspondence between $\delta^{\text{out}}(U)$ and S. Clearly, every $P \in S$ has such an edge uv. Suppose P contains two edges leaving U; then it must also have an edge vu going *into* U, with $v \notin U$, $u \in U$. But this edge would have been reversed in G', which implies $v \in U$, contradiction.
So we have a primal feasible solution x corresponding to S and a dual feasible solution z corresponding to $\delta^{\text{out}}(U)$, with $\sum x_P = |S| = |\delta^{\text{out}}(U)| = \sum z_e$. By the duality theorem, both must be optimal solutions. □

3.2 Flows

The maximum flow problem is a generalization of the disjoint paths problem, and it is best introduced directly as a linear program. It is similar to the first LP for disjoint paths above, but instead of $x \le 1$ it has $x_e \le c_e$.

LP form of Flow Problem

$$\text{maximize} \sum_{e\in\delta^{\text{out}}(s)} x_e \quad \text{with}$$

$$\sum_{e\in\delta^{\text{in}}(v)} x_e - \sum_{e\in\delta^{\text{out}}(v)} x_e = 0 \quad \text{for } v \in V\backslash\{s,t\},$$

$$0 \le x_e \le c_e \quad \text{for } e \in E.$$

Definitions: A *network* is a directed graph G with a nonnegative function $c : E(G) \to \mathbb{R}$; $c_e = c(e)$ is called the *capacity* of e. A *flow* in a network is a function $f : E(G) \to \mathbb{R}$ such that $0 \le f(e) \le c_e$, and for every vertex, other than s and t, inflow equals outflow, i.e.

$$\sum_{e\in\delta^{\text{in}}(v)} x_e = \sum_{e\in\delta^{\text{out}}(v)} x_e.$$

The *value* of a flow f is $\sum_{e\in\delta^{\text{out}}(s)} x_e$; it is what we want to maximize.

Augmenting Path Algorithm
The algorithm for disjoint paths above also generalizes to an algorithm for flows. The way to see it is that when we reverse the edges of the paths we have so far, we are saying that we cannot use these edges for the next path, but we could remove them again.
For flows, we do something similar with the flow that we have so far. For each edge, we see if we could still add to it (to increase the flow along that edge), and we see if we could remove from it (to be able to increase the flow along other edges). We store this information in an auxiliary graph, with a reverse edge wherever flow could be removed, find an st-path in that graph, and then increase the flow along that path as much as possible.

Algorithm for maximum flow in a network

1. Set all $f(e) = 0$, define G_f by $V(G_f) = V(G)$, $E(G_f) = \emptyset$;
2. Set $E_1 = \{e : f(e) < c_e\}$ and $E_2 = \{e : 0 < f(e)\}$;
   Take $E(G_f) = E_1 \cup E_2^{-1}$;
3. Find an st-path Q in G_f; if none exists, go to 6;
4. Compute $\alpha = \min\left(\min_{e \in Q \cap E_1}(c_e - f(e)), \min_{e \in Q \cap E_2^{-1}}(f(e))\right)$;
5. Augment along Q: $f(e) := \begin{cases} f(e) + \alpha & \text{for } e \in Q \cap E_1, \\ f(e) - \alpha & \text{for } e^{-1} \in Q \cap E_2^{-1}; \end{cases}$
   go back to 2;
6. Return f.

Theorem 3.3. *If all capacities c_e are rational, the algorithm terminates.*

Proof. Take M to be the least common multiple of the denominators of the c_e. Then in the algorithm α is always a positive integer multiple of $1/M$, so after each loop the value of f is increased by at least $1/M$. Since the value is bounded by $\sum_{e \in \delta^{\text{out}}(s))} c(e) = c(\delta^{\text{out}}(s))$ (or the capacity of any other cut, see below), the algorithm can take at most $M \cdot c(\delta^{\text{out}}(s))$ iterations. □

Non-termination: There are graphs with some irrational capacities such that the algorithm as given above does not terminate.

Non-polynomiality: There are also graphs with integer capacities for which the algorithm does not have polynomial running time.

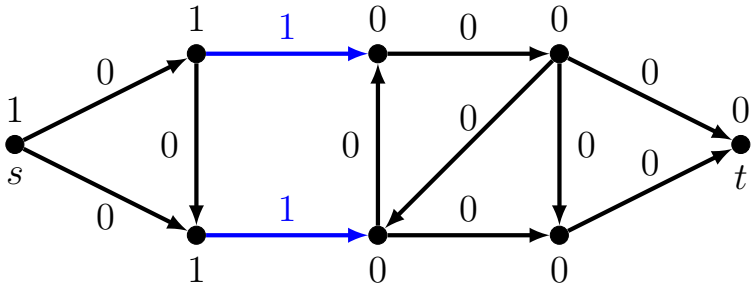
For both of these problems you will see an example in the problem set. Fortunately, both can be fixed by not picking the path Q randomly. But before we get to that, let's first check that when the algorithm terminates, it does actually give a maximum flow. For that we will need the dual. (recall that we assumed $\delta^{\text{in}}(s) = \emptyset$ and $\delta^{\text{out}}(t) = \emptyset$):

Dual of relaxation

$$\begin{aligned} \text{minimize } \sum_{e \in E} c_e z_e, \quad & z \geq 0,\ y \in \mathbb{R}^{|V|-2}, \\ y_v - y_u + z_{uv} \geq 0 \quad & \text{for } uv \in E, u \neq s, v \neq t, \\ y_w + z_{sw} \geq 1 \quad & \text{for } sw \in \delta^{\text{out}}(s), \\ -y_w + z_{wt} \geq 0 \quad & \text{for } wt \in \delta^{\text{in}}(t). \end{aligned}$$

Note that one could simplify the constraints by setting $y_s = 1$, $y_t = 0$, so that for really every $uv \in E$ the constraint has the form $y_v - y_u + z_{uv} \geq 0$. Indeed, for sw we then have $y_w - 1 + z_{sw} \geq 0$, and for wt we have $0 - y_w + z_{wt} \geq 0$, so in both cases we have the same constraint as above.

Cuts: A cut $\delta^{\text{out}}(U)$ with $s \in U, t \notin U$ gives a feasible dual solution: Set $y_u = 1$ for $u \in U$, $y_v = 0$ for $v \notin U$, and set $z_e = 1$ for $e \in \delta^{\text{out}}(U)$, $z_e = 0$ otherwise. In a picture it would look like this (the numbers at the vertices are the y_v, the ones at the edges are the z_e):



Theorem 3.4. *If the augmenting path algorithm terminates, then f is a maximum flow.*

Proof. We will show that there is a cut whose capacity equals the value of f, which proves by duality that f is maximum. Let

$$U = \{u : \exists \text{ an } su\text{-path in } G_f\}.$$

We have $s \in U$, and $t \notin U$, since otherwise the algorithm would not have terminated with f. So $\delta^{\text{out}}(U)$ is an st-cut.
For every $e = uv \in \delta^{\text{out}}(U)$ we must have $f(e) = c_e$, otherwise e would be an edge in G_f, and we would have $v \in U$, a contradiction. Furthermore, for every $e' = v'u' \in \delta^{\text{in}}(U)$ we must have $f(e') = 0$, since otherwise $e'^{-1} = u'v'$ would be an edge in G_f, which would mean $v' \in U$, again a contradiction.
It follows that

$$\sum_{e \in \delta^{\text{out}}(U)} c_e = \sum_{e \in \delta^{\text{out}}(U)} f(e) - \sum_{e \in \delta^{\text{in}}(U)} f(e) = \sum_{e \in \delta^{\text{out}}(s)} f(e).$$

This says exactly that the capacity of the cut equals the value of f. The last equality holds because in general, the value of f across any cut is the same. To prove it, sum up all the constraints of the linear program, then observe that for each edge, the terms will cancel, except for edges that leave U, enter U, or leave s:

$$0 = \sum_{v \in V(G)} \left(\sum_{e \in \delta^{\text{in}}(v)} f(e) - \sum_{e \in \delta^{\text{out}}(v)} f(e) \right) = \sum_{e \in \delta^{\text{out}}(U)} f(e) - \sum_{e \in \delta^{\text{in}}(U)} f(e) - \sum_{e \in \delta^{\text{out}}(s)} f(e).$$

□

Theorem 3.5 (Max-Flow Min-Cut)**.** *The maximum value of a flow equals the minimum capacity of an st-cut:*

$$\max_{\text{flows } f} \sum_{e \in \delta^{out}(s)} f(e) = \min_{\text{cuts } S} \sum_{e \in S} c_e.$$

Proof. If f is a maximum flow, then G_f has no st-path. As above, this gives a cut whose capacity equals the value of f. By duality, this cut must be minimum. □

Theorem 3.6. *If the algorithm always chooses a shortest path in G_f in step 3, then its running time is polynomial.*

Proof. Clearly, in every iteration some edge disappears from G_f, but others may appear when their reverse is on the augmenting path Q_f. We claim that thanks to choosing Q_f to be a shortest path, no edge can (re)appear more than $|V|$ times. This implies that there are at most $|E| \cdot |V|$ iterations, which proves the theorem.
To prove the claim, consider $d_f(v) = \text{dist}_{G_f}(s, v)$ (which we know is easily calculated by BFS). We first prove that d_f does not decrease, i.e. if the algorithm augments f to f', then $d_{f'}(v) \geq d_f(v)$ for all v. This follows by letting uv be the last edge on a shortest path from s to v in $G_{f'}$, so that

$$d_{f'}(v) = d_{f'}(u) + 1 \geq d_f(u) + 1 \geq d_f(v).$$

Here the equality follows because u comes before v on a shortest path, and the first inequality follows by induction. If $uv \in G_f$, then we clearly have $d_f(v) \leq d_f(u)+1$, the third inequality. If $uv \notin G_f$, then it must have just appeared, which can only happen if $vu \in Q_f$, which implies $d_f(u) = d_f(v) + 1$, hence also $d_f(u) + 1 \geq d_f(v)$. This proves that $d_{f'}(v) \geq d_f(v)$ for all v.
To prove the claim, suppose $uv \in G_f$ but uv disappears in the augmentation from f to f'; and that later uv reappears in the augmentation from f'' to f'''. It follows that $uv \in Q_f$ before it disappears, and that $vu \in Q_{f''}$ before it reappears. These two facts imply

$$d_{f''}(u) = d_{f''}(v) + 1 \geq d_f(v) + 1 = d_f(u) + 2.$$

The two equalities follow because Q_f and $Q_{f''}$ are shortest paths, and the inequality follows from the observation that d_f only increases.
So whenever an edge uv reappears, $d_f(u)$ must have increased by 2. Since $d_f(u) \leq |V|$, the edge can certainly not reappear more than $|V|$ times. □