"Who are you?" said Lunkwill, rising angrily from his seat. "What do you want?"

"I am Majikthise!" announced the older one.

"And I demand that I am Vroomfondel!" shouted the younger one.

Majikthise turned on Vroomfondel. "It's alright," he explained angrily, "you don't need to demand that."

"Alright!" bawled Vroomfondel banging on an nearby desk. "I am Vroomfondel, and that is not a demand, that is a solid fact! What we demand is solid facts!"

"No we don't!" exclaimed Majikthise in irritation. "That is precisely what we don't *demand!"*

Scarcely pausing for breath, Vroomfondel shouted, "We don't demand solid facts! What we demand is a total absence of solid facts. I demand that I may or may not be Vroomfondel!"

— Douglas Adams, *The Hitchhiker's Guide to the Galaxy* (1979)

⋆24 Extensions of Maximum Flow

24.1 Maximum Flows with Edge Demands

Now suppose each directed edge e in has both a capacity $c(e)$ and a *demand* $d(e) \le c(e)$, and we want a flow f of maximum value that satisfies $d(e) \le f(e) \le c(e)$ at every edge e. We call a flow that satisfies these constraints a *feasible* flow. In our original setting, where $d(e) = 0$ for every edge e, the zero flow is feasible; however, in this more general setting, even determining whether a feasible flow exists is a nontrivial task.

Perhaps the easiest way to find a feasible flow (or determine that none exists) is to reduce the problem to a standard maximum flow problem, as follows. The input consists of a directed graph $G = (V, E)$, nodes s and t, demand function $d : E \to \mathbb{R}$, and capacity function $c : E \to \mathbb{R}$. Let D denote the sum of all edge demands in G:

$$D := \sum_{u \to v \in E} d(u \to v).$$

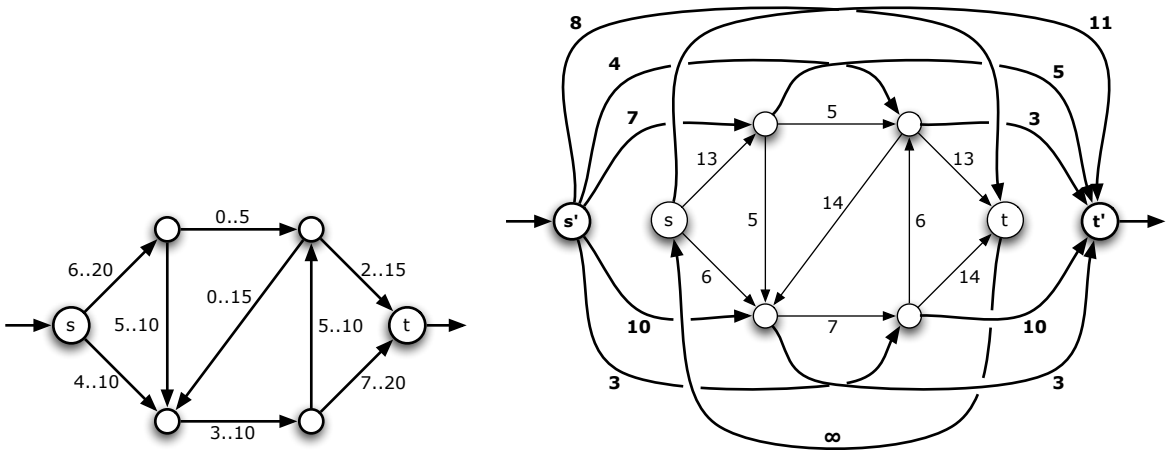
We construct a new graph $G' = (V', E')$ from G by adding new source and target vertices s' and t', adding edges from s' to each vertex in V, adding edges from each vertex in V to t', and finally adding an edge from t to s. We also define a new capacity function $c' : E' \to \mathbb{R}$ as follows:

- For each vertex $v \in V$, we set $c'(s' \to v) = \sum_{u \in V} d(u \to v)$ and $c'(v \to t') = \sum_{w \in V} d(v \to w)$.
- For each edge $u \to v \in E$, we set $c'(u \to v) = c(u \to v) - d(u \to v)$.
- Finally, we set $c'(t \to s) = \infty$.

Intuitively, we construct G' by replacing any edge $u \to v$ in G with three edges: an edge $u \to v$ with capacity $c(u \to v) - d(u \to v)$, an edge $s' \to v$ with capacity $d(u \to v)$, and an edge $u \to t'$ with capacity $d(u \to v)$. If this construction produces multiple edges from s' to the same vertex v (or to t' from the same vertex v), we merge them into a single edge with the same total capacity.

In G', the total capacity out of s' and the total capacity into t' are both equal to D. We call a flow with value exactly D a *saturating* flow, since it saturates all the edges leaving s' or entering t'. If G' has a saturating flow, it must be a maximum flow, so we can find it using any max-flow algorithm.

Lemma 1. *G has a feasible (s, t)-flow if and only if G' has a saturating (s', t')-flow.*



A flow network G with demands and capacities (written $d \,..\, c$), and the transformed network G'.

Proof: Let $f : E \to \mathbb{R}$ be a feasible (s,t)-flow in the original graph G. Consider the following function $f' \colon E' \to \mathbb{R}$:

$$\begin{aligned} f'(u{\to}v) &= f(u{\to}v) - d(u{\to}v) && \text{for all } u{\to}v \in E \\ f'(s'{\to}v) &= \sum_{u \in V} d(u{\to}v) && \text{for all } v \in V \\ f'(v{\to}t') &= \sum_{w \in V} d(u \to w) && \text{for all } v \in V \\ f'(t{\to}s) &= |f| \end{aligned}$$

We easily verify that f' is a saturating (s',t')-flow in G. The admissibility of f implies that $f(e) \geq d(e)$ for every edge $e \in E$, so $f'(e) \geq 0$ everywhere. Admissibility also implies $f(e) \leq c(e)$ for every edge $e \in E$, so $f'(e) \leq c'(e)$ everywhere. Tedious algebra implies that

$$\sum_{u \in V'} f'(u{\to}v) = \sum_{w \in V'} f(v{\to}w)$$

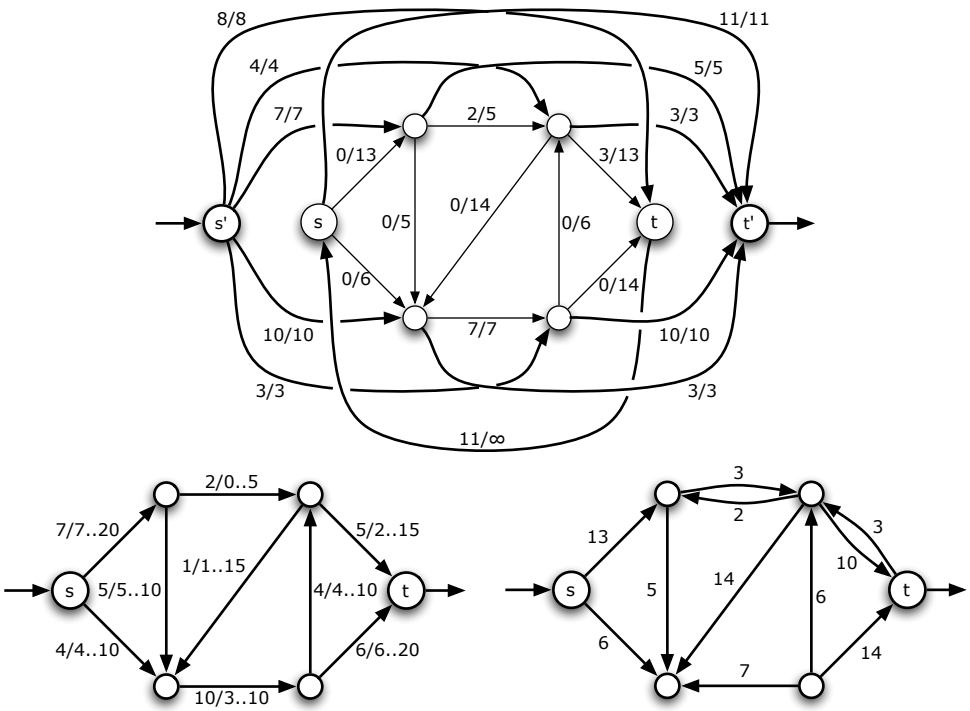
for every vertex $v \in V$ (including s and t). Thus, f' is a legal (s',t')-flow, and every edge out of s' or into t' is clearly saturated. Intuitively, f' diverts $d(u{\to}v)$ units of flow from u directly to the new target t', and injects the same amount of flow into v directly from the new source s'.

The same tedious algebra implies that for any saturating (s',t')-flow $f' \colon E' \to \mathbb{R}$ for G', the function $f = f'|_E + d$ is a feasible (s,t)-flow in G. □

Thus, we can compute a feasible (s,t)-flow for G, if one exists, by searching for a maximum (s',t')-flow in G' and checking that it is saturating. Once we've found a feasible (s,t)-flow in G, we can transform it into a maximum flow using an augmenting-path algorithm, but with one small change. To ensure that *every* flow we consider is feasible, we must redefine the residual capacity of an edge as follows:

$$c_f(u{\to}v) = \begin{cases} c(u{\to}v) - f(u{\to}v) & \text{if } u{\to}v \in E, \\ f(v{\to}u) - d(v{\to}u) & \text{if } v{\to}u \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Otherwise, the algorithm is unchanged. If we use the Dinitz/Edmonds-Karp fat-pipe algorithm, we get an overall running time of $\boldsymbol{O(VE^2)}$.



A saturating flow f' in G', the corresponding feasible flow f in G, and the corresponding residual network G_f.

24.2 Node Supplies and Demands

Another useful variant to consider allows flow to be injected or extracted from the flow network at vertices other than s or t. Let $x : (V \setminus \{s,t\}) \to \mathbb{R}$ be an *excess* function describing how much flow is to be injected (or extracted if the value is negative) at each vertex. We now want a maximum 'flow' that satisfies the variant balance condition

$$\sum_{u \in V} f(u \to v) - \sum_{w \in V} f(v \to w) = x(v)$$

for every node v except s and t, or prove that no such flow exists. As above, call such a function f a *feasible flow*.

As for flows with edge demands, the only real difficulty in finding a maximum flow under these modified constraints is finding a feasible flow (if one exists). We can reduce this problem to a standard max-flow problem, just as we did for edge demands.

To simplify the transformation, let us assume without loss of generality that the total excess in the network is zero: $\sum_v x(v) = 0$. If the total excess is positive, we add an infinite capacity edge $t \to \tilde{t}$, where $\tilde{t}$ is a new target node, and set $x(t) = -\sum_v x(v)$. Similarly, if the total excess is negative, we add an infinite capacity edge $\tilde{s} \to s$, where $\tilde{s}$ is a new source node, and set $x(s) = -\sum_v x(v)$. In both cases, every feasible flow in the modified graph corresponds to a feasible flow in the original graph.

As before, we modify G to obtain a new graph G' by adding a new source s', a new target t', an infinite-capacity edge $t \to s$ from the old target to the old source, and several edges from s' and to t'. Specifically, for each vertex v, if $x(v) > 0$, we add a new edge $s' \to v$ with capacity $x(v)$, and if $x(v) < 0$, we add an edge $v \to t'$ with capacity $-x(v)$. As before, we call an (s', t')-flow in G' *saturating* if every edge leaving s' or entering t' is saturated; any saturating flow is a maximum flow. It is easy to check that saturating flows in G' are in direct correspondence with feasible flows in G; we leave details as an exercise (hint, hint).

Similar reductions allow us to several other variants of the maximum flow problem using the same path-augmentation techniques. For example, we could associate capacities and demands with the vertices instead of (or in addition to) the edges, as well as a *range* of excesses with every vertex, instead of a single excess value.

24.3 Minimum-Cost Flows

Now imagine that each edge e in the network has both a capacity $c(e)$ and a *cost* $\$(e)$. The cost function describes the cost of sending a unit of flow through the edges; thus, the cost any flow f is defined as follows:

$$\$(f) = \sum_{e \in E} \$(e) \cdot f(e).$$

The ***minimum-cost maximum-flow*** problem is to compute a maximum flow of minimum cost. If the network has only one maximum flow, that's what we want, but if there is more than one maximum flow, we want the maximum flow whose cost is as small as possible. Costs can either be positive, negative, or zero. However, if an edge $u \to v$ and its reversal $v \to u$ both appear in the graph, their costs must sum to zero: $\$(u \to v) = -\$(v \to u)$. Otherwise, we could make an infinite profit by pushing flow back and forth along the edge!

Each augmentation step in the standard Ford-Fulkerson algorithm both increases the value of the flow and changes its cost. If the total cost of the augmenting path is positive, the cost of the flow decreases; conversely, if the total cost of the augmenting path is negative, the cost of the flow decreases. We can also change the cost of the flow without changing its value, by augmenting along a directed *cycle* in the residual graph. Again, augmenting along a negative-cost cycle decreases the cost of the flow, and augmenting along a positive-cost cycle increases the cost of the flow.

It follows immediately that a flow f is a minimum-cost maximum flow in G if and only if the residual graph G_f has no directed paths from s to t *and no negative-cost cycles*.

We can compute a min-cost max-flow using the so-called *cycle cancelling* algorithm first proposed by Morton Klein in 1967. The algorithm has two phases; in the first, we compute an arbitrary maximum flow f, using any method we like. The second phase repeatedly decreases the cost of f, by augmenting f along a negative-cost cycle in the residual graph G_f, until no such cycle exists. As in Ford-Fulkerson, the amount of flow we push around each cycle is equal to the minimum residual capacity of any edge on the cycle.

In each iteration of the second phase, we can use a modification of Shimbel's shortest path algorithm (often called "Bellman-Ford") to find a negative-cost cycle in $O(VE)$ time. To bound the number of iterations in the second phase, we assume that both the capacity and the cost of each edge is an integer, and we define

$$C = \max_{e \in E} c(e) \qquad \text{and} \qquad D = \max_{e \in E} |\$(e)|.$$

The cost of any feasible flow is clearly between $-ECD$ and ECD, and each augmentation step decreases the cost of the flow by a positive integer, and therefore by at least 1. We conclude that the second phase requires at most $2ECD$ iterations, and therefore runs in $\boldsymbol{O(VE^2CD)}$ time. As with the raw Ford-Fulkerson algorithm, this running time is exponential in the complexity of the input, and it may never terminate if the capacities and/or costs are irrational.

Like Ford-Fulkerson, more careful choices of *which* cycle to cancel can lead to more efficient algorithms. Unfortunately, some obvious choices are NP-hard to compute, including the cycle with most negative cost and the negative cycle with the fewest edges. In the late 1980s, Andrew Goldberg and Bob Tarjan developed a min-cost flow algorithm that repeatedly cancels the so-called *minimum-mean cycle*, which is the cycle whose *average cost per edge* is smallest. By combining an algorithm of Karp to compute

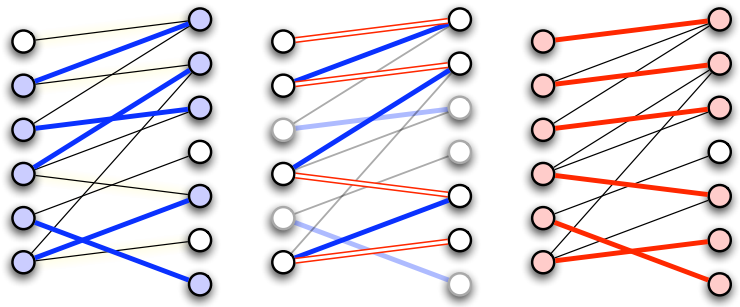
minimum-mean cycles in $O(EV)$ time, efficient dynamic tree data structures, and other sophisticated techniques that are (unfortunately) beyond the scope of this class, their algorithm achieves a running time of $\boldsymbol{O(E^2V\log^2 V)}$. The fastest min-cost max-flow algorithm currently known,[1] due to James Orlin, reduces the problem to $O(E\log V)$ iterations of Dijkstra's shortest-path algorithm; Orlin's algorithm runs in $\boldsymbol{O(E^2\log V + EV\log^2 V)}$ ***time***.

24.4 Maximum-Weight Matchings

Recall from the previous lecture that we can find a maximum-cardinality matching in any bipartite graph in $O(VE)$ time by reduction to the standard maximum flow problem.

Now suppose the input graph has *weighted* edges, and we want to find the matching with maximum total *weight*. Given a bipartite graph $G = (U \times W, E)$ and a non-negative weight function $w : E \to \mathbb{R}$, the goal is to compute a matching M whose total weight $w(M) = \sum_{uw\in M} w(uw)$ is as large as possible. Max-weight matchings can't be found directly using standard max-flow algorithms[2], but we can modify the algorithm for maximum-cardinality matchings described above.

It will be helpful to reinterpret the behavior of our earlier algorithm directly in terms of the original bipartite graph instead of the derived flow network. Our algorithm maintains a matching M, which is initially empty. We say that a vertex is *matched* if it is an endpoint of an edge in M. At each iteration, we find an *alternating path* π that starts and ends at unmatched vertices and alternates between edges in $E \setminus M$ and edges in M. Equivalently, let G_M be the directed graph obtained by orienting every edge in M from W to U, and every edge in $E \setminus M$ from U to W. An alternating path is just a directed path in G_M between two unmatched vertices. Any alternating path has odd length and has exactly one more edge in $E \setminus M$ than in M. The iteration ends by setting $M \leftarrow M \oplus \pi$, thereby increasing the number of edges in M by one. The max-flow/min-cut theorem implies that when there are no more alternating paths, M is a maximum matching.



A matching M with 5 edges, an alternating path π, and the augmented matching $M \oplus \pi$ with 6 edges.

If the edges of G are weighted, we only need to make two changes to the algorithm. First, instead of looking for an arbitrary alternating path at each iteration, we look for the alternating path π such that $M \oplus \pi$ has largest weight. Suppose we weight the edges in the residual graph G_M as follows:

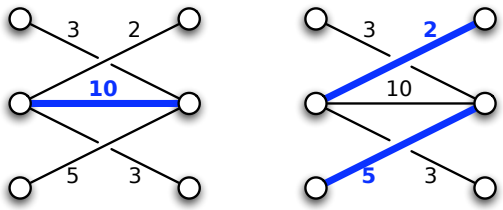
$$w'(u\to w) = -w(uw) \quad \text{for all } uw \notin M$$
$$w'(w\to u) = w(uw) \quad \text{for all } uw \in M$$

We now have $w(M \oplus \pi) = w(M) - w'(\pi)$. ***Thus, the correct augmenting path π must be the directed path in G_M with minimum total residual weight $w'(\pi)$.*** Second, because the matching with the

[1] at least, among algorithms whose running times do not depend on C and D

[2] However, max-flow algorithms can be modified to compute maximum *weighted* flows, where every edge has both a capacity and a weight, and the goal is to maximize $\sum_{u\to v} w(u\to v)f(u\to v)$.

maximum *weight* may not be the matching with the maximum *cardinality*, we return the heaviest matching considered in *any* iteration of the algorithm.



A maximum-weight matching is not necessarily a maximum-cardinality matching.

Before we determine the running time of the algorithm, we need to check that it actually finds the maximum-weight matching. After all, it's a greedy algorithm, and greedy algorithms don't work unless you prove them into submission! Let M_i denote the maximum-weight matching in G with exactly i edges. In particular, $M_0 = \emptyset$, and the global maximum-weight matching is equal to M_i for some i. (The figure above show M_1 and M_2 for the same graph.) Let G_i denote the directed residual graph for M_i, let w_i denote the residual weight function for M_i as defined above, and let π_i denote the directed path in G_i such that $w_i(\pi_i)$ is minimized. To simplify the proof, I will assume that there is a unique maximum-weight matching M_i of any particular size; this assumption can be enforced by applying a consistent tie-breaking rule. With this assumption in place, the correctness of our algorithm follows inductively from the following lemma.

Lemma 2. *If G contains a matching with $i+1$ edges, then $M_{i+1} = M_i \oplus \pi_i$.*

Proof: I will prove the equivalent statement $M_{i+1} \oplus M_i = \pi_{i-1}$. To simplify notation, call an edge in $M_{i+1} \oplus M_i$ *red* if it is an edge in M_{i+1}, and *blue* if it is an edge in M_i.

The graph $M_{i+1} \oplus M_i$ has maximum degree 2, and therefore consists of pairwise disjoint paths and cycles, each of which alternates between red and blue edges. Since G is bipartite, every cycle must have even length. The number of edges in $M_{i+1} \oplus M_i$ is odd; specifically, $M_{i+1} \oplus M_i$ has $2i + 1 - 2k$ edges, where k is the number of edges that are in both matchings. Thus, $M_{i+1} \oplus M_i$ contains an odd number of paths of odd length, some number of paths of even length, and some number of cycles of even length.

Let γ be a cycle in $M_{i+1} \oplus M_i$. Because γ has an equal number of edges from each matching, $M_i \oplus \gamma$ is another matching with i edges. The total weight of this matching is exactly $w(M_i) - w_i(\gamma)$, which must be less than $w(M_i)$, so $w_i(\gamma)$ must be positive. On the other hand, $M_{i+1} \oplus \gamma$ is a matching with $i+1$ edges whose total weight is $w(M_{i+1}) + w_i(\gamma) < w(M_{i+1})$, so $w_i(\gamma)$ must be negative! We conclude that no such cycle γ exists; $M_{i+1} \oplus M_i$ consists entirely of disjoint *paths*.

Exactly the same reasoning implies that no path in $M_{i+1} \oplus M_i$ has an even number of edges.

Finally, since the number of red edges in $M_{i+1} \oplus M_i$ is one more than the number of blue edges, the number of paths that start with a red edge is exactly one more than the number of paths that start with a blue edge. The same reasoning as above implies that $M_{i+1} \oplus M_i$ does not contain a blue-first path, because we can pair it up with a red-first path.

We conclude that $M_{i+1} \oplus M_i$ consists of a single alternating path π whose first edge is red. Since $w(M_{i+1}) = w(M_i) - w_i(\pi)$, the path π must be the one with minimum weight $w_i(\pi)$. □

We can find the alternating path π_i using a single-source shortest path algorithm. Modify the residual graph G_i by adding zero-weight edges from a new source vertex s to every *unmatched* node in U, and from every *unmatched* node in W to a new target vertex t, exactly as in out unweighted matching algorithm. Then π_i is the shortest path from s to t in this modified graph. Since M_i is the maximum-weight matching with i vertices, G_i has no negative cycles, so this shortest path is well-defined.

We can compute the shortest path in G_i in $O(VE)$ time using Shimbel's algorithm, so the overall running time our algorithm is $\boldsymbol{O(V^2E)}$.

The residual graph G_i has negative-weight edges, so we can't speed up the algorithm by replacing Shimbel's algorithm with Dijkstra's. However, we can use a variant of Johnson's all-pairs shortest path algorithm to improve the running time to $\boldsymbol{O(VE + V^2 \log V)}$. Let $d_i(v)$ denote the distance from s to v in the residual graph G_i, using the distance function w_i. Let $\tilde{w}_i$ denote the modified distance function $\tilde{w}_i(u{\to}v) = d_{i-1}(u) + w_i(u{\to}v) - d_{i-1}(v)$. As we argued in the discussion of Johnson's algorithm, shortest paths with respect to w_i are still shortest paths with respect to $\tilde{w}_i$. Moreover, $\tilde{w}_i(u{\to}v) > 0$ for every edge $u{\to}v$ in G_i:

- If $u{\to}v$ is an edge in G_{i-1}, then $w_i(u{\to}v) = w_{i-1}(u{\to}v)$ and $d_{i-1}(v) \le d_{i-1}(u) + w_{i-1}(u{\to}v)$.
- If $u{\to}v$ is not in G_{i-1}, then $w_i(u{\to}v) = -w_{i-1}(v{\to}u)$ and $v{\to}u$ is an edge in the shortest path π_{i-1}, so $d_{i-1}(u) = d_{i-1}(v) + w_{i-1}(v{\to}u)$.

Let $\tilde{d}_i(v)$ denote the shortest path distance from s to v with respect to the distance function $\tilde{w}_i$. Because $\tilde{w}_i$ is positive everywhere, we can quickly compute $\tilde{d}_i(v)$ for all v using Dijkstra's algorithm. This gives us both the shortest alternating path π_i and the distances $d_i(v) = \tilde{d}_i(v) + d_{i-1}(v)$ needed for the next iteration.

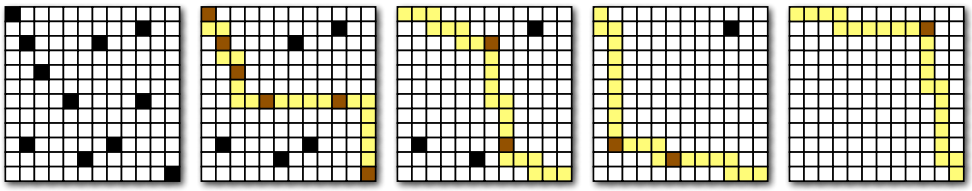
Exercises

1. Suppose we are given a directed graph $G = (V, E)$, two vertices s an t, and a capacity function $c : V \to \mathbb{R}^+$. A flow f is *feasible* if the total flow into every vertex v is at most $c(v)$:

$$\sum_u f(u{\to}v) \le c(v) \quad \text{for every vertex } v.$$

   Describe and analyze an efficient algorithm to compute a feasible flow of maximum value.

2. Suppose we are given an $n \times n$ grid, some of whose cells are marked; the grid is represented by an array $M[1..n, 1..n]$ of booleans, where $M[i, j] = \textsc{True}$ if and only if cell (i, j) is marked. A *monotone* path through the grid starts at the top-left cell, moves only right or down at each step, and ends at the bottom-right cell. Our goal is to cover the marked cells with as few monotone paths as possible.



Greedily covering the marked cells in a grid with four monotone paths.

   (a) Describe an algorithm to find a monotone path that covers the largest number of marked cells.

   (b) There is a natural greedy heuristic to find a small cover by monotone paths: If there are any marked cells, find a monotone path π that covers the largest number of marked cells, unmark any cells covered by π those marked cells, and recurse. Show that this algorithm does *not* always compute an optimal solution.

(c) Describe and analyze an efficient algorithm to compute the smallest set of monotone paths that covers every marked cell.

3. Suppose we are given a set of boxes, each specified by their height, width, and depth in centimeters. All three side lengths of every box lie strictly between 10cm and 20cm. As you should expect, one box can be placed inside another if the smaller box can be rotated so that its height, width, and depth are respectively smaller than the height, width, and depth of the larger box. Boxes can be nested recursively. Call a box is *visible* if it is not inside another box.

   Describe and analyze an algorithm to nest the boxes so that the number of visible boxes is as small as possible.

4. Let G be a directed flow network whose edges have costs, but which contains no negative-cost cycles. Prove that one can compute a minimum-cost maximum flow in G using a variant of Ford-Fulkerson that repeatedly augments the (s,t)-path of *minimum total cost* in the current residual graph. What is the running time of this algorithm?

5. An ***(s, t)-series-parallel*** graph is an directed acyclic graph with two designated vertices s (the *source*) and t (the *target* or *sink*) and with one of the following structures:

   - **Base case:** A single directed edge from s to t.
   - **Series:** The union of an (s,u)-series-parallel graph and a (u,t)-series-parallel graph that share a common vertex u but no other vertices or edges.
   - **Parallel:** The union of two smaller (s,t)-series-parallel graphs with the same source s and target t, but with no other vertices or edges in common.

   (a) Describe an efficient algorithm to compute a maximum flow from s to t in an (s,t)-series-parallel graph with arbitrary edge capacities.

   (b) Describe an efficient algorithm to compute a *minimum-cost* maximum flow from s to t in an (s,t)-series-parallel graph whose edges have *unit* capacity and arbitrary costs.

   *(c) Describe an efficient algorithm to compute a *minimum-cost* maximum flow from s to t in an (s,t)-series-parallel graph whose edges have *arbitrary* capacities and costs.