Man muss immer generalisieren. [One must always generalize.]

— attributed to Carl Gustav Jacob Jacobi (c. 1830) by
Philip J. Davis and Reuben Hersh, *The Mathematical Experience* (1981)

Life is like riding a bicycle.
To keep your balance you must keep moving.

— Albert Einstein, in a letter to his son Eduard (February 5, 1930)

A process cannot be understood by stopping it.
Understanding must move with the flow of the process, must join it and flow with it.

— The First Law of Mentat, from Frank Herbert's *Dune* (1965)

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)

★25 Extensions of Maximum Flow

In this note, we consider several natural generalizations of the maximum flow problem. Some of these generalizations can be solved by reduction back to the standard problem; for others, we will need to generalize the augmenting-path algorithms we used to solve the standard problem.

25.1 Lower Bounds

In a standard maximum-flow problem, each edge e has a capacity $c(e)$, and we seek a flow f that satisfies the inequalities $0 \le f(e) \le c(e)$. One natural generalization is to allow each edge to specify a lower bound on its flow value, in addition to an upper bound or capacity. In this modified problem, the input consists of a directed graph $G = (V, E)$, nodes s and t, and two functions $\ell, u \colon E \to \mathbb{R}$, and we seek a flow f with maximum value such that $\ell(e) \le f(e) \le u(e)$ at every edge e. We call a flow that satisfies these constraints a ***feasible*** flow. In our original setting, where $\ell(e) = 0$ 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.

First, we assume without loss of generality that the bounds on every edge are non-negative. Negative flow values may strain our physical intuition, but they are still completely sensible mathematically; we simply interpret any negative flow from v to w as a positive flow from w to v, and vice versa, just as we already do in the residual graph. If any of the edge bounds are negative, we modify the graph as follows:

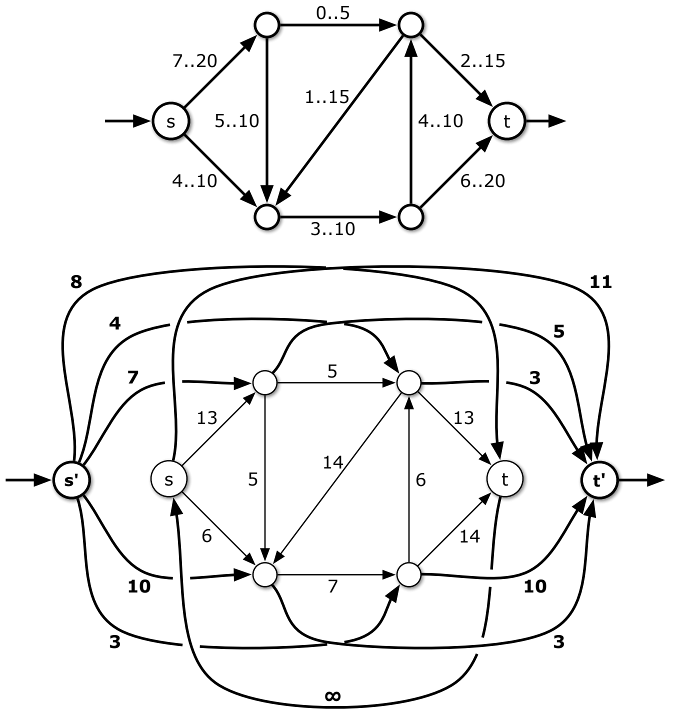
- For every edge $v{\to}w$ such that $u(v{\to}w) < 0$, we replace $v{\to}w$ with its reversal $w{\to}v$ and define $\ell(w{\to}v) = -u(v{\to}w)$ and $u(w{\to}v) = -\ell(v{\to}w)$.

- For every edge $v{\to}w$ such that $\ell(u{\to}v) < 0 \le u(v{\to}w)$, we add the reversed edge $w{\to}v$ and define $\ell(w{\to}v) = 0$ and $u(w{\to}v) = -\ell(v{\to}w)$, and finally change $\ell(v{\to}w)$ to zero.

Next, 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 define the capacity $c'(e)$ of each edge $e \in E'$ as follows:

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

Intuitively, our construction replaces each edge $u \to v$ in G with three edges: an edge $u \to v$ with capacity $u(u \to v) - \ell(u \to v)$, an edge $s' \to v$ with capacity $\ell(u \to v)$, and an edge $u \to t'$ with capacity $\ell(u \to v)$. 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. Because all lower bounds in G are non-negative, all capacities in G' are also non-negative.



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

In G', the total capacity out of s' and the total capacity into t' are both equal to the sum of all lower bounds in G:

$$L := \sum_{v \in V} c'(s' \to v) = \sum_{v \in V} c'(v \to t') = \sum_{u \to v \in E} \ell(u \to v).$$

We call a flow in G' ***saturating*** if it saturates every edge leaving s' and every edge entering t', or equivalently, if it has value L. Every saturating flow is a maximum flow, so we can find it using any maximum-flow algorithm.

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

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

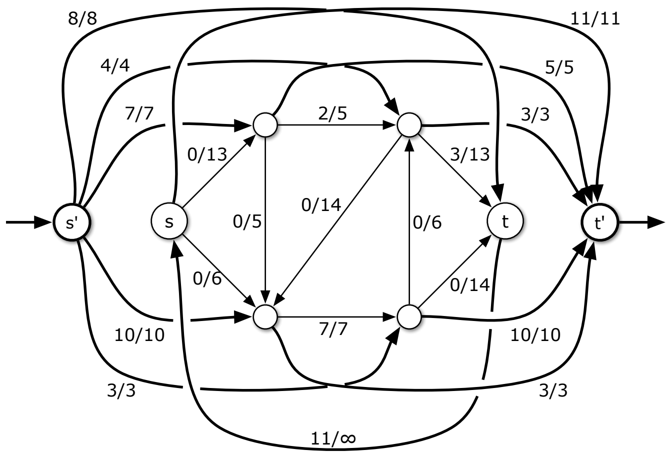
$$\begin{aligned}
f'(u{\to}v) &= f(u{\to}v) - \ell(u{\to}v) && \text{for all } u{\to}v \in E\\
f'(s'{\to}v) &= \sum_{u\in V} \ell(u{\to}v) && \text{for all } v \in V\\
f'(v{\to}t') &= \sum_{w\in V} \ell(u \to w) && \text{for all } v \in V\\
f'(t{\to}s) &= |f|
\end{aligned}$$

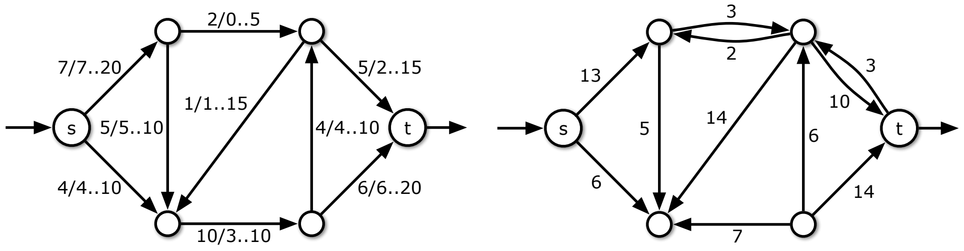
We easily verify that f' is a saturating (s', t')-flow in G. The definition of feasibility implies that for every edge e, we have $\ell(e) \le f(e) \le u(e)$ and therefore $0 \le f'(e) \le c'(e)$. Tedious algebra implies that

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

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 $\ell(v{\to}w)$ units of flow from v directly to the new target t', and injects the same amount of flow into w 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. $\square$





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

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. (If we discover that the maximum flow in G' is not saturating, we correctly conclude that there is no feasible flow in G'.) Once we have a feasible flow in G, we then repeatedly improve it to a maximum flow using the standard 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 Ford-Fulkerson algorithm is unchanged. Orlin's maximum-flow algorithm can also be adapted to this setting, giving us an overall running time of $\boldsymbol{O(VE)}$.

25.2 Node Supplies and Demands

Another natural and useful variant to consider allows a fixed amount of flow to be injected or extracted from the flow network at vertices other than s or t. Let $b\colon (V \setminus \{s,t\}) \to \mathbb{R}$ be an ***balance*** 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 balance condition

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

for every node v except s and t, or prove that no such flow exists. Thus, we interpret positive balances as *demand* and negative balances as (negated) *supply*. As usual, we call such a function f that satisfies these balance constraints (and the usual capacity constraints) 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 maximum-flow problem, just as we did for edge demands.

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

★★★ Need a figure here!

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 $b(v) > 0$, we add a new edge $s'\to v$ with capacity $b(v)$, and if $b(v) < 0$, we add an edge $v\to t'$ with capacity $-b(v)$. As before, we call an (s',t')-flow in G' *saturating* if every edge leaving s' or entering t' is saturated; every 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.

25.3 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:

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

Costs can either be positive, negative, or zero. The ***minimum-cost flow*** problem is to compute a feasible flow with minimum cost, instead of a feasible flow with maximum value.

★★★ Reduce maximum flow to min-cost circulation: add $t{\to}s$ with infinite capacity and cost -1, and give all other edges cost 0.

25.3.1 Cycle Cancelling

Without loss of generality, it suffices to consider the ***minimum-cost circulation*** problem, where the network has no specified source or target, and we seek a minimum-cost flow with value 0. For flow networks with a source s and target t, we can reduce to the minimum-cost circulation problem by adding a single edge $t{\to}s$ with infinite capacity and cost 0. For flow networks with nontrivial balance constraints at the vertices, we first find a feasible flow f using the algorithm in the previous section and then seek a minimum-cost circulation in the residual graph G_f.

Fix a flow network $G = (V, E)$. Without loss of generality, assume that whenever G contains an edge $u{\to}v$, it does not contain the reversed edge $v{\to}u$. If G contains both $u{\to}v$ and $v{\to}u$, we can introduce a new vertex x and then replace $v{\to}u$ with two edges $v{\to}x$ and $x{\to}u$, each with capacity $c(v{\to}u)$ and cost $\$(v{\to}u)/2$.

★★★ Figure!

Fix a circulation f in G. Each edge in G and its reversal has both a ***residual capacity*** and a ***residual cost***:

$$c_f(u{\to}v) := \begin{cases} c(u{\to}v) - f(u{\to}v) & \text{if } u{\to}v \in E \\ f(v{\to}u) & \text{if } v{\to}u \in E \end{cases}$$

$$\$_f(u{\to}v) := \begin{cases} \$(u{\to}v) & \text{if } u{\to}v \in E \\ -\$(v{\to}u) & \text{if } v{\to}u \in E \end{cases}$$

As before, the ***residual network*** $\boldsymbol{G_f}$ consists of all edges with non-zero residual capacity.

★★★ Figure showing flow f and residual graph G_f

Now let γ be a directed cycle in the residual graph G_f. Let $F = \min_{e \in \gamma} c_f(e)$ be the minimum residual capacity of edges in γ, and let ¢ $= \sum_{e \in \gamma} \$_f(e)$ be the *sum* of the residual costs of edges in γ. Just as in the Ford-Fulkerson algorithm, we can ***augment*** the circulation f, by pushing F units of flow around γ, to obtain a new circulation f':

$$f'(u{\to}v) = \begin{cases} f(u{\to}v) + R & \text{if } u{\to}v \in \gamma \\ f(u{\to}v) - R & \text{if } v{\to}u \in \gamma \\ f(u{\to}v) & \text{otherwise} \end{cases}$$

Straightforward calculation now implies that

$$\$(f') = \$(f) + ¢ \cdot R.$$

In particular, if $¢ < 0$, the new circulation f' has lower cost than the original circulation f. We immediately conclude that ***f is a minimum-cost circulation if and only if G_f contains no negative cycles.***

★★★ Figure showing negative cycle and updated flow f'

This observation implies that we can compute minimum-cost circulations using a ***cycle cancelling*** algorithm first proposed by Morton Klein in 1967, which is a straightforward variant of Ford and Fulkerson's augmenting path algorithm. We start by setting f to the all-zero circulation. Then we repeatedly augment f along an arbitrary negative-cost cycle in the residual graph G_f, until there are no more negative residual cycles.

In each iteration of the cycle canceling algorithm, we can find a negative-cost cycle in $O(VE)$ time using a straightforward modification of the Shimbel-Bellman-Ford shortest path algorithm. To bound the number of iterations, 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 total cost of any feasible circulation clearly lies between $-ECD$ and ECD, and each augmentation step decreases the cost of the flow by a positive integer, and therefore by at least 1. Since we start with the zero circulation, we conclude that the algorithm requires at most ECD 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 the algorithm may never terminate if capacities and/or costs are irrational.

★★★ For the network produced by our earlier reduction from maximum flows to min-cost flows, cycle cancelling **IS** Ford-Fulkerson, because a cycle in G_f is negative if and only if it contains $t \rightarrow s$.

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 minimum-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 $O(VE)$ 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 $\boldsymbol{O(VE^2 \log^2 V)}$.

25.3.2 Successive Shortest Paths

★★★

- Proposed by Ford and Fulkerson.
- Instead of trivial balance, assume non-negative costs, by saturating all negative-cost edges and considering the residual graph.
- Feasible flow f is optimal iff G_f has no negative cycles.
- Maintain pseudoflow f such that G_f has no negative cycles. If f is not a flow, augment f along the minimum-cost path in G_f from any node s with excess to any node t with deficit, and recurse.
- Lemma: Pushing flow along a shortest excess-to-deficit path cannot introduce negative cycles.
- Integer balances $\Longrightarrow$ at most $B = \sum_v |b(v)|$ iterations $\Longrightarrow$ $O(BVE)$ time via Bellman-Ford.

25.3.3 Node Potentials and Reduced Costs

★★★

- Improvement due to Edmonds and Karp in 1969 (published 1972) and independently Nobuaki Tomizawa in 1970 (published 1971).
- Define node potentials and reduced costs. Shortest paths don't change. Cycle lengths don't change.
- No negative residual cycles $\Longleftrightarrow$ for some potential function, reduced residual costs are all non-negative
- Fix any node s. Let $\mathit{dist}_\pi(s, v)$ denote the shortest-path distance in G_f from s to v, using reduced costs with respect to π as distances. If reduced costs are non-negative with respect to π, then reduced costs are also non-negative with respect to π', where $\pi'(v) = \pi(v) - \mathit{dist}(s, v)$ for all nodes v.
- Moreover, reduced costs with respect to π' are zero for all shortest-path edges in G_f. (Complementary slackness!)
- Modify successive shortest-path algorithm to maintain node potentials. At each iteration, set $\pi(v) \leftarrow \pi(v) - \mathit{dist}(s, v)$. (Alternatively, just let $\pi(v) = -\mathit{dist}(s, v)$?)
- Non-negative reduced costs $\Longrightarrow$ shortest path via Dijkstra in $O(E \log V)$ time $\Longrightarrow$ $O(BE \log V)$ time overall.

The fastest minimum-cost circulation algorithm currently known,[1] due to James Orlin in the early 1990s, reduces the problem to $O(E \log V)$ iterations of Dijkstra's shortest-path algorithm and therefore runs in ***$O(E^2 \log^2 V)$ time***.

25.4 Maximum-Weight Matchings

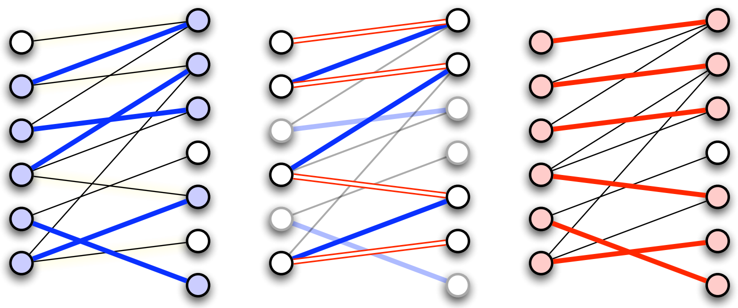
Recall from the previous lecture note that we can find a maximum-cardinality matching in any bipartite graph in $O(VE)$ time by reduction to the standard maximum flow problem.

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 $w\colon E \to \mathbb{R}$, the goal is to compute a matching M whose total weight $w(M) = \sum_{uw \in M} w(uw)$ is as large as possible. Maximum-weight matchings can't be found directly using standard maximum-flow algorithms[2], but we can modify the algorithm for maximum-cardinality matchings described in the previous note.

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

[2]However, maximum-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 $\sum_{u \to v} w(u \to v) f(u \to v)$.

It will be helpful to reinterpret our earlier maximum-matching 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.

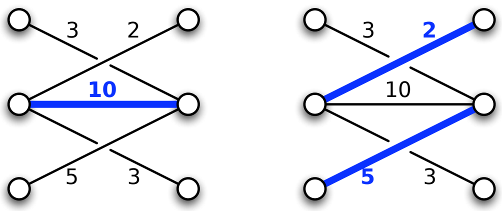
To find the maximum-weight matching in a bipartite graph with weighted edges, we need to make only two changes to this algorithm.

- 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:

$$\begin{aligned} w'(u \to w) &= -w(uw) && \text{for all } uw \notin M \\ w'(w \to u) &= w(uw) && \text{for all } uw \in M \end{aligned}$$

  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 $w'(\pi)$.*** In other words, we are implicitly applying the successive shortest-path algorithm!

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



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, $M_0 = \varnothing$, and the global maximum-weight matching is equal to M_i for some i. (The figure on the previous page shows 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. (See, I told you we were using successive shortest paths!) 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-Bellman-Ford, 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 immediately speed up the algorithm by replacing Shimbel-Bellman-Ford with Dijkstra's algorithm. However, we can use the same repricing trick as for the more general successive shortest-path algorithm, or equivalently, a variant of Johnson's all-pairs shortest path algorithm, to improve the running time to $\boldsymbol{O(VE \log V)}$. Let $d_i(v)$ denote the distance from s to v in the residual graph G_i, using the distance function w_i.

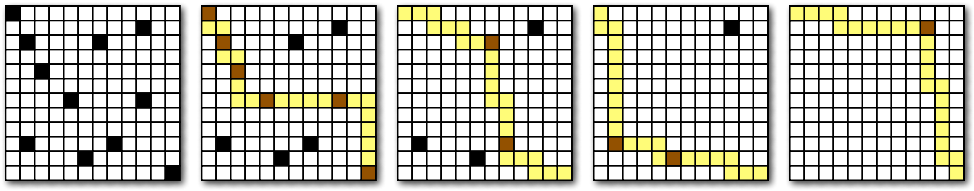
Let $\tilde{w}_i$ denote the modified distance function $\tilde{w}_i(u\to v) := d_{i-1}(u) + w_i(u\to v) - d_{i-1}(v)$. As we argued above (and in the discussion of Johnson's algorithm), shortest paths with respect to w_i are still shortest paths with respect to $\tilde{w}_i$. Moreover, $\tilde{w}_i(u\to v) > 0$ 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 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.

2. Describe and analyze an algorithm for the following problem, first posed and solved by the German mathematician Carl Jacobi in the early 1800s.[3]

   Disponantur nn quantitates $h_k^{(i)}$ quaecunque in schema Quadrati, ita ut k habeantur n series horizontales et n series verticales, quarum quaeque est n terminorum. Ex illis quantitatibus eligantur n transversales, i.e. in seriebus horizontalibus simul atque verticalibus diversis positae, quod fieri potest $1.2\ldots n$ modis; ex omnibus illis modis quaerendum est is, qui summam n numerorum electorum suppeditet maximam.

[3]Carl Gustav Jacob Jacobi. De investigando ordine systematis aequationum differentialum vulgarium cujuscunque. *J. Reine Angew. Math.* 64(4):297–320, 1865. Posthumously published by Carl Borchardt.

For the few students who are not fluent in mid-19th century academic Latin, here is a modern English translation of Jacobi's problem. Suppose we are given an $n \times n$ matrix M. Describe and analyze an algorithm that computes a permutation σ that maximizes the sum $\sum_{i=1}^{n} M_{i,\sigma(i)}$, or equivalently, permutes the columns of M so that the sum of the elements along the diagonal is as large as possible.

3. 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?

4. 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.

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.

6. Every year, Professor Dumbledore assigns the instructors at Hogwarts to various faculty committees. There are n faculty members and c committees. Each committee member has submitted a list of their *prices* for serving on each committee; each price could be positive, negative, zero, or even infinite. For example, Professor Snape might declare that he would serve on the Student Recruiting Committee for 1000 Galleons, that he would *pay* 10000 Galleons to serve on the Defense Against the Dark Arts Course Revision Committee, and that he would not serve on the Muggle Relations committee for any price.

   Conversely, Dumbledore knows how many instructors are needed for each committee, as well as a list of instructors who would be suitable members for each committee. (For example: "Dark Arts Revision: 5 members, anyone but Snape.") If Dumbledore assigns an instructor to a committee, he must pay that instructor's price from the Hogwarts treasury.

Dumbledore needs to assign instructors to committees so that (1) each committee is full, (3) no instructor is assigned to more than three committees, (2) only suitable and willing instructors are assigned to each committee, and (4) the total cost of the assignment is as small as possible. Describe and analyze an efficient algorithm that either solves Dumbledore's problem, or correctly reports that there is no valid assignment whose total cost is finite.

7. Vince wants to borrow a certain amount of money from his friends as cheaply as possible, possibly after first arranging a sequence of intermediate loans. Each of Vince's friends have a different amount of money that they can lend (possibly zero). For any two people x and y, there is a maximum amount of money (possibly zero or infinite) that x is willing to lend to y and a certain profit (possibly zero or even negative) that x expects from any loan to y.

   For example, suppose Vince wants to borrow $100 from his friends Ben and Naomi, who have the following constraints:

   - Ben has $500 available to lend.
   - Ben is willing to lend up to $150 to Vince at a profit of 20¢ per dollar.
   - Ben is willing to lend up to $50 to Naomi, at a loss of 10¢ per dollar.
   - Naomi has $50 available to lend.
   - Naomi is willing to lend any amount of money to Vince, at a profit of 10¢ per dollar.
   - Naomi is not willing to lend money to Ben.

   If Vince borrows $100 directly from Ben, he needs $120 to pay off the loan. If Vince borrows $50 from Ben and $50 from Naomi, he needs $115 to pay off the loan: $60 for Ben and $55 for Naomi. But if Vince asks Naomi to borrow $50 from Ben and then borrows the entire $100 from Naomi, then he needs only $110 to pay off Naomi, who can then pay off Ben with just $45. With the same constraints, the maximum amount of money that Vince can borrow is $250.

   Describe and analyze an algorithm that finds a sequence of loans that minimizes the amount Vince needs to pay everyone off, or correctly reports that Vince cannot borrow his desired amount. The input has the following components:

   - An array $Money[1..n]$, where $Money[i]$ is the amount of money that friend i has.
   - An array $MaxLoan[1..n, 0..n]$, where $MaxLoan[i, j]$ is the amount of money that friend i is willing to lend to friend j. "Friend 0" is Vince.
   - An array $Profit[1..n, 0..n]$, where $Profit[i, j]$ is the profit per dollar that friend i expects from any load to friend j. Again, "friend 0" is Vince.
   - The total amount T that Vince wants to borrow.