For a long time it puzzled me how something so expensive, so leading edge, could be so useless, and then it occurred to me that a computer is a stupid machine with the ability to do incredibly smart things, while computer programmers are smart people with the ability to do incredibly stupid things. They are, in short, a perfect match.

— Bill Bryson, *Notes from a Big Country* (1999)

20 Applications of Maximum Flow

20.1 Edge-Disjoint Paths

One of the easiest applications of maximum flows is computing the maximum number of edge-disjoint paths between two specified vertices s and t in a directed graph G using maximum flows. A set of paths in G is *edge-disjoint* if each edge in G appears in at most one of the paths; several edge-disjoint paths may pass through the same vertex, however.

If we give each edge capacity 1, then the maxflow from s to t assigns a flow of either 0 or 1 to every edge. Since any vertex of G lies on at most two saturated edges (one in and one out, or none at all), the subgraph S of saturated edges is the union of several edge-disjoint paths and cycles. Moreover, the number of paths is exactly equal to the value of the flow. Extracting the actual paths from S is easy—just follow any directed path in S from s to t, remove that path from S, and recurse.

Conversely, we can transform any collection of k edge-disjoint paths into a flow by pushing one unit of flow along each path from s to t; the value of the resulting flow is exactly k. It follows that the maxflow algorithm actually computes the largest possible set of edge-disjoint paths. The overall running time is $O(VE)$, just like for maximum bipartite matchings.

The same algorithm can also be used to find edge-disjoint paths in *undirected* graphs. We simply replace every undirected edge in G with a pair of directed edges, each with unit capacity, and compute a maximum flow from s to t in the resulting directed graph G' using the Ford-Fulkerson algorithm. For any edge uv in G, if our max flow saturates both directed edges $u \to v$ and $v \to u$ in G', we can remove *both* edges from the flow without changing its value. Thus, without loss of generality, the maximum flow assigns a direction to every saturated edge, and we can extract the edge-disjoint paths by searching the graph of directed saturated edges.

20.2 Vertex Capacities and Vertex-Disjoint Paths

Suppose we have capacities on the vertices as well as the edges. Here, in addition to our other constraints, we require that for any vertex v other than s and t, the total flow into v (and therefore the total flow out of v) is at most some non-negative value $c(v)$. How can we compute a maximum flow with these new constraints?

One possibility is to modify our existing algorithms to take these vertex capacities into account. Given a flow f, we can define the *residual capacity* of a vertex v to be its original capacity minus the total flow into v:

$$c_f(v) = c(v) - \sum_u f(u \to v).$$

Since we cannot send any more flow into a vertex with residual capacity 0 we remove from the residual graph G_f every edge $u \to v$ that appears in G whose head vertex v is saturated. Otherwise, the augmenting-path algorithm is unchanged.

But an even simpler method is to transform the input into a traditional flow network, with only edge capacities. Specifically, we replace every vertex v with two vertices v_{in} and v_{out}, connected by

an edge $v_{\text{in}} \to v_{\text{out}}$ with capacity $c(v)$, and then replace every directed edge $u \to v$ with the edge $u_{\text{out}} \to v_{\text{in}}$ (keeping the same capacity). Finally, we compute the maximum flow from s_{out} to t_{in} in this modified flow network.

It is now easy to compute the maximum number of *vertex*-disjoint paths from s to t in any directed graph. Simply give every vertex capacity 1, and compute a maximum flow!

20.3 Maximum Matchings in Bipartite Graphs

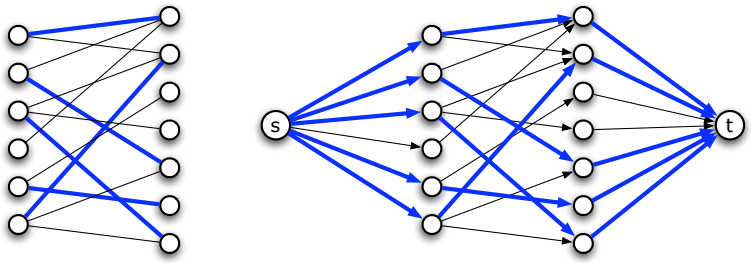
Another natural application of maximum flows is finding large *matchings* in bipartite graphs. A matching is a subgraph in which every vertex has degree at most one, or equivalently, a collection of edges such that no two share a vertex. The problem is to find the matching with the maximum number of edges in a given bipartite graph.

We can solve this problem by reducing it to a maximum flow problem as follows. Let G be the given bipartite graph with vertex set $U \cup W$, such that every edge joins a vertex in U to a vertex in W. We create a new *directed* graph G' by (1) orienting each edge from U to W, (2) adding two new vertices s and t, (3) adding edges from s to every vertex in U, and (4) adding edges from each vertex in W to t. Finally, we assign every edge in G' a capacity of 1.

Any matching M in G can be transformed into a flow f_M in G' as follows: For each edge uw in M, push one unit of flow along the path $s \to u \to w \to t$. These paths are disjoint except at s and t, so the resulting flow satisfies the capacity constraints. Moreover, the value of the resulting flow is equal to the number of edges in M.

Conversely, consider any (s,t)-flow f in G' computed using the Ford-Fulkerson augmenting path algorithm. Because the edge capacities are integers, the Ford-Fulkerson algorithm assigns an integer flow to every edge. (This is easy to verify by induction, hint, hint.) Moreover, since each edge has *unit* capacity, the computed flow either saturates ($f(e) = 1$) or avoids ($f(e) = 0$) every edge in G'. Finally, since at most one unit of flow can enter any vertex in U or leave any vertex in W, the saturated edges from U to W form a matching in G. The size of this matching is exactly $|f|$.

Thus, the size of the maximum matching in G is equal to the value of the maximum flow in G', and provided we compute the maxflow using augmenting paths, we can convert the actual maxflow into a maximum matching. The maximum flow has value at most $\min\{|U|, |W|\} = O(V)$, so the Ford-Fulkerson algorithm runs in $\boldsymbol{O(VE)}$ ***time***.



A maximum matching in a bipartite graph G, and the corresponding maximum flow in G'.

20.4 Binary Assignment Problems

Maximum-cardinality matchings are a special case of a general family of so-called *assignment* problems.[1] An unweighted *binary* assignment problem involves two disjoint finite sets X and Y, which typically represent two different kinds of resources, such as web pages and servers, jobs and machines, rows and

[1] Most authors refer to finding a maximum-weight matching in a bipartite graph as *the* assignment problem.

columns of a matrix, hospitals and interns, or customers and pints of ice cream. Our task is to choose the largest possible collection of pairs (x, y) as possible, where $x \in X$ and $y \in Y$, subject to several constraints of the following form:

- Each element $x \in X$ can appear in at most $c(x)$ pairs.
- Each element $y \in Y$ can appear in at most $c(y)$ pairs.
- Each pair $(x, y) \in X \times Y$ can appear in the output at most $c(x, y)$ times.

Each upper bound $c(x)$, $c(y)$, and $c(x, y)$ is either a (typically small) non-negative integer or ∞. Intuitively, we create each pair in our output by *assigning* an element of X to an element of y.

The maximum-matching problem is a special case, where $c(z) = 1$ for all $z \in X \cup Y$, and each $c(x, y)$ is either 0 or 1, depending on whether the pair xy defines an edge in the underlying bipartite graph.

Here is a slightly more interesting example. A nearby school, famous for its onerous administrative hurdles, decides to organize a dance. Every pair of students (one boy, one girl) who wants to dance must register in advance. School regulations limit each boy-girl pair to at most three dances together, and limits each student to at most ten dances overall. How can we maximize the number of dances? This is a binary assignment problem for the set X of girls and the set Y of boys. For each girl x and boy y, we have $c(x) = 10$, $c(y) = 10$, and either $c(x, y) = 3$ (if x and y registered to dance) or $c(x, y) = 0$ (if they didn't).

This binary assignment problem can be reduces to a standard maximum flow problem as follows. We construct a flow network $G = (V, E)$ with vertices $X \cup Y \cup \{s, t\}$ and the following edges:

- an edge $s \to x$ with capacity $c(x)$ for each $x \in X$,
- an edge $y \to t$ with capacity $c(y)$ for each $y \in Y$.
- an edge $x \to y$ with capacity $c(x, y)$ for each $x \in X$ and $y \in Y$, and

Because all the edges have integer capacities, the Ford-Fulkerson algorithm constructs an integer maximum flow f^*. This flow can be decomposed into the sum of $|f^*|$ paths of the form $s \to x \to y \to t$ for some $x \in X$ and $y \in Y$. For each such path, we report the pair (x, y). (Equivalently, the pair (x, y) appears in our output collection $f(x \to y)$ times.) It is easy to verify (hint, hint) that this collection of pairs satisfies all the necessary constraints. Conversely, any legal collection of r pairs can be transformed into a feasible integer flow with value r in G. Thus, the largest legal collection of pairs corresponds to a maximum flow in G. So our algorithm is correct.

20.5 Baseball Elimination

Every year millions of baseball fans eagerly watch their favorite team, hoping they will win a spot in the playoffs, and ultimately the World Series. Sadly, most teams are "mathematically eliminated" days or even weeks before the regular season ends. Often, it is easy to spot when a team is eliminated—they can't win enough games to catch up to the current leader in their division. But sometimes the situation is more subtle.

For example, here are the actual standings from the American League East on August 30, 1996.

Team	Won–Lost	Left	NYY	BAL	BOS	TOR	DET
New York Yankees	75–59	28		3	8	7	3
Baltimore Orioles	71–63	28	3		2	7	4
Boston Red Sox	69–66	27	8	2		0	0
Toronto Blue Jays	63–72	27	7	7	0		0
Detroit Lions	49–86	27	3	4	0	0	

Detroit is clearly behind, but some die-hard Lions fans may hold out hope that their team can still win. After all, if Detroit wins all 27 of their remaining games, they will end the season with 76 wins, more than any other team has now. So as long as every other team loses every game... but that's not possible, because some of those other teams still have to play each other. Here is one complete argument:[2]

> *By winning all of their remaining games, Detroit can finish the season with a record of 76 and 86. If the Yankees win just 2 more games, then they will finish the season with a 77 and 85 record which would put them ahead of Detroit. So, let's suppose the Tigers go undefeated for the rest of the season and the Yankees fail to win another game.*
>
> *The problem with this scenario is that New York still has 8 games left with Boston. If the Red Sox win all of these games, they will end the season with at least 77 wins putting them ahead of the Tigers. Thus, the only way for Detroit to even have a chance of finishing in first place, is for New York to win exactly one of the 8 games with Boston and lose all their other games. Meanwhile, the Sox must loss all the games they play agains teams other than New York. This puts them in a 3-way tie for first place....*
>
> *Now let's look at what happens to the Orioles and Blue Jays in our scenario. Baltimore has 2 games left with with Boston and 3 with New York. So, if everything happens as described above, the Orioles will finish with at least 76 wins. So, Detroit can catch Baltimore only if the Orioles lose all their games to teams other than New York and Boston. In particular, this means that Baltimore must lose all 7 of its remaining games with Toronto. The Blue Jays also have 7 games left with the Yankees and we have already seen that for Detroit to finish in first place, Toronto must will all of these games. But if that happens, the Blue Jays will win at least 14 more games giving them at final record of 77 and 85 or better which means they will finish ahead of the Tigers. So, no matter what happens from this point in the season on, Detroit can not finish in first place in the American League East.*

There has to be a better way to figure this out!

Here is a more abstract formulation of the problem. Our input consists of two arrays $W[1..n]$ and $G[1..n, 1..n]$, where $W[i]$ is the number of games team i has already won, and $G[i, j]$ is the number of upcoming games between teams i and j. We want to determine whether team n can end the season with the most wins (possibly tied with other teams).[3]

We model this question as an assignment problem: We want to ***assign*** *a winner to each game*, so that team n comes in first place. We have an assignment problem! Let $R[i] = \sum_j G[i, j]$ denote the number of remaining games for team i. We will assume that team n wins all $R[n]$ of its remaining games. Then team n can come in first place if and only if every other team i wins at most $W[n] + R[n] - W[i]$ of its $R[i]$ remaining games.

Since we want to ***assign*** winning teams to games, we start by building a bipartite graph, whose nodes represent the games and the teams. We have $\binom{n}{2}$ *game* nodes $g_{i,j}$, one for each pair $1 \le i < j < n$, and $n-1$ *team* nodes t_i, one for each $1 \le i < n$. For each pair i, j, we add edges $g_{i,j} \to t_i$ and $g_{i,j} \to t_j$ with *infinite* capacity. We add a source vertex s and edges $s \to g_{i,j}$ with capacity $G[i, j]$ for each pair i, j. Finally, we add a target node t and edges $t_i \to t$ with capacity $W[n] - W[i] + R[n]$ for each team i.

Theorem: *Team n can end the season in first place if and only if there is a feasible flow in this graph that saturates every edge leaving s.*

Proof: Suppose it is possible for team n to end the season in first place. Then every team $i < n$ wins at most $W[n] + R[n] - W[i]$ of the remaining games. For each game between team i and team j that team i wins, add one unit of flow along the path $s \to g_{i,j} \to t_i \to t$. Because there are exactly $G[i, j]$ games between teams i and j, every edge leaving s is saturated. Because each team i wins at most $W[n] + R[n] - W[i]$ games, the resulting flow is feasible.

Conversely, Let f be a feasible flow that saturates every edge out of s. Suppose team i wins exactly $f(g_{i,j} \to t_i)$ games against team j, for all i and j. Then teams i and j play $f(g_{i,j} \to t_i) + f(g_{i,j} \to t_j) =$

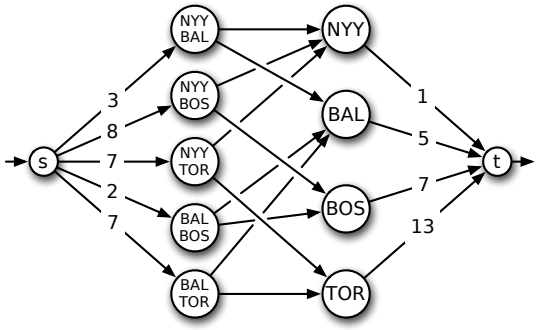
[2] Both the example and this argument are taken from http://riot.ieor.berkeley.edu/~baseball/detroit.html.

[3] We assume here that no games end in a tie (always true for Major League Baseball), and that every game is actually played (not always true).

$f(s \to g_{i,j}) = G[i,j]$ games, so every upcoming game is played. Moreover, each team i wins a total of $\sum_j f(g_{i,j} \to t_i) = f(t_i \to t) \le W[n] + R[n] - W[i]$ upcoming games, and therefore at most $W[n] + R[n]$ games overall. Thus, if team n win all their upcoming games, they end the season in first place. □

So, to decide whether our favorite team can win, we construct the flow network, compute a maximum flow, and report whether than maximum flow saturates the edges leaving s. The flow network has $O(n^2)$ vertices and $O(n^2)$ edges, and it can be constructed in $O(n^2)$ time. Using Dinitz's algorithm, we can compute the maximum flow in $O(VE^2) = O(n^6)$ time.

The graph derived from the 1996 American League East standings is shown below. The total capacity of the edges leaving s is 27 (there are 27 remaining games), but the total capacity of the edges entering t is only 26. So the maximum flow has value at most 26, which means that Detroit is mathematically eliminated.



The flow graph for the 1996 American League East standings. Unlabeled edges have infinite capacity.

Exercises

1. Given an undirected graph $G = (V, E)$, with three vertices u, v, and w, describe and analyze an algorithm to determine whether there is a path from u to w that passes through v.

2. Let $G = (V, E)$ be a directed graph where for each vertex v, the in-degree and out-degree of v are equal. Let u and v be two vertices G, and suppose G contains k edge-disjoint paths from u to v. Under these conditions, must G also contain k edge-disjoint paths from v to u? Give a proof or a counterexample with explanation.

3. A *cycle cover* of a given directed graph $G = (V, E)$ is a set of vertex-disjoint cycles that cover all the vertices. Describe and analyze an efficient algorithm to find a cycle cover for a given graph, or correctly report that no cycle cover exists. *[Hint: Use bipartite matching!]*

4. Consider a directed graph $G = (V, E)$ with multiple source vertices $s_1, s_2, \ldots, s_\sigma$ and multiple target vertices $t_1, t_1, \ldots, t_\tau$, where no vertex is both a source and a target. A *multiterminal flow* is a function $f : E \to \mathbb{R}_{\geq 0}$ that satisfies the flow conservation constraint at every vertex that is neither a source nor a target. The value $|f|$ of a multiterminal flow is the total excess flow out of *all* the source vertices:

$$|f| := \sum_{i=1}^{\sigma} \left(\sum_{w} f(s_i \to w) - \sum_{u} f(u \to s_i) \right)$$

   As usual, we are interested in finding flows with maximum value, subject to capacity constraints on the edges. (In particular, we don't care how much flow moves from any particular source to any particular target.)

   (a) Consider the following algorithm for computing multiterminal flows. The variables f and f' represent flow functions. The subroutine MaxFlow(G, s, t) solves the standard maximum flow problem with source s and target t.

```
MaxMultiFlow(G, s[1..σ], t[1..τ]):
  f ← 0                                  ⟨⟨Initialize the flow⟩⟩
  for i ← 1 to σ
      for j ← 1 to τ
          f' ← MaxFlow(G_f, s[i], t[j])
          f ← f + f'                     ⟨⟨Update the flow⟩⟩
  return f
```

   Prove that this algorithm correctly computes a maximum multiterminal flow in G.

   (b) Describe a more efficient algorithm to compute a maximum multiterminal flow in G.

5. *Ad-hoc networks* are made up of low-powered wireless devices. In principle[4], these networks can be used on battlefields, in regions that have recently suffered from natural disasters, and in other hard-to-reach areas. The idea is that a large collection of cheap, simple devices could be distributed through the area of interest (for example, by dropping them from an airplane); the devices would then automatically configure themselves into a functioning wireless network.

[4] but not really in practice

These devices can communication only within a limited range. We assume all the devices are identical; there is a distance D such that two devices can communicate if and only if the distance between them is at most D.

We would like our ad-hoc network to be reliable, but because the devices are cheap and low-powered, they frequently fail. If a device detects that it is likely to fail, it should transmit its information to some other *backup* device within its communication range. We require each device x to have k potential backup devices, all within distance D of x; we call these k devices the ***backup set*** of x. Also, we do not want any device to be in the backup set of too many other devices; otherwise, a single failure might affect a large fraction of the network.

So suppose we are given the communication radius D, parameters b and k, and an array $d[1..n, 1..n]$ of distances, where $d[i, j]$ is the distance between device i and device j. Describe an algorithm that either computes a backup set of size k for each of the n devices, such that that no device appears in more than b backup sets, or reports (correctly) that no good collection of backup sets exists.

6. Suppose we are given an array $A[1..m][1..n]$ of non-negative real numbers. We want to *round* A to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A. For example:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \longmapsto \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

Describe an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible.

*7. A *rooted tree* is a directed acyclic graph, in which every vertex has exactly one incoming edge, except for the *root*, which has no incoming edges. Equivalently, a rooted tree consists of a root vertex, which has edges pointing to the roots of zero or more smaller rooted trees. Describe a polynomial-time algorithm to compute, given two rooted trees A and B, the largest common rooted subtree of A and B.

[Hint: Let $LCS(u, v)$ denote the largest common subtree whose root in A is u and whose root in B is v. Your algorithm should compute $LCS(u, v)$ for all vertices u and v using dynamic programming. This would be easy if every vertex had $O(1)$ children, and still straightforward if the children of each node were ordered from left to right and the common subtree had to respect that ordering. But for unordered trees with large degree, you need another trick to combine recursive subproblems efficiently. Don't waste your time trying to reduce the polynomial running time.]