

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)

Shortly after the “iron curtain” fell in 1990, an American and a Russian, who had both worked on the development of weapons, met. The American asked: “When you developed the Bomb, how were you able to perform such an enormous amount of computing with your weak computers?” The Russian responded: “We used better algorithms.”

— Yefim Dinitz, in “Dinitz’ Algorithm: The Original Version and Even’s Version” (2006)

It is to be noted, however, that the problem of determining an effective method of actually acquiring the solution to a specific problem is not solved in this paper. In the category of development of such methods we seem to be, currently, ahead of the Russians.

— Abraham Charnes, introducing Leonid Kantorovich’s seminal 1942 paper “On the Translocation of Masses” (1958)

CHAPTER 11

Applications of Flows and Cuts

11.1 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 maximum flow from s to t pushes either 0 or 1 units of flow along each edge. The flow-decomposition theorem implies that the subgraph S of saturated edges is the union of several edge-disjoint paths and cycles. Moreover, the number of paths in this decomposition is exactly equal to the value of the flow. Extracting the actual paths from S is easy—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 any maximum flow algorithm actually computes the largest

© Copyright 2017 Jeff Erickson.

This work is licensed under a Creative Commons License (<http://creativecommons.org/licenses/by-nc-sa/4.0/>).

Free distribution is strongly encouraged; commercial distribution is expressly forbidden.

See <http://jeffe.cs.illinois.edu/teaching/algorithms> for the most recent revision.

possible set of edge-disjoint paths.

If we use Orlin's algorithm to compute the maximum (s, t) -flow, we can compute edge-disjoint paths in $O(VE)$ time, but Orlin's algorithm is overkill for this simple application. The cut $(\{s\}, V \setminus \{s\})$ has capacity at most $V - 1$, so the maximum flow has value at most $V - 1$. Thus, Ford and Fulkerson's original augmenting path algorithm already runs in $O(|f^*|E) = O(VE)$ time.

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 $u \rightarrow v$ and $v \rightarrow u$ 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.

11.2 Vertex Capacities and Vertex-Disjoint Paths

Now 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?

The simplest 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 $v_{\text{in}} \rightarrow v_{\text{out}}$ with capacity $c(v)$, and then replace every directed edge $u \rightarrow v$ with the edge $u_{\text{out}} \rightarrow v_{\text{in}}$ (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 graph: Assign capacity 1 to every vertex and compute a maximum flow!



Figure!

11.3 Bipartite Matching

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 $s \rightarrow u \rightarrow w \rightarrow t$. 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 in $O(E)$ time. Again, we can compute the maximum flow in $O(VE)$ time using either Orlin's algorithm or off-the-shelf Ford-Fulkerson.

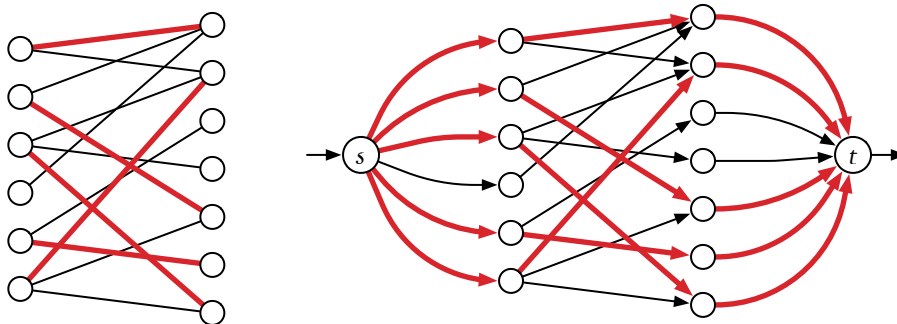


Figure 11.1. A maximum matching in a bipartite graph G , and the corresponding maximum flow in G' .

A more sophisticated algorithm, first published by John Hopcroft and Richard Karp in 1973, computes maximum matchings in bipartite graphs in $O(\sqrt{VE})$ time. I might discuss the Hopcroft-Karp algorithm in more detail in another note.

11.4 Exam Scheduling

Suppose Shampoo-Banana University has hired you to write an algorithm to schedule their final exams. There are n different classes, each of which needs to schedule a final exam in one of r rooms during one of t different time slots. At most one class's final exam can be scheduled in each room during each time slot; conversely, classes cannot be split into multiple rooms or multiple times. Moreover, each exam must be overseen by one of p proctors. Each proctor can oversee at most one exam at a time; each proctor

is available for only certain time slots; and no proctor is allowed oversee more than 5 exams total. The input to the scheduling problem consists of three arrays:

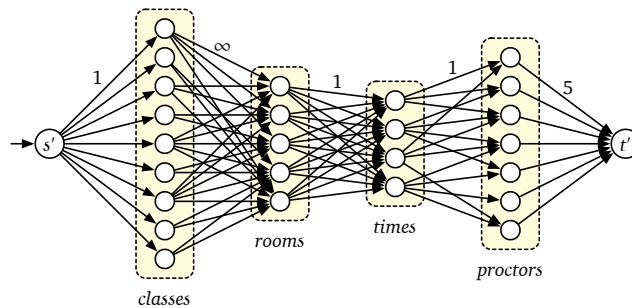
- An integer array $E[1..n]$ where $E[i]$ is the number of students enrolled in the i th class.
- An integer array $S[1..r]$, where $S[j]$ is the number of seats in the j th room. The i th class's final exam *can* be held in the j th room if and only if $E[i] \leq S[j]$.
- A boolean array $A[1..t, 1..p]$ where $A[k, \ell] = \text{TRUE}$ if and only if the ℓ th proctor is available during the k th time slot.

We want to design an algorithm that either schedules a room, a time, and a proctor for every class's final exam, or correctly reports that no such schedule is possible.

We can solve this scheduling problem by reducing it to maximum flow as follows. We construct a flow network G with six types of vertices—a source vertex s' , a vertex c_i for each class, a vertex r_j for each room, a vertex t_k for each time slot, a vertex p_ℓ for each proctor, and a target vertex t' —and five types of edges:

- An edge $s' \rightarrow c_i$ with capacity 1 for each class i . (“Each class can hold at most one final exam.”)
- An uncapacitated edge $c_i \rightarrow r_j$ for each class i and room j such that $E[i] \leq S[j]$. (“Class i can hold exams in room j if and only if the room has enough seats.”)
- An edge $r_j \rightarrow t_k$ with capacity 1 for each room j and time slot k . (“At most one exam can be held in room j at time k .”)
- An edge $t_k \rightarrow p_\ell$ with capacity 1 for time slot k and proctor ℓ such that $A[\ell, k] = \text{TRUE}$. (“A proctor can oversee at most one exam at any time, and only during times that they are available.”)
- An edge $p_\ell \rightarrow t'$ with capacity 5 for each proctor ℓ . (“Each proctor can oversee at most 5 exams.”)

Altogether G has $n + r + t + p + 2$ vertices and $O(nr + rt + tp)$ edges. (I'm calling the source and target vertices s' and t' instead of s and t only because the problem statement already uses the variable t to be the number of distinct time slots.)



A flow network for the exam scheduling problem.

Each path from s' to t' in G represents a valid choice of class, room, time, and proctor for one final exam; specifically, the class fits into the room, and the proctor is available at that time. Conversely, for each valid choice (class, room, time, proctor), there is a corresponding path from s' to t' in G .

More generally, let f be an arbitrary feasible integer (s', t') -flow in G . By the Flow Decomposition Theorem, f can be decomposed into $|f|$ paths from s' to t' , each carrying exactly one unit of flow. Straightforward definition-chasing implies that the resulting set of tuples is a valid schedule for $|f|$ final exams. Conversely, for any valid schedule for k final exams, the sum of the k corresponding paths is a feasible integer (s', t') -flow with value k .

Thus, we can construct a valid schedule for the maximum possible number of exams by computing an maximum (s', t') -flow f^* in G . Because all finite capacities in G are integers, we can assume without loss of generality that f^* is an integer flow, and therefore by the previous paragraph corresponds to a valid schedule for $|f^*|$ final exams. Thus, if $|f^*| = n$, we can return the resulting schedule; otherwise, we can correctly report that scheduling all n final exams is impossible.

We can construct G from the given input data by brute force in $O(nr + rt + tp)$ time. We can compute the maximum flow in $O(VE)$ time using either Ford-Fulkerson (because $|f^*| \leq n < V$) or Orlin's algorithm, and we can compute the flow decomposition in $O(VE)$ time.

Tuple Selection

Both the maximum matching problem and the previous scheduling problem are examples of a class of problems that I call **tuple selection**.¹ The input to a tuple selection problem consists of several finite sets X_1, X_2, \dots, X_d , each representing a different discrete resource. Our task is to select the largest possible set of d -tuples, each containing exactly one element from each set X_i , subject to several constraints of the following forms:

- For each index i , Each element $x \in X_i$ can appear in at most $c(x)$ selected tuples.
- For each index i , any two elements $x \in X_i$ and $y \in X_{i+1}$ can appear in at most $c(x, y)$ selected tuples.

Each of the upper bounds $c(x)$ and $c(x, y)$ is either a (typically small) non-negative integer or ∞ .

In the maximum-matching problem, we have $d = 2$ types of resources, each element x has capacity $c(x) = 1$, and each pair (x, y) has capacity $c(x, y) = 1$ or $c(x, y) = 0$, depending on whether xy is an edge in the underlying bipartite graph. In the scheduling problem, we have $d = 4$ types of resources.

¹I could not find a standard name for this class of problems, so I made up my own. I used to refer to these as "assignment problems", but everyone else in the world understands that phrase to mean weighted bipartite matching.

As long as the resources X_i can be ordered so that all pairwise constraints $c(x, y)$ involve only vertices in *adjacent* subsets, the tuple selection problem can be reduced to a maximum-flow problem in a graph $G = (V, E)$ defined as follows:

- G contains a vertex for each element of each set X_i , as well as a source vertex s and a target vertex t . Each vertex x (except s and t) has capacity $c(x)$.
- G contains an edge $s \rightarrow w$ for each $w \in X_1$, an edge $z \rightarrow t$ for each $z \in X_d$, and an edge $x \rightarrow y$ with capacity $c(x, y)$ for each $x \in X_i$ and $y \in X_{i+1}$, for all i . (Optionally, we can omit edges $x \rightarrow y$ with $c(x, y) = 0$.)

Every path from s to t in this graph corresponds to a d -tuple that we could select. Thus, we can compute a maximum valid set of tuples by computing an integer maximum (s, t) -flow f^* in G , decomposing f^* into $|f^*|$ paths from s to t , and then transcribing each path into a tuple.

In other words, if H is the graph whose vertices are the resources and whose edges represent constraints, we can solve the tuple selection problem in polynomial time via flows *when H is a path*. On the other hand, if the constraint graph H contains either a cycle or a vertex with degree greater than 2, the tuple selection problem is NP-hard.

11.5 Disjoint-Path Covers

A **path cover** of a directed graph G is a collection of directed paths in G such that every vertex of G lies on *at least* one path. A **disjoint-path cover** of G is a path cover such that every vertex of G lies on *exactly* one path. Every directed graph has an disjoint-path cover, consisting of several paths of length zero, one for each vertex, but that's boring. Instead, let's look for disjoint-path covers that contain as few paths as possible. This problem is NP-hard in general—a graph has a disjoint-path cover of size 1 if and only if it contains a Hamiltonian path—but there is an efficient flow-based algorithm for directed *acyclic* graphs.

To solve this problem for a given directed acyclic graph $G = (V, E)$, we construct a new bipartite graph $G' = (V', E')$ as follows.

- H contains two vertices v^b and v^\sharp for every vertex v of G .
- H contains an undirected edge $u^b v^\sharp$ for every directed edge $u \rightarrow v$ in G .

(If G is represented as an adjacency matrix, then G' is just the bipartite graph represented by the same adjacency matrix!)

Now I claim that G has an disjoint path cover consisting of k paths if and only if the new graph G' has a matching of size $V - k$. As usual, we prove the equivalence in two stages:

- ⇐ Suppose G has an disjoint path cover P with k paths; think of P as a subgraph of G . Every vertex in P has in-degree either 0 or 1; moreover, there is exactly one vertex with in-degree 0 in each path in P . It follows that the subgraph P has exactly $V - k$

edges. Now define a subset M of the edges of G' as follows:

$$M := \{u^b v^\# \in E' \mid u \rightarrow v \in P\}.$$

By definition of disjoint path cover, every vertex of G has at most one incoming edge in P and at most one outgoing edge in P . We conclude that every vertex of G' is incident to at most one edge in M ; that is, M is a matching of size $V - k$.

\Rightarrow Suppose G' has a matching M of size $V - k$. We project M' back to G by defining a subgraph $P = (V, M')$, where

$$M' := \{u \rightarrow v \in E \mid u^b v^\# \in M\}.$$

By definition of matching, every vertex of G has at most one incoming edge in P and at most one outgoing edge in P . It follows that P is a collection of disjoint directed paths in G ; since P includes every vertex, P defines an disjoint path cover with $V - k$ edges. The number of paths in P is equal to the number of vertices in G that have no incoming edge in M' . We conclude that P contains exactly k paths.

It follows immediately that we can find a minimum disjoint-path cover in G by computing a maximum matching in G' , using Ford-Fulkerson's maximum-flow algorithm, in $O(V'E') = O(VE)$ time.

Despite its formulation in terms of dags and paths, this is really a matching problem: We want to match as many vertices as possible to distinct *successors* in the graph. The number of paths required to cover the dag is equal to the number of vertices with no successor.

Minimal Teaching Assignment

Now consider the following “real”-world scheduling problem.² Shampoo-Banana University offers several thousand courses every day. Due to extreme budget cuts, the university needs to significantly reduce the number of professors. However, because students pay tuition (and the university cannot afford lawyers), the university must retain enough professors that every class advertised in the course catalog is actually taught. How few professors can the university get away with? Each remaining faculty member will be assigned a sequence of classes to teach on any given day. The classes assigned to each professor must not overlap; moreover, there must be enough slack in each professor's schedule for them to walk from one class to the next. For purposes of this problem, let's assume that every professor is capable of teaching every class, and that professors will not have office hours, lunches, or bathroom breaks.³

Concretely, suppose there are n classes offered in m different locations. The input to our problem consists of the following data:

²For a somewhat more realistic (and less depressing) formulation of this problem, consider airplanes and flights, or buses and bus routes, instead of professors and classes.

³They will, however, be expected to answer student emails as they walk between classes.

- An array $C[1..n]$ of classes, where each class $C[i]$ has three fields: the starting time $C[i].start$, the ending time $C[i].end$, and the location $C[i].loc$.
- A two-dimensional array $T[1..m, 1..m]$, where $T[u, v]$ is the time required to walk from location u to location v .

We want to find the minimum number of professors that can collectively teach every class, such that whenever a professor is assigned to teach two classes i and j where $C[j].start \geq C[i].start$, we actually have

$$C[j].start \geq C[i].end + T[C[i].loc, C[j].loc].$$

We can solve this problem by reducing it to a disjoint-path cover problem as follows. We construct a dag $G = (V, E)$ whose vertices are classes and whose edges represent pairs of classes that could be taught by the same professor. Specifically, a directed edge $i \rightarrow j$ indicates that class j can be taught after class i . We can easily construct this dag in $O(n^2)$ time by brute force. Then we can find a disjoint-path cover of G using the matching algorithm described above; each directed path in G represents a legal class schedule for one professor. The entire algorithm runs in $O(n^2 + VE) = O(n^3)$ time.⁴

Despite its initial description in terms of intervals and distances, this is really a matching problem. Specifically, we want to *match* as many classes as possible to the next class to be taught by the same professor. The number of professors we need is equal to the number of classes with no assigned successor; each class without an assigned successor is the last class that some professor teaches.

11.6 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 Tigers	49–86	27	3	4	0	0	

⁴Most American universities schedule ten-minute breaks between classes, under the remarkable belief that a normal human being can get from *any* two classroom on campus to *any* other classroom in ten minutes. If we assume that the time interval $T[u, v]$ is identical for all u and v , this scheduling problem can be solved in $O(n \log n)$ time using a simple greedy algorithm.

Detroit is clearly behind, but some die-hard Tigers 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:⁵

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 lose all the games they play against 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 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 win all of these games. But if that happens, the Blue Jays will win at least 14 more games giving them a 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 got 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).⁶

In the mid-1960s, Benjamin Schwartz observed that this question can be modeled as a maximum flow problem; about 20 years later, Dan Gusfield, Charles Martel, and David Fernández-Baca simplified Schwartz's flow formulation to a bipartite assignment problem. Specifically, we want to know whether it is possible *assign a winner to each game*, so that team n comes in first place. 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 $g_{i,j}$,

⁵Both the example and this argument are taken from <http://riot.ieor.berkeley.edu/~baseball/detroit.html>.

⁶We 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).

one for each pair $1 \leq i < j < n$, and $n - 1$ team nodes t_i , one for each $1 \leq i < n$. For each pair i, j , we add edges $g_{i,j} \rightarrow t_i$ and $g_{i,j} \rightarrow t_j$ with infinite capacity. We add a source vertex s and edges $s \rightarrow g_{i,j}$ with capacity $G[i, j]$ for each pair i, j . Finally, we add a target node t and edges $t_i \rightarrow t$ 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 $s \rightarrow g_{i,j} \rightarrow t_i \rightarrow t$. 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 $f(g_{i,j} \rightarrow t_i)$ games against team j , for all i and j . Then teams i and j play $f(g_{i,j} \rightarrow t_i) + f(g_{i,j} \rightarrow t_j) = f(s \rightarrow g_{i,j}) = G[i, j]$ games, so every upcoming game is played. Moreover, each team i wins a total of $\sum_j f(g_{i,j} \rightarrow t_i) = f(t_i \rightarrow t) \leq W[n] + R[n] - W[i]$ 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. \square

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 Orlin’s algorithm, we can compute the maximum flow in $O(VE) = O(n^4)$ 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.

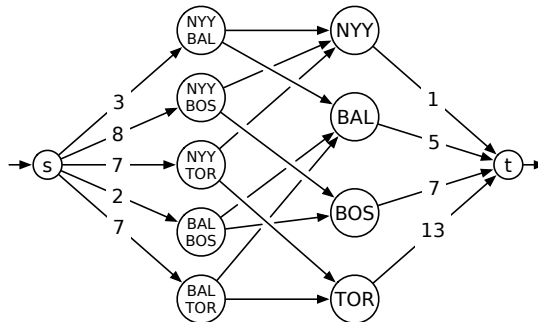


Figure 11.2. The flow graph for the 1996 American League East standings. Unlabeled edges have infinite capacity.

This is not the fastest algorithm for this problem. Kevin Wayne⁷ proved that one can determine *all* teams that are mathematically eliminated in only $O(n^3)$ time, essentially using a single maximum-flow computation.

11.7 Project Selection

In our final example, suppose we are given a set of n projects that we could possibly perform; for simplicity, we identify each project by an integer between 1 and n . Some projects cannot be started until certain other projects are completed. This set of dependencies is described by a directed acyclic graph, where an edge $i \rightarrow j$ indicates that project i depends on project j . Finally, each project i has an associated *profit* p_i which is given to us if the project is completed; however, some projects have negative profits, which we interpret as positive *costs*. We can choose to finish any subset X of the projects that includes all its dependents; that is, for every project $x \in X$, every project that x depends on is also in X . Our goal is to find a valid subset of the projects whose total profit is as large as possible. In particular, if all of the jobs have negative profit, the correct answer is to do nothing.

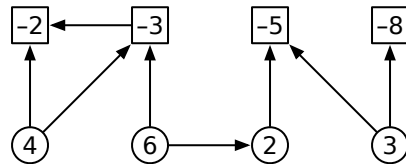


Figure 11.3. A dependency graph for a set of eight projects. Circles indicate profitable projects; squares indicate costly projects.

At a high level, our task to partition the projects into two subsets S and T , the jobs we *Select* and the jobs we *Turn down*. So intuitively, we'd like to model our problem as a minimum cut problem in a certain graph. But in which graph? How do we enforce prerequisites? We want to *maximize* profit, but we only know how to find *minimum* cuts. And how do we convert negative profits into positive capacities?

We define a new graph G by adding a source vertex s and a target vertex t to the dependency graph, with an edge $s \rightarrow j$ for every profitable job (with $p_j > 0$), and an edge $i \rightarrow t$ for every costly job (with $p_i < 0$). Intuitively, we can think of s as a new job (“To the bank!”) with profit/cost 0 that we must perform last. We assign edge capacities as follows:

- $c(s \rightarrow j) = p_j$ for every profitable job j ;
- $c(i \rightarrow t) = -p_i$ for every costly job i ;
- $c(i \rightarrow j) = \infty$ for every dependency edge $i \rightarrow j$.

⁷Kevin D. Wayne. A new property and a faster algorithm for baseball elimination. *SIAM J. Discrete Math* 14(2):223–229, 2001.

All edge-capacities are positive, so this is a legal input to the maximum cut problem.

Now consider an (s, t) -cut (S, T) in G . If the capacity $\|S, T\|$ is finite, then for every dependency edge $i \rightarrow j$, projects i and j are on the same side of the cut, which implies that S is a valid solution. Moreover, we claim that selecting the jobs in S earns us a total profit of $C - \|S, T\|$, where C is the sum of all the positive profits. This claim immediately implies that we can *maximize* our total profit by computing a *minimum* cut in G .

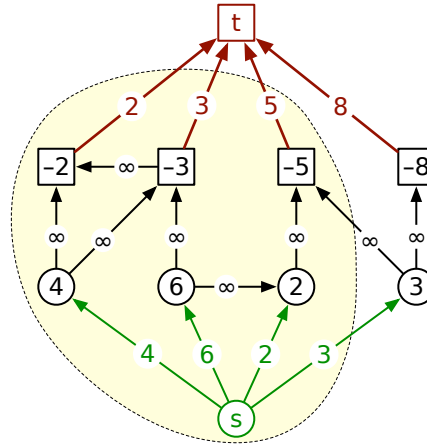


Figure 11.4. The flow network for the example dependency graph, along with its minimum cut. The cut has capacity 13 and $C = 15$, so the total profit for the selected jobs is 2.

We prove our key claim as follows. For any subset A of projects, we define three functions:

$$\begin{aligned}
 \text{cost}(A) &:= \sum_{i \in A: p_i < 0} -p_i = \sum_{i \in A} c(i \rightarrow t) \\
 \text{benefit}(A) &:= \sum_{j \in A: p_j > 0} p_j = \sum_{j \in A} c(s \rightarrow j) \\
 \text{profit}(A) &:= \sum_{i \in A} p_i = \text{benefit}(A) - \text{cost}(A).
 \end{aligned}$$

By definition, $C = \text{benefit}(S) + \text{benefit}(T)$. Because the cut (S, T) has finite capacity, only edges of the form $s \rightarrow j$ and $i \rightarrow t$ can cross the cut. By construction, every edge $s \rightarrow j$ points to a profitable job and each edge $i \rightarrow t$ points from a costly job. Thus, $\|S, T\| = \text{cost}(S) + \text{benefit}(T)$. We immediately conclude that $C - \|S, T\| = \text{benefit}(S) - \text{cost}(S) = \text{profit}(S)$, as claimed.

Exercises

1. 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.

2. Consider a directed graph $G = (V, E)$ with multiple source vertices $s_1, s_2, \dots, s_\sigma$ and multiple target vertices t_1, t_2, \dots, t_τ , where no vertex is both a source and a target. A *multiterminal flow* is a function $f : E \rightarrow \mathbb{R}_{\geq 0}$ 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:

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

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 $\text{MAXFLOW}(G, s, t)$ solves the standard maximum flow problem with source s and target t .

```

MAXMULTIFLOW( $G, s[1.. \sigma], t[1.. \tau]$ ):
   $f \leftarrow 0$                                 ⟨⟨Initialize the flow⟩⟩
  for  $i \leftarrow 1$  to  $\sigma$ 
    for  $j \leftarrow 1$  to  $\tau$ 
       $f' \leftarrow \text{MAXFLOW}(G_f, s[i], t[j])$ 
       $f \leftarrow f + f'$                        ⟨⟨Update the flow⟩⟩
  return  $f$ 

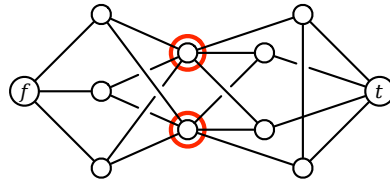
```

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 .
3. 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 . [Hint: If G were a directed graph, this problem would be NP-hard!]
4. The Island of Sodor is home to a large number of towns and villages, connected by an extensive rail network. Recently, several cases of a deadly contagious disease (either swine flu or zombies; reports are unclear) have been reported in the village of Ffarquhar. The controller of the Sodor railway plans to close down certain railway stations to prevent the disease from spreading to Tidmouth, his home town. No trains can pass through a closed station. To minimize expense (and public notice), he wants to close down as few stations as possible. However, he cannot close the Ffarquhar station, because that would expose him to the disease, and he cannot close the Tidmouth station, because then he couldn't visit his favorite pub.

Describe and analyze an algorithm to find the minimum number of stations that must be closed to block all rail travel from Ffarquhar to Tidmouth. The Sodor rail network is represented by an undirected graph, with a vertex for each station and an edge for each rail connection between two stations. Two special vertices f and t represent the stations in Ffarquhar and Tidmouth.

For example, given the following input graph, your algorithm should return the integer 2.



5. An $n \times n$ grid is an undirected graph with n^2 vertices organized into n rows and n columns. We denote the vertex in the i th row and the j th column by (i, j) . Every vertex (i, j) has exactly four neighbors $(i - 1, j)$, $(i + 1, j)$, $(i, j - 1)$, and $(i, j + 1)$, except the *boundary* vertices, for which $i = 1$, $i = n$, $j = 1$, or $j = n$.

Let $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$ be distinct vertices, called *terminals*, in the $n \times n$ grid. The **escape problem** is to determine whether there are m vertex-disjoint paths in the grid that connect the terminals to any m distinct boundary vertices.

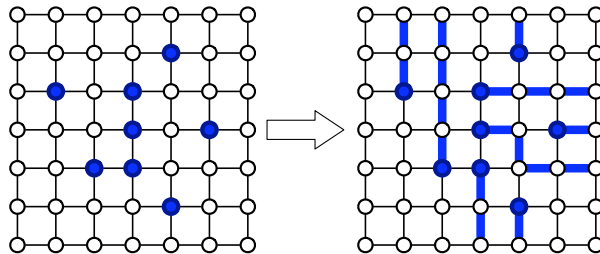


Figure 11.5. A positive instance of the escape problem, and its solution.

- (a) Describe and analyze an efficient algorithm to solve the escape problem. The running time of your algorithm should be a small polynomial function of n .
- (b) Now suppose the input to the escape problem consists of a single integer n and the list of m terminal vertices. If m is very small, the previous running time is actually *exponential* in the input size! Describe and analyze an algorithm to solve the escape problem in time *polynomial* in m .
- (c) Modify the previous algorithm to output an explicit description of the escape paths (if they exist), still in time polynomial in m .
6. The UIUC Computer Science Department is installing a mini-golf course in the basement of Siebel Center! The playing field is a closed polygon bounded by m

horizontal and vertical line segments, meeting at right angles. The course has n starting points and n holes, in one-to-one correspondence. It is always possible hit the ball along a straight line directly from each starting point to the corresponding hole, without touching the boundary of the playing field. (Players are not allowed to bounce golf balls off the walls; too much glass.) The n starting points and n holes are all at distinct locations.

Sadly, the architect's computer crashed just as construction was about to begin. Thanks to the herculean efforts of their sysadmins, they were able to recover the *locations* of the starting points and the holes, but all information about which starting points correspond to which holes was lost!

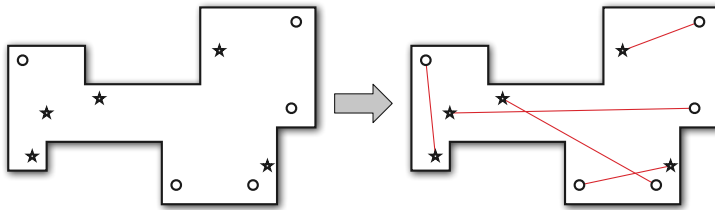
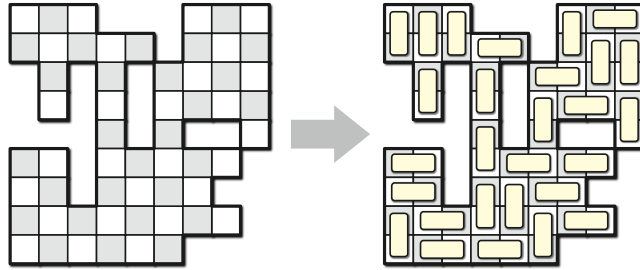


Figure 11.6. A minigolf course with five starting points (★) and holes (○), and a legal correspondence between them.

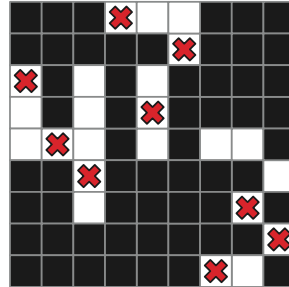
Describe and analyze an algorithm to compute a one-to-one correspondence between the starting points and the holes that meets the straight-line requirement, or to report that no such correspondence exists. The input consists of the x - and y -coordinates of the m corners of the playing field, the n starting points, and the n holes. Assume you can determine in constant time whether two line segments intersect, given the x - and y -coordinates of their endpoints.

7. A **cycle cover** of a given directed graph $G = (V, E)$ is a set of vertex-disjoint cycles that cover every vertex in G . 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!]
8. Suppose you are given an $n \times n$ checkerboard with some of the squares deleted. You have a large set of dominos, just the right size to cover two squares of the checkerboard. Describe and analyze an algorithm to determine whether one tile the board with dominos—each domino must cover exactly two undeleted squares, and each undeleted square must be covered by exactly one domino.



Your input is a two-dimensional array $Deleted[1..n, 1..n]$ of bits, where $Deleted[i, j] = \text{TRUE}$ if and only if the square in row i and column j has been deleted. Your output is a single bit; you do **not** have to compute the actual placement of dominos. For example, for the board shown above, your algorithm should return **TRUE**.

9. Suppose we are given an $n \times n$ square grid, some of whose squares are colored black and the rest white. Describe and analyze an algorithm to determine whether tokens can be placed on the grid so that
- every token is on a white square;
 - every row of the grid contains exactly one token; and
 - every column of the grid contains exactly one token.



Your input is a two dimensional array $IsWhite[1..n, 1..n]$ of booleans, indicating which squares are white. Your output is a single boolean. For example, given the grid above as input, your algorithm should return **TRUE**.

10. 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.

11. 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 $M[i, j] = \text{TRUE}$ 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.

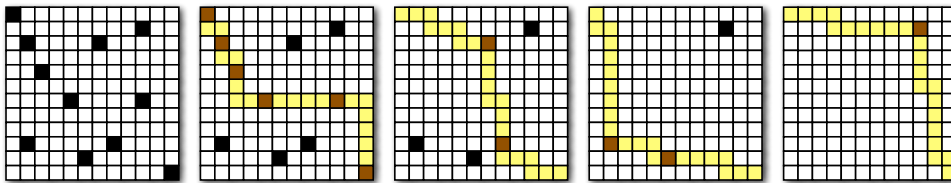


Figure 11.7. Greedily covering the marked cells in a grid with four monotone paths.

- (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.
12. You're organizing the First Annual UIUC Computer Science 72-Hour Dance Exchange, to be held all day Friday, Saturday, and Sunday. Several 30-minute sets of music will be played during the event, and a large number of DJs have applied to perform. You need to hire DJs according to the following constraints.
- Exactly k sets of music must be played each day, and thus $3k$ sets altogether.
 - Each set must be played by a single DJ in a consistent music genre (ambient, bubblegum, dubstep, horrorcore, hyphy, trip-hop, Nitzhonot, Kwaito, J-pop, Nashville country, ...).
 - Each genre must be played at most once per day.
 - Each candidate DJ has given you a list of genres they are willing to play.
 - Each DJ can play at most three sets during the entire event.
- Suppose there are n candidate DJs and g different musical genres available. Describe and analyze an efficient algorithm that either assigns a DJ and a genre to each of the $3k$ sets, or correctly reports that no such assignment is possible.
13. Suppose you are running a web site that is visited by the same set of people every day. Each visitor claims membership in one or more *demographic groups*; for example, a visitor might describe himself as male, 40–50 years old, a father, a resident of

Illinois, an academic, a blogger, and a fan of Joss Whedon.⁸ Your site is supported by advertisers. Each advertiser has told you which demographic groups should see its ads and how many of its ads you must show each day. Altogether, there are n visitors, k demographic groups, and m advertisers.

Describe an efficient algorithm to determine, given all the data described in the previous paragraph, whether you can show each visitor exactly *one* ad per day, so that every advertiser has its desired number of ads displayed, and every ad is seen by someone in an appropriate demographic group.

14. 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:

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

- (a) Describe and analyze an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible.
- (b) Prove that a legal rounding is possible *if and only if* the sum of entries in each row is an integer, and the sum of entries in each column is an integer. In other words, prove that either your algorithm from part (a) returns a legal rounding, or a legal rounding is *obviously* impossible.
- ♥(c) Suppose we are guaranteed that none of the entries in the input matrix A is an integer. Describe and analyze an even faster algorithm that either rounds A or reports correctly that no such rounding is possible. For full credit, your algorithm must run in $O(mn)$ time. [Hint: **Don't** use flows.]
15. **Ad-hoc networks** are made up of low-powered wireless devices. In principle⁹, 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.

These devices can communicate 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

⁸Har har har! Mine is an evil laugh! Now *die!*

⁹but not so much in practice

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 no device appears in more than b backup sets, or reports (correctly) that no good collection of backup sets exists.

16. Faced with the threat of brutally severe budget cuts, Potemkin University has decided to hire actors to sit in classes as “students”, to ensure that every class they offer is completely full. Because actors are expensive, the university wants to hire as few of them as possible.

Building on their previous experience at the now-defunct Shampoo-Banana University, the administrators at Potemkin have given you a directed acyclic graph $G = (V, E)$, whose vertices represent classes, and where each edge $i \rightarrow j$ indicates that the same “student” can attend class i and then later attend class j . In addition, you are also given an array $cap[1..V]$ listing the maximum number of “students” who can take each class. Describe an algorithm to compute the minimum number of “students” that would allow every class to be filled to capacity.

17. Quentin, Alice, and the other Brakebills Physical Kids are planning an excursion through the Neitherlands to Fillory. The Neitherlands is a vast, deserted city composed of several plazas, each containing a single fountain that can magically transport people to a different world. Adjacent plazas are connected by gates, which have been cursed by the Beast. The gates between plazas are open only for five minutes every hour, all simultaneously—from 12:00 to 12:05, then from 1:00 to 1:05, and so on—and are otherwise locked. During those five minutes, if more than one person passes through any single gate, the Beast will detect their presence.¹⁰ Moreover, anyone attempting to open a locked gate, or attempting to pass through more than one gate within the same five-minute period will turn into a niffin.¹¹ However, any number of people can safely pass through *different* gates at the same time and/or pass through the same gate at *different* times.

You are given a map of the Neitherlands, which is a graph G with a vertex for each fountain and an edge for each gate, with the fountains to Earth and Fillory clearly marked.

¹⁰This is very bad.

¹¹This is very very bad.

- (a) Suppose you are also given a positive integer h . Describe and analyze an algorithm to compute the maximum number of people that can walk from the Earth fountain to the Fillory fountain in at most h hours—that is, after the gates have opened at most h times—without anyone alerting the Beast or turning into a niffin. [Hint: Build a different graph.]
- (b) On the other hand, suppose you are also given an integer k . Describe and analyze an algorithm to compute the minimum number of hours that allow k people to walk from the Earth fountain to the Fillory fountain, without anyone alerting the Beast or turning into a niffin. [Hint: Use part (a).]
- ♥18. 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 an efficient algorithm to compute, given two rooted trees A and B , the largest rooted tree that is isomorphic to both a subgraph of A and a subgraph of B . More briefly, describe an algorithm to find the largest common subtree of two rooted trees.
- [Hint: This would be a relatively straightforward dynamic programming problem if either every node had $O(1)$ children or the children of each node were ordered from left to right. But for unordered trees with large degree, you need another technique to combine recursive subproblems efficiently.]