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 $O(V'E') = \boldsymbol{O(VE)}$ ***time***.

Despite its formulation in terms of dags and paths, this is really a maximum 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. (And of course, every bipartite maximum matching problem is really a flow problem.)

Minimal Teaching Assignment

Let's go back to Sham-Poobanana University for another "real-world" scheduling problem.[6] SPU offers several thousand courses every day. Due to extreme budget cuts, the university needs to significantly reduce the size of its faculty. However, because students pay tuition (and the university cannot afford lawyers), the university must retain enough professors to guarantee that every class advertised in the course catalog is actually taught. How few professors can SPU 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.[7]

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

- 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

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

[6] For 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.

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

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 are scheduled far enough apart to be taught by the same professor. Specifically, a directed edge $i \to j$ indicates that the same professor can teach class i and then class j. It is easy to 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 $O(n^2 + VE) = \boldsymbol{O(n^3)}$ ***time***.[8]

Despite its initial description in terms of intervals and distances, this is really a maximum matching problem. Specifically, we want to *match* as many classes as possible to the *next* class 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 American 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	

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,

[8]Most 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 actually be solved in $O(n \log n)$ time using a simple greedy algorithm.

because some of those other teams still have to play each other. Here is one complete argument:[9]

> 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 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).[10]

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 pair selection problem. Specifically, we want to know whether it is possible to ***select*** *a winner for 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.

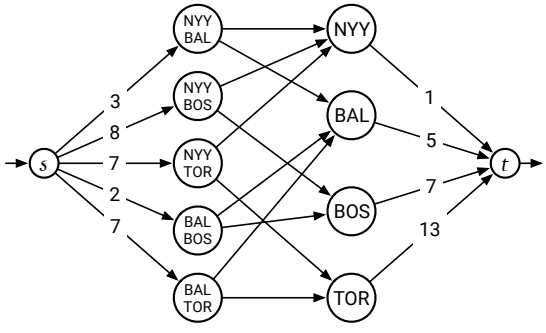
Since we want to ***select*** a winning team for each game, we start by building a bipartite graph, whose nodes represent the games and the teams. We have

[9] Both the example and this argument are taken from Eli V. Olinick's web site https://s2.smu.edu/~olinick/riot/detroit.html, which is based on Olinick's joint research with Ilan Adler, Alan Erera, and Dorit Hochbaum.

[10] We assume here that no games end in a tie (true for Major League Baseball games that affect postseason standing, and *mostly* true otherwise), and that every game is actually played (not true for Major League Baseball games during the regular season).

$\binom{n}{2}$ *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.

Here is the graph derived from the 1996 American League East standings, where "team n" is the Detroit Tigers. All unlabeled edges have infinite capacity.



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 $f(g_{i,j} \to t_i) + f(g_{i,j} \to t_j) = f(s \to g_{i,j}) = G[i,j]$ 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. □

In summary, to decide whether our favorite team can win, we construct the flow network, compute a maximum flow, and report whether than maximum flow saturates every edge out of s. For example, in the graph shown above, the total capacity of the edges leaving s is 27 (because there are 27 remaining games). On the other hand, the total capacity of the edges entering t is only 26,

which implies that the maximum flow value is at most 26. We conclude that Detroit is mathematically eliminated.[11]

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 $O(VE) = \boldsymbol{O(n^4)}$ ***time***.

This is not the fastest algorithm for the baseball elimination problem. In 2001, Kevin Wayne proved that one can determine ***all*** teams that are mathematically eliminated in only $\boldsymbol{O(n^3)}$ ***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. Some projects cannot be started until certain other projects are completed. The projects and their dependencies are described by a directed acyclic graph G whose vertices are the projects, where each edge $u \to v$ indicates that project u cannot be performed before project v. (This is exactly the form of dependency graphs we considered in Chapter 6.4.) Finally, each project v has an associated *profit* $\$(v)$ which will be given to us if the project is completed; 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.

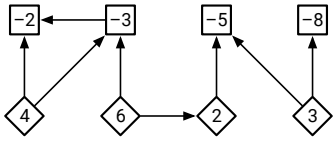


Figure 11.6. A dependency graph for a set of eight projects. Diamonds indicate profitable projects; squares indicate costly projects. Each edge $u \to v$ means u depends on v.

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?

[11] We got lucky here; it is possible for a team to be eliminated even if the total capacity of all edges into t is no smaller than the total capacity of edges out of s.

To transform our given constraint graph G into a flow network G', we add a source vertex s and a target vertex t to the dependency graph, with an edge $s \to v$ for every profitable job v (with $\$(v) > 0$), and an edge $u \to t$ for every costly job u (with $\$(u) < 0$). Intuitively, we can think of s as a new job ("Sleep!") with profit/cost 0 that we must perform last. We assign capacities to the edges of G' as follows:

- $c(s \to v) = \$(v)$ for every profitable job v;
- $c(u \to t) = -\$(u)$ for every costly job u;
- $c(u \to v) = \infty$ for every dependency edge $u \to v$.

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

Now consider an arbitrary (s,t)-cut (S,T) in G'. For any edge $u \to v$ in the original dependency graph, if $u \in S$ and $v \in T$, then $\|S,T\| = \infty$. Thus, we can legally select the jobs in S if and only if the capacity of the cut (S,T) is finite.

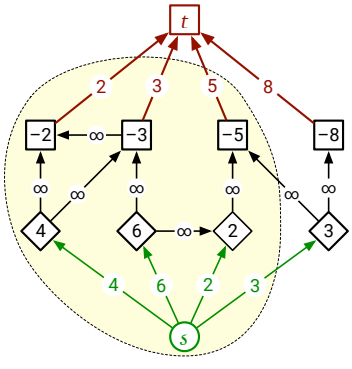


Figure 11.7. The flow network for the example dependency graph, along with its minimum cut. The cut has capacity 13 and $P = 15$, so the total profit for the selected jobs is 2.

In fact, it turns out that cuts with smaller capacity correspond to job selections with higher profit. Specifically, I claim that selecting the jobs in S earns a total profit of $P - \|S,T\|$, where P is the sum of all the positive profits:

$$P = \sum_v \max\{0, \$(v)\} = \sum_{\$(v)>0} \$(v).$$

We can prove this claim by straightforward definition-chasing, as follows. For any subset X of projects, we define three values. (Here, as usual, we define $c(u \to v) = 0$ when $u \to v$ is not an edge.)

$$cost(X) := \sum_{\substack{u \in X \\ \$(u)<0}} -\$(u) = \sum_{u \in X} c(u \to t)$$

$$yield(X) := \sum_{\substack{v \in X \\ \$(v)>0}} \$(v) = \sum_{v \in X} c(s \to v)$$

$$profit(X) := \sum_{v \in X} \$(v) = yield(X) - cost(X).$$

By definition, $P = yield(V) = yield(S) + yield(T)$. Because the cut (S, T) has finite capacity, only edges of the form $s \to v$ and $u \to t$ can cross the cut. By construction, every edge $s \to v$ points to a profitable job and each edge $u \to t$ points from a costly job. Thus, $\|S, T\| = cost(S) + yield(T)$. We immediately conclude that $P - \|S, T\| = yield(S) - cost(S) = profit(S)$, as claimed.

It follows immediately that we can *maximize* our total profit by computing a *minimum* cut in G'. We can easily construct G' from G in $O(V + E)$ time, and we can compute the minimum (s, t)-cut in G' in $O(VE)$ time using Orlin's algorithm. We conclude that the entire project-selection algorithm runs in $\boldsymbol{O(VE)}$ ***time***.

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. Suppose G contains k edge-disjoint paths from some vertex u to another vertex 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. 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!]*

3. Consider a directed graph $G = (V, E)$ with several source vertices $s_1, s_2, \ldots, s_\sigma$ and target vertices $t_1, t_1, \ldots, t_\tau$, where no vertex is both a source and a target. A *multi-terminal 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 multi-terminal 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 multi-terminal 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
```

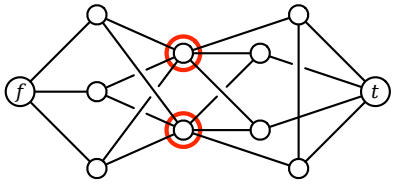
Prove that this algorithm correctly computes a maximum multi-terminal flow in G.

(b) Describe a more efficient algorithm to compute a maximum multi-terminal flow in G.

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 ith row and the jth 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 $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ 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.

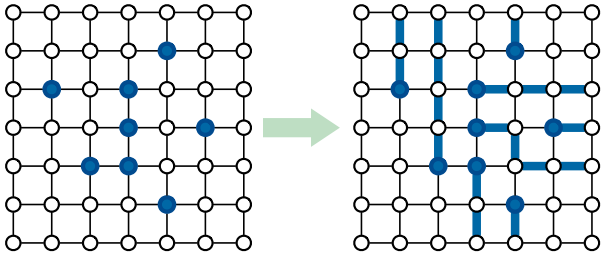


Figure 11.8. A positive instance of the escape problem, and its solution.

(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!

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

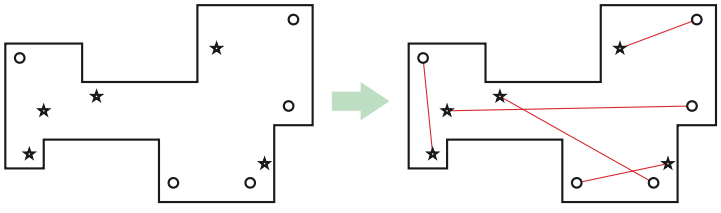


Figure 11.9. A mini-golf course with five starting points (⋆) and holes (○), and a legal correspondence between them.

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.

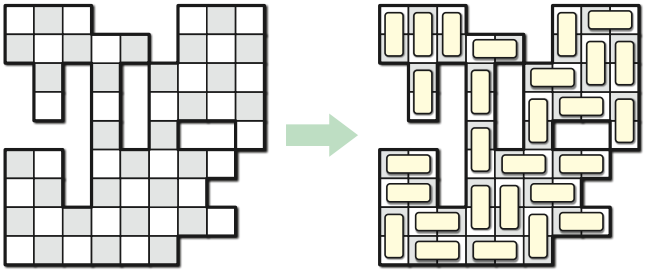


Figure 11.10. Covering a partial checkerboard with dominos.

Your input is a boolean array $Deleted[1..n, 1..n]$, where $Deleted[i, j] =$ TRUE if and only if the square in row i and column j has been deleted. Your output is a single boolean; 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.

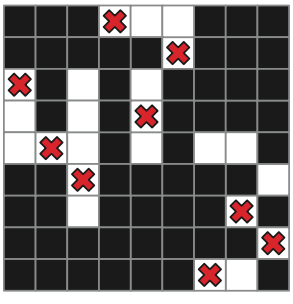


Figure 11.11. Marking every row and column in a grid.

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.

    (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

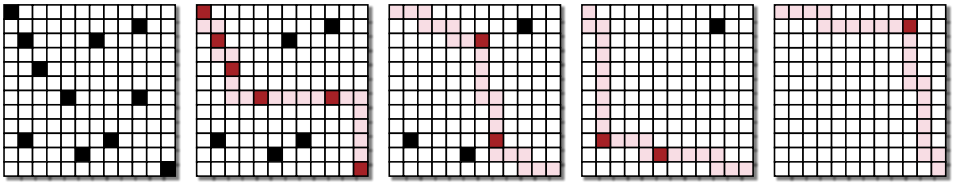


Figure 11.12. Greedily covering the marked cells in a grid with four monotone paths.

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. The Faculty Senate at Sham-Poobanana University has decided to convene a committee to determine whether Uncle Gabby, Professor Bobo Cornelius, or Mofo the Psychic Gorilla should replace the recently disgraced Baron Factotum as the new official ~~mascot~~ *symbol* of SPU's athletic teams (The Fighting Pooh-bahs). Exactly one faculty member must be chosen from each academic department to serve on this committee. Some faculty members have appointments in multiple departments, but each committee member can represent only one department. For example, if Prof. Blagojevich is affiliated with both the Department of Corruption and the Department of Stupidity, and he is chosen as the Stupidity representative, then someone else must represent Corruption. Finally, University policy requires that every faculty committee must contain exactly the same number of assistant professors, associate professors, and full professors. Fortunately, the number of departments is a multiple of 3.

Describe and analyze an algorithm to choose a subset of the SPU faculty to staff The Post-Factotum Simian ~~Mascot~~ *Symbol* Committee, or correctly report that no valid committee is possible. Your input is a bipartite graph indicating which professors belong to which departments; each professor vertex is labeled with that professor's rank (assistant, associate, or full).

13. The Department of Commuter Silence at Sham-Poobanana University has a flexible curriculum with a complex set of graduation requirements. The department offers n different courses, and there are m different requirements. Each requirement specifies a subset of the n courses and the number of courses that must be taken from that subset. The subsets for different requirements may overlap, but each course can only be used to satisfy *at most one* requirement.

For example, suppose there are $n = 5$ courses A, B, C, D, E and $m = 2$ graduation requirements:

- You must take at least 2 courses from the subset $\{A, B, C\}$.
- You must take at least 2 courses from the subset $\{C, D, E\}$.

Then a student who has only taken courses B, C, D cannot graduate, but a student who has taken either A, B, C, D or B, C, D, E can graduate.

Describe and analyze an algorithm to determine whether a given student can graduate. The input to your algorithm is the list of m requirements (each specifying a subset of the n courses and the number of courses that must be taken from that subset) and the list of courses the student has taken.

14. You're organizing the First Annual SPU Commuter Silence 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, K-pop, Kwaito, mariachi, straight-ahead jazz, trip-hop, Nashville country, parapara, ska, . . .).
    - 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.

15. 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.[12] 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.

[12] Har har har! Mine is an evil laugh! Now *die!*

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.

16. 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:

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

(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.]*

17. ***Ad-hoc networks*** are made up of low-powered wireless devices. In principle[13], 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 to fail, it should transmit its information to some other *backup* device within its communication range. We require each device x to have k

[13]but not so much in practice

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.

18. 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 leadership experience at the now-defunct Sham-Poobanana University, the administrators at Potemkin have given you a directed acyclic graph $G = (V, E)$, whose vertices represent classes, and where each edge $i \to j$ 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 analyze an algorithm to compute the minimum number of "students" that would allow every class to be filled to capacity.

19. 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.[14] 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.[15] 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.

[14] This is very bad.

[15] This is very very bad.

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.

(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. The running time of your algorithm should depend on h. *[Hint: Build a different graph.]*

♣♥(b) Describe an analyze an algorithm for part (a) whose running time is polynomial in V and E, with *no* dependence on h.

(c) 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).]*

♥20. Let $G = (L \sqcup R, E)$ be a bipartite graph, whose left vertices L are indexed $\ell_1, \ell_2, \ldots, \ell_n$ and whose right vertices are indexed $r_1, r_2, \ldots, r_n$. A matching M in G is ***non-crossing*** if, for every pair of edges $\ell_i r_j$ and $\ell_{i'} r_{j'}$ in M, we have $i < i'$ if and only if $j < j'$.

(a) Describe and analyze an algorithm to find the largest non-crossing matching in G. *[Hint: This is not really a flow problem.]*

(b) Describe and analyze an algorithm to find the smallest number of non-crossing matchings $M_1, M_2, \ldots, M_k$ such that each edge in G is in exactly one matching M_i. *[Hint: This is really a flow problem.]*

♥21. Let $G = (L \sqcup R, E)$ be a bipartite graph, whose left vertices L are indexed $\ell_1, \ell_2, \ldots, \ell_n$ in some arbitrary order.

(a) A matching M in G is ***dense*** if there are no consecutive unmatched vertices in L; that is, for each index i, either ℓ_i or ℓ_{i+1} is incident to an edge in M. Describe an algorithm to determine whether G has a dense matching.

(b) A matching M in G is ***sparse*** if there are no consecutive *matched* vertices in L; that is, for each index i, either ℓ_i or ℓ_{i+1} is *not* incident to an edge in M. (In particular, the empty matching is sparse.) Describe an algorithm to find the largest sparse matching in G.

(c) A matching M in G is ***palindromic*** if, for every index i, either ℓ_i and ℓ_{n-i+1} are both incident to edges in M, or neither ℓ_i nor ℓ_{n-i+1} is incident

to an edge in M. (In particular, the empty matching is palindromic.) Describe an algorithm to find the largest palindromic matching in G.

None of these problems restrict which vertices in R are matched or unmatched.

♥22. 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.]

[I]n his short and broken treatise he provides an eternal example—not of laws, or even of method, for there is no method except to be very intelligent, but of intelligence itself swiftly operating the analysis of sensation to the point of principle and definition.

— T. S. Eliot on Aristotle, "The Perfect Critic", *The Sacred Wood* (1921)

The nice thing about standards is that you have so many to choose from; furthermore, if you do not like any of them, you can just wait for next year's model.

— Andrew S. Tannenbaum, *Computer Networks* (1981)

It is a rare mind indeed that can render the hitherto non-existent blindingly obvious. The cry "I could have thought of that" is a very popular and misleading one, for the fact is that they didn't, and a very significant and revealing fact it is too.

— Dirk Gently to Richard McDuff
in Douglas Adams' *Dirk Gently's Holistic Detective Agency* (1987)

If a problem has no solution, it may not be a problem, but a fact — not to be solved, but to be coped with over time.

— Shimon Peres, as quoted by David Rumsfeld, *Rumsfeld's Rules* (2001)

12

NP-Hardness

12.1 A Game You Can't Win

Imagine that a salesman in a red suit, who looks suspiciously like Tom Waits, presents you with a black steel box with n binary switches on the front and a light bulb on the top. The salesman tells you that the state of the light bulb is controlled by a complex *boolean circuit*—a collection of AND, OR, and NOT gates connected by wires, with one input wire for each switch and a single output wire for the light bulb. He then asks you the following question: Is there a way to set the switches so that the light bulb turns on? If you can answer this question correctly, he will give you the box and a ~~million~~ ~~billion~~ trillion dollars; if you answer incorrectly, or if you die without answering at all, he will take your soul.

As far as you can tell, the Adversary hasn't connected the switches to the light bulb at all, so no matter how you set the switches, the light bulb will stay off. If you declare that it *is* possible to turn on the light, the Adversary will open

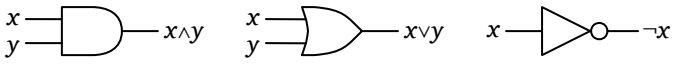


Figure 12.1. An AND gate, an OR gate, and a NOT gate.

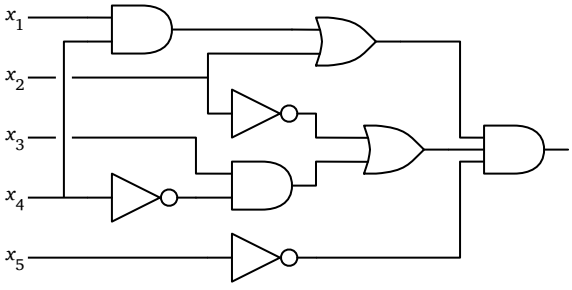


Figure 12.2. A boolean circuit. Inputs enter from the left, and the output leaves to the right.

the box and reveal that there is no circuit at all. But if you declare that it is *not* possible to turn on the light, before testing all 2^n settings, the Adversary will magically create a circuit inside the box that turns on the light *if and only if* the switches are in one of the settings you haven't tested, and then flip the switches to that setting, turning on the light. (You can't detect the Adversary's cheating, because you can't see inside the box until the end.) The only way to *provably* answer the Adversary's question correctly is to try all 2^n possible settings. You quickly realize that this will take *far* longer than you expect to live, so you gracefully decline the Adversary's offer.

The Adversary smiles and says, in a growl like Heath Ledger's Joker after smoking a carton of Marlboros, "Ah, yes, of course, you have no reason to trust me. But perhaps I can set your mind at ease." He hands you a large roll of parchment—which you hope was made from *sheep* skin—with a circuit diagram drawn (or perhaps tattooed) on it. "Here are the complete plans for the circuit inside the box. Feel free to poke around inside the box to make sure the plans are correct. Or build your own circuit from these plans. Or write a computer program to simulate the circuit. Whatever you like. If you discover that the plans don't match the actual circuit in the box, you win the trillion bucks." A few spot checks convince you that the plans have no obvious flaws; subtle cheating appears to be impossible.

But you should still decline the Adversary's "generous" offer. The problem that the Adversary is posing is called ***circuit satisfiability*** or ***CIRCUITSAT***: Given a boolean circuit, is there is a set of inputs that makes the circuit output TRUE, or conversely, whether the circuit *always* outputs FALSE. For any *particular* input setting, we can calculate the output of the circuit in polynomial (actually, *linear*) time using depth-first-search. But nobody knows how to solve CIRCUITSAT faster than just trying all 2^n possible inputs to the circuit by brute force, which requires

exponential time. Admittedly, nobody has actually formally *proved* that we can't beat brute force—maybe, just *maybe*, there's a clever algorithm that just hasn't been discovered yet—but nobody has actually formally proved that anti-gravity unicorns don't exist, either. For all practical purposes, it's safe to assume that there is no fast algorithm for CIRCUITSAT.

You tell the salesman no. He smiles and says, "You're smarter than you look, kid," and then flies away on his anti-gravity unicorn.

12.2 P versus NP

A minimal requirement for an algorithm to be considered "efficient" is that its running time is bounded by a polynomial function of the input size: $O(n^c)$ for some constant c, where n is the size of the input.[1] Researchers recognized early on that not all problems can be solved this quickly, but had a hard time figuring out exactly which ones could and which ones couldn't. There are several so-called ***NP-hard*** problems, which most people believe *cannot* be solved in polynomial time, even though nobody can prove a super-polynomial lower bound.

A *decision problem* is a problem whose output is a single boolean value: YES or NO. Let me define three classes of decision problems:

- ***P*** is the set of decision problems that can be solved in polynomial time. Intuitively, P is the set of problems that can be solved quickly.
- ***NP*** is the set of decision problems with the following property: If the answer is YES, then there is a *proof* of this fact that can be checked in polynomial time. Intuitively, NP is the set of decision problems where we can verify a YES answer quickly if we have the solution in front of us.
- ***co-NP*** is essentially the opposite of NP. If the answer to a problem in co-NP is NO, then there is a proof of this fact that can be checked in polynomial time.

For example, the circuit satisfiability problem is in NP. If a given boolean circuit is satisfiable, then any set of m input values that produces TRUE output is a proof that the circuit is satisfiable; we can check the proof by evaluating the circuit in polynomial time. It is widely believed that circuit satisfiability is *not* in P or in co-NP, but nobody actually knows.

Every decision problem in P is also in NP. If a problem is in P, we can verify YES answers in polynomial time recomputing the answer from scratch! Similarly, every problem in P is also in co-NP.

[1] This notion of efficiency was independently formalized by Alan Cobham in 1965, Jack Edmonds in 1965, and Michael Rabin in 1966, although similar notions were considered more than a decade earlier by Kurt Gödel, John Nash, and John von Neumann.

Perhaps the single most important unanswered question in theoretical computer science—if not all of computer science—if not all of ***science***—is whether the complexity classes P and NP are actually different. Intuitively, it seems obvious to most people that $P \neq NP$; the homeworks and exams in your algorithms and data structures classes have (I hope) convinced you that problems can be incredibly hard to solve, even when the solutions are simple in retrospect. It's completely obvious; *of course* solving problems from scratch is harder than just checking that a solution is correct. We can reasonably accept—and most algorithm designers *do* accept—the statement "$P \neq NP$" as a law of nature, similar to other laws of nature like Maxwell's equations, general relativity, and the sun rising tomorrow morning that are strongly supported by evidence, but have no mathematical proof.

But if we're being mathematically rigorous, we have to admit that nobody knows how to *prove* that that $P \neq NP$. In fact, there has been little or no real progress toward a proof for decades.[2] The Clay Mathematics Institute lists P versus NP as the first of its seven Millennium Prize Problems, offering a $1,000,000 reward for its solution. And yes, in fact, several people *have* lost their souls, or at least their sanity, attempting to solve this problem.

A more subtle but still open question is whether the complexity classes NP and co-NP are different. Even if we can verify every YES answer quickly, there's no reason to believe we can also verify NO answers quickly. For example, as far as we know, there is no short proof that a boolean circuit is *not* satisfiable. It is generally believed that NP ≠ co-NP, but again, nobody knows how to prove it.

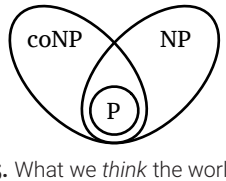


Figure 12.3. What we *think* the world looks like.

12.3 NP-hard, NP-easy, and NP-complete

A problem Π is ***NP-hard*** if a polynomial-time algorithm for Π would imply a polynomial-time algorithm for *every problem in NP*. In other words:

Π is NP-hard ⟺ If Π can be solved in polynomial time, then P=NP

[2]Perhaps the most significant progress has taken the form of *barrier* results, which imply that entire avenues of attack are doomed to fail. In a very real sense, not only do we have no idea how to prove $P \neq NP$, but we can actually *prove* that we have no idea how to prove $P \neq NP$!

Intuitively, if we could solve one particular NP-hard problem quickly, then we could quickly solve *any* problem whose solution is easy to understand, using the solution to that one special problem as a subroutine. NP-hard problems are at least as hard as every problem in NP.

Finally, a problem is ***NP-complete*** if it is both NP-hard and an element of NP (or "NP-easy"). Informally, NP-complete problems are the hardest problems in NP. A polynomial-time algorithm for even one NP-complete problem would immediately imply a polynomial-time algorithm for *every* NP-complete problem. Literally *thousands* of problems have been shown to be NP-complete, so a polynomial-time algorithm for one (and therefore all) of them seems incredibly unlikely.

Calling a problem NP-hard is like saying "If I own a dog, then it can speak fluent English." You probably don't know whether or not I own a dog, but I bet you're pretty sure that I don't own a *talking* dog. Nobody has a mathematical *proof* that dogs can't speak English—the fact that no one has ever heard a dog speak English is *evidence*, as are the hundreds of examinations of dogs that lacked the proper mouth shape and brainpower, but mere evidence is not a mathematical *proof*. Nevertheless, no sane person would believe me if I said I owned a dog that spoke fluent English. So the statement "If I own a dog, then it can speak fluent English" has a natural corollary: No one in their right mind should believe that I own a dog! Similarly, if a problem is NP-hard, no one in their right mind should believe it can be solved in polynomial time.

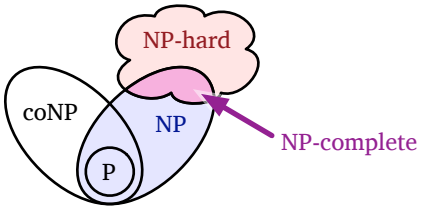


Figure 12.4. More of what we *think* the world looks like.

It is not obvious that *any* problems are NP-hard. The following remarkable theorem was first published by Steve Cook in 1971 and independently by Leonid Levin in 1973.[3] I won't even sketch the proof here, because I've been (deliberately) vague about the definitions; interested readers find a proof in my lecture notes on nondeterministic Turing machines.

[3]Levin first reported his results at seminars in Moscow in 1971, while still a PhD student. News of Cook's result did not reach the Soviet Union until at least 1973, after Levin's announcement of his results had been published; in accordance with Stigler's Law, this result is often called "Cook's Theorem". Levin was denied his PhD at Moscow University for political reasons; he emigrated to the US in 1978 and earned a PhD at MIT a year later. Cook was denied tenure at Berkeley in 1970, just one year before publishing his seminal paper; he (but not Levin) later won the Turing award for his proof.

The Cook-Levin Theorem. *Circuit satisfiability is NP-hard.*

♥12.4 Formal Definitions (*HC SVNT DRACONES*)

Formally, the complexity classes P, NP, and co-NP are defined in terms of *languages* and *Turing machines*. A language is a set of strings over some finite alphabet Σ; without loss of generality, we can assume that $\Sigma = \{0, 1\}$. A Turing machine is a very restrictive type of computer whose precise definition are surprisingly unimportant. P is the set of languages that can be decided in ***P***olynomial time by a deterministic single-tape Turing machine. Similarly, NP is the set of all languages that can be decided in polynomial time by a nondeterministic Turing machine; NP is an abbreviation for ***N***ondeterministic ***P***olynomial-time.

The requirement of polynomial time is sufficiently crude that we do not have to specify the precise form of Turing machine (number of tapes, number of heads, number of tracks, size of the tape alphabet, and so on). In fact, any algorithm that runs on a random-access machine[4] in $T(n)$ time can be simulated by a single-tape, single-track, single-head Turing machine that runs in $O(T(n)^4)$ time. This simulation result allows us to argue formally about computational complexity in terms of standard high-level programming constructs like for-loops and recursion, instead of describing everything directly in terms of Turing machines.

Formally, a problem Π is NP-hard if and only if, for every language $\Pi' \in$ NP, there is a polynomial-time ***Turing reduction*** from Π' to Π. A Turing reduction just means a reduction that can be executed on a Turing machine; that is, a Turing machine M that can solve Π' using another Turing machine M' for Π as a black-box subroutine. Turing reductions are also called *oracle reductions*; polynomial-time Turing reductions are also called *Cook reductions*.

Researchers in complexity theory prefer to define NP-hardness in terms of polynomial-time ***many-one reductions***, which are also called *Karp reductions*. A *many-one* reduction from one language $L' \subseteq \Sigma^*$ to another language $L \subseteq \Sigma^*$ is an function $f : \Sigma^* \to \Sigma^*$ such that $x \in L'$ if and only if $f(x) \in L$. Then we can

[4]Random-access machines are a model of computation that more faithfully models physical computers. A standard random-access machine has unbounded random-access memory, modeled as an unbounded array $M[0..\infty]$ where each address $M[i]$ holds a single w-bit integer, for some fixed integer w, and can read to or write from any memory addresses in constant time. RAM algorithms are formally written in assembly-like language, using instructions like **ADD** i, j, k (meaning "$M[i] \leftarrow M[j] + M[k]$"), **INDIR** i, j (meaning "$M[i] \leftarrow M[M[j]]$"), and **IF0GOTO** i, ℓ (meaning "if $M[i] = 0$, go to line ℓ"). The precise instruction set is surprisingly irrelevant. By definition, each instruction executes in unit time. In practice, RAM algorithms can be faithfully described using higher-level pseudocode, as long as we're careful about arithmetic precision.

define a *language* L to be NP-hard if and only if, for any language $L' \in$ NP, there is a many-one reduction from L' to L that can be computed in polynomial time.

Every Karp reduction "is" a Cook reduction, but not vice versa. Specifically, any Karp reduction from one decision problem Π to another decision Π' is equivalent to transforming the input to Π into the input for Π', invoking an oracle (that is, a subroutine) for Π', and then returning the answer verbatim. However, as far as we know, not every Cook reduction can be simulated by a Karp reduction.

Complexity theorists prefer Karp reductions primarily because NP is closed under Karp reductions, but is *not* closed under Cook reductions (unless NP=co-NP, which is considered unlikely). There are natural problems that are (1) NP-hard with respect to Cook reductions, but (2) NP-hard with respect to Karp reductions only if P=NP. One trivial example is of such a problem is UNSAT: Given a boolean formula, is it *always false*? On the other hand, many-one reductions apply *only* to decision problems (or more formally, to languages); formally, no optimization or construction problem is Karp-NP-hard.

To make things even more confusing, both Cook and Karp originally defined NP-hardness in terms of ***logarithmic-space*** reductions. Every logarithmic-space reduction is a polynomial-time reduction, but (as far as we know) not vice versa. It is an open question whether relaxing the set of allowed (Cook or Karp) reductions from logarithmic-space to polynomial-time changes the set of NP-hard problems.

Fortunately, none of these subtleties raise their ugly heads in practice—in particular, every algorithmic reduction described in these notes can be formalized as a logarithmic-space many-one reduction—so you can wake up now.

12.5 Reductions and SAT

To prove that any problem other than Circuit satisfiability is NP-hard, we use a *reduction argument*. Reducing problem A to another problem B means describing an algorithm to solve problem A under the assumption that an algorithm for problem B already exists. You're already used to doing reductions, only you probably call it something else, like writing subroutines or utility functions, or modular programming. To prove something is NP-hard, we describe a similar transformation between problems, but not in the direction that most people expect.

You should tattoo the following rule of onto the back of your hand, right next to your mom's birthday and the *actual* rules of Monopoly.[5]

[5]If a player lands on an available property and declines (or is unable) to buy it, that property is immediately auctioned off to the highest bidder; the player who originally declined the property may bid, and bids may be arbitrarily higher or lower than the list price. Players in Jail can still

To prove that problem *A* is NP-hard, reduce a known NP-hard problem to *A*.

In other words, to prove that your problem is hard, you need to describe an algorithm to solve a *different* problem, which you already know is hard, using a magical mystery algorithm for *your* problem as a subroutine. The essential logic is a proof by contradiction. The reduction implies that if your problem were easy, then the other problem would be easy, which it ain't. Equivalently, since you know the other problem is hard, the reduction implies that your problem must also be hard.

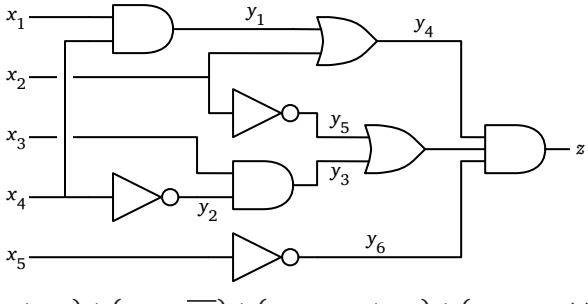
As a canonical example, consider the *formula satisfiability* problem, usually just called ***SAT***. The input to SAT is a boolean *formula* like

$$(a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee \overline{(\bar{a} \Rightarrow d)} \vee (c \neq a \wedge b)),$$

and the question is whether it is possible to assign boolean values to the variables $a, b, c, \ldots$ so that the entire formula evaluates to TRUE.

To prove that SAT is NP-hard, we need to give a reduction from a known NP-hard problem. The only problem we know is NP-hard so far is CIRCUITSAT, so let's start there.

Let K be an arbitrary boolean circuit. We can transform K into a boolean formula Φ by creating new output variables for each gate, and then just writing down the list of gates separated by ANDS. For example, our example circuit would be transformed into a formula as follows:



$$(y_1 = x_1 \wedge x_4) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1 \vee x_2) \wedge$$
$$(y_5 = \overline{x_2}) \wedge (y_6 = \overline{x_5}) \wedge (y_7 = y_3 \vee y_5) \wedge (z = y_4 \wedge y_7 \wedge y_6) \wedge z$$

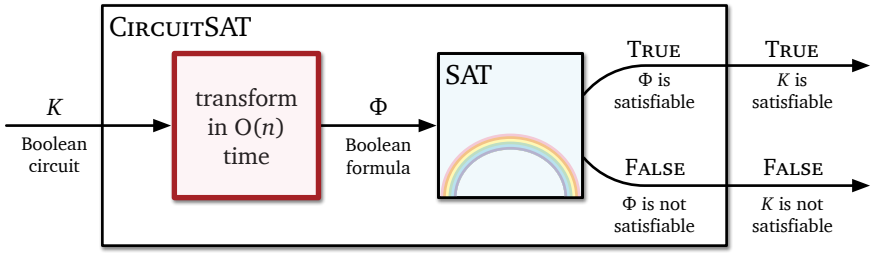
buy and sell property, buy and sell houses and hotels, and collect rent. The game has 32 houses and 12 hotels; once they're gone, they're gone. In particular, if all houses are already on the board, you cannot downgrade a hotel to four houses; you must raze all the hotels in the group to the ground. Players can sell or exchange *undeveloped* properties with each other, but cannot sell property back to the bank; on the other hand, players can sell buildings to the bank (at half price), but cannot sell or exchange buildings with each other. All penalties are paid directly to the bank. A player landing on Free Parking does not win anything. A player landing on Go gets exactly $200, no more. Railroads are not magic transporters. Finally, Jeff *always* gets the boot. No, not the T-Rex or the penguin—the *boot*, dammit.

Now we claim that the original circuit K is satisfiable ***if and only if*** the resulting formula Φ is satisfiable. Like every other "if and only if" statement, we prove this claim in two steps:

- $\Rightarrow$ Given a set of inputs that satisfy the circuit K, we can obtain a satisfying assignment for the formula Φ by computing the output of every gate in K.
- $\Leftarrow$ Given a satisfying assignment for the formula Φ, we can obtain a satisfying input the circuit by simply ignoring the internal gate variables y_i and the output variable z.

The entire transformation from circuit to formula can be carried out in linear time. Moreover, the size of the resulting formula is at most a constant factor larger than any reasonable representation of the circuit.

Now suppose, for the sake of argument, there is an algorithm that can determine in polynomial time whether a given boolean formula is satisfiable. Then given any boolean circuit K, we can decide whether K is satisfiable by first transforming K into a boolean formula Φ as described above, and then asking our magical mystery SAT algorithm whether Φ is satisfiable, as suggested by the following cartoon. Each box represents an algorithm. The red box on the left is the transformation subroutine. The box on the right is the magic SAT algorithm. It *must* be magic, because it has a *rainbow* on it.[6]



If you prefer magic pseudocode to magic boxes:

```
CIRCUITSAT(K):
  transcribe K into a boolean formula Φ
  return SAT(Φ)      ⟨⟨ ★★★ MAGIC ★★★ ⟩⟩
```

Transcribing K into Φ requires only polynomial time (in fact, only *linear* time, but whatever), so the entire CIRCUITSAT algorithm also runs in polynomial time.

$$T_{\text{CIRCUITSAT}}(n) \leq O(n) + T_{\text{SAT}}(O(n))$$

We conclude that any polynomial-time algorithm for SAT would give us a polynomial-time algorithm for CIRCUITSAT, which in turn would imply P=NP. So SAT is NP-hard!

[6] Kate Erickson, personal communication, 2011. For those of you reading black-and-white printed copies: Yes, that round thing is a rainbow.

12.6 3SAT (from SAT)

A special case of SAT that is particularly useful in proving NP-hardness results is called ***3CNF-SAT*** or more often just ***3SAT***.

A boolean formula is in *conjunctive normal form* (CNF) if it is a conjunction (AND) of several *clauses*, each of which is the disjunction (OR) of several *literals*, each of which is either a variable or its negation. For example:

$$\overbrace{(a \vee b \vee c \vee d)}^{\text{clause}} \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b})$$

A *3CNF* formula is a CNF formula with exactly three literals per clause; the previous example is not a 3CNF formula, since its first clause has four literals and its last clause has only two. 3SAT is just SAT restricted to 3CNF formulas: Given a 3CNF formula, is there an assignment to the variables that makes the formula evaluate to TRUE?

We could prove that 3SAT is NP-hard by a reduction from the more general SAT problem, but it's easier just to start over from scratch, by reducing directly from CIRCUITSAT.

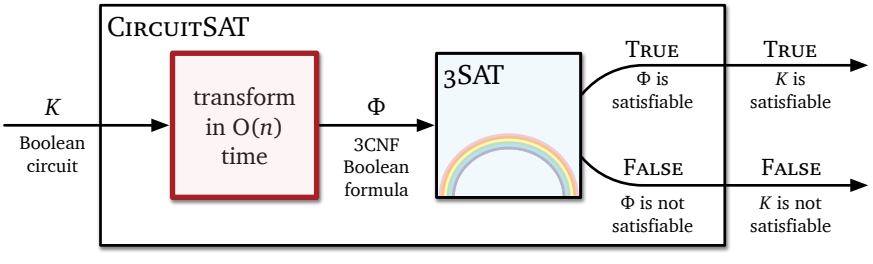


Figure 12.5. A polynomial-time reduction from CIRCUITSAT to 3SAT.

Given an arbitrary boolean circuit K, we transform K into an equivalent 3CNF formula in several stages. As we describe each stage, we also prove that stage is correct.

- *Make sure every AND and OR gate in K has exactly two inputs.* If any gate has $k > 2$ inputs, replace it with a binary tree of $k-1$ two-input gates. Call the resulting circuit K'. The circuits K and K' are logically equivalent circuits, so every satisfying input for K is a satisfying input for K' and vice versa.

- *Transcribe K' into a boolean formula Φ_1 with one clause per gate,* exactly as in our previous reduction to SAT. We already proved that every satisfying input for Φ_1 can be transformed into a satisfying assignment for Φ_1 and vice versa.

- *Replace each clause in Φ_1 with a CNF formula.* There are only three types of clauses in Φ_1, one for each type of gate in K':

$$\begin{aligned} a = b \wedge c &\longmapsto (a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee c) \\ a = b \vee c &\longmapsto (\bar{a} \vee b \vee c) \wedge (a \vee \bar{b}) \wedge (a \vee \bar{c}) \\ a = \bar{b} &\longmapsto (a \vee b) \wedge (\bar{a} \vee \bar{b}) \end{aligned}$$

  Call the resulting CNF formula Φ_2. Because Φ_1 and Φ_2 are logically equivalent formulas, every satisfying assignment for Φ_1 is also a satisfying assignment for Φ_2, and vice versa.[7]

- *Replace each clause in Φ_2 with a 3CNF formula.* Every clause in Φ_2 has at most three literals. We can keep the three-literal clauses as-is. We expand each two-literal clause into two three-literal clauses by introducing a new variable. Finally, we expand any one-literal clause into four three-literal clauses by introducing two new variables.

$$\begin{aligned} a \vee b &\longmapsto (a \vee b \vee x) \wedge (a \vee b \vee \bar{x}) \\ a &\longmapsto (a \vee x \vee y) \wedge (a \vee \bar{x} \vee y) \wedge (a \vee x \vee \bar{y}) \wedge (a \vee \bar{x} \vee \bar{y}) \end{aligned}$$

  Call the final 3CNF formula Φ_3. Every satisfying assignment for Φ_2 can be transformed into a satisfying assignment for Φ_3 by assigning *arbitrary* values to the new variables (x and y). Conversely, every satisfying assignment for Φ_3 can be transformed into a satisfying assignment for Φ_2 by ignoring the new variables.

For example, our example circuit is transformed into the following 3CNF formula:

$$\begin{gathered} (y_1 \vee \overline{x_1} \vee \overline{x_4}) \wedge (\overline{y_1} \vee x_1 \vee z_1) \wedge (\overline{y_1} \vee x_1 \vee \overline{z_1}) \wedge (\overline{y_1} \vee x_4 \vee z_2) \wedge (\overline{y_1} \vee x_4 \vee \overline{z_2}) \\ \wedge (y_2 \vee x_4 \vee z_3) \wedge (y_2 \vee x_4 \vee \overline{z_3}) \wedge (\overline{y_2} \vee \overline{x_4} \vee z_4) \wedge (\overline{y_2} \vee \overline{x_4} \vee \overline{z_4}) \\ \wedge (y_3 \vee \overline{x_3} \vee \overline{y_2}) \wedge (\overline{y_3} \vee x_3 \vee z_5) \wedge (\overline{y_3} \vee x_3 \vee \overline{z_5}) \wedge (\overline{y_3} \vee y_2 \vee z_6) \wedge (\overline{y_3} \vee y_2 \vee \overline{z_6}) \\ \wedge (\overline{y_4} \vee y_1 \vee x_2) \wedge (y_4 \vee \overline{x_2} \vee z_7) \wedge (y_4 \vee \overline{x_2} \vee \overline{z_7}) \wedge (y_4 \vee \overline{y_1} \vee z_8) \wedge (y_4 \vee \overline{y_1} \vee \overline{z_8}) \\ \wedge (y_5 \vee x_2 \vee z_9) \wedge (y_5 \vee x_2 \vee \overline{z_9}) \wedge (\overline{y_5} \vee \overline{x_2} \vee z_{10}) \wedge (\overline{y_5} \vee \overline{x_2} \vee \overline{z_{10}}) \\ \wedge (y_6 \vee x_5 \vee z_{11}) \wedge (y_6 \vee x_5 \vee \overline{z_{11}}) \wedge (\overline{y_6} \vee \overline{x_5} \vee z_{12}) \wedge (\overline{y_6} \vee \overline{x_5} \vee \overline{z_{12}}) \\ \wedge (\overline{y_7} \vee y_3 \vee y_5) \wedge (y_7 \vee \overline{y_3} \vee z_{13}) \wedge (y_7 \vee \overline{y_3} \vee \overline{z_{13}}) \wedge (y_7 \vee \overline{y_5} \vee z_{14}) \wedge (y_7 \vee \overline{y_5} \vee \overline{z_{14}}) \\ \wedge (y_8 \vee \overline{y_4} \vee \overline{y_7}) \wedge (\overline{y_8} \vee y_4 \vee z_{15}) \wedge (\overline{y_8} \vee y_4 \vee \overline{z_{15}}) \wedge (\overline{y_8} \vee y_7 \vee z_{16}) \wedge (\overline{y_8} \vee y_7 \vee \overline{z_{16}}) \\ \wedge (y_9 \vee \overline{y_8} \vee \overline{y_6}) \wedge (\overline{y_9} \vee y_8 \vee z_{17}) \wedge (\overline{y_9} \vee y_6 \vee z_{18}) \wedge (\overline{y_9} \vee y_6 \vee \overline{z_{18}}) \wedge (\overline{y_9} \vee y_8 \vee \overline{z_{17}}) \\ \wedge (y_9 \vee z_{19} \vee z_{20}) \wedge (y_9 \vee \overline{z_{19}} \vee z_{20}) \wedge (y_9 \vee z_{19} \vee \overline{z_{20}}) \wedge (y_9 \vee \overline{z_{19}} \vee \overline{z_{20}}) \end{gathered}$$

[7] This transformation from boolean circuit K to boolean formula Φ_2 in conjunctive normal form with *at most* three literals per clause was actually presented by Grigorii Tseitin in 1966, five years before Levin announced his results about *perebor* problems; in the same paper, Tseitin described the problem we now call CNF-SAT, perhaps for the first time.

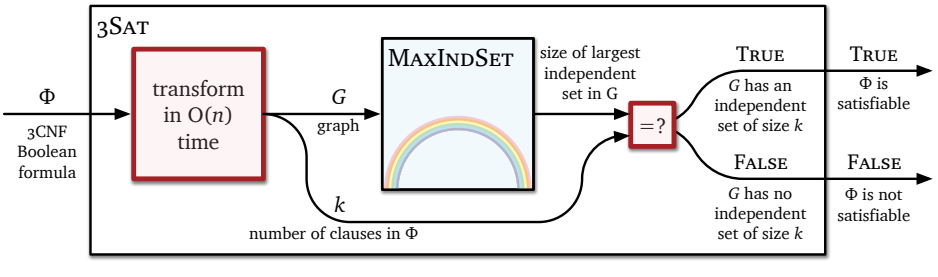
At first glance, this formula may look a *lot* more complicated than the original circuit, but in fact, it's only larger by a constant factor—each binary gate in the original circuit has been transformed into at most five clauses. Even if the formula size were a large *polynomial* function (like n^{374}) of the circuit size, we would still have a valid reduction.

Our reduction transforms an arbitrary boolean circuit K into a 3CNF formula Φ_3 in polynomial time. Moreover, any satisfying input for the input circuit K can be transformed into a satisfying assignment for Φ_3, and any satisfying assignment for Φ_3 can be transformed into a satisfying input for K. In other words, K is satisfiable if and only if Φ_3 is satisfiable. Thus, if 3SAT can be solved in polynomial time, then CIRCUITSAT can be solved in polynomial time, which implies that P = NP. We conclude 3SAT is NP-hard.

12.7 Maximum Independent Set (from 3SAT)

For the next few problems we consider, the input is a simple, unweighted, undirected graph, and the problem asks for the size of the largest or smallest subgraph satisfying some structural property.

Let G be an arbitrary graph. An ***independent set*** in G is a subset of the vertices of G with no edges between them. The *maximum independent set* problem, or simply ***MAXINDSET***, asks for the size of the largest independent set in a given graph. I will prove that MAXINDSET is NP-hard using a reduction from 3SAT, as suggested by the following figure:



Given an arbitrary 3CNF formula Φ, we construct a graph G as follows. Let k denote the number of clauses in Φ. The graph G contains exactly $3k$ vertices, one for each literal in Φ. Two vertices in G are connected by an edge if and only if either (1) they correspond to literals in the same clause, or (2) they correspond to a variable and its inverse. For example, the formula $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$ is transformed into the graph shown in Figure 12.6.

Each independent set in G contains at most one vertex from each clause triangle, because any two vertices in each triangle are connected. Thus, the largest independent set in G has size *at most* k. I claim that G contains an

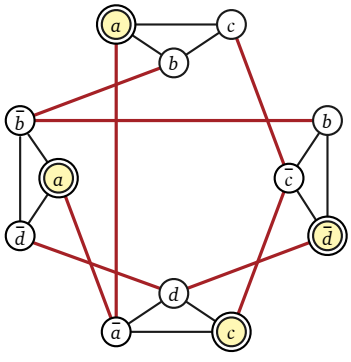


Figure 12.6. A graph derived from a satisfiable 3CNF formula with 4 clauses, and an independent set of size 4.

independent set of size *exactly* k if and only if the original formula Φ is satisfiable. As usual for "if and only if" statements, the proof consists of two parts.

$\Rightarrow$ Suppose Φ is satisfiable. Fix an arbitrary satisfying assignment. By definition, each clause in Φ contains at least one TRUE literal. Thus, we can choose a subset S of k vertices in G that contains exactly one vertex per clause triangle, such that the corresponding literals are all TRUE. Because each triangle contains at most one vertex in S, no two vertices in S are connected by a triangle edge. Because every literal in S is TRUE, no two vertices in S are connected by a negation edge. We conclude that S is an independent set of size k in G.

$\Leftarrow$ On the other hand, suppose G contains an independent set S of size k. Each vertex in S must lie in a different clause triangle. Suppose we assign the value TRUE to each literal in S; because contradictory literals are connected by edges, this assignment is consistent. There may be variables x such that neither x nor $\bar{x}$ corresponds to a vertex in S; we can set these variables to any value we like. Because S contains one vertex in each clause triangle, each clause in Φ contains (at least) one TRUE literal. We conclude that Φ is satisfiable.

Transforming the 3CNF formula Φ into the graph G takes polynomial time, even if we do everything by brute force. Thus, if we could solve MAXINDSET in polynomial time, then we could also solve 3SAT in polynomial time, by transforming the input formula Φ into a graph G and comparing the size of the largest independent set in G with the number of clauses in Φ. But that would imply P=NP, which is ridiculous! We conclude that MAXINDSET is NP-hard.

12.8 The General Pattern

All NP-hardness proofs—and more generally, all polynomial-time reductions—follow the same general pattern. To reduce problem X to problem Y in polynomial time, we need to do three things:

1. Describe a polynomial-time algorithm to transform an ***arbitrary*** instance of x of X into a *special* instance y of Y.
2. Prove that if x is a "good" instance of X, then y is a "good" instance of Y.
3. Prove that if y is a "good" instance of Y, then x is a "good" instance of X. (This is usually the part that causes the most trouble.)

Of course, *developing* a correct reduction doesn't mean handling these three tasks one at a time. *First* writing down an algorithm that *seems* to work and *then* trying prove that it *actually* works is rarely successful, especially in time-limited settings like exams. We *must* develop the algorithm, the "if" proof, and the "only if" proof simultaneously.

To quote the late great Ricky Jay:[8] **This is an acquired skill.**

One point that confuses many students is that the reduction algorithm only reduces "one way"—from X to Y—but the correctness proof needs to work "both ways". But the proofs are not actually symmetric. The "if" proof needs to handle *arbitrary* instances of x, but the "only if" only needs to handle the *special* instances of Y produced by the reduction algorithm. Exploiting this asymmetry is actually the key to successfully designing correct reductions.

I find it useful to think in terms of transforming *certificates*—proofs that a given instance is "good"—along with the instances themselves. For example, a certificate for CIRCUITSAT is a set of inputs that turns on the light bulb; a certificate for SAT or 3SAT is a satisfying assignment; a certificate for MAXINDSET is a large independent set. To reduce X to Y, we actually need to design *three* algorithms, one for each of the following tasks:

- Transform an arbitrary instance x of X into a special instance y of Y in polynomial time.
- Transform an arbitrary certificate for x into a certificate for y, and
- Transform an arbitrary certificate for y into a certificate for x.

The second and third tasks refer to the input and output of the first algorithm. The *certificate* transformation needs to be reversible, not the *instance* transformation. We never have to transform instances of Y, and we don't need to think about *arbitrary* instances of Y at all. Only the first algorithm needs to run in polynomial time (although in practice, the second and third algorithms are almost always simpler than the first).

[8] from his 1996 off-Broadway show *Ricky Jay and his 52 Assistants*

For example, our reduction from CircuitSat to 3Sat consists of three algorithms:

- The first transforms an *arbitrary* boolean circuit K into a special 3CNF boolean formula Φ_3, in polynomial time. (Encode each wire as a variable and each gate as a sub-formula, and then expand each sub-formula into 3CNF.)
- The second transforms an *arbitrary* satisfying input for K into a satisfying assignment for Φ_3. (Trace the input through the circuit, transfer values from each wire to the corresponding variable, and give any additional variables arbitrary values.)
- The third transforms an *arbitrary* satisfying assignment for Φ_3 into a satisfying input for K. (Transfer values from each wire variable in Φ_3 to the corresponding wire in K.)

The reduction works because it *encodes* any boolean circuit K into a highly structured 3CNF formula Φ_3. The specific structure of Φ_3 restricts *how* it can be satisfied; every satisfying assignment for Φ_3 must "come from" some satisfying input for K. We don't have to think about arbitrary 3CNF formulas at all.

Similarly, our reduction from 3Sat to MaxIndSet consists of three algorithms:

- The first transforms an *arbitrary* 3CNF formula Φ into a special graph G and a specific integer k, in polynomial time.
- The second transforms an *arbitrary* satisfying assignment for Φ into an independent set in G of size k.
- The third transforms an *arbitrary* independent set in G of size k into a satisfying assignment for Φ.

Again, our first transformation *encodes* the input formula Φ into a highly structured graph G and a specific integer k. The structure of G ensures that every independent set of size k "comes from" a satisfying assignment for Φ. We don't have to think about arbitrary graphs or arbitrary independent set sizes at all.

12.9 Clique and Vertex Cover (from Independent Set)

A ***clique*** is another name for a complete graph, that is, a graph where every pair of vertices is connected by an edge. The MaxClique problem asks for the number of nodes in its largest complete subgraph in a given graph. A ***vertex cover*** of a graph is a set of vertices that touches every edge in the graph. The MinVertexCover problem asks for the size of the smallest vertex cover in a given graph.

We can prove that MaxClique is NP-hard using the following easy reduction from MaxIndSet. Any graph G has an *edge-complement* $\overline{G}$ with the same vertices,

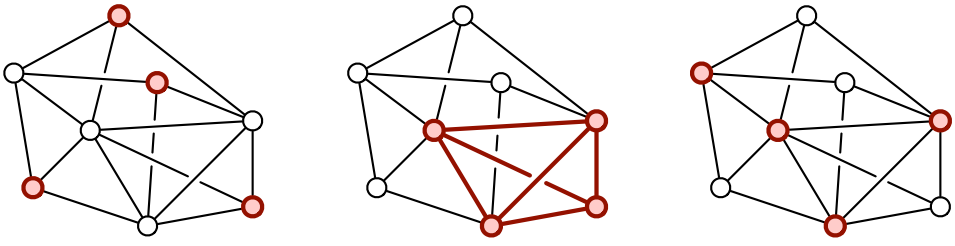


Figure 12.7. A graph whose largest independent set, largest clique, and smallest vertex cover all have size 4.

but with exactly the opposite set of edges—uv is an edge in $\overline{G}$ if and only if uv is *not* an edge in G. A set of vertices is independent in G if and only if the same vertices define a clique in $\overline{G}$. Thus, the largest independent in G has the same vertices (and thus the same size) as the largest clique in the complement of G.

The proof that MINVERTEXCOVER is NP-hard is even simpler, because it relies on the following easy observation: I is an independent set in a graph $G = (V, E)$ if and only if its complement $V \setminus I$ is a vertex cover of the same graph G. Thus, the *largest* independent set in any graph is just the complement of the *smallest* vertex cover of the same graph! Thus, if the smallest vertex cover in an n-vertex graph has size k, then the largest independent set has size $n - k$.

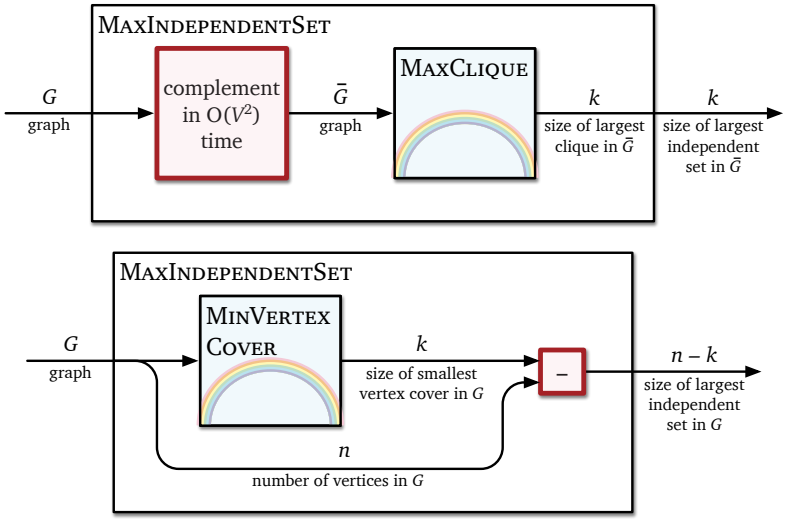


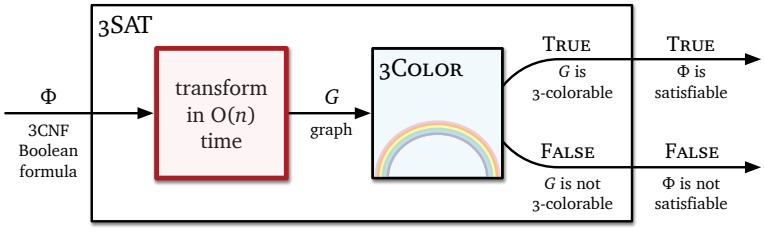
Figure 12.8. Easy reductions from MAXINDSET to MAXCLIQUE and MINVERTEXCOVER.

12.10 Graph Coloring (from 3SAT)

A ***proper k-coloring*** of a graph $G = (V, E)$ is a function $C : V \to \{1, 2, \ldots, k\}$ that assigns one of k "colors" to each vertex, so that every edge has two different

colors at its endpoints. (The "colors" are really just arbitrary labels, which for simplicity we represent by small positive integers, rather than electromagnetic frequencies, CMYK vectors, or Pantone numbers, for example.) The graph coloring problem asks for the smallest possible number of colors in a legal coloring of a given graph.

To prove that graph coloring is NP-hard, it suffices to consider the decision problem ***3Color***: Given a graph, does it have a proper 3-coloring? We prove 3Color is NP-hard using a reduction from 3Sat. (Why 3Sat? Because it has a 3 in it. You probably think I'm joking, but I'm not.) Given a 3CNF formula Φ, we construct a graph G that is 3-colorable if and only if Φ is satisfiable, as suggested by the usual diagram.



We describe the reduction using a standard strategy of decomposing the output graph G into ***gadgets***, subgraphs that enforce various semantics of the input formula Φ. Decomposing reductions into separate gadgets is not only helpful for understanding existing reductions and proving them correct, but for designing new NP-hardness reductions. Our formula-to-graph reduction uses three types of gadgets:

- There is a single *truth gadget*: a triangle with three vertices T, F, and X, which intuitively stand for True, False, and Other. Since these vertices are all connected, they must have different colors in any 3-coloring. For the sake of convenience, we will *name* those colors True, False, and Other. Thus, when we say that a node is colored True, we mean that it has the same color as vertex T.
- For each variable a, the graph contains a *variable gadget*, which is a triangle joining two new nodes labeled a and $\overline{a}$ to node X in the truth gadget. Node a must be colored either True or False, and therefore node $\overline{a}$ must be colored either False or True, respectively.

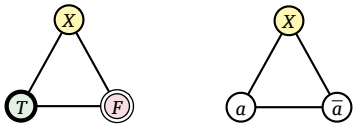


Figure 12.9. The truth gadget and a variable gadget for a.

- Finally, for each clause in Φ, the graph contains a *clause gadget*. Each clause gadget joins three literal nodes (from the corresponding variable gadgets) to node T (from the truth gadget) using five new unlabeled nodes and ten edges, as shown below.

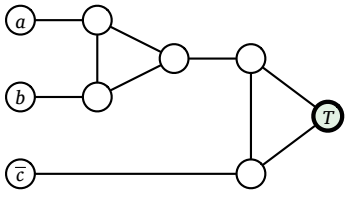


Figure 12.10. A clause gadget for $(a \vee b \vee \bar{c})$.

In effect, each triangle in the clause gadget behaves like a "majority gate". In any valid 3-coloring, if the two vertices to the left of the triangle have the same color, the rightmost vertex of the triangle must have the same color; on the other hand, if the two left vertices have different colors, the color of the right vertex can be chosen arbitrarily.

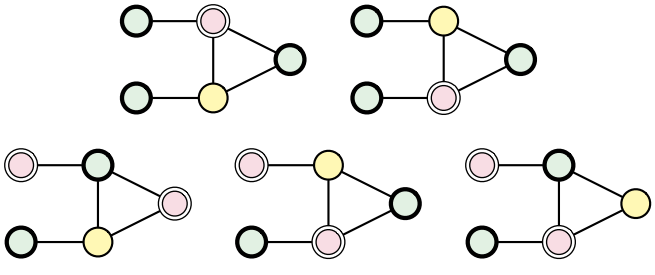


Figure 12.11. All valid 3-colorings of a "half-gadget", up to permutations of the colors

It follows that there is no valid 3-coloring of a clause gadget where all three literal nodes are colored FALSE. On the other hand, any coloring of the literal nodes with more than one color can be extended to a valid 3-coloring of the clause gadget. The variable gadgets force each literal node to be colored either TRUE or FALSE; thus, in any valid 3-coloring of the clause gadget, at least one literal node is colored TRUE.

The final graph G contains exactly *one* node T, exactly *one* node F, and exactly *two* nodes a and $\bar{a}$ for each variable. For example, the formula $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$ that we previously used to illustrate the MAXCLIQUE reduction would be transformed into the graph shown in Figure 12.12. The 3-coloring is one of several that correspond to the satisfying assignment $a = c = \text{TRUE}$, $b = d = \text{FALSE}$.

We've already done most of the work for a proof of correctness. If the formula is satisfiable, then we can color the literal nodes according to any satisfying assignment, and then (because each clause is satisfied) extend the coloring