The wonderful thing about standards is that
there are so many of them to choose from.

— Real Admiral Grace Murray Hopper

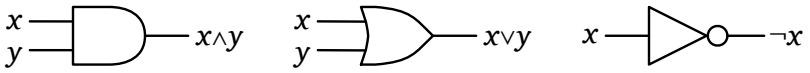
If a problem has no solution, it may not be a problem, but a fact —
not to be solved, but to be coped with over time.

— Shimon Peres

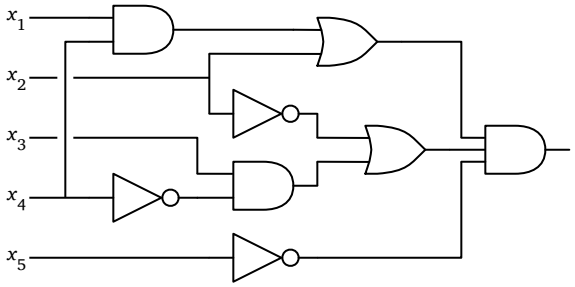
30 NP-Hard Problems

30.1 A Game You Can't Win

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 billion dollars; if you answer incorrectly, or if you die without answering at all, he will take your soul.



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



A boolean circuit. inputs enter from the left, and the output leaves to the right.

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 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.) Thus, the only way to *provably* answer the Adversary's question correctly is to try all 2^n possible settings of the switches. 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, "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 paper with a circuit diagram drawn on it. "Here

are the complete plans for the circuit inside the box. Feel free to open the box and poke around to make sure the plans are correct. Or build your own box following these plans. Or write a computer program to simulate the box. Whatever you like. If you discover that the plans don't match the actual circuit in the box, you win the billion dollars." A few spot checks convince you that the plans have no obvious flaws; cheating appears to be impossible.

But you should still decline the Adversary's bet. 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, but this requires exponential time. On the other hand, nobody has ever actually *proved* that this is the best we can do; maybe there's a clever algorithm that just hasn't been discovered yet!

30.2 P versus NP

A minimal requirement for an algorithm to be considered "efficient" is that its running time is polynomial: $O(n^c)$ 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 the answer is YES, then any set of m input values that produces TRUE output is a proof of this fact; 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.

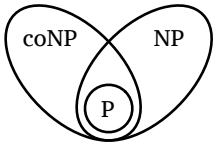
Every decision problem in P is also in NP. If a problem is in P, we can verify YES answers in polynomial time recomputing the answer from scratch! Similarly, every problem in P is also in co-NP.

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 this class and others have (I hope) convinced you that problems can be incredibly hard to solve, even when the solutions are obvious in retrospect. It's completely obvious; *of course* solving problems from scratch

[1] This notion of efficiency was independently formalized by Alan Cobham (The intrinsic computational difficulty of functions. *Logic, Methodology, and Philosophy of Science (Proc. Int. Congress)*, 24–30, 1965), Jack Edmonds (Paths, trees, and flowers. *Canadian Journal of Mathematics* 17:449–467, 1965), and Michael Rabin (Mathematical theory of automata. *Proceedings of the 19th ACM Symposium in Applied Mathematics*, 153–175, 1966), although similar notions were considered more than a decade earlier by Kurt Gödel and John von Neumann.

is harder than just checking that a solution is correct. But nobody knows how to prove it! 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 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 $\neq$ co-NP, but nobody knows how to prove it.



What we *think* the world looks like.

30.3 NP-hard, NP-easy, and NP-complete

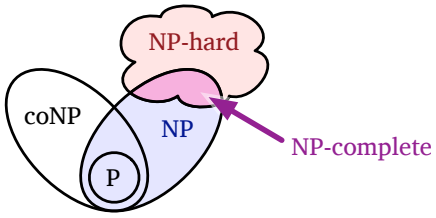
A problem Π is ***NP-hard*** if a polynomial-time algorithm for Π would imply a polynomial-time algorithm for *every problem in NP*. In other words:

Π is NP-hard $\iff$ If Π can be solved in polynomial time, then P=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 any problem in NP.

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'repretty 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! Likewise, if a problem is NP-hard, no one in their right mind should believe it can be solved in polynomial time.

Finally, a problem is ***NP-complete*** if it is both NP-hard and an element of NP (or 'NP-easy'). NP-complete problems are the hardest problems in NP. If anyone finds a polynomial-time algorithm for even one NP-complete problem, then that would 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.



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

It is not immediately clear that *any* decision problems are NP-hard or NP-complete. NP-hardness is already a lot to demand of a problem; insisting that the problem also have a nondeterministic polynomial-time algorithm seems almost completely unreasonable. The following remarkable theorem was first published by Steve Cook in 1971 and independently by Leonid Levin in 1973.[2] I won't even sketch the proof, since I've been (deliberately) vague about the definitions.

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

*30.4 Formal Definition (*HC SVNT DRACONES*)

Formally, the complexity classes P, NP, and co-NP are defined in terms of *languages* and *Turing machines*. A language is just a set of strings over some finite alphabet Σ; without loss of generality, we can assume that $\Sigma = \{0, 1\}$. P is the set of languages that can be recognized in polynomial time by a deterministic single-tape Turing machine. It is elementary but *extremely* tedious to prove that any algorithm that can be executed on a random-access machine[3] in time $T(n)$ can be simulated on a single-tape Turing machine in time $O(T(n)^2)$. This simulation result, which we'll simply take on faith, 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.

A problem Π is formally 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. Polynomial-time Turing reductions are also called *oracle reductions* or *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 $\Pi' \subseteq \Sigma^*$ to another language $\Pi \subseteq \Sigma^*$ is an function $f : \Sigma^* \to \Sigma^*$ such that $x \in \Pi'$ if and only if $f(x) \in \Pi$. Then we could define a *language* Π to be NP-hard if and only if, for any language $\Pi' \in$ NP, there is a many-one reduction from Π' to Π that can be computed in polynomial time.

Every Karp reduction "is" a Cook reduction, but not vice versa. Specifically, any Karp reduction from Π to 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.

[2] 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.

[3] Random-access machines are a model of computation that more faithfully models physical computers. A random-access machine has unbounded random-access memory, modeled as an 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 **`IFZGOTO i,ℓ`** (meaning "if $M[i] = 0$, go to line ℓ"). In practice, RAM algorithms can be faithfully described using higher-level pseudocode, as long as we're careful about arithmetic precision.

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.

30.5 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.[4]

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 mythical algorithm for *your* problem as a subroutine. The essential logic is a proof by contradiction. Your reduction shows implies that if your problem were easy, then the other problem would be easy, too. Equivalently, since you know the other problem is hard, your problem must also be hard.

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

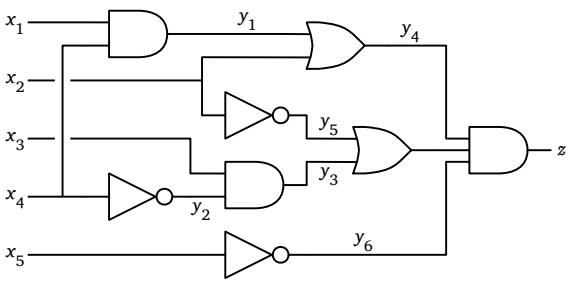
$$(a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee \overline{(\bar{a} \Rightarrow d)} \vee (c \neq a \wedge b)),$$

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

To show 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 circuit satisfiability, so let's start there. Given a boolean circuit, we can transform it into a boolean formula by creating new output variables for each gate, and then just writing down the list of gates separated by ANDS. For example, we can transform the example circuit into a formula as follows:

Now the original circuit is satisfiable if and only if the resulting formula is satisfiable. Given a satisfying input to the circuit, we can get a satisfying assignment for the formula by computing the output of every gate. Given a satisfying assignment for the formula, we can get a satisfying input the the circuit by just ignoring the internal gate variables y_i and the output variable z.

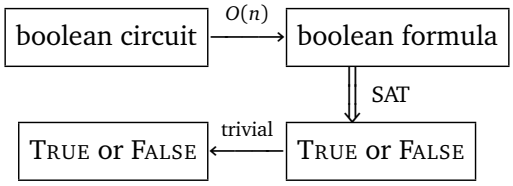
[4] If a player lands on an available property and declines (or is unable) to buy it, that property is immediately auctioned off to the highest bidder; the player who originally declined the property may bid, and bids may be arbitrarily higher or lower than the list price. Players in Jail can still buy 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 sell all three hotels in the group. Players can sell/exchange undeveloped properties, but not buildings or cash. A player landing on Free Parking does not win anything. A player landing on Go gets $200, no more. Railroads are not magic transporters. Finally, Jeff *always* gets the car.



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

A boolean circuit with gate variables added, and an equivalent boolean formula.

We can transform any boolean circuit into a formula in linear time using depth-first search, and the size of the resulting formula is only a constant factor larger than the size of the circuit. Thus, we have a polynomial-time reduction from circuit satisfiability to SAT:



$$T_{\text{CSAT}}(n) \le O(n) + T_{\text{SAT}}(O(n)) \quad \Longrightarrow \quad T_{\text{SAT}}(n) \ge T_{\text{CSAT}}(\Omega(n)) - O(n)$$

The reduction implies that if we had a polynomial-time algorithm for SAT, then we'd have a polynomial-time algorithm for circuit satisfiability, which would imply that P=NP. So SAT is NP-hard.

To prove that a boolean formula is satisfiable, we only have to specify an assignment to the variables that makes the formula TRUE. We can check the proof in linear time just by reading the formula from left to right, evaluating as we go. So SAT is also in NP, and thus is actually NP-complete.

30.6 3SAT (from SAT)

A special case of SAT that is particularly useful in proving NP-hardness results is called *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, with a boolean circuit. We perform the reduction in several stages.

1. *Make sure every* AND *and* OR *gate has only two inputs.* If any gate has $k > 2$ inputs, replace it with a binary tree of $k - 1$ two-input gates.

2. *Write down the circuit as a formula, with one clause per gate.* This is just the previous reduction.

3. *Change every gate clause into a CNF formula.* There are only three types of clauses, one for each type of gate:

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

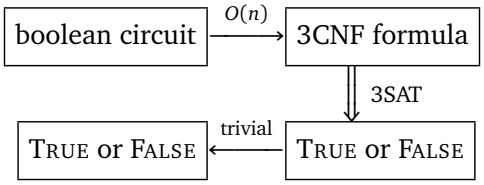
4. *Make sure every clause has exactly three literals.* Introduce new variables into each one- and two-literal clause, and expand it into two clauses as follows:

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

For example, if we start with the same example circuit we used earlier, we obtain the following 3CNF formula. Although this may look a lot more ugly and complicated than the original circuit at first glance, it's actually only a constant factor larger—every 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^{573}) of the circuit size, we would still have a valid reduction.

$$\begin{aligned}
&(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_8 \vee \overline{z_{17}}) \wedge (\overline{y_9} \vee y_6 \vee z_{18}) \wedge (\overline{y_9} \vee y_6 \vee \overline{z_{18}}) \\
&\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{aligned}$$

This process transforms the circuit into an equivalent 3CNF formula; the output formula is satisfiable if and only if the input circuit is satisfiable. As with the more general SAT problem, the formula is only a constant factor larger than any reasonable description of the original circuit, and the reduction can be carried out in polynomial time. Thus, we have a polynomial-time reduction from circuit satisfiability to 3SAT:



$$T_{\text{CSAT}}(n) \le O(n) + T_{\text{3SAT}}(O(n)) \quad \Longrightarrow \quad T_{\text{3SAT}}(n) \ge T_{\text{CSAT}}(\Omega(n)) - O(n)$$

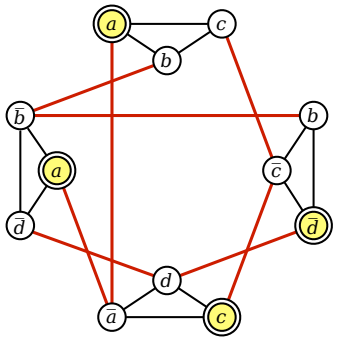
We conclude 3SAT is NP-hard. And because 3SAT is a special case of SAT, it is also in NP. Therefore, 3SAT is NP-complete.

30.7 Maximum Independent Set (from 3SAT)

For the next few problems we consider, the input is a simple, unweighted 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'll prove that MAXINDSET is NP-hard (but not NP-complete, since it isn't a decision problem) using a reduction from 3SAT. I'll describe a reduction from a 3CNF formula into a graph that has an independent set of a certain size if and only if the formula is satisfiable. The graph has one node for each instance of each literal in the formula. Two nodes are connected by an edge if (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 following graph.



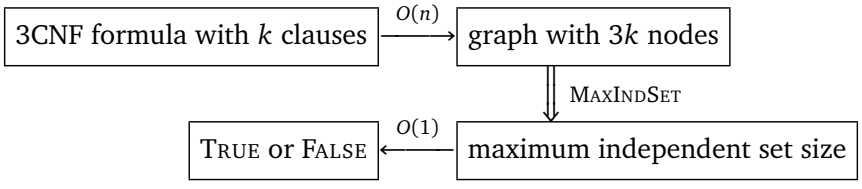
A graph derived from a 3CNF formula, and an independent set of size 4.
Black edges join literals from the same clause; red (heavier) edges join contradictory literals.

Now suppose the original formula had k clauses. Then I claim that the formula is satisfiable if and only if the graph has an independent set of size k.

1. **independent set $\Longrightarrow$ satisfying assignment:** If the graph has an independent set of k vertices, then each vertex must come from a different clause. To obtain a satisfying assignment, we assign the value TRUE to each literal in the independent set. Since contradictory literals are connected by edges, this assignment is consistent. There may be variables that have no literal in the independent set; we can set these to any value we like. The resulting assignment satisfies the original 3CNF formula.

2. **satisfying assignment $\Longrightarrow$ independent set:** If we have a satisfying assignment, then we can choose one literal in each clause that is TRUE. Those literals form an independent set in the graph.

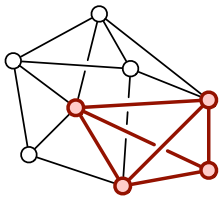
Thus, the reduction is correct. Since the reduction from 3CNF formula to graph takes polynomial time, we conclude that MAXINDSET is NP-hard. Here's a diagram of the reduction:



$$T_{3SAT}(n) \le O(n) + T_{MAXINDSET}(O(n)) \quad \Longrightarrow \quad T_{MAXINDSET}(n) \ge T_{3SAT}(\Omega(n)) - O(n)$$

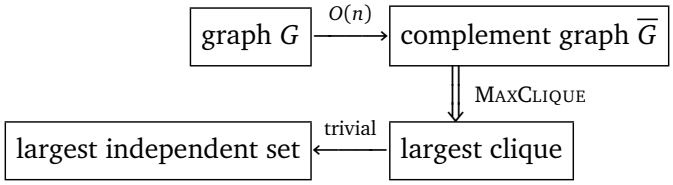
30.8 Clique (from Independent Set)

A *clique* is another name for a complete graph, that is, a graph where every pair of vertices is connected by an edge. The *maximum clique size* problem, or simply MAXCLIQUE, is to compute, given a graph, the number of nodes in its largest complete subgraph.



A graph with maximum clique size 4.

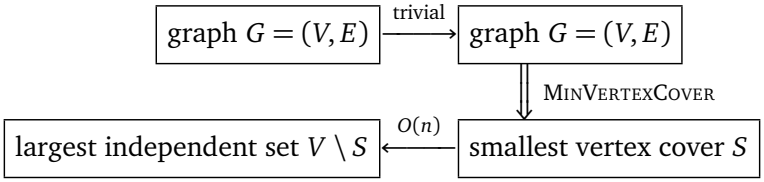
There is an easy proof that MAXCLIQUE is NP-hard, using a reduction from MAXINDSET. Any graph G has an *edge-complement* $\overline{G}$ with the same vertices, but with exactly the opposite set of edges—(u, v) is an edge in $\overline{G}$ if and only if it is *not* an edge in G. A set of vertices is independent in G if and only if the same vertices define a clique in $\overline{G}$. Thus, we can compute the largest independent in a graph simply by computing the largest clique in the complement of the graph.



30.9 Vertex Cover (from Independent Set)

A *vertex cover* of a graph is a set of vertices that touches every edge in the graph. The MINVERTEXCOVER problem is to find the smallest vertex cover in a given graph.

Again, the proof of NP-hardness is simple, and relies on just one fact: If I is an independent set in a graph $G = (V, E)$, then $V \setminus I$ is a vertex cover. Thus, to find the *largest* independent set, we just need to find the vertices that aren't in the *smallest* vertex cover of the same graph.

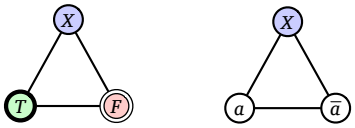


30.10 Graph Coloring (from 3SAT)

A *k-coloring* of a graph is a map $C : V \to \{1, 2, \ldots, k\}$ that assigns one of k 'colors' to each vertex, so that every edge has two different colors at its endpoints. The graph coloring problem is to find the smallest possible number of colors in a legal coloring. To show that this problem is NP-hard, it's enough to consider the special case 3COLORABLE: Given a graph, does it have a 3-coloring?

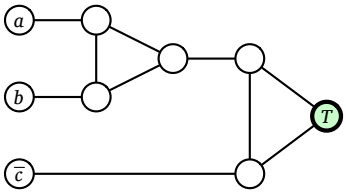
To prove that 3COLORABLE is NP-hard, we use a reduction from 3SAT. Given a 3CNF formula Φ, we produce a graph G_Φ as follows. The graph consists of a *truth* gadget, one *variable* gadget for each variable in the formula, and one *clause* gadget for each clause in the formula.

- The truth gadget is just 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, all we mean is that it must be colored the same as the node T.
- The variable gadget for a variable a is also a triangle joining two new nodes labeled a and $\bar{a}$ to node X in the truth gadget. Node a must be colored either TRUE or FALSE, and so node $\bar{a}$ must be colored either FALSE or TRUE, respectively.



The truth gadget and a variable gadget for a.

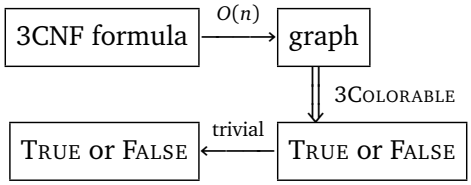
- Finally, each clause gadget joins three literal nodes to node T in the truth gadget using five new unlabeled nodes and ten edges; see the figure below. A straightforward case analysis implies that if all three literal nodes in the clause gadget are colored FALSE, then some edge in the gadget must be monochromatic. Since the variable gadgets force each literal node to be colored either TRUE or FALSE, in any valid 3-coloring, at least one of the three literal nodes is colored TRUE. On the other hand, for any coloring of the literal nodes where at least one literal node is colored TRUE, there is a valid 3-coloring of the clause gadget.



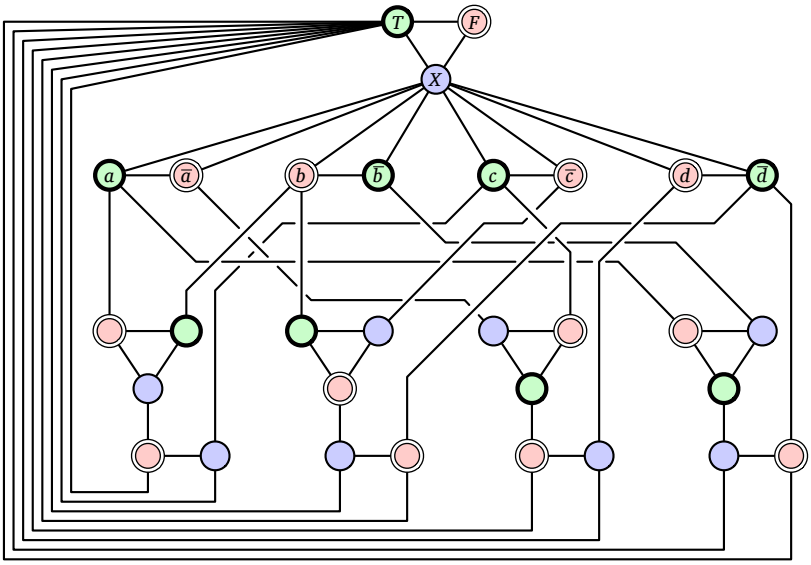
A clause gadget for $(a \vee b \vee \bar{c})$.

The final graph G_Φ 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 I used to illustrate the MAXCLIQUE reduction would be transformed into the graph shown on the next page. The 3-coloring is one of several that correspond to the satisfying assignment $a = c =$ TRUE, $b = d =$ FALSE.

Now the proof of correctness is just brute force case analysis. If the graph is 3-colorable, then we can extract a satisfying assignment from any 3-coloring—at least one of the three literal nodes in every clause gadget is colored TRUE. Conversely, if the formula is satisfiable, then we can color the graph according to any satisfying assignment.



We can easily verify that a graph has been correctly 3-colored in linear time: just compare the endpoints of every edge. Thus, 3COLORING is in NP, and therefore NP-complete. Moreover, since 3COLORING is a special case of the more general graph coloring problem—What is the minimum number of colors?—the more problem is also NP-hard, but *not* NP-complete, because it's not a decision problem.



A 3-colorable graph derived from the satisfiable 3CNF formula
$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$

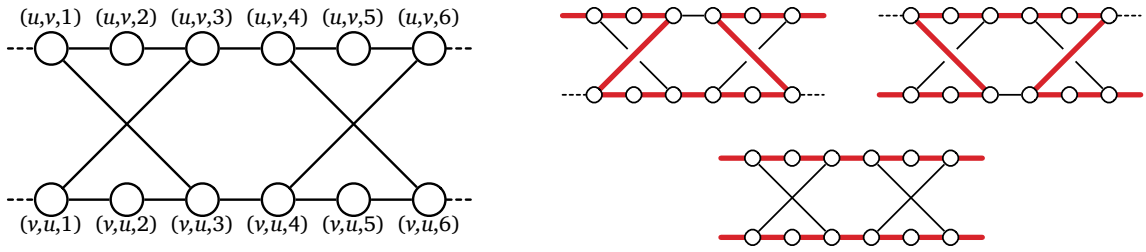
30.11 Undirected Hamiltonian Cycle (from Vertex Cover)

Decompose into a reduction from vertex cover to *directed* Hamiltonian cycle and a reduction from directed Hamiltonian cycle to undirected Hamiltonian cycle. This note needs more graph→graph reductions with simple gadgets. Steiner tree from vertex cover?

A ***Hamiltonian cycle*** in a graph is a cycle that visits every vertex exactly once. This is very different from an *Eulerian cycle*, which is actually a closed *walk* that traverses every *edge* exactly once. Eulerian cycles are easy to find and construct in linear time using a variant of depth-first search. Determining whether a graph contains a Hamiltonian cycle, on the other hand, is NP-hard.

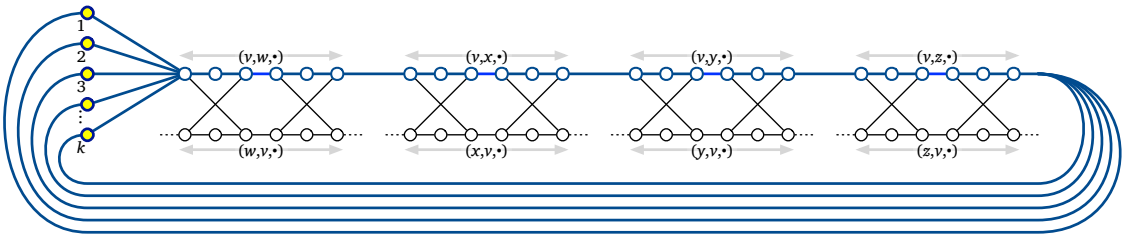
To prove that finding a Hamiltonian cycle in an undirected graph is NP-hard, we describe a reduction from the vertex cover problem. Given a graph G and an integer k, we need to transform it into another graph G', such that G' has a Hamiltonian cycle if and only if G has a vertex cover of size k. As usual, our transformation uses several gadgets.

- For each edge uv in G, we have an *edge gadget* in G' consisting of twelve vertices and fourteen edges, as shown below. The four corner vertices $(u, v, 1)$, $(u, v, 6)$, $(v, u, 1)$, and $(v, u, 6)$ each have an edge leaving the gadget. A Hamiltonian cycle can only pass through an edge gadget in only three ways. Eventually, these options will correspond to one or both vertices u and v being in the vertex cover.



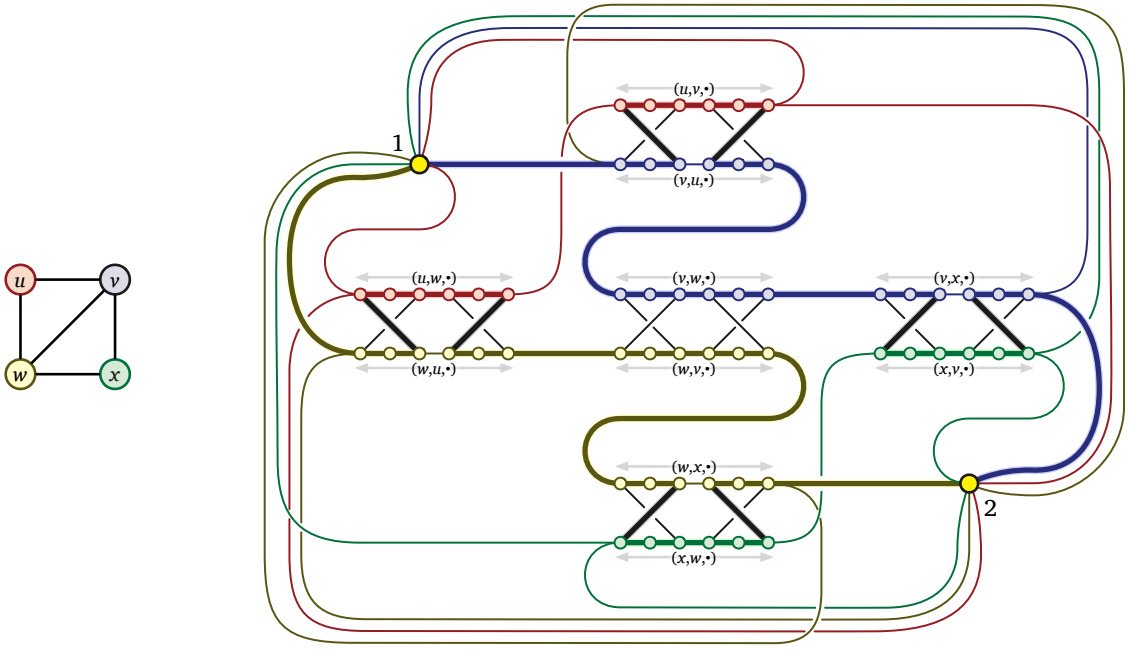
An edge gadget for (u, v) and the only possible Hamiltonian paths through it.

- G' also contains k *cover vertices*, simply numbered 1 through k.
- Finally, for each vertex u in G, we string together all the edge gadgets for edges (u, v) into a single *vertex chain*, and then connect the ends of the chain to all the cover vertices. Specifically, suppose vertex u has d neighbors $v_1, v_2, \ldots, v_d$. Then G' has $d - 1$ edges between $(u, v_i, 6)$ and $(u, v_{i+1}, 1)$, plus k edges between the cover vertices and $(u, v_1, 1)$, and finally k edges between the cover vertices and $(u, v_d, 6)$.



The vertex chain for v: all edge gadgets involving v are strung together and joined to the k cover vertices.

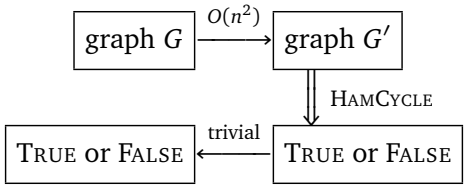
An example of our complete transformation is shown below.



The original graph G with vertex cover $\{v, w\}$, and the transformed graph G' with a corresponding Hamiltonian cycle. Vertex chains are colored to match their corresponding vertices.

It is now straightforward but tedious to prove that if $\{v_1, v_2, \ldots, v_k\}$ is a vertex cover of G, then G' has a Hamiltonian cycle—start at cover vertex 1, through traverse the vertex chain for v_1, then visit cover vertex 2, then traverse the vertex chain for v_2, and so forth, eventually returning to cover vertex 1. Conversely, any Hamiltonian cycle in G' alternates between cover vertices and vertex chains, and the vertex chains correspond to the k vertices in a vertex cover of G. (This is a little harder to prove.) Thus, G has a vertex cover of size k if and only if G' has a Hamiltonian cycle.

The transformation from G to G' takes at most $O(n^2)$ time; we conclude that the Hamiltonian cycle problem is NP-hard. Moreover, since we can easily verify a Hamiltonian cycle in linear time, the Hamiltonian cycle problem is in NP, and therefore is NP-complete.



A closely related problem to Hamiltonian cycles is the famous *traveling salesman problem*—Given a *weighted* graph G, find the shortest cycle that visits every vertex. Finding the shortest cycle is obviously harder than determining if a cycle exists at all, so the traveling salesman problem is also NP-hard.

30.12 Directed Hamiltonian Cycle (from 3SAT)

The previous reduction showed that finding Hamiltonian cycles in *undirected* graphs is NP-hard; it is natural to conjecture the problem is still hard when the input graph is directed. In fact, there are easy polynomial-time reductions between the directed and undirected versions of the problem; we leave the details of this double reduction as an exercise. However, one can also prove directly that the directed version is NP-hard using the following reduction from 3SAT, which is quite different from the undirected reduction.

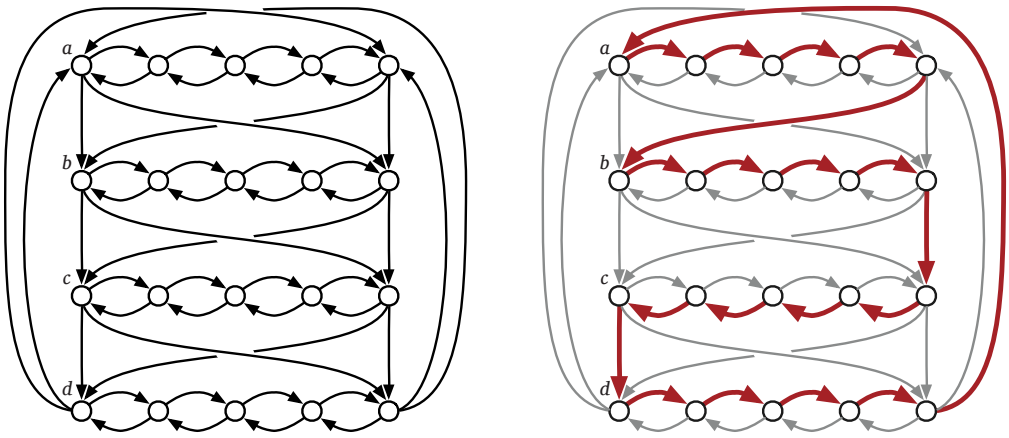
Given a 3CNF formula Φ with n variables $x_1, x_2, \ldots, x_n$ and k clauses $c_1, c_2, \ldots, c_k$, we construct a directed graph $G(\Phi)$ as follows. For each variable x_i, we construct a *variable gadget* consisting of a doubly-linked list of $k+1$ vertices $(i,0),(i,1),\ldots,(i,k)$, connected by edges $(i,j-1)\to(i,j)$ and $(i,j)\to(i,j-1)$ for each index j. Then we connect successive variable gadgets by adding edges

$$(i,0)\to(i+1,0) \qquad (i,k)\to(i+1,0) \qquad (i,0)\to(i+1,k) \qquad (i,k)\to(i+1,k)$$

for each index i; we also connect the first and last variable gadgets with the edges

$$(n,0)\to(1,0) \qquad (n,k)\to(1,0) \qquad (n,0)\to(1,k) \qquad (n,k)\to(1,k).$$

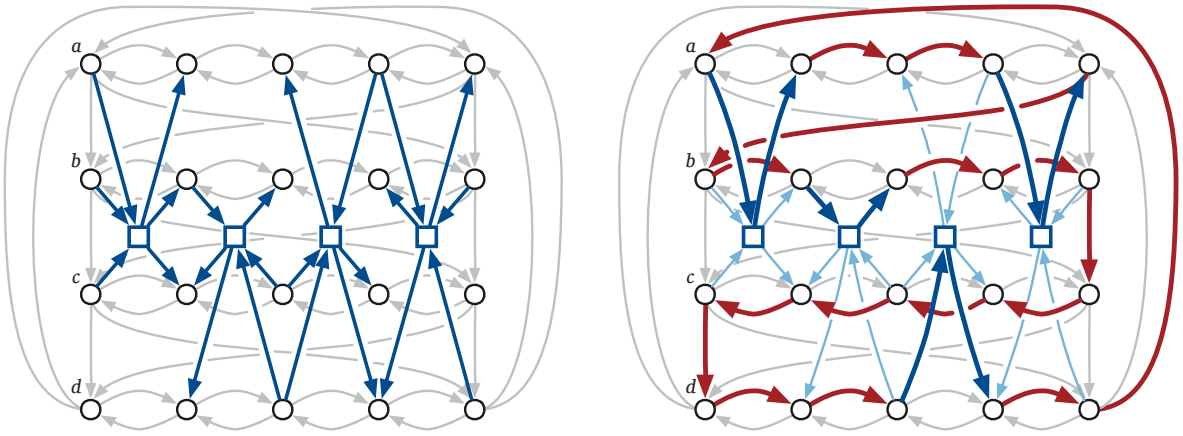
The resulting graph $G(\Phi)$ has exactly 2^n Hamiltonian cycles, one for each assignment of boolean values to the n variables of Φ. Specifically, for each i, we traverse the ith variable gadget from left to right if $x_i = \text{TRUE}$ and right to left if $x_i = \text{FALSE}$.



Left: Variable gadgets for a formulawith four variables a, b, c, d and four clauses.
Right: A Hamiltonian cycle corresponding to the assignment $a = b = d = \text{TRUE}$, $c = \text{FALSE}$.

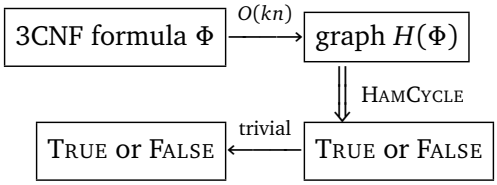
Now we extend $G(\Phi)$ to a larger graph $H(\Phi)$ by adding a *clause vertex* $[j]$ for each clause c_j, connected to the variable gadgets by six edges. Specifically, for each positive literal x_i in c_j, we add the

edges $(i, j-1) \to [j] \to (i, j)$, and for each negative literal $\bar{x}_i$ in c_j, we add the edges $(i, j) \to [j] \to (i, j-1)$. The connections to the clause vertices guarantee that a Hamiltonian cycle in $G(\Phi)$ can be extended to a Hamiltonian cycle in $H(\Phi)$ if and only if the corresponding variable assignment satisfies Φ. Exhaustive case analysis now implies that $H(\Phi)$ has a Hamiltonian cycle if and only if Φ is satisfiable.



Left: Clause vertices in $H(\Phi)$, where $\Phi = (a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$.
Right: The Hamiltonian cycle in $H(\Phi)$ corresponding to the satisfying assignment $a = b = d = \text{TRUE}$, $c = \text{FALSE}$.

Transforming the formula Φ into the graph $H(\Phi)$ takes $O(kn)$ time, which is at most quadratic in the total length of the formula; we conclude that the directed Hamiltonian cycle problem is NP-hard. Moreover, since we can easily verify a Hamiltonian cycle in linear time, the directed Hamiltonian cycle problem is in NP, and therefore is NP-complete.



30.13 Subset Sum (from Vertex Cover)

The next problem that we prove NP-hard is the SUBSETSUM problem considered in the very first lecture on recursion: Given a set X of positive integers and an integer t, determine whether X has a subset whose elements sum to t.

To prove this problem is NP-hard, we once again reduce from VERTEXCOVER. Given a graph G and an integer k, we compute a set X of integer and an integer t, such that X has a subset that sums to t if and only if G has an vertex cover of size k. Our transformation uses just two 'gadgets', which are *integers* representing vertices and edges in G.

Number the *edges* of G arbitrarily from 0 to $m-1$. Our set X contains the integer $b_i := 4^i$ for each edge i, and the integer

$$a_v := 4^m + \sum_{i \in \Delta(v)} 4^i$$

for each vertex v, where $\Delta(v)$ is the set of edges that have v has an endpoint. Alternately, we can think of each integer in X as an $(m+1)$-digit number written in base 4. The mth digit is 1 if the integer represents a vertex, and 0 otherwise; and for each $i < m$, the ith digit is 1 if the integer represents edge i

or one of its endpoints, and 0 otherwise. Finally, we set the target sum

$$t := k \cdot 4^m + \sum_{i=0}^{m-1} 2 \cdot 4^i.$$

Now let's prove that the reduction is correct. First, suppose there is a vertex cover of size k in the original graph G. Consider the subset $X_C \subseteq X$ that includes a_v for every vertex v in the vertex cover, and b_i for every edge i that has *exactly one* vertex in the cover. The sum of these integers, written in base 4, has a 2 in each of the first m digits; in the most significant digit, we are summing exactly k 1's. Thus, the sum of the elements of X_C is exactly t.

On the other hand, suppose there is a subset $X' \subseteq X$ that sums to t. Specifically, we must have

$$\sum_{v \in V'} a_v + \sum_{i \in E'} b_i = t$$

for some subsets $V' \subseteq V$ and $E' \subseteq E$. Again, if we sum these base-4 numbers, there are no carries in the first m digits, because for each i there are only three numbers in X whose ith digit is 1. Each edge number b_i contributes only one 1 to the ith digit of the sum, but the ith digit of t is 2. Thus, for each edge in G, at least one of its endpoints must be in V'. In other words, V is a vertex cover. On the other hand, only vertex numbers are larger than 4^m, and $\lfloor t/4^m \rfloor = k$, so V' has at most k elements. (In fact, it's not hard to see that V' has *exactly* k elements.)

For example, given the four-vertex graph used on the previous page to illustrate the reduction to Hamiltonian cycle, our set X might contain the following base-4 integers:

$$\begin{array}{rr} a_u := 111000_4 = 1344 & b_{uv} := 010000_4 = 256 \\ a_v := 110110_4 = 1300 & b_{uw} := 001000_4 = 64 \\ a_w := 101101_4 = 1105 & b_{vw} := 000100_4 = 16 \\ a_x := 100011_4 = 1029 & b_{vx} := 000010_4 = 4 \\ & b_{wx} := 000001_4 = 1 \end{array}$$

If we are looking for a vertex cover of size 2, our target sum would be $t := 222222_4 = 2730$. Indeed, the vertex cover $\{v, w\}$ corresponds to the subset $\{a_v, a_w, b_{uv}, b_{uw}, b_{vx}, b_{wx}\}$, whose sum is $1300 + 1105 + 256 + 64 + 4 + 1 = 2730$.

The reduction can clearly be performed in polynomial time. Since VERTEXCOVER is NP-hard, it follows that SUBSETSUM is NP-hard.

There is one subtle point that needs to be emphasized here. Way back at the beginning of the semester, we developed a dynamic programming algorithm to solve SUBSETSUM in time $O(nt)$. Isn't this a polynomial-time algorithm? idn't we just prove that P=NP? Hey, where's our million dollars? Alas, life is not so simple. True, the running time is polynomial in n and t, but in order to qualify as a true polynomial-time algorithm, the running time must be a polynomial function *of the size of the input*. The *values* of the elements of X and the target sum t could be exponentially larger than the number of input bits. Indeed, the reduction we just described produces a value of t that is exponentially larger than the size of our original input graph, which would force our dynamic programming algorithm to run in exponential time.

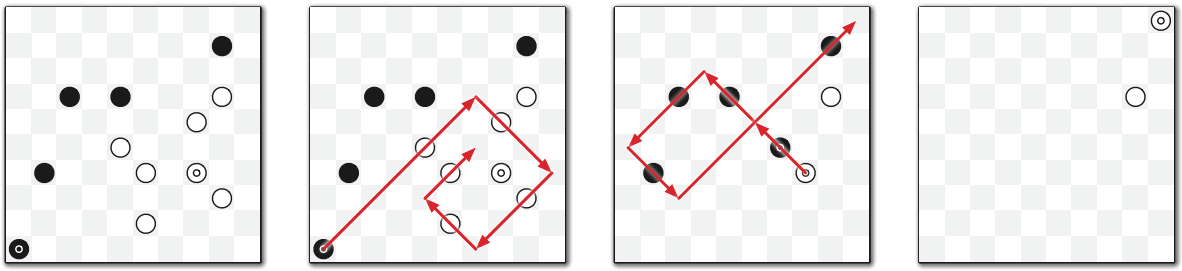
Algorithms like this are said to run in ***pseudo-polynomial time***, and any NP-hard problem with such an algorithm is called ***weakly NP-hard***. Equivalently, a weakly NP-hard problem is one that can be solved in polynomial time when all input numbers are represented in *unary* (as a sum of 1s), but becomes NP-hard when all input numbers are represented in *binary*. If a problem is NP-hard even when all the input numbers are represented in unary, we say that the problem is ***strongly NP-hard***.

30.14 A Frivolous Example

Draughts is a family of board games that have been played for thousands of years. Most American students are familiar with the version called *checkers* or *English draughts*, but the most common variant worldwide, known as ***international draughts*** or ***Polish draughts***, originated in the Netherlands in the 16th century. For a complete set of rules, the reader should consult Wikipedia; here a few important differences from the Anglo-American game:

- **Flying kings:** As in checkers, a piece that ends a move in the row closest to the opponent becomes a *king* and gains the ability to move backward. Unlike in checkers, however, a king in international draughts can move any distance along a diagonal line in a single turn, as long as the intermediate squares are empty or contain exactly one opposing piece (which is captured).
- **Forced maximum capture:** In each turn, the moving player must capture as many opposing pieces as possible. This is distinct from the forced-capture rule in checkers, which requires only that each player must capture if possible, and that a capturing move ends only when the moving piece cannot capture further. In other words, checkers requires capturing a *maximal* set of opposing pieces on each turn; whereas, international draughts requires a *maximum* capture.
- **Capture subtleties:** As in checkers, captured pieces are removed from the board only at the end of the turn. Any piece can be captured at most once. Thus, when an opposing piece is jumped, that piece remains on the board *but cannot be jumped again* until the end of the turn.

For example, in the first position shown below, each circle represents a piece, and doubled circles represent kings. Black must make the indicated move, capturing five white pieces, because it is not possible to capture more than five pieces, and there is no other move that captures five. Black cannot extend his capture further northeast, because the captured White pieces are still on the board.



Two forced(!) moves in international draughts.

The actual game, which is played on a 10×10 board with 20 pieces of each color, is computationally trivial; we can precompute the optimal move for both players in every possible board configuration and hard-code the results into a lookup table of constant size. Sure, it's a *big* constant, but it's still just a constant!

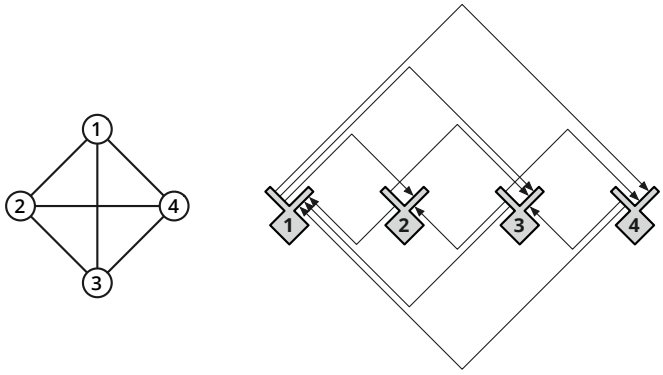
But consider the natural generalization of international draughts to an $n \times n$ board. In this setting, ***finding a legal move is actually NP-hard!*** The following reduction from the Hamiltonian cycle problem in directed graphs was discovered by Bob Hearn in 2010.[5] In most two-player games, finding the *best* move is NP-hard (or worse); this is the only example I know of a game where just *following the rules* is an intractable problem!

Given a graph G with n vertices, we construct a board configuration for international draughts, such that White can capture a certain number of black pieces in a single move if and only if G has

[5]Posted on Theoretical Computer Science Stack Exchange: http://cstheory.stackexchange.com/a/1999/111.

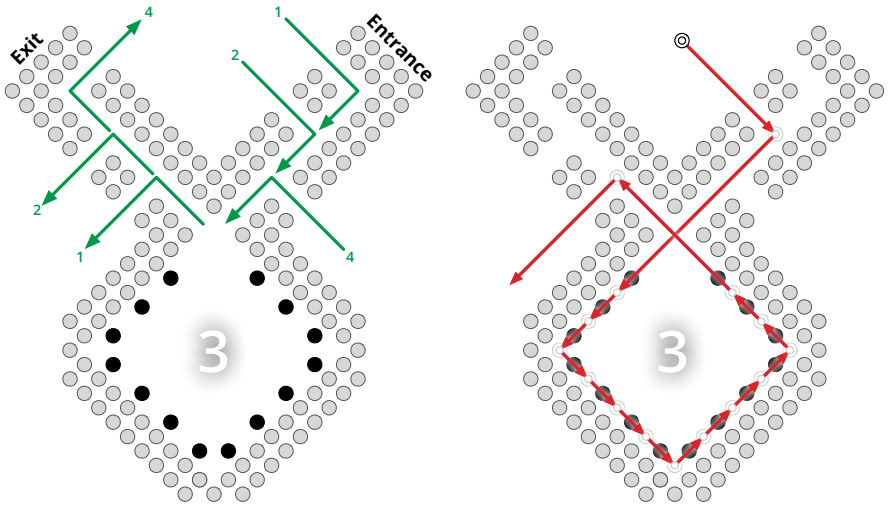
a Hamiltonian cycle. We treat G as a directed graph, with two arcs $u \to v$ and $v \to u$ in place of each undirected edge uv. Number the vertices arbitrarily from 1 to n. The final draughts configuration has several gadgets.

- The vertices of G are represented by rabbit-shaped *vertex gadgets*, which are evenly spaced along a horizontal line. Each arc $i \to j$ is represented by a path of two diagonal line segments from the "right ear" of vertex gadget i to the "left ear" of vertex gadget j. The path for arc $i \to j$ is located above the vertex gadgets if $i < j$, and below the vertex gadgets if $i > j$.



A high level view of the reduction from Hamiltonian cycle to international draughts.

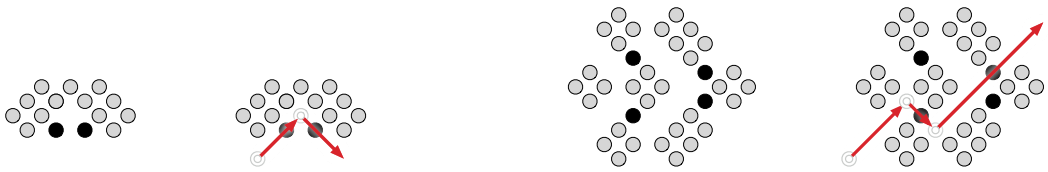
- The bulk of each vertex gadget is a diamond-shaped region called a *vault*. The walls of the vault are composed of two solid layers of black pieces, which cannot be captured; these pieces are drawn as gray circles in the figures. There are N capturable black pieces inside each vault, for some large integer N to be determined later. A white king can enter the vault through the "right ear", capture every internal piece, and then exit through the "left ear". Both ears are hallways, again with walls two pieces thick, with gaps where the arc paths end to allow the white king to enter and leave. The lengths of the "ears" can be adjusted easily to align with the other gadgets.



Left: A vertex gadget. Right: A white king emptying the vault.
Gray circles are black pieces that cannot be captured.

- For each arc $i \to j$, we have a *corner gadget*, which allows a white king leaving vertex gadget i to be redirected to vertex gadget j.

- Finally, wherever two arc paths cross, we have a *crossing gadget*; these gadgets allow the white king to traverse either arc path, but forbid switching from one arc path to the other.



A corner gadget and a crossing gadget.

A single white king starts at the bottom corner of one of the vaults. In any legal move, this king must alternate between traversing entire arc paths and clearing vaults. The king can traverse the various gadgets backward, entering each vault through the exit and vice versa. But the reversal of a Hamiltonian cycle in G is another Hamiltonian cycle in G, so walking backward is fine.

If there is a Hamiltonian cycle in G, the white king can capture at least nN black pieces by visiting each of the other vaults and returning to the starting vault. On the other hand, if there is no Hamiltonian cycle in G, the white king can can capture at most half of the pieces in the starting vault, and thus can capture at most $(n - 1/2)N + O(n^3)$ enemy pieces altogether. The $O(n^3)$ term accounts for the corner and crossing gadgets; each edge passes through one corner gadget and at most $n^2/2$ crossing gadgets.

To complete the reduction, we set $N = n^4$. Summing up, we obtain an $O(n^5) \times O(n^5)$ board configuration, with $O(n^5)$ black pieces and one white king. We can clearly construct this board configuration in polynomial time. A complete example of the construction appears on the next page.

It is still open whether the following related question is NP-hard: Given an $n \times n$ board configuration for international draughts, can (and therefore *must*) White capture *all* the black pieces in a single turn?

30.15 Other Useful NP-hard Problems

Literally thousands of different problems have been proved to be NP-hard. I want to close this note by listing a few NP-hard problems that are useful in deriving reductions. I won't describe the NP-hardness proofs for these problems in detail, but you can find most of them in Garey and Johnson's classic *Scary Black Book of NP-Completeness*.[6]

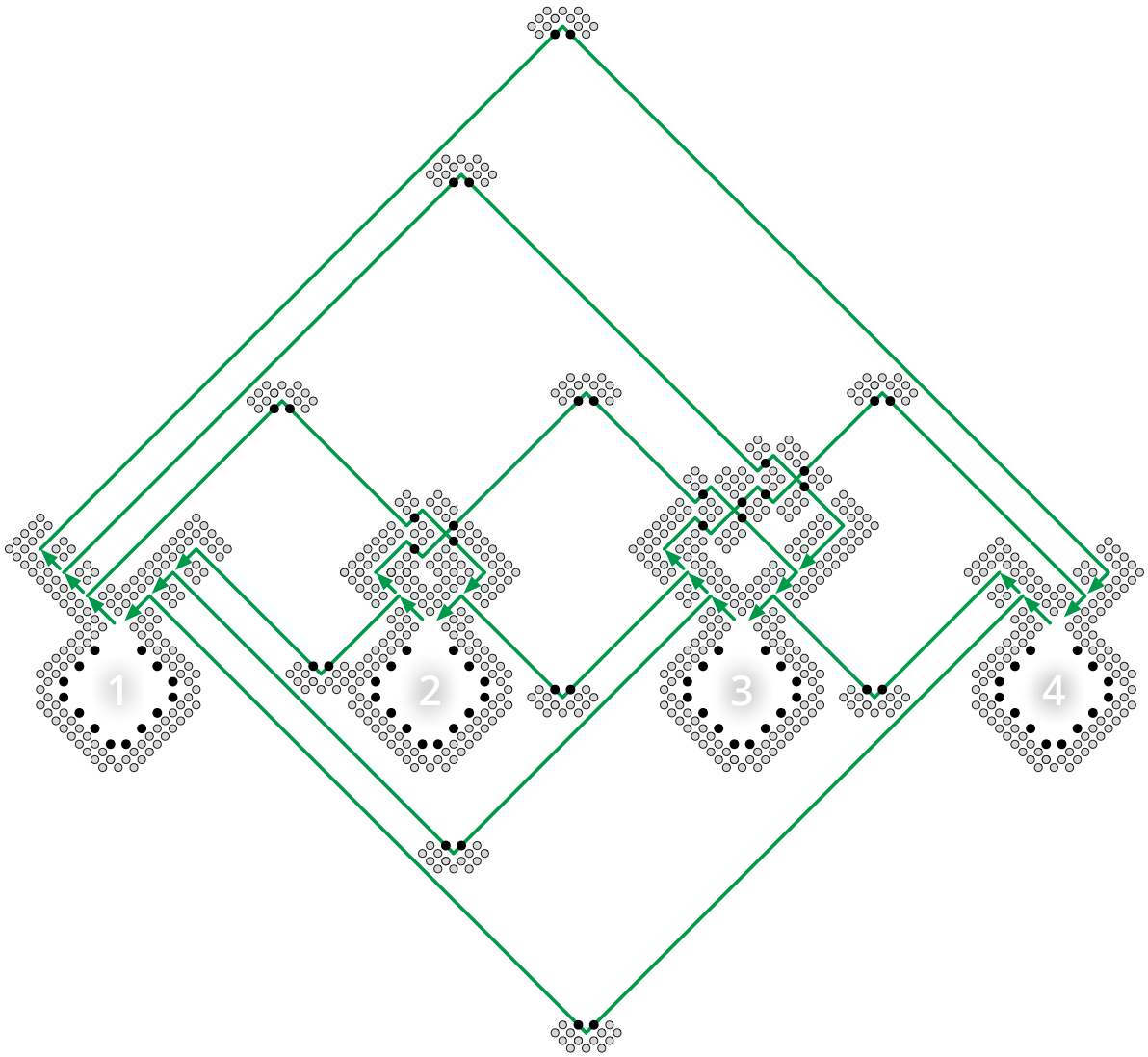
- PLANARCIRCUITSAT: Given a boolean circuit that can be embedded in the plane so that no two wires cross, is there an input that makes the circuit output TRUE? This problem can be proved NP-hard by reduction from the general circuit satisfiability problem, by replacing each crossing with a small series of gates.

- NOTALLEQUAL3SAT: Given a 3CNF formula, is there an assignment of values to the variables so that every clause contains at least one TRUE literal *and* at least one FALSE literal? This problem can be proved NP-hard by reduction from the usual 3SAT.

- PLANAR3SAT: Given a 3CNF boolean formula, consider a bipartite graph whose vertices are the clauses and variables, where an edge indicates that a variable (or its negation) appears in a clause. If this graph is planar, the 3CNF formula is also called planar. The PLANAR3SAT problem asks, given a planar 3CNF formula, whether it has a satisfying assignment. This problem can be proved NP-hard by reduction from PLANARCIRCUITSAT.[7]

[6] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Co., 1979.

[7] Surprisingly, PLANARNOTALLEQUAL3SAT is solvable in polynomial time!



The final draughts configuration for the example graph.

- EXACT3DIMENSIONALMATCHING or X3M: Given a set S and a collection of three-element subsets of S, called *triples*, is there a sub-collection of disjoint triples that exactly cover S? This problem can be proved NP-hard by a reduction from 3SAT.

- PARTITION: Given a set S of n integers, are there subsets A and B such that $A \cup B = S$, $A \cap B = \emptyset$, and

$$\sum_{a \in A} a = \sum_{b \in B} b?$$

  This problem can be proved NP-hard by a simple reduction from SUBSETSUM. Like SUBSETSUM, the PARTITION problem is only weakly NP-hard.

- 3PARTITION: Given a set S of $3n$ integers, can it be partitioned into n disjoint three-element subsets, such that every subset has exactly the same sum? Despite the similar names, this problem is *very* different from PARTITION; sorry, I didn't make up the names. This problem can be proved NP-hard by reduction from X3M. Unlike PARTITION, the 3PARTITION problem is *strongly* NP-hard, that is, it remains NP-hard even if the input numbers are less than some polynomial in n.

- SETCOVER: Given a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, find the smallest sub-collection of S_i's that contains all the elements of $\bigcup_i S_i$. This problem is a generalization of both VERTEXCOVER and X3M.

- HITTINGSET: Given a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, find the minimum number of elements of $\bigcup_i S_i$ that hit every set in $\mathcal{S}$. This problem is also a generalization of VERTEXCOVER.

- HAMILTONIANPATH: Given an graph G, is there a path in G that visits every vertex exactly once? This problem can be proved NP-hard either by modifying the reductions from 3SAT or VERTEXCOVER to HAMILTONIANCYCLE, or by a direct reduction from HAMILTONIANCYCLE.

- LONGESTPATH: Given a non-negatively weighted graph G and two vertices u and v, what is the longest simple path from u to v in the graph? A path is *simple* if it visits each vertex at most once. This problem is a generalization of the HAMILTONIANPATH problem. Of course, the corresponding *shortest* path problem is in P.

- STEINERTREE: Given a weighted, undirected graph G with some of the vertices marked, what is the minimum-weight subtree of G that contains every marked vertex? If *every* vertex is marked, the minimum Steiner tree is just the minimum spanning tree; if exactly two vertices are marked, the minimum Steiner tree is just the shortest path between them. This problem can be proved NP-hard by reduction from VERTEXCOVER.

In addition to these dry but useful problems, most interesting puzzles and solitaire games have been shown to be NP-hard, or to have NP-hard generalizations. (Arguably, if a game or puzzle isn't at least NP-hard, it isn't interesting!) Some familiar examples include Minesweeper[8] (by reduction from CIRCUITSAT), Tetris[9] (by reduction from 3PARTITION), Sudoku[10] (by reduction from 3SAT), and Pac-Man[11] (by reduction from Hamiltonian cycle).

*30.16 On Beyond Zebra

P and NP are only the first two steps in an enormous hierarchy of complexity classes. To close these notes, let me describe a few more classes of interest.

Polynomial Space. ***PSPACE*** is the set of decision problems that can be solved using polynomial *space*. Every problem in NP (and therefore in P) is also in PSPACE. It is generally believed that NP $\neq$ PSPACE, but nobody can even prove that P $\neq$ PSPACE. A problem Π is ***PSPACE-hard*** if, for any problem Π' that can be solved using polynomial *space*, there is a polynomial-*time* many-one reduction from Π' to Π. A problem is ***PSPACE-complete*** if it is both PSPACE-hard and in PSPACE. If any PSPACE-hard problem is in NP, then PSPACE=NP; similarly, if any PSPACE-hard problem is in P, then PSPACE=P.

[8] Richard Kaye. Minesweeper is NP-complete. *Mathematical Intelligencer* 22(2):9–15, 2000. http://www.mat.bham.ac.uk/R.W.Kaye/minesw/minesw.pdf

[9] Ron Breukelaar*, Erik D. Demaine, Susan Hohenberger*, Hendrik J. Hoogeboom, Walter A. Kosters, and David Liben-Nowell*. Tetris is hard, even to approximate. *International Journal of Computational Geometry and Applications* 14:41–68, 2004. The reduction was *considerably* simplified between its discovery in 2002 and its publication in 2004.

[10] Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E86-A(5):1052–1060, 2003. http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/MasterThesis.pdf.

[11] Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, to appear, 2013. http://giovanniviglietta.com/papers/gaming2.pdf.

The canonical PSPACE-complete problem is the *quantified boolean formula* problem, or ***QBF***: Given a boolean formula Φ that may include any number of universal or existential quantifiers, but no free variables, is Φ equivalent to TRUE? For example, the following expression is a valid input to QBF:

$$\exists a : \forall b : \exists c : (\forall d : a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee (\exists e : \overline{(\bar{a} \Rightarrow e)}) \vee (c \neq a \wedge e))).$$

SAT is provably equivalent the special case of QBF where the input formula contains only existential quantifiers. QBF remains PSPACE-hard even when the input formula must have all its quantifiers at the beginning, the quantifiers strictly alternate between $\exists$ and $\forall$, and the quantified proposition is in conjunctive normal form, with exactly three literals in each clause, for example:

$$\exists a : \forall b : \exists c : \forall d : \big((a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})\big)$$

This restricted version of QBF can also be phrased as a two-player strategy question. Suppose two players, Alice and Bob, are given a 3CNF predicate with free variables $x_1, x_2, \ldots, x_n$. The players alternately assign values to the variables in order by index—Alice assigns a value to x_1, Bob assigns a value to x_2, and so on. Alice eventually assigns values to every variable with an odd index, and Bob eventually assigns values to every variable with an even index. Alice wants to make the expression TRUE, and Bob wants to make it FALSE. Assuming Alice and Bob play perfectly, who wins this game? Not surprisingly, most two-player games[12] like tic-tac-toe, reversi, checkers, go, chess, and mancala—or more accurately, appropriate generalizations of these constant-size games to arbitrary board sizes—are PSPACE-hard.

Another canonical PSPACE-hard problem is *NFA totality*: Given a non-deterministic finite-state automaton M over some alphabet Σ, does M accept every string in Σ^*? The closely related problems *NFA equivalence* (Do two given NFAs accept the same language?) and *NFA minimization* (Find the smallest NFA that accepts the same language as a given NFA) are also PSPACE-hard, as are the corresponding questions about regular expressions. (The corresponding questions about *deterministic* finite-state automata are all solvable in polynomial time.)

Exponential time. The next significantly larger complexity class, ***EXP*** (also called EXPTIME), is the set of decision problems that can be solved in exponential time, that is, using at most 2^{n^c} steps for some constant $c > 0$. Every problem in PSPACE (and therefore in NP (and therefore in P)) is also in EXP. It is generally believed that PSPACE $\subsetneq$ EXP, but nobody can even prove that NP $\neq$ EXP. A problem Π is ***EXP-hard*** if, for any problem Π' that can be solved in *exponential* time, there is a *polynomial*-time many-one reduction from Π' to Π. A problem is ***EXP-complete*** if it is both EXP-hard and in EXP. If any EXP-hard problem is in PSPACE, then EXP=PSPACE; similarly, if any EXP-hard probelm is in NP, then EXP=NP. We *do* know that P $\neq$ EXP; in particular, no EXP-hard problem is in P.

Natural generalizations of many interesting 2-player games—like checkers, chess, mancala, and go—are actually EXP-hard. The boundary between PSPACE-complete games and EXP-hard games is rather subtle. For example, there are three ways to draw in chess (the standard 8×8 game): stalemate (the player to move is not in check but has no legal moves), repeating the same board position three times, or moving fifty times without capturing a piece. The $n \times n$ generalization of chess is either in PSPACE or EXP-hard depending on how we generalize these rules. If we declare a draw after (say) n^3 capture-free moves, then every game must end after a polynomial number of moves, so we can simulate all possible games from any given position using only polynomial space. On the other hand, if we ignore

[12] For a good (but now slightly dated) overview of known results on the computational complexity of games and puzzles, see Erik D. Demaine and Robert Hearn's survey "Playing Games with Algorithms: Algorithmic Combinatorial Game Theory" at http://arxiv.org/abs/cs.CC/0106019.

the capture-free move rule entirely, the resulting game can last an exponential number of moves, so there no obvious way to detect a repeating position using only polynomial space; indeed, this version of $n \times n$ chess is EXP-hard.

Excelsior! Naturally, even exponential time is not the end of the story. ***NEXP*** is the class of decision problems that can be solve in *nondeterministic* exponential time; equivalently, a decision problem is in NEXP if and only if, for every YES instance, there is a *proof* of this fact that can be checked in exponential time. ***EXPSPACE*** is the set of decision problems that can be solved using exponential *space*. Even these larger complexity classes have hard and complete problems; for example, if we add the intersection operator $\cap$ to the syntax of regular expressions, deciding whether two such expressions describe the same language is EXPSPACE-hard. Beyond EXPSPACE are complexity classes with *doubly*-exponential resource bounds (EEXP, NEEXP, and EEXPSPACE), then *triply* exponential resource bounds (EEEXP, NEEEXP, and EEEXPSPACE), and so on ad infinitum.

All these complexity classes can be ordered by inclusion as follows:

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXP} \subseteq \text{NEXP} \subseteq \text{EXPSPACE} \subseteq \text{EEXP} \subseteq \text{NEEXP} \subseteq \text{EEXPSPACE} \subseteq \text{EEEXP} \subseteq \cdots,$$

Most complexity theorists strongly believe that every inclusion in this sequence is strict; that is, no two of these complexity classes are equal. However, the strongest result that has been proved is that every class in this sequence is strictly contained in the class *three* steps later in the sequence. For example, we have proofs that P≠EXP and PSPACE≠EXPSPACE, but not whether P≠PSPACE or NP≠EXP.

The limit of this series of increasingly exponential complexity classes is the class ***ELEMENTARY*** of decision problems that can be solved using time or space bounded by a function the form $2 \uparrow^k n$ for some integer k, where

$$2 \uparrow^k n := \begin{cases} n & \text{if } k = 0, \\ 2^{2 \uparrow^{k-1} n} & \text{otherwise.} \end{cases}$$

For example, $2 \uparrow^1 n = 2^n$ and $2 \uparrow^2 n = 2^{2^n}$.

You might be tempted to conjecture that every natural decidable problem can be solved in elementary time, but then you would be wrong. Consider the *extended regular expressions* defined by recursively combining (possibly empty) strings over some finite alphabet by concatenation (xy), union ($x + y$), Kleene closure (x^*), and negation ($\overline{x}$). For example, the extended regular expression $\overline{(0+1)^*00(0+1)^*}$ represents the set of strings in $\{0,1\}^*$ that do *not* contain two 0s in a row. It is possible to determine algorithmically whether two extended regular expressions describe identical languages, by recursively converting each expression into an equivalent NFA, converting each NFA into a DFA, and then minimizing the DFA. Unfortunately, however, this equivalence problem cannot be decided in only elementary time, intuitively because each layer of recursive negation exponentially increases the number of states in the final DFA.

Exercises

1. (a) Describe and analyze and algorithm to solve PARTITION in time $O(nM)$, where n is the size of the input set and M is the sum of the absolute values of its elements.

   (b) Why doesn't this algorithm imply that P=NP?

2. Consider the following problem, called BOXDEPTH: Given a set of n axis-aligned rectangles in the plane, how big is the largest subset of these rectangles that contain a common point?

(a) Describe a polynomial-time reduction from BOXDEPTH to MAXCLIQUE.

(b) Describe and analyze a polynomial-time algorithm for BOXDEPTH. *[Hint: $O(n^3)$ time should be easy, but $O(n \log n)$ time is possible.]*

(c) Why don't these two results imply that P=NP?

3. A boolean formula is in *disjunctive normal form* (or *DNF*) if it consists of a *disjunction* (OR) or several *terms*, each of which is the conjunction (AND) of one or more literals. For example, the formula

$$(\overline{x} \wedge y \wedge \overline{z}) \vee (y \wedge z) \vee (x \wedge \overline{y} \wedge \overline{z})$$

is in disjunctive normal form. DNF-SAT asks, given a boolean formula in disjunctive normal form, whether that formula is satisfiable.

(a) Describe a polynomial-time algorithm to solve DNF-SAT.

(b) What is the error in the following argument that P=NP?

> *Suppose we are given a boolean formula in conjunctive normal form with at most three literals per clause, and we want to know if it is satisfiable. We can use the distributive law to construct an equivalent formula in disjunctive normal form. For example,*
>
> $$(x \vee y \vee \overline{z}) \wedge (\overline{x} \vee \overline{y}) \iff (x \wedge \overline{y}) \vee (y \wedge \overline{x}) \vee (\overline{z} \wedge \overline{x}) \vee (\overline{z} \wedge \overline{y})$$
>
> *Now we can use the algorithm from part (a) to determine, in polynomial time, whether the resulting DNF formula is satisfiable. We have just solved 3SAT in polynomial time. Since 3SAT is NP-hard, we must conclude that P=NP!*

4. (a) Describe a polynomial-time reduction from PARTITION to SUBSETSUM.

(b) Describe a polynomial-time reduction from SUBSETSUM to PARTITION.

5. (a) Describe a polynomial-time reduction from HAMILTONIANPATH to HAMILTONIANCYCLE.

(b) Describe a polynomial-time reduction from HAMILTONIANCYCLE to HAMILTONIANPATH. *[Hint: A polynomial-time reduction may call the black-box subroutine more than once.]*

6. (a) Describe a polynomial-time reduction from HAMILTONIANCYCLE to DIRECTEDHAMILTONIANCYCLE.

(b) Describe a polynomial-time reduction from DIRECTEDHAMILTONIANCYCLE to HAMILTONIANCYCLE.

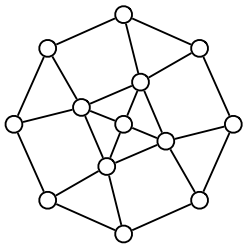
7. (a) Prove that PLANARCIRCUITSAT is NP-complete. *[Hint: Construct a gadget for crossing wires.]*

(b) Prove that NOTALLEQUAL3SAT is NP-complete.

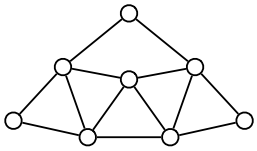
(c) Prove that the following variant of 3SAT is NP-complete: Given a boolean formula Φ in conjunctive normal form where each clause contains at most 3 literals and each variable appears in at most 3 clauses, does Φ have a satisfying assignment?

8. (a) Using the gadget on the right below, prove that deciding whether a given *planar* graph is 3-colorable is NP-complete. *[Hint: Show that the gadget can be 3-colored, and then replace any crossings in a planar embedding with the gadget appropriately.]*

(b) Using part (a) and the middle gadget below, prove that deciding whether a planar graph *with maximum degree 4* is 3-colorable is NP-complete. *[Hint: Replace any vertex with degree greater than 4 with a collection of gadgets connected so that no degree is greater than four.]*



(a) Gadget for planar 3-colorability.



(b) Gadget for degree-4 planar 3-colorability.

9. Prove that the following problems are NP-complete.

   (a) Given two undirected graphs G and H, is G isomorphic to a subgraph of H?

   (b) Given an undirected graph G, does G have a spanning tree in which every node has degree at most 17?

   (c) Given an undirected graph G, does G have a spanning tree with at most 42 leaves?

10. **There's something special about the number 3.**

    (a) Describe and analyze a polynomial-time algorithm for 2PARTITION. Given a set S of $2n$ positive integers, your algorithm will determine in polynomial time whether the elements of S can be split into n disjoint pairs whose sums are all equal.

    (b) Describe and analyze a polynomial-time algorithm for 2COLOR. Given an undirected graph G, your algorithm will determine in polynomial time whether G has a proper coloring that uses only two colors.

    (c) Describe and analyze a polynomial-time algorithm for 2SAT. Given a boolean formula Φ in conjunctive normal form, with exactly *two* literals per clause, your algorithm will determine in polynomial time whether Φ has a satisfying assignment.

11. **There's nothing special about the number 3.**

    (a) The problem 12PARTITION is defined as follows: Given a set S of $12n$ positive integers, determine whether the elements of S can be split into n subsets of 12 elements each whose sums are all equal. Prove that 12PARTITION is NP-hard. *[Hint: Reduce from* 3PARTITION*. It may be easier to consider multisets first.]*

    (b) The problem 12COLOR is defined as follows: Given an undirected graph G, determine whether we can color each vertex with one of twelve colors, so that every edge touches two different colors. Prove that 12COLOR is NP-hard. *[Hint: Reduce from* 3COLOR*.]*

    (c) The problem 12SAT is defined as follows: Given a boolean formula Φ in conjunctive normal form, with exactly twelve literals per clause, determine whether Φ has a satisfying assignment. Prove that 12SAT is NP-hard. *[Hint: Reduce from 3SAT.]*

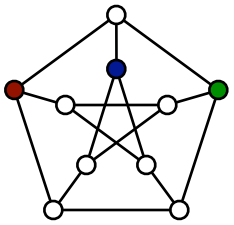
12. This exercise asks you to prove that a certain reduction from VERTEXCOVER to STEINERTREE is correct. Suppose we want to find the smallest vertex cover in a given undirected graph $G = (V, E)$. We construct a new graph $H = (V', E')$ as follows:

    - $V' = V \cup E \cup \{z\}$
    - $E' = \{ve \mid v \in V \text{ is an endpoint of } e \in W\} \cup \{vz \mid v \in V\}$.

    Equivalently, we construct H by subdividing each edge in G with a new vertex, and then connecting all the original vertices of G to a new *apex* vertex z.

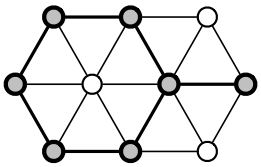
    Prove that G has a vertex cover of size k if and only if there is a subtree of H with $k + |E| + 1$ vertices that contains every vertex in $E \cup \{z\}$.

13. Let $G = (V, E)$ be a graph. A *dominating set* in G is a subset S of the vertices such that every vertex in G is either in S or adjacent to a vertex in S. The DOMINATINGSET problem asks, given a graph G and an integer k as input, whether G contains a dominating set of size k. Prove that this problem is NP-complete.



A dominating set of size 3 in the Peterson graph.

14. A subset S of vertices in an undirected graph G is called *triangle-free* if, for every triple of vertices $u, v, w \in S$, at least one of the three edges uv, uw, vw is *absent* from G. ***Prove*** that finding the size of the largest triangle-free subset of vertices in a given undirected graph is NP-hard.

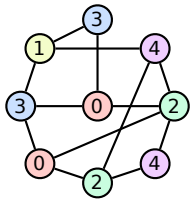


A triangle-free subset of 7 vertices.
This is ***not*** the largest triangle-free subset in this graph.

15. *Pebbling* is a solitaire game played on an undirected graph G, where each vertex has zero or more *pebbles*. A single *pebbling move* consists of removing two pebbles from a vertex v and adding one pebble to an arbitrary neighbor of v. (Obviously, the vertex v must have at least two pebbles before the move.) The PEBBLEDESTRUCTION problem asks, given a graph $G = (V, E)$ and a pebble count $p(v)$ for each vertex v, whether is there a sequence of pebbling moves that removes all but one pebble. Prove that PEBBLEDESTRUCTION is NP-complete.

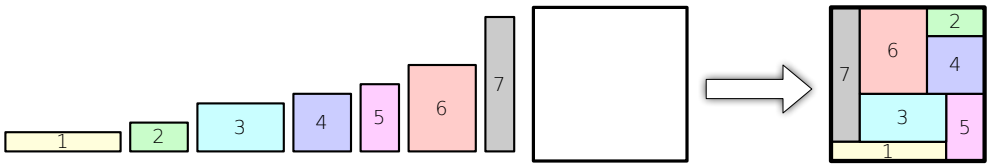
16. Recall that a 5-coloring of a graph G is a function that assigns each vertex of G an 'color' from the set $\{0, 1, 2, 3, 4\}$, such that for any edge uv, vertices u and v are assigned different 'colors'.

A 5-coloring is *careful* if the colors assigned to adjacent vertices are not only distinct, but differ by more than 1 (mod 5). Prove that deciding whether a given graph has a careful 5-coloring is NP-complete. *[Hint: Reduce from the standard* 5COLOR *problem.]*



A careful 5-coloring.

17. The RECTANGLETILING problem is defined as follows: Given one large rectangle and several smaller rectangles, determine whether the smaller rectangles can be placed inside the large rectangle with no gaps or overlaps. Prove that RECTANGLETILING is NP-complete.



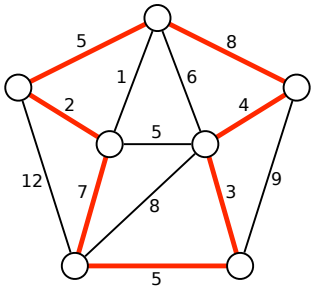
A positive instance of the RECTANGLETILING problem.

18. For each problem below, either describe a polynomial-time algorithm or prove that the problem is NP-complete.

    (a) A *double-Eulerian circuit* in an undirected graph G is a closed walk that traverses every edge in G exactly twice. Given a graph G, does G have a double-Eulerian circuit?

    (b) A *double-Hamiltonian circuit* in an undirected graph G is a closed walk that visits every vertex in G exactly twice. Given a graph G, does G have a double-Hamiltonian circuit?

19. Let G be an undirected graph with weighted edges. A *heavy Hamiltonian cycle* is a cycle C that passes through each vertex of G exactly once, such that the total weight of the edges in C is at least half of the total weight of all edges in G. Prove that deciding whether a graph has a heavy Hamiltonian cycle is NP-complete.

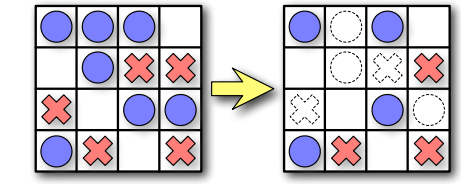


A heavy Hamiltonian cycle. The cycle has total weight 34; the graph has total weight 67.

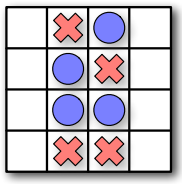
20. (a) A *tonian path* in a graph G is a path that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian path is NP-complete.

    (b) A *tonian cycle* in a graph G is a cycle that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian cycle is NP-complete. *[Hint: Use part (a).]*

21. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions: (1) every row contains at least one stone, and (2) no column contains stones of both colors. For some initial configurations of stones, reaching this goal is impossible.



A solvable puzzle and one of its many solutions.



An unsolvable puzzle.

    Prove that it is NP-hard to determine, given an initial configuration of red and blue stones, whether the puzzle can be solved.

22. A boolean formula in *exclusive-or conjunctive normal form* (XCNF) is a conjunction (AND) of several *clauses*, each of which is the *exclusive*-or of several literals; that is, a clause is true if and only if it contains an odd number of true literals. The XCNF-SAT problem asks whether a given XCNF formula is satisfiable. Either describe a polynomial-time algorithm for XCNF-SAT or prove that it is NP-hard.

23. You're in charge of choreographing a musical for your local community theater, and it's time to figure out the final pose of the big show-stopping number at the end. ("Streetcar!"") You've decided that each of the n cast members in the show will be positioned in a big line when the song finishes, all with their arms extended and showing off their best spirit fingers.

    The director has declared that during the final flourish, each cast member must either point both their arms up or point both their arms down; it's your job to figure out who points up and who points down. Moreover, in a fit of unchecked power, the director has also given you a list of arrangements that will upset his delicate artistic temperament. Each forbidden arrangement is a subset of the cast members paired with arm positions; for example: "Marge may not point her arms up while Ned, Apu, and Smithers point their arms down."

    Prove that finding an acceptable arrangement of arm positions is NP-hard.

24. Jeff tries to make his students happy. At the beginning of class, he passes out a questionnaire that lists a number of possible course policies in areas where he is flexible. Every student is asked to respond to each possible course policy with one of "strongly favor", "mostly neutral", or "strongly oppose". Each student may respond with "strongly favor" or "strongly oppose" to at most five questions. Because Jeff's students are very understanding, each student is happy if (but only if) he or she prevails in just one of his or her strong policy preferences. Either describe a polynomial-time

algorithm for setting course policy to maximize the number of happy students, or show that the problem is NP-hard.

25. The next time you are at a party, one of the guests will suggest everyone play a round of Three-Way Mumbledypeg, a game of skill and dexterity that requires three teams and a knife. The official Rules of Three-Way Mumbledypeg (fixed during the Holy Roman Three-Way Mumbledypeg Council in 1625) require that (1) each team *must* have at least one person, (2) any two people on the same team *must* know each other, and (3) everyone watching the game *must* be on one of the three teams. Of course, it will be a really *fun* party; nobody will want to leave. There will be several pairs of people at the party who don't know each other. The host of the party, having heard thrilling tales of your prowess in all things algorithmic, will hand you a list of which pairs of party-goers know each other and ask you to choose the teams, while he sharpens the knife.

    Either describe and analyze a polynomial time algorithm to determine whether the party-goers can be split into three legal Three-Way Mumbledypeg teams, or prove that the problem is NP-hard.

26. The party you are attending is going great, but now it's time to line up for ***The Algorithm March (アルゴリズムこうしん)***! This dance was originally developed by the Japanese comedy duo Itsumo Kokokara (いつもここから) for the children's television show PythagoraSwitch (ピタゴラスイッチ). The Algorithm March is performed by a line of people; each person in line starts a specific sequence of movements one measure later than the person directly in front of them. Thus, the march is the dance equivalent of a musical round or canon, like "Row Row Row Your Boat".

    Proper etiquette dictates that each marcher must know the person directly in front of them in line, lest a minor mistake during lead to horrible embarrassment between strangers. Suppose you are given a complete list of which people at your party know each other. ***Prove*** that it is NP-hard to determine the largest number of party-goers that can participate in the Algorithm March. You may assume without loss of generality that there are no ninjas at your party.

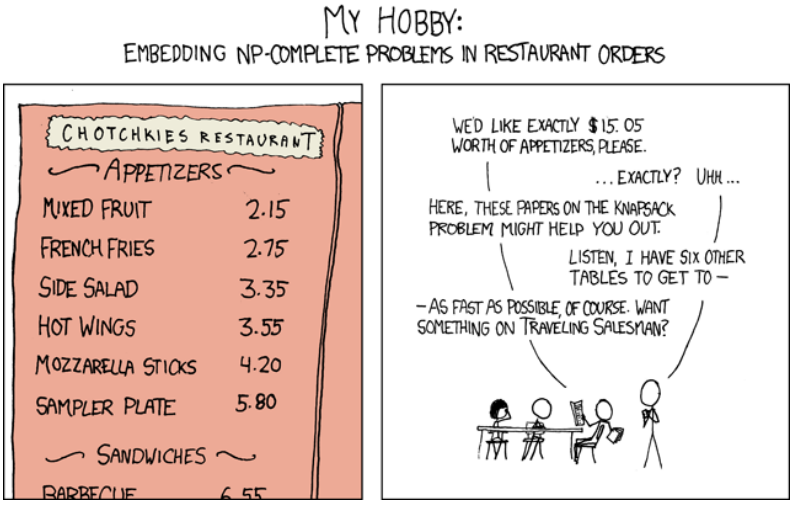
27. (a) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary weighted graph G, the length of the shortest Hamiltonian cycle in G. Describe and analyze a **polynomial-time** algorithm that computes, given an arbitrary weighted graph G, the shortest Hamiltonian cycle in G, using this magic black box as a subroutine.

    (b) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary graph G, the number of vertices in the largest complete subgraph of G. Describe and analyze a **polynomial-time** algorithm that computes, given an arbitrary graph G, a complete subgraph of G of maximum size, using this magic black box as a subroutine.

    (c) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary graph G, whether G is 3-colorable. Describe and analyze a **polynomial-time** algorithm that either computes a proper 3-coloring of a given graph or correctly reports that no such coloring exists, using the magic black box as a subroutine. *[Hint: The input to the magic black box is a graph. Just a graph. Vertices and edges. Nothing else.]*

    (d) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary boolean formula Φ, whether Φ is satisfiable. Describe and analyze a **polynomial-time** algorithm that either computes a satisfying assignment for a given boolean formula or correctly reports that no such assignment exists, using the magic black box as a subroutine.

(e) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary set X of positive integers, whether X can be partitioned into two sets A and B such that $\sum A = \sum B$. Describe and analyze a **polynomial-time** algorithm that either computes an equal partition of a given set of positive integers or correctly reports that no such partition exists, using the magic black box as a subroutine.



[General solutions give you a 50% tip.]

— Randall Munroe, *xkcd* (http://xkcd.com/287/)

Le mieux est l'ennemi du bien. [The best is the enemy of the good.]

— Voltaire, *La Bégueule* (1772)

Who shall forbid a wise skepticism, seeing that there is no practical question on which any thing more than an approximate solution can be had?

— Ralph Waldo Emerson, *Representative Men* (1850)

Now, distrust of corporations threatens our still-tentative economic recovery; it turns out greed is bad, after all.

— Paul Krugman, "Greed is Bad", *The New York Times*, June 4, 2002.

*31 Approximation Algorithms

31.1 Load Balancing

On the future smash hit reality-TV game show *Grunt Work*, scheduled to air Thursday nights at 3am (2am Central) on ESPNπ, the contestants are given a series of utterly pointless tasks to perform. Each task has a predetermined time limit; for example, "Sharpen this pencil for 17 seconds", or "Pour pig's blood on your head and sing The Star-Spangled Banner for two minutes", or "Listen to this 75-minute algorithms lecture". The directors of the show want you to assign each task to one of the contestants, so that the last task is completed as early as possible. When your predecessor correctly informed the directors that their problem is NP-hard, he was immediately fired. "Time is money!" they screamed at him. "We don't need perfection. Wake up, dude, this is *television*!"

Less facetiously, suppose we have a set of n jobs, which we want to assign to m machines. We are given an array $T[1..n]$ of non-negative numbers, where $T[j]$ is the running time of job j. We can describe an *assignment* by an array $A[1..n]$, where $A[j] = i$ means that job j is assigned to machine i. The *makespan* of an assignment is the maximum time that any machine is busy:

$$makespan(A) = \max_i \sum_{A[j]=i} T[j]$$

The *load balancing* problem is to compute the assignment with the smallest possible makespan.

It's not hard to prove that the load balancing problem is NP-hard by reduction from PARTITION: The array $T[1..n]$ can be evenly partitioned if and only if there is an assignment to two machines with makespan exactly $\sum_i T[i]/2$. A slightly more complicated reduction from 3PARTITION implies that the load balancing problem is *strongly* NP-hard. If we really need the optimal solution, there is a dynamic programming algorithm that runs in time $O(nM^m)$, where M is the minimum makespan, but that's just horrible.

There is a fairly natural and efficient greedy heuristic for load balancing: consider the jobs one at a time, and assign each job to the machine i with the earliest finishing time $Total[i]$.

```
GREEDYLOADBALANCE(T[1..n], m):
  for i ← 1 to m
      Total[i] ← 0

  for j ← 1 to n
      mini ← arg min_i Total[i]
      A[j] ← mini
      Total[mini] ← Total[mini] + T[j]

  return A[1..m]
```

Theorem 1. *The makespan of the assignment computed by* GREEDYLOADBALANCE *is at most twice the makespan of the optimal assignment.*

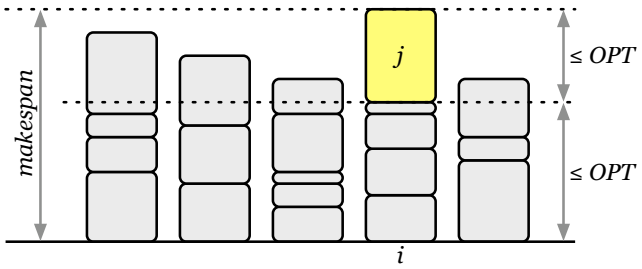
Proof: Fix an arbitrary input, and let *OPT* denote the makespan of its optimal assignment. The approximation bound follows from two trivial observations. First, the makespan of any assignment (and therefore of the optimal assignment) is at least the duration of the longest job. Second, the makespan of any assignment is at least the total duration of all the jobs divided by the number of machines.

$$OPT \geq \max_j T[j] \qquad \text{and} \qquad OPT \geq \frac{1}{m}\sum_{j=1}^{n} T[j]$$

Now consider the assignment computed by GREEDYLOADBALANCE. Suppose machine i has the largest total running time, and let j be the last job assigned to machine i. Our first trivial observation implies that $T[j] \leq OPT$. To finish the proof, we must show that $Total[i] - T[j] \leq OPT$. Job j was assigned to machine i because it had the smallest finishing time, so $Total[i] - T[j] \leq Total[k]$ for all k. (Some values $Total[k]$ may have increased since job j was assigned, but that only helps us.) In particular, $Total[i] - T[j]$ is less than or equal to the *average* finishing time over all machines. Thus,

$$Total[i] - T[j] \leq \frac{1}{m}\sum_{i=1}^{m} Total[i] = \frac{1}{m}\sum_{j=1}^{n} T[j] \leq OPT$$

by our second trivial observation. We conclude that the makespan $Total[i]$ is at most $2 \cdot OPT$. □



Proof that GREEDYLOADBALANCE is a 2-approximation algorithm

GREEDYLOADBALANCE is an *online* algorithm: It assigns jobs to machines in the order that the jobs appear in the input array. Online approximation algorithms are useful in settings where inputs arrive in a stream of unknown length—for example, real jobs arriving at a real scheduling algorithm. In this online setting, it may be *impossible* to compute an optimum solution, even in cases where the offline problem (where all inputs are known in advance) can be solved in polynomial time. The study of online algorithms could easily fill an entire one-semester course (alas, not this one).

In our original offline setting, we can improve the approximation factor by sorting the jobs before piping them through the greedy algorithm.

```
SORTEDGREEDYLOADBALANCE(T[1..n], m):
  sort T in decreasing order
  return GREEDYLOADBALANCE(T, m)
```

Theorem 2. *The makespan of the assignment computed by* SORTEDGREEDYLOADBALANCE *is at most* $3/2$ *times the makespan of the optimal assignment.*

Proof: Let i be the busiest machine in the schedule computed by SORTEDGREEDYLOADBALANCE. If only one job is assigned to machine i, then the greedy schedule is actually optimal, and the theorem is trivially true. Otherwise, let j be the last job assigned to machine i. Since each of the first m jobs is assigned to a unique machine, we must have $j \geq m+1$. As in the previous proof, we know that $Total[i] - T[j] \leq OPT$.

In *any* schedule, at least two of the first $m+1$ jobs, say jobs k and ℓ, must be assigned to the same machine. Thus, $T[k] + T[\ell] \leq OPT$. Since $\max\{k,\ell\} \leq m+1 \leq j$, and the jobs are sorted in decreasing order by duration, we have

$$T[j] \leq T[m+1] \leq T[\max\{k,\ell\}] = \min\{T[k], T[\ell]\} \leq OPT/2.$$

We conclude that the makespan $Total[i]$ is at most $3 \cdot OPT/2$, as claimed. □

31.2 Generalities

Consider an arbitrary optimization problem. Let $OPT(X)$ denote the value of the optimal solution for a given input X, and let $A(X)$ denote the value of the solution computed by algorithm A given the same input X. We say that A is an ***α(n)-approximation algorithm*** if and only if

$$\frac{OPT(X)}{A(X)} \leq \alpha(n) \quad \text{and} \quad \frac{A(X)}{OPT(X)} \leq \alpha(n)$$

for all inputs X of size n. The function $\alpha(n)$ is called the ***approximation factor*** for algorithm A. For any given algorithm, only one of these two inequalities will be important. For maximization problems, where we want to compute a solution whose cost is as small as possible, the first inequality is trivial. For maximization problems, where we want a solution whose value is as large as possible, the second inequality is trivial. A 1-approximation algorithm always returns the exact optimal solution.

Especially for problems where exact optimization is NP-hard, we have little hope of completely characterizing the optimal solution. The secret to proving that an algorithm satisfies some approximation ratio is to find a useful function of the input that provides both lower bounds on the cost of the optimal solution and upper bounds on the cost of the approximate solution. For example, if $OPT(X) \geq f(X)/2$ and $A(X) \leq 5f(X)$ for *any* function f, then A is a 10-approximation algorithm. Finding the right intermediate function can be a delicate balancing act.

31.3 Greedy Vertex Cover

Recall that the *vertex color* problem asks, given a graph G, for the smallest set of vertices of G that cover every edge. This is one of the first NP-hard problems introduced in the first week of class. There is a natural and efficient greedy heuristic[1] for computing a small vertex cover: mark the vertex with the largest degree, remove all the edges incident to that vertex, and recurse.

```
GREEDYVERTEXCOVER(G):
  C ← ∅
  while G has at least one edge
      v ← vertex in G with maximum degree
      G ← G \ v
      C ← C ∪ v
  return C
```

Obviously this algorithm doesn't compute the optimal vertex cover—that would imply P=NP!—but it does compute a reasonably close approximation.

[1] A *heuristic* is an algorithm that doesn't work.

Theorem 3. GREEDYVERTEXCOVER *is an* $O(\log n)$*-approximation algorithm.*

Proof: For all i, let G_i denote the graph G after i iterations of the main loop, and let d_i denote the maximum degree of any node in G_{i-1}. We can define these variables more directly by adding a few extra lines to our algorithm:

```
GREEDYVERTEXCOVER(G):
  C ← ∅
  G_0 ← G
  i ← 0
  while G_i has at least one edge
      i ← i + 1
      v_i ← vertex in G_{i-1} with maximum degree
      d_i ← deg_{G_{i-1}}(v_i)
      G_i ← G_{i-1} \ v_i
      C ← C ∪ v_i
  return C
```

Let $|G_{i-1}|$ denote the number of edges in the graph G_{i-1}. Let C^* denote the optimal vertex cover of G, which consists of OPT vertices. Since C^* is also a vertex cover for G_{i-1}, we have

$$\sum_{v \in C^*} \deg_{G_{i-1}}(v) \geq |G_{i-1}|.$$

In other words, the *average* degree in G_i of any node in C^* is at least $|G_{i-1}|/OPT$. It follows that G_{i-1} has at least one node with degree at least $|G_{i-1}|/OPT$. Since d_i is the maximum degree of any node in G_{i-1}, we have

$$d_i \geq \frac{|G_{i-1}|}{OPT}$$

Moreover, for any $j \geq i-1$, the subgraph G_j has no more edges than G_{i-1}, so $d_i \geq |G_j|/OPT$. This observation implies that

$$\sum_{i=1}^{OPT} d_i \;\geq\; \sum_{i=1}^{OPT} \frac{|G_{i-1}|}{OPT} \;\geq\; \sum_{i=1}^{OPT} \frac{|G_{OPT}|}{OPT} \;=\; |G_{OPT}| \;=\; |G| - \sum_{i=1}^{OPT} d_i.$$

In other words, the first OPT iterations of GREEDYVERTEXCOVER remove at least half the edges of G. Thus, after at most $OPT \lg|G| \leq 2\,OPT \lg n$ iterations, all the edges of G have been removed, and the algorithm terminates. We conclude that GREEDYVERTEXCOVER computes a vertex cover of size $O(OPT \log n)$. □

So far we've only proved an *upper bound* on the approximation factor of GREEDYVERTEXCOVER; perhaps a more careful analysis would imply that the approximation factor is only $O(\log\log n)$, or even $O(1)$. Alas, no such improvement is possible. For any integer n, a simple recursive construction gives us an n-vertex graph for which the greedy algorithm returns a vertex cover of size $\Omega(OPT \cdot \log n)$. Details are left as an exercise for the reader.

31.4 Set Cover and Hitting Set

The greedy algorithm for vertex cover can be applied almost immediately to two more general problems: *set cover* and *hitting set*. The input for both of these problems is a *set system* $(X, \mathcal{F})$, where X is a finite *ground set*, and $\mathcal{F}$ is a family of subsets of X.[2] A *set cover* of a set system $(X, \mathcal{F})$ is a subfamily of sets

[2] A matroid (see the lecture note on greedy algorithms) is a special type of set system.

in $\mathcal{F}$ whose union is the entire ground set X. A *hitting set* for $(X, \mathcal{F})$ is a subset of the ground set X that intersects every set in $\mathcal{F}$.

An undirected graph can be cast as a set system in two different ways. In one formulation, the ground set X contains the vertices, and each edge defines a set of two vertices in $\mathcal{F}$. In this formulation, a vertex cover is a hitting set. In the other formulation, the *edges* are the ground set, the *vertices* define the family of subsets, and a vertex cover is a set cover.

Here are the natural greedy algorithms for finding a small set cover and finding a small hitting set. GREEDYSETCOVER finds a set cover whose size is at most $O(\log|\mathcal{F}|)$ times the size of smallest set cover. GREEDYHITTINGSET finds a hitting set whose size is at most $O(\log|X|)$ times the size of the smallest hitting set.

```
GreedySetCover(X, F):
  C ← ∅
  while X ≠ ∅
      S ← arg max_{S∈F} |S ∩ X|
      X ← X \ S
      C ← C ∪ {S}
  return C
```

```
GreedyHittingSet(X, F):
  H ← ∅
  while F ≠ ∅
      x ← arg max_{x∈X} |{S ∈ F | x ∈ S}|
      F ← F \ {S ∈ F | x ∈ S}
      H ← H ∪ {x}
  return H
```

The similarity between these two algorithms is no coincidence. For any set system $(X, \mathcal{F})$, there is a *dual* set system $(\mathcal{F}, X^*)$ defined as follows. For any element $x \in X$ in the ground set, let x^* denote the subfamily of sets in $\mathcal{F}$ that contain x:

$$x^* = \{S \in \mathcal{F} \mid x \in S\}.$$

Finally, let X^* denote the collection of all subsets of the form x^*:

$$X^* = \left\{x^* \;\middle|\; x \in S\right\}.$$

As an example, suppose X is the set of letters of alphabet and $\mathcal{F}$ is the set of last names of student taking CS 573 this semester. Then X^* has 26 elements, each containing the subset of CS 573 students whose last name contains a particular letter of the alphabet. For example, m^* is the set of students whose last names contain the letter m.

There is a direct one-to-one correspondence between the ground set X and the dual set family X^*. It is a tedious but instructive exercise to prove that the dual of the dual of any set system is isomorphic to the original set system—$(X^*, \mathcal{F}^*)$ is essentially the same as $(X, \mathcal{F})$. It is also easy to prove that a set cover for any set system $(X, \mathcal{F})$ is also a hitting set for the dual set system $(\mathcal{F}, X^*)$, and therefore a hitting set for any set system $(X, \mathcal{F})$ is isomorphic to a set cover for the dual set system $(\mathcal{F}, X^*)$.

31.5 Vertex Cover, Again

The greedy approach doesn't always lead to the best approximation algorithms. Consider the following alternate heuristic for vertex cover:

```
DumbVertexCover(G):
  C ← ∅
  while G has at least one edge
      (u, v) ← any edge in G
      G ← G \ {u, v}
      C ← C ∪ {u, v}
  return C
```

The minimum vertex cover—in fact, *every* vertex cover—contains at least one of the two vertices u and v chosen inside the while loop. It follows immediately that DUMBVERTEXCOVER is a 2-approximation algorithm!

The same idea can be extended to approximate the minimum hitting set for any set system $(X, \mathcal{F})$, where the approximation factor is the size of the largest set in $\mathcal{F}$.

31.6 Traveling Salesman: The Bad News

The *traveling salesman problem*[3] problem asks for the shortest Hamiltonian cycle in a weighted undirected graph. To keep the problem simple, we can assume without loss of generality that the underlying graph is always the complete graph K_n for some integer n; thus, the input to the traveling salesman problem is just a list of the $\binom{n}{2}$ edge lengths.

Not surprisingly, given its similarity to the Hamiltonian cycle problem, it's quite easy to prove that the traveling salesman problem is NP-hard. Let G be an arbitrary undirected graph with n vertices. We can construct a length function for K_n as follows:

$$\ell(e) = \begin{cases} 1 & \text{if } e \text{ is an edge in } G, \\ 2 & \text{otherwise.} \end{cases}$$

Now it should be obvious that if G has a Hamiltonian cycle, then there is a Hamiltonian cycle in K_n whose length is exactly n; otherwise every Hamiltonian cycle in K_n has length at least $n+1$. We can clearly compute the lengths in polynomial time, so we have a polynomial time reduction from Hamiltonian cycle to traveling salesman. Thus, the traveling salesman problem is NP-hard, even if all the edge lengths are 1 and 2.

There's nothing special about the values 1 and 2 in this reduction; we can replace them with any values we like. By choosing values that are sufficiently far apart, we can show that even approximating the shortest traveling salesman tour is NP-hard. For example, suppose we set the length of the 'absent' edges to $n+1$ instead of 2. Then the shortest traveling salesman tour in the resulting weighted graph either has length exactly n (if G has a Hamiltonian cycle) or has length at least $2n$ (if G does not have a Hamiltonian cycle). Thus, if we could approximate the shortest traveling salesman tour within a factor of 2 in polynomial time, we would have a polynomial-time algorithm for the Hamiltonian cycle problem.

Pushing this idea to its limits us the following negative result.

Theorem 4. *For any function $f(n)$ that can be computed in time polynomial in n, there is no polynomial-time $f(n)$-approximation algorithm for the traveling salesman problem on general weighted graphs, unless P=NP.*

31.7 Traveling Salesman: The Good News

Even though the general traveling salesman problem can't be approximated, a common special case can be approximated fairly easily. The special case requires the edge lengths to satisfy the so-called *triangle inequality*:

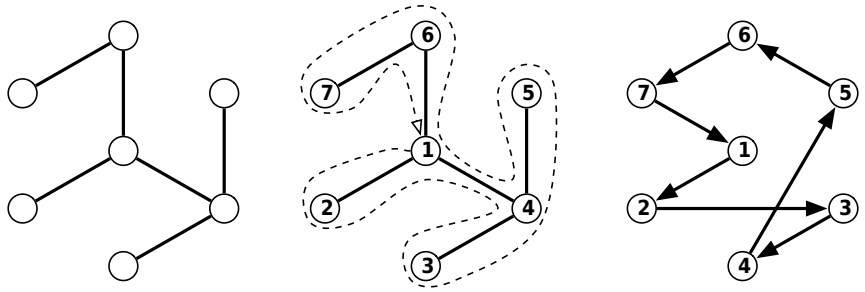
$$\ell(u,w) \le \ell(u,v) + \ell(v,w) \quad \text{for any vertices } u, v, w.$$

This inequality is satisfied for *geometric graphs*, where the vertices are points in the plane (or some higher-dimensional space), edges are straight line segments, and lengths are measured in the usual

[3]This is sometimes bowdlerized into the traveling sales*person* problem. That's just silly. Who ever heard of a traveling salesperson sleeping with the farmer's child?

Euclidean metric. Notice that the length functions we used above to show that the general TSP is hard to approximate do not (always) satisfy the triangle inequality.

With the triangle inequality in place, we can quickly compute a 2-approximation for the traveling salesman tour as follows. First, we compute the minimum spanning tree T of the weighted input graph; this can be done in $O(n^2 \log n)$ time (where n is the number of vertices of the graph) using any of several classical algorithms. Second, we perform a depth-first traversal of T, numbering the vertices in the order that we first encounter them. Because T is a spanning tree, every vertex is numbered. Finally, we return the cycle obtained by visiting the vertices according to this numbering.



A minimum spanning tree T, a depth-first traversal of T, and the resulting approximate traveling salesman tour.

Theorem 5. *A depth-first ordering of the minimum spanning tree gives a 2-approximation of the shortest traveling salesman tour.*

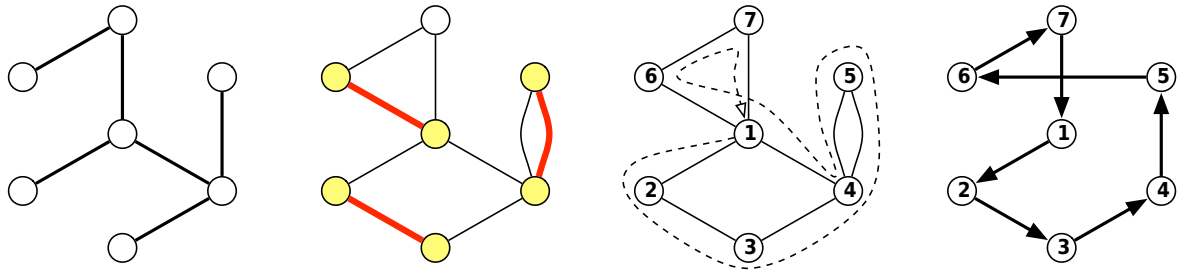
Proof: Let *OPT* denote the cost of the optimal TSP tour, let *MST* denote the total length of the minimum spanning tree, and let A be the length of the tour computed by our approximation algorithm. Consider the 'tour' obtained by walking through the minimum spanning tree in depth-first order. Since this tour traverses every edge in the tree exactly twice, its length is $2 \cdot MST$. The final tour can be obtained from this one by removing duplicate vertices, moving directly from each node to the next *unvisited* node.; the triangle inequality implies that taking these shortcuts cannot make the tour longer. Thus, $A \leq 2 \cdot MST$. On the other hand, if we remove any edge from the optimal tour, we obtain a spanning tree (in fact a spanning *path*) of the graph; thus, $MST \geq OPT$. We conclude that $A \leq 2 \cdot OPT$; our algorithm computes a 2-approximation of the optimal tour. □

We can improve this approximation factor using the following algorithm discovered by Nicos Christofides in 1976. As in the previous algorithm, we start by constructing the minimum spanning tree T. Then let O be the set of vertices with *odd* degree in T; it is an easy exercise (hint, hint) to show that the number of vertices in O is even.

In the next stage of the algorithm, we compute a *minimum-cost perfect matching* M of these odd-degree vertices. A prefect matching is a collection of edges, where each edge has both endpoints in O and each vertex in O is adjacent to exactly one edge; we want the perfect matching of minimum total length. Later in the semester, we will see an algorithm to compute M in polynomial time.

Now consider the multigraph $T \cup M$; any edge in both T and M appears twice in this multigraph. This graph is connected, and every vertex has even degree. Thus, it contains an *Eulerian circuit*: a closed walk that uses every edge exactly once. We can compute such a walk in $O(n)$ time with a simple modification of depth-first search. To obtain the final approximate TSP tour, we number the vertices in the order they first appear in some Eulerian circuit of $T \cup M$, and return the cycle obtained by visiting the vertices according to that numbering.

Theorem 6. *Given a weighted graph that obeys the triangle inequality, the Christofides heuristic computes a (3/2)-approximation of the shortest traveling salesman tour.*



A minimum spanning tree T, a minimum-cost perfect matching M of the odd vertices in T, an Eulerian circuit of $T \cup M$, and the resulting approximate traveling salesman tour.

Proof: Let A denote the length of the tour computed by the Christofides heuristic; let OPT denote the length of the optimal tour; let MST denote the total length of the minimum spanning tree; let MOM denote the total length of the minimum odd-vertex matching.

The graph $T \cup M$, and therefore any Euler tour of $T \cup M$, has total length $MST + MOM$. By the triangle inequality, taking a shortcut past a previously visited vertex can only shorten the tour. Thus, $A \leq MST + MOM$.

By the triangle inequality, the optimal tour of the odd-degree vertices of T cannot be longer than OPT. Any cycle passing through of the odd vertices can be partitioned into two perfect matchings, by alternately coloring the edges of the cycle red and green. One of these two matchings has length at most $OPT/2$. On the other hand, both matchings have length at least MOM. Thus, $MOM \leq OPT/2$.

Finally, recall our earlier observation that $MST \leq OPT$.

Putting these three inequalities together, we conclude that $A \leq 3 \cdot OPT/2$, as claimed. □

31.8 k-center Clustering

The *k-center clustering* problem is defined as follows. We are given a set $P = \{p_1, p_2, \ldots, p_n\}$ of n points in the plane[4] and an integer k. Our goal to find a collection of k circles that collectively enclose all the input points, such that the radius of the largest circle is as large as possible. More formally, we want to compute a set $C = \{c_1, c_2, \ldots, c_k\}$ of k *center points*, such that the following cost function is minimized:

$$cost(C) := \max_i \min_j |p_i\, c_j|.$$

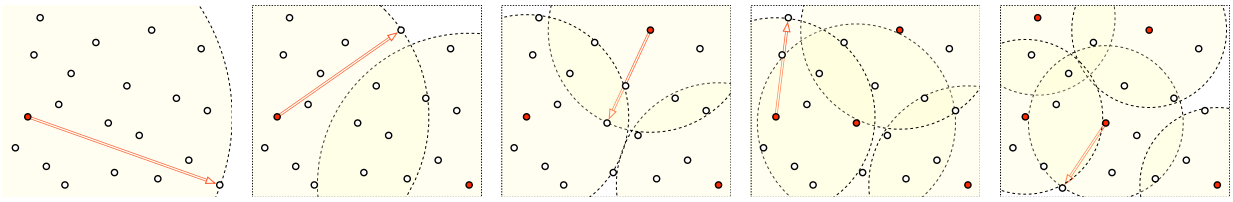
Here, $|p_i\, c_j|$ denotes the Euclidean distance between input point p_i and center point c_j. Intuitively, each input point is assigned to its closest center point; the points assigned to a given center c_j comprise a *cluster*. The distance from c_j to the farthest point in its cluster is the *radius* of that cluster; the cluster is contained in a circle of this radius centered at c_j. The k-center clustering cost $cost(C)$ is precisely the maximum cluster radius.

This problem turns out to be NP-hard, even to approximate within a factor of roughly 1.8. However, there is a natural greedy strategy, first analyzed in 1985 by Teofilo Gonzalez[5], that is guaranteed to produce a clustering whose cost is at most twice optimal. Choose the k center points one at a time, starting with an arbitrary input point as the first center. In each iteration, choose the input point that is farthest from any earlier center point to be the next center point.

In the pseudocode below, d_i denotes the current distance from point p_i to its nearest center, and r_j denotes the maximum of all d_i (or in other words, the cluster radius) after the first j centers have

[4]The k-center problem can be defined over *any* metric space, and the approximation analysis in this section holds in any metric space as well. The analysis in the *next* section, however, does require that the points come from the Euclidean plane.

[5]Teofilo F. Gonzalez. Clustering to minimize the maximum inter-cluster distance. *Theoretical Computer Science* 38:293-306, 1985.



The first five iterations of Gonzalez's k-center clustering algorithm.

been chosen. The algorithm includes an extra iteration to compute the final clustering radius r_k (and the next center c_{k+1}).

```
GonzalezKCenter(P, k):
  for i ← 1 to n
      d_i ← ∞
  c_1 ← p_1
  for j ← 1 to k
      r_j ← 0
      for i ← 1 to n
          d_i ← min{d_i, |p_i c_j|}
          if r_j < d_i
              r_j ← d_i;  c_{j+1} ← p_i
  return {c_1, c_2, ..., c_k}
```

GonzalezKCenter clearly runs in $O(nk)$ time. Using more advanced data structures, Tomas Feder and Daniel Greene[6] described an algorithm to compute exactly the same clustering in only $O(n \log k)$ time.

Theorem 7. GonzalezKCenter *computes a 2-approximation to the optimal k-center clustering.*

Proof: Let *OPT* denote the optimal k-center clustering radius for P. For any index i, let c_i and r_i denote the ith center point and ith clustering radius computed by GonzalezKCenter.

By construction, each center point c_j has distance at least r_{j-1} from any center point c_i with $i < j$. Moreover, for any $i < j$, we have $r_i \geq r_j$. Thus, $|c_i c_j| \geq r_k$ for all indices i and j.

On the other hand, at least one cluster in the optimal clustering contains at least two of the points $c_1, c_2, \ldots, c_{k+1}$. Thus, by the triangle inequality, we must have $|c_i c_j| \leq 2 \cdot OPT$ for some indices i and j. We conclude that $r_k \leq 2 \cdot OPT$, as claimed. □

*31.9 Approximation Schemes

With just a little more work, we can compute an arbitrarily close approximation of the optimal k-clustering, using a so-called *approximation scheme*. An approximation scheme accepts both an instance of the problem and a value $\varepsilon > 0$ as input, and it computes a $(1+\varepsilon)$-approximation of the optimal output for that instance. As I mentioned earlier, computing even a 1.8-approximation is NP-hard, so we cannot expect our approximation scheme to run in polynomial time; nevertheless, at least for small values of k, the approximation scheme will be considerably more efficient than any exact algorithm.

Our approximation scheme works in three phases:

[6] Tomas Feder* and Daniel H. Greene. Optimal algorithms for approximate clustering. *Proc. 20th STOC*, 1988. Unlike Gonzalez's algorithm, Feder and Greene's faster algorithm does not work over arbitrary metric spaces; it requires that the input points come from some $\mathbb{R}^d$ and that distances are measured in some L_p metric. The time analysis also assumes that the distance between any two points can be computed in $O(1)$ time.

1. Compute a 2-approximate clustering of the input set P using GONZALEZKCENTER. Let r be the cost of this clustering.

2. Create a regular grid of squares of width $\delta = \varepsilon r/2\sqrt{2}$. Let Q be a subset of P containing one point from each non-empty cell of this grid.

3. Compute an *optimal* set of k centers for Q. Return these k centers as the approximate k-center clustering for P.

The first phase requires $O(nk)$ time. By our earlier analysis, we have $r^* \le r \le 2r^*$, where r^* is the optimal k-center clustering cost for P.

The second phase can be implemented in $O(n)$ time using a hash table, or in $O(n \log n)$ time by standard sorting, by associating approximate coordinates $(\lfloor x/\delta \rfloor, \lfloor y/\delta \rfloor)$ to each point $(x, y) \in P$ and removing duplicates. The key observation is that the resulting point set Q is significantly smaller than P. We know P can be covered by k balls of radius r^*, each of which touches $O(r^*/\delta^2) = O(1/\varepsilon^2)$ grid cells. It follows that $|Q| = O(k/\varepsilon^2)$.

Let $T(n, k)$ be the running time of an *exact* k-center clustering algorithm, given n points as input. If this were a computational geometry class, we might see a "brute force" algorithm that runs in time $T(n, k) = O(n^{k+2})$; the fastest algorithm currently known[7] runs in time $T(n, k) = n^{O(\sqrt{k})}$. If we use this algorithm, our third phase requires $(k/\varepsilon^2)^{O(\sqrt{k})}$ time.

It remains to show that the optimal clustering for Q implies a $(1 + \varepsilon)$-approximation of the optimal clustering for P. Suppose the optimal clustering of Q consists of k balls $B_1, B_2, \ldots, B_k$, each of radius $\tilde{r}$. Clearly $\tilde{r} \le r^*$, since any set of k balls that cover P also cover any subset of P. Each point in $P \setminus Q$ shares a grid cell with some point in Q, and therefore is within distance $\delta\sqrt{2}$ of some point in Q. Thus, if we increase the radius of each ball B_i by $\delta\sqrt{2}$, the expanded balls must contain every point in P. We conclude that the optimal centers for Q gives us a k-center clustering for P of cost at most $r^* + \delta\sqrt{2} \le r^* + \varepsilon r/2 \le r^* + \varepsilon r^* = (1 + \varepsilon)r^*$.

The total running time of the approximation scheme is $O(nk + (k/\varepsilon^2)^{O(\sqrt{k})})$. This is still exponential in the input size if k is large (say $\sqrt{n}$ or $n/100$), but if k and ε are fixed constants, the running time is linear in the number of input points.

*31.10 An FPTAS for Subset Sum

An approximation scheme whose running time, for any fixed ε, is polynomial in n is called a *polynomial-time approximation scheme* or *PTAS* (usually pronounced "pee taz"). If in addition the running time depends only polynomially on ε, the algorithm is called a *fully* polynomial-time approximation scheme or *FPTAS* (usually pronounced "eff pee taz"). For example, an approximation scheme with running time $O(n^2/\varepsilon^2)$ is an FPTAS; an approximation scheme with running time $O(n^{1/\varepsilon^6})$ is a PTAS but not an FPTAS; and our approximation scheme for k-center clustering is not a PTAS.

The last problem we'll consider is the SUBSETSUM problem: Given a set X containing n positive integers and a target integer t, determine whether X has a subset whose elements sum to t. The lecture notes on NP-completeness include a proof that SUBSETSUM is NP-hard. As stated, this problem doesn't allow any sort of approximation—the answer is either TRUE or FALSE.[8] So we will consider a related optimization problem instead: Given set X and integer t, find the subset of X whose sum is as large as possible but no larger than t.

[7] R. Z. Hwang, R. C. T. Lee, and R. C. Chan. The slab dividing approach to solve the Euclidean p-center problem. *Algorithmica* 9(1):1–22, 1993.

[8] Do, or do not. There is no 'try'. (Are old one thousand when years you, alphabetical also in order talk will you.)

We have already seen a dynamic programming algorithm to solve the decision version SUBSETSUM in time $O(nt)$; a similar algorithm solves the optimization version in the same time bound. Here is a different algorithm, whose running time does not depend on t:

```
SUBSETSUM(X[1..n], t):
  S_0 ← {0}
  for i ← 1 to n
      S_i ← S_{i-1} ∪ (S_{i-1} + X[i])
      remove all elements of S_i bigger than t
  return max S_n
```

Here $S_{i-1} + X[i]$ denotes the set $\{s + X[i] \mid s \in S_{i-1}\}$. If we store each S_i in a sorted array, the ith iteration of the for-loop requires time $O(|S_{i-1}|)$. Each set S_i contains all possible subset sums for the first i elements of X; thus, S_i has at most 2^i elements. On the other hand, since every element of S_i is an integer between 0 and t, we also have $|S_i| \leq t + 1$. It follows that the total running time of this algorithm is $\sum_{i=1}^{n} O(|S_{i-1}|) = O(\min\{2^n, nt\})$.

Of course, this is only an estimate of worst-case behavior. If several subsets of X have the same sum, the sets S_i will have fewer elements, and the algorithm will be faster. The key idea for finding an approximate solution quickly is to 'merge' nearby elements of S_i—if two subset sums are nearly equal, ignore one of them. On the one hand, merging similar subset sums will introduce some error into the output, but hopefully not too much. On the other hand, by reducing the size of the set of sums we need to maintain, we will make the algorithm faster, hopefully significantly so.

Here is our approximation algorithm. We make only two changes to the exact algorithm: an initial sorting phase and an extra FILTERing step inside the main loop.

```
FILTER(Z[1..k], δ):
  SORT(Z)
  j ← 1
  Y[j] ← Z[i]
  for i ← 2 to k
      if Z[i] > (1 + δ) · Y[j]
          j ← j + 1
          Y[j] ← Z[i]
  return Y[1..j]
```

```
APPROXSUBSETSUM(X[1..n], k, ε):
  SORT(X)
  R_0 ← {0}
  for i ← 1 to n
      R_i ← R_{i-1} ∪ (R_{i-1} + X[i])
      R_i ← FILTER(R_i, ε/2n)
      remove all elements of R_i bigger than t
  return max R_n
```

Theorem 8. APPROXSUBSETSUM *returns a* $(1+\varepsilon)$*-approximation of the optimal subset sum, given any* ε *such that* $0 < \varepsilon \leq 1$.

Proof: The theorem follows from the following claim, which we prove by induction:

> For any element $s \in S_i$, there is an element $r \in R_i$ such that $r \leq s \leq r \cdot (1 + \varepsilon n/2)^i$.

The claim is trivial for $i = 0$. Let s be an arbitrary element of S_i, for some $i > 0$. There are two cases to consider: either $x \in S_{i-1}$, or $x \in S_{i-1} + x_i$.

(1) Suppose $s \in S_{i-1}$. By the inductive hypothesis, there is an element $r' \in R_{i-1}$ such that $r' \leq s \leq r' \cdot (1 + \varepsilon n/2)^{i-1}$. If $r' \in R_i$, the claim obviously holds. On the other hand, if $r' \notin R_i$, there must be an element $r \in R_i$ such that $r < r' \leq r(1 + \varepsilon n/2)$, which implies that

$$r < r' \leq s \leq r' \cdot (1 + \varepsilon n/2)^{i-1} \leq r \cdot (1 + \varepsilon n/2)^i,$$

so the claim holds.

(2) Suppose $s \in S_{i-1} + x_i$. By the inductive hypothesis, there is an element $r' \in R_{i-1}$ such that $r' \le s - x_i \le r' \cdot (1 + \varepsilon n/2)^{i-1}$. If $r' + x_i \in R_i$, the claim obviously holds. On the other hand, if $r' + x_i \notin R_i$, there must be an element $r \in R_i$ such that $r < r' + x_i \le r(1 + \varepsilon n/2)$, which implies that

$$\begin{aligned} r < r' + x_i \le s &\le r' \cdot (1 + \varepsilon n/2)^{i-1} + x_i \\ &\le (r - x_i) \cdot (1 + \varepsilon n/2)^i + x_i \\ &\le r \cdot (1 + \varepsilon n/2)^i - x_i \cdot ((1 + \varepsilon n/2)^i - 1) \\ &\le r \cdot (1 + \varepsilon n/2)^i. \end{aligned}$$

so the claim holds.

Now let $s^* = \max S_n$ and $r^* = \max R_n$. Clearly $r^* \le s^*$, since $R_n \subseteq S_n$. Our claim implies that there is some $r \in R_n$ such that $s^* \le r \cdot (1 + \varepsilon/2n)^n$. But r cannot be bigger than r^*, so $s^* \le r^* \cdot (1 + \varepsilon/2n)^n$. The inequalities $e^x \ge 1 + x$ for all x, and $e^x \le 2x + 1$ for all $0 \le x \le 1$, imply that $(1 + \varepsilon/2n)^n \le e^{\varepsilon/2} \le 1 + \varepsilon$. □

Theorem 9. APPROXSUBSETSUM *runs in* $O((n^3 \log n)/\varepsilon)$ *time.*

Proof: Assuming we keep each set R_i in a sorted array, we can merge the two sorted arrays R_{i-1} and $R_{i-1} + x_i$ in $O(|R_{i-1}|)$ time. FILTERin R_i and removing elements larger than t also requires only $O(|R_{i-1}|)$ time. Thus, the overall running time of our algorithm is $O(\sum_i |R_i|)$; to express this in terms of n and ε, we need to prove an upper bound on the size of each set R_i.

Let $\delta = \varepsilon/2n$. Because we consider the elements of X in increasing order, every element of R_i is between 0 and $i \cdot x_i$. In particular, every element of $R_{i-1} + x_i$ is between x_i and $i \cdot x_i$. After FILTERing, at most one element $r \in R_i$ lies in the range $(1 + \delta)^k \le r < (1 + \delta)^{k+1}$, for any k. Thus, at most $\lceil \log_{1+\delta} i \rceil$ elements of $R_{i-1} + x_i$ survive the call to FILTER. It follows that

$$\begin{aligned} |R_i| &= |R_{i-1}| + \left\lceil \frac{\log i}{\log(1+\delta)} \right\rceil \\ &\le |R_{i-1}| + \left\lceil \frac{\log n}{\log(1+\delta)} \right\rceil && [i \le n] \\ &\le |R_{i-1}| + \left\lceil \frac{2 \ln n}{\delta} \right\rceil && [e^x \le 1 + 2x \text{ for all } 0 \le x \le 1] \\ &\le |R_{i-1}| + \left\lceil \frac{n \ln n}{\varepsilon} \right\rceil && [\delta = \varepsilon/2n] \end{aligned}$$

Unrolling this recurrence into a summation gives us the upper bound $|R_i| \le i \cdot \lceil (n \ln n)/\varepsilon \rceil = O((n^2 \log n)/\varepsilon)$.

We conclude that the overall running time of APPROXSUBSETSUM is $O((n^3 \log n)/\varepsilon)$, as claimed. □

Exercises

1. (a) Prove that for any set of jobs, the makespan of the greedy assignment is at most $(2 - 1/m)$ times the makespan of the optimal assignment, where m is the number of machines.
   (b) Describe a set of jobs such that the makespan of the greedy assignment is exactly $(2 - 1/m)$ times the makespan of the optimal assignment, where m is the number of machines.
   (c) Describe an efficient algorithm to solve the minimum makespan scheduling problem *exactly* if every processing time $T[i]$ is a power of two.

2. (a) Find the smallest graph (minimum number of edges) for which GREEDYVERTEXCOVER does not return the smallest vertex cover.
   (b) For any integer n, describe an n-vertex graph for which GREEDYVERTEXCOVER returns a vertex cover of size $OPT \cdot \Omega(\log n)$.

3. (a) Find the smallest graph (minimum number of edges) for which DUMBVERTEXCOVER does not return the smallest vertex cover.
   (b) Describe an infinite family of graphs for which DUMBVERTEXCOVER returns a vertex cover of size $2 \cdot OPT$.

4. Consider the following heuristic for constructing a vertex cover of a connected graph G: return the set of non-leaf nodes in any depth-first spanning tree of G.
   (a) Prove that this heuristic returns a vertex cover of G.
   (b) Prove that this heuristic returns a 2-approximation to the minimum vertex cover of G.
   (c) Describe an infinite family of graphs for which this heuristic returns a vertex cover of size $2 \cdot OPT$.

5. Consider the following optimization version of the PARTITION problem. Given a set X of positive integers, our task is to partition X into disjoint subsets A and B such that $\max\{\sum A, \sum B\}$ is as small as possible. This problem is clearly NP-hard. Determine the approximation ratio of the following polynomial-time approximation algorithm. Prove your answer is correct.

```
PARTITION(X[1..n]):
  Sort X in increasing order
  a ← 0;  b ← 0
  for i ← 1 to n
      if a < b
          a ← a + X[i]
      else
          b ← b + X[i]
  return max{a, b}
```

6. The *chromatic number* $\chi(G)$ of a graph G is the minimum number of colors required to color the vertices of the graph, so that every edge has endpoints with different colors. Computing the chromatic number exactly is NP-hard.

Prove that the following problem is also NP-hard: Given an n-vertex graph G, return any integer between $\chi(G)$ and $\chi(G)+573$. *[Note: This does not contradict the possibility of a constant* ***factor*** *approximation algorithm.]*

7. Let $G = (V, E)$ be an undirected graph, each of whose vertices is colored either red, green, or blue. An edge in G is *boring* if its endpoints have the same color, and *interesting* if its endpoints have different colors. The *most interesting 3-coloring* is the 3-coloring with the maximum number of interesting edges, or equivalently, with the fewest boring edges. Computing the most interesting 3-coloring is NP-hard, because the standard 3-coloring problem is a special case.

   (a) Let $zzz(G)$ denote the number of boring edges in the most interesting 3-coloring of a graph G. Prove that it is NP-hard to approximate $zzz(G)$ within a factor of $10^{10^{100}}$.

   (b) Let $wow(G)$ denote the number of interesting edges in the most interesting 3-coloring of G. Suppose we assign each vertex in G a *random* color from the set $\{\text{red}, \text{green}, \text{blue}\}$. Prove that the expected number of interesting edges is at least $\frac{2}{3}wow(G)$.

8. Consider the following algorithm for coloring a graph G.

```
TreeColor(G):
  T ← any spanning tree of G
  Color the tree T with two colors
  c ← 2

  for each edge (u, v) ∈ G \ T
      T ← T ∪ {(u, v)}

      if color(u) = color(v)    ⟨⟨Try recoloring u with an existing color⟩⟩
          for i ← 1 to c
              if no neighbor of u in T has color i
                  color(u) ← i

      if color(u) = color(v)    ⟨⟨Try recoloring v with an existing color⟩⟩
          for i ← 1 to c
              if no neighbor of v in T has color i
                  color(v) ← i

      if color(u) = color(v)    ⟨⟨Give up and create a new color⟩⟩
          c ← c + 1
          color(u) ← c
```

   (a) Prove that this algorithm colors any bipartite graph with just two colors.

   (b) Let $\Delta(G)$ denote the maximum degree of any vertex in G. Prove that this algorithm colors any graph G with at most $\Delta(G)$ colors. This trivially implies that TreeColor is a $\Delta(G)$-approximation algorithm.

   (c) Prove that TreeColor is *not* a constant-factor approximation algorithm.

9. The Knapsack problem can be defined as follows. We are given a finite set of elements X where each element $x \in X$ has a non-negative *size* and a non-negative *value*, along with an integer *capacity* c. Our task is to determine the maximum total value among all subsets of X whose total size is at most c. This problem is NP-hard. Specifically, the optimization version of SubsetSum is a special case, where each element's value is equal to its size.

Determine the approximation ratio of the following polynomial-time approximation algorithm. Prove your answer is correct.

```
APPROXKNAPSACK(X, c):
  return max{GREEDYKNAPSACK(X, c), PICKBESTONE(X, c)}
```

```
GREEDYKNAPSACK(X, c):
  Sort X in decreasing order by the ratio value/size
  S ← 0; V ← 0
  for i ← 1 to n
      if S + size(x_i) > c
          return V
      S ← S + size(x_i)
      V ← V + value(x_i)
  return V
```

```
PICKBESTONE(X, c):
  Sort X in increasing order by size
  V ← 0
  for i ← 1 to n
      if size(x_i) > c
          return V
      if value(x_i) > V
          V ← value(x_i)
  return V
```

10. In the *bin packing* problem, we are given a set of n items, each with weight between 0 and 1, and we are asked to load the items into as few bins as possible, such that the total weight in each bin is at most 1. It's not hard to show that this problem is NP-Hard; this question asks you to analyze a few common approximation algorithms. In each case, the input is an array $W[1..n]$ of weights, and the output is the number of bins used.

```
NEXTFIT(W[1..n]):
  b ← 0
  Total[0] ← ∞

  for i ← 1 to n
      if Total[b] + W[i] > 1
          b ← b + 1
          Total[b] ← W[i]
      else
          Total[b] ← Total[b] + W[i]

  return b
```

```
FIRSTFIT(W[1..n]):
  b ← 0

  for i ← 1 to n
      j ← 1; found ← FALSE

      while j ≤ b and found = FALSE
          if Total[j] + W[i] ≤ 1
              Total[j] ← Total[j] + W[i]
              found ← TRUE
          j ← j + 1

      if found = FALSE
          b ← b + 1
          Total[b] = W[i]

  return b
```

(a) Prove that NEXTFIT uses at most twice the optimal number of bins.

(b) Prove that FIRSTFIT uses at most twice the optimal number of bins.

*(c) Prove that if the weight array W is initially sorted in decreasing order, then FIRSTFIT uses at most $(4 \cdot OPT + 1)/3$ bins, where OPT is the optimal number of bins. The following facts may be useful (but you need to prove them if your proof uses them):

- In the packing computed by FIRSTFIT, every item with weight more than $1/3$ is placed in one of the first OPT bins.
- FIRSTFIT places at most $OPT - 1$ items outside the first OPT bins.

11. Given a graph G with edge weights and an integer k, suppose we wish to partition the the vertices of G into k subsets $S_1, S_2, \ldots, S_k$ so that the sum of the weights of the edges that cross the partition (that is, have endpoints in different subsets) is as large as possible.

(a) Describe an efficient $(1 - 1/k)$-approximation algorithm for this problem.

(b) Now suppose we wish to minimize the sum of the weights of edges that do *not* cross the partition. What approximation ratio does your algorithm from part (a) achieve for the new problem? Justify your answer.

12. The lecture notes describe a $(3/2)$-approximation algorithm for the metric traveling salesman problem. Here, we consider computing minimum-cost Hamiltonian *paths*. Our input consists of a graph G whose edges have weights that satisfy the triangle inequality. Depending upon the problem, we are also given zero, one, or two endpoints.

(a) If our input includes zero endpoints, describe a $(3/2)$-approximation to the problem of computing a minimum cost Hamiltonian path.

(b) If our input includes one endpoint u, describe a $(3/2)$-approximation to the problem of computing a minimum cost Hamiltonian path that starts at u.

(c) If our input includes two endpoints u and v, describe a $(5/3)$-approximation to the problem of computing a minimum cost Hamiltonian path that starts at u and ends at v.

13. Suppose we are given a collection of n *jobs* to execute on a machine containing a row of m processors. When the ith job is executed, it occupies a *contiguous* set of $prox[i]$ processors for $time[i]$ seconds. A *schedule* for a set of jobs assigns each job an interval of processors and a starting time, so that no processor works on more than one job at any time. The *makespan* of a schedule is the time from the start to the finish of all jobs. Finally, the *parallel scheduling problem* asks us to compute the schedule with the smallest possible makespan.

(a) Prove that the parallel scheduling problem is NP-hard.

(b) Give an algorithm that computes a 3-approximation of the minimum makespan of a set of jobs in $O(m \log m)$ time. That is, if the minimum makespan is M, your algorithm should compute a schedule with make-span at most $3M$. You can assume that n is a power of 2.

14. Consider the greedy algorithm for metric TSP: start at an arbitrary vertex u, and at each step, travel to the closest unvisited vertex.

(a) Show that the greedy algorithm for metric TSP is an $O(\log n)$-approximation, where n is the number of vertices. *[Hint: Argue that the kth least expensive edge in the tour output by the greedy algorithm has weight at most $OPT/(n - k + 1)$; try $k = 1$ and $k = 2$ first.]*

*(b) Show that the greedy algorithm for metric TSP is no better than an $O(\log n)$-approximation. That is, describe an infinite family of weighted graphs such that the greedy algorithm returns a cycle whose weight is $\Omega(\log n)$ times the optimal TSP tour.