Some Hardness Proofs

Magnus Lie Hetland

January 2011

This is a very brief overview of some well-known hard (NP Hard and NP complete) problems, and the main ideas behind their hardness proofs. The document is based in Chapter 11 of my book, *Python Algorithms* (Apress, 2010).

Hardness by Reduction

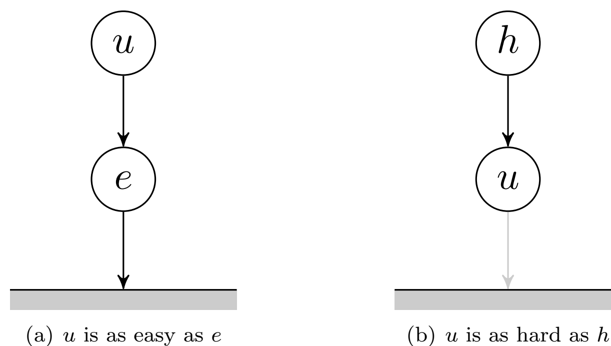
The basis for hardness here is the idea of *reductions*. Reducing problem X to problem Y means that you can solve X *by means of* Y. If Y is easy, then that rubs off on X. Conversely, if X is hard, that rubs off on Y. As long as the reduction (the "by means of" transformation) doesn't do any heavy lifting, there's no way Y can be *easier* than X. The mere fact that you can use Y to solve X would then lead to a contradiction.

Figure 1 illustrates the idea. If solving a problem is visualized as reducing it "to ground," it's all a matter of finding paths. The edges (reductions) are assumed to be without any appreciable cost. In the first subfigure, the unknown problem u must clearly be as easy as e. The fact that we can find our way to ground from e means that we have also (transitively) solved u.

The converse situation is shown in the second subfigure. Here, we use u as the linchpin in a potential (grayed out) solution for the known-to-be-hard problem h. We might, for example, know that it is *impossible* to solve h in polynomial time (e.g., the Towers of Hanoi), or even that it can't be solved *at all* (e.g., the Halting problem). Somehow finding an easy way to solve u won't change the status of h; it will lead to a *contradiction*. That means that such a solution *cannot exist*.

We could also have a situation where we *believe* that h is hard. Establishing the reduction $h \to u$ will still establish u as *at least as hard* as h (by the same logic). One important idea here is that of *completeness*. Within a class of problems, we can establish the set of *complete* problems—the set of problems that are *at least as hard as all the others in the class*. We don't necessarily know how hard those problems are. In cases where such absolute hardness can be (extremely) difficult to establish, the relative hardness of complete problems can still be highly useful.

The idea is illustrated in Figure 2. Problem c is complete for the class of problems under consideration. As can be seen in the first subfigure, all other problems can be reduced to c; whether c itself can be solved is unknown, but if it *can* be, so can all the others. If the class is very large, that implication in itself might support the hypothesis that c cannot be solved. The second



(a) u is as easy as e (b) u is as hard as h

Figure 1: The two directions of reduction. Reducing from an unknown problem u to a known, easy one e, or reducing from a known, hard one h to u.

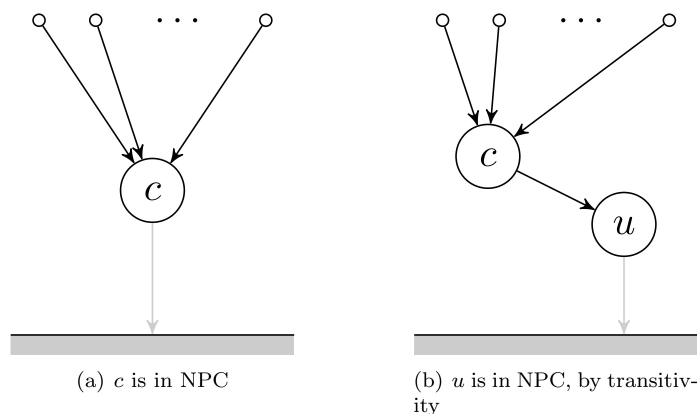
subfigure shows how completeness rubs off by reduction, because reductions are transitive.

In this document, I'm talking about the class of NP: decision problems that can be solved in polynomial time using a nondeterministic Turing machine (or those for whose yes-instances there exist polynomial-size certificates that can be verified in polynomial time by a deterministic Turing machine). This is a huge class, encompassing most decision problems of any practical interest. Completeness in NP is defined under polynomial-time reductions, which means that if there is any overlap between NPC (the class of NP complete problems) and P (decision problems solvable in polynomial time), then P = NP. No polynomial solutions have as yet been found for any problems in NPC, and it is almost universally assumed that this is not possible (that is, P $\neq$ NP).

While decision problems are useful for formal reasoning, we usually work with more general problem formulations in practice. If we use only "half" the definition of NP completeness—the reduction part—we end up with *NP hardness*. Any problem in NP can be reduced to any NP hard problem in polynomial time, but we do not require NP hard problems to actually be in NP. This means that we're allowed to include all kinds of optimization problems and the like as well.

In the following, I won't distinguish too clearly between NP completeness and NP hardness, but simply describing problems a hard. This means that they are NP hard. For any decision problems, though, it also means they are NP complete. In many cases, the same problem can even be formulated both as a search/optimization problem ("What is the shortest path from s to t?") and as a decision problem ("Is there a path from s to t with a length less than or equal to d?"), both of which are hard. In the interest of brevity, I have chosen to be a bit ambiguous in this area, leaving the details as an exercise for the reader.

One final (and important) note about these reduction-based proofs: you need to make sure that the reduction covers all the cases. To establish equivalence between two decision problems, you need to show implication *both ways*: instances of problem A have yes-answers if *and only if* the corresponding instances (according to the reduction) of problem B have. This does *in no way* mean that the reduction goes both ways, however. (Do you see the difference?)



(a) c is in NPC

(b) u is in NPC, by transitivity

Figure 2: Reducing from an NP complete problem to any NP problem confers NP completeness.

Getting Started

The satisfaction problem, or SAT, involves assigning truth values to a set of Boolean variables so that a given formula is satisfied (that is, evaluates to *true*).

Fact 1. *SAT is NP complete.*

This is basically the Cook–Levin theorem. I'm not going to give a full proof for it here, as it is rather involved. The general idea is to simulate nondeterministic Turing machines using the machinery of Boolean logic. Let A be any problem in NP that we're trying to reduce to SAT. Let's say that M is a nondeterministic Turing machine that solves A. When given some input for A, we then create a Boolean formula that *simulates the computation of M*. That is, our formula is satisfiable if and only if M accepts the input. Think of the Boolean variables as representing choices made by M, and the value of the formula as the output of M. This kind of simulation is, in fact, possible in general. (See, e.g., *Introduction to Algorithms* by Cormen et al., 3rd ed., § 34.3, or the Wikipedia entry on the Cook–Levin theorem.[1])

A related problem is Circuit SAT, where a Boolean circuit is used, rather than a formula.

Fact 2. *Circuit SAT is NP complete.*

Proof sketch. Using a logical circuit rather than a Boolean formula doesn't really change much. Any Boolean formula can be expressed directly as a logical circuit. □

Reduction in the other direction (which is not required here) isn't *quite* so straightforward, but still pretty simple. For example, introduce a Boolean variable for each gate in the circuit, and add a bidirectional implication between that variable and the expression it represents. For example, an AND gate with inputs x, y and z might get the variable a, and we'd encoded it in our formula

[1] `http://en.wikipedia.org/w/index.php?title=Cook-Levin_theorem&oldid=395479548`

as $(a \leftrightarrow (x \wedge y \wedge z))$. We can then use a in other subformulas, as the output of the AND gate is routed as input to other gates. (The total formula is then a conjunction of all these subformulas, as well as the final output variable on its own.)

Next: A useful simplification. Recall that any Boolean formula can be rewritten in disjunctive or conjunctive normal form (DNF and CNF, respectively). We now wish to work with a CNF version where each clause has exactly k distinct (where a literal is either a variable or a negated variable).

Fact 3. *k-CNF-SAT is NP complete for $k > 2$.*

Proof sketch. All that's needed is to show that this holds for $k = 3$, which relies that any Boolean formula can be expressed in 3-CNF form without exponential blowup in size. First, make a "parse tree" where each operator has at most two children. Add a variable for each node, and link it to the (binary) subexpression with a double implication. So, for example, if the node representing $(x \to y)$ is given the name z, we create the new (sub)formula $(z \leftrightarrow (x \to y))$. The new variables are used to represent the subexpressions in their parent nodes. So, if we originally had the subexpression $(a \wedge (x \to y))$, that now becomes $(a \wedge z)$ (which, again, receives its own node variable—b, say—and becomes $(b \leftrightarrow (a \wedge z)))$. Thus we're able to encode an arbitrary Boolean formula as a conjunction of subformulas with at most three variables.

For each of these subformulas, build a truth table. Use rows that evaluate to 0 to build a formula in DNF form of the complement of what we're looking for. Then negate that, converting it to CNF using De Morgan's law (complement literals and change $\wedge$ to $\vee$ and vice versa). Now we need to make sure each clause has exactly three (distinct) literals. First, introduce two auxiliary variables, p and q. For each clause we've got so far:

- If it has 3 distinct literals, just use it in the global conjunction.
- If it has 2 distinct literals, that is, it is of the form $(A \vee B)$, rewrite as

$$(A \vee B \vee p) \wedge (A \vee B \vee \neg p) .$$

- If it has only 1 literal A, rewrite as

$$(A \vee p \vee q) \wedge (A \vee p \vee \neg q) \wedge (A \vee \neg p \vee q) \wedge (A \vee \neg p \vee \neg q) .$$

We now have a 3-CNF formula that is satisfiable if and only if our original formula is. The question is: can this reduction be performed in polynomial time? First, when representing our parse tree, we introduce one variable per connective. Then, for each clause in the resulting formula, we introduce at most 8 clauses (from our truth tables) in the next step. In the final step, we introduce at most 4 clauses for each clause. In other words, the final formula is of polynomial size, and each of the (constant number of) transformations can easily be performed in polynomial time. □

Now, let's take a step beyond Boolean formulas. Consider the problem of finding a Hamilton cycle (or determining whether one exists). A Hamilton cycle for a graph $G = (V, E)$ is a cycle in G (that is, a subgraph of G) that contains (visits) every node in V.

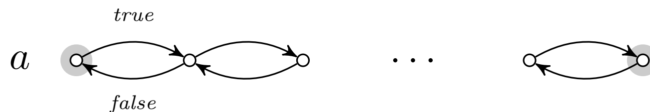


Figure 3: Representing the variable a by a "row." A Hamilton cycle must either pass entirely from left to right (meaning that a is true) or right to left (a is false).

Fact 4. *The Hamilton cycle problem for directed graphs is hard.*

Proof sketch. By reduction from the 3-CNF-SAT problem.[2] Represent each variable A as a row of nodes, as shown in Figure 3. The number of nodes is not crucial, as long as we have enough. (We can just insert nodes as needed, during the rest of the reduction.) We let the *left-to-right* and *right-to-left* directions represent *true* and *false*, respectively. That is, if the cycle goes through our row from left to right, the variable A is true (and vice versa).

Figure 4 shows how we connect two variables (rows) A and B. We just link up the ends of the rows so that the cycle can go from either end of A to either end of B. We link all variables like this in a series, making sure to link the last variable to the first. Once we have finished this setup, it's clear that for n variables we 2^n possible (directed) Hamilton cycles, each representing a truth assignment. Now, it's time to encode the actual formula (thereby restricting the valid/satisfying truth assignments—or cycles).

For each clause in our 3-CNF formula, we introduce a node. In order to satisfy the formula, each clause must be satisfied. Correspondingly, every clause node must be visited. We link these nodes to our variable rows as shown in Figure 5. The clause in question is $(a \vee \neg b)$. (The third literal has been omitted to simplify the figure; adding more literals does not change the line of reasoning.) In order to satisfy this clause, either a must be true, b must be false, or both. Equivalently, either the Hamilton cycle must go from left to right in (and take a detour via the node from) row a, or from right to left in row b, or both.

Once all these clause nodes have been added, there will be a Hamilton cycle if and only if the original formula is satisfiable. The reduction is clearly polynomial. □

Note that the reduction works both for the decision and search versions of the problems. Detecting a Hamilton cycle lets you decide satisfiability; *finding* a Hamilton cycle lets you *find* a satisfying truth assignment for your formula.

A Bunch of Problems

Now that we have the basic problem (SAT) in place, as well as a rather involved reduction example (SAT → Hamilton cycle), let's zoom through a bunch of other problems. The first one, the partition problem, is so seemingly simple, it makes it easy to show that lots of other problems are hard. The partition problem simply involves partitioning a set of positive integers into two sets of equal sums (or, in the optimization version, as equal as possible).

[2]It actually works quite easily for more the more general CNF-SAT problem.

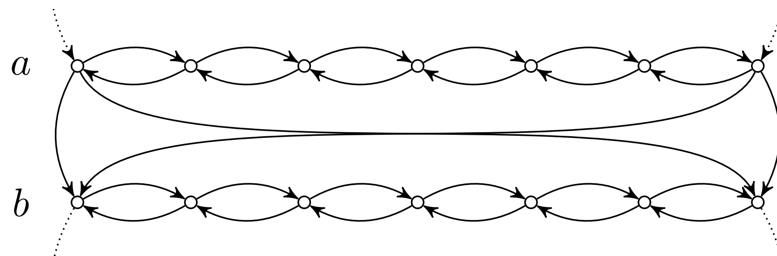


Figure 4: Two rows (variables), linked so that the Hamilton cycle is free to choose either direction (that is, truth value) when moving from one variable to the next.

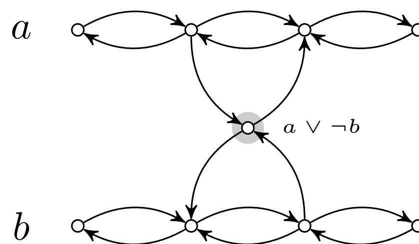


Figure 5: The clause $(a \vee \neg b)$, represented as an added node (highlighted), with detours requiring a to be true (going from left to right) and b to be false (right to left) in order to satisfy the clause (visit the node).

Fact 5. *The partition problem is hard.*

Proving this is a bit tricky. For details, see *Computers and Intractability*, 24th printing, by Garey and Johnson (W. H. Freeman and Company, 2003). For now, let's just use the problem as the basis for some other easy proofs.

Bin packing involves packing items of sizes up to k into bins of size k. We either want to know whether a given set of bins is enough, or (in the optimization version) try to minimize the number of bins.

Fact 6. *Bin packing is hard.*

Proof sketch. Reduction from partition. Let S be the sum of the numbers, and see if you can pack them into two bins of size $S/2$. □

The subset sum problem involves finding a subset of a set of integers that sums to a given value. This value is sometimes arbitrarily set to 0. Also, the set of values is sometimes restricted to positive integers.[3] Let's consider the latter version. (Including negative numbers only generalizes the problem, and fixing the sum to 0 involves only a minor transform.)

Fact 7. *The subset sum problem is hard.*

Proof sketch. Reduction from partition. Let S be the sum of the numbers, and see if you can find a subset that sums to $S/2$. □

The 0-1 knapsack problem involves a set of objects, each with a weight and a value. We are to select a subset of these. The total weight of the subset is constrained, and we want to optimize the value (or achieve a given minimum value).

Fact 8. *The 0-1 Knapsack problem is hard.*

Proof sketch. Reduction from subset sum. For each number x, create an object with weight and value equal to x. Set the knapsack capacity to the desired sum. □

The bounded integral knapsack problem is a version where you have object *types*, rather than individual objects, meaning that you can use a limited number of objects from each types (rather than just 0 or 1).

Fact 9. *The bounded integral knapsack problem is hard.*

Proof sketch. Just set the limit for each object type to 1. □

It is also easy to construct a reduction in the other direction here.

The *unbounded* integral knapsack problem, however, is a bit different. Here, you're allowed to take as many objects as you wish from each type. If we had an unbounded version of the subset sum problem—one where each number could be used more than once—we could use a reduction like before. Let this unbounded subset sum problem be one where all the numbers $w_1 \ldots w_n$ are positive, and we're trying to achieve the sum k by using each w_i zero or more times.

Fact 10. *The unbounded subset sum problem is hard.*

[3] There is, of course, no point in adding *both* of these restrictions...

Proof sketch. We want to set up an unbounded problem where each number is still only used once, accurately simulating the bounded case. Let's focus on the indexes of the numbers at first, ignoring the actual values. For n indexes we try to create n "slots" using powers of two; we represent the slot for index i by 2^i. If we used $\sum_{i=1}^{n} 2^i$ as capacity, though, our optimization wouldn't care if we had one instance of 2^n or two instances of 2^{n-1}, and we want to make sure we have at most one of each. We add another constraint, representing i by $2^i + 2^{n+1}$, and setting the capacity to $\sum_{i=1}^{n} 2^i + n2^{n+1}$. It will still pay to fill every slot from 1 to n when maximizing, but we can now include only n occurrences of 2^{n+1}, so a single instance of 2^n is better than two instances of 2^{n-1}.

If we stop here, we can force the optimization to include exactly one of each number, which isn't really what we want. We need two versions of each item: one representing inclusion and one, exclusion. If number i is included, we will add its value x_i, and if it's excluded, we add 0. We also have the original capacity k to deal with. These constraints are vital, but still subordinate to the ones enforcing a single item per slot. To enforce this, we want to have two "digits" in our representation, with the most significant digit representing the slots, and the least significant representing the weights/values we're maximizing. We implement this by multiplying the most significant part by a huge number (which acts as the radix). If the largest number (in the original subset sum problem) is $x_{\max}$, we can multiply by $nx_{\max}$, and we'll be safe. We then represent the number x_i from the original problem by the following two numbers, representing inclusion and exclusion, respectively:

$$(2^{n+1} + 2^i)nx_{\max} + x_i$$
$$(2^{n+1} + 2^i)nx_{\max}$$

The capacity becomes $(n2^{n+1} + \sum_{i=1}^{n})nx_{\max} + k$. □

Fact 11. *The unbounded integral knapsack problem is hard.*

Proof sketch. Reduction from unbounded subset sum. As in the bounded case, we can simply set value equal to weight, and use that to emulate the numbers. Finding a subset with the correct sum is then equivalent to filling up the knapsack exactly. □

Integer programming is linear programming where the variables are integers. (Mixed integer programming also involves reals, and is clearly just as hard.)

Fact 12. *Integer programming is hard.*

Proof sketch. Any of the knapsack or subsets sum problems could easily be solved by integer programming. □

Fact 13. *0-1 Integer programming is hard.*

Proof sketch. Just reduce from subset sum or 0-1 knapsack. □

A k-coloring of a graph is a partitioning of the nodes into k sets so that no edge is between two nodes in the same set.

Fact 14. *3-coloring (and k-coloring for $k > 3$) is hard.*

Proof sketch. Reduction from 3-SAT. First, create a triangle where two nodes represent true and false, while the last is the so-called *base* node. For a Boolean variable a, create a triangle consisting of one node for a, one for $\neg a$, and the third being the base node. Thus, if a gets the same color as the true node, $\neg a$ will get the same color as the false node, and vice versa.

Now create a graph widget for each clause, linking the nodes for either a or $\neg a$ to other nodes, including the true and false nodes, so that the only way to find a three-coloring is if one of the variable nodes gets the same color as the true node. (Details left as an exercise for the reader.) □

The chromatic number of a graph is the minimum number k of colors needed to permit a k-coloring.

Fact 15. *Determining the chromatic number of a graph is hard.*

Proof sketch. The chromatic number can be used to answer the decision version of 3-coloring (or k-coloring, in general). □

A clique is simply a complete graph, but the term is often used to refer to subgraphs. Finding a clique of size k in a graph G then means finding a complete subgraph of G with k nodes.

Fact 16. *The clique problem (as well as the max clique problem) is hard.*

Proof sketch. Reduction from 3-SAT once again. Three nodes for each clause (one for each literal), edges between all nodes representing compatible literals (those that can be true at the same time; that is, not between a and $\neg a$). Do *not*, however, add any edges between literals from the same clause. That way, if you have k clauses, finding a clique of size k will force all clauses to be true. (Cormen et al. give a more detailed version of this proof.) □

An independent set is a subset of the nodes of a graph where no node has an edge to any of the others.

Fact 17. *The independent set problem is hard.*

Proof sketch. This is just the clique problem on the edge-complement graph. □

Fact 18. *Partitioning into cliques (clique cover) or independent sets of size k is hard.*

Proof sketch. These two are equivalent. The independent set version can easily be used to solve the k-coloring problem. □

A vertex cover for the graph $G = (V, E)$ is a set of nodes (vertices) whose edges constitute all of E. The problem involves finding one of at most a given size (or of minimum size).

Fact 19. *The vertex cover problem is hard.*

Proof sketch. A graph with n nodes has a vertex cover of size at most k if and only if it has an independent set of size at most $n - k$ (giving a reduction both ways). This can be seen as follows: a set of nodes is a cover if and only if the remaining nodes form an independent set. First, note that if there were an edge between any of the remaining nodes, that edge would not have been covered—a

contradiction. The implication goes the other way as well. Let's say you have an independent set. Any edge *not* connected to the independent set must, of course, be covered by the remaining nodes. However, any edge *connected* to the independent set must *also* be covered by a node outside it, because both ends of the edge cannot be inside the independent set. □

Note that the *edge cover* problem is can be solved in polynomial time.[4]

The set cover problem is just a generalization of the vertex cover problem. You have a set S and another set F consisting of subsets of S. You then try to find a small subset of F whose union is equal to S.

Fact 20. *The set cover problem is hard.*

Proof sketch. The vertex cover problem can be represented directly. □

Fact 21. *The undirected Hamilton cycle problem is hard.*

Proof sketch. Reduction from the directed Hamilton cycle problem. Split every node into three, replacing it by a length-two path. Imagine coloring the nodes. The original is blue, but you ad a red in-node and a green out-node. All (directed) in-edges now become (undirected) edges linked to the red in-node, and the out-edges are linked to the green out-node. Clearly, if the original graph had a Hamilton cycle, the new one will as well. We need, however, to establish that the implication goes both ways. (Note that this is very different from reducing both ways.) In other words: if the new (undirected) graph has a Hamilton cycle, will that imply the existence of a directed one in the original?

If the new graph *does* have a Hamilton cycle, the node colors of such a cycle would be either cycling through red, blue, and green (in that order) or green, blue, and red. Because we can choose direction freely, we choose the former, which necessarily corresponds to a directed Hamilton cycle in the original graph. □

A Hamilton *path* is like a Hamilton cycle, except it's missing one edge (that is, it need not end up where it started).

Fact 22. *The directed Hamilton path problem is hard.*

Proof sketch. Reduction from the directed Hamilton cycle problem. Split any node v that has both in- and out-edges into v and v'. (Without such a node, there is no Hamilton path, assuming that $|V| > 2$.) Keep all in-edges pointing to v and all out-edges starting at v'. If the original graph had a directed Hamilton cycle, the new one will have a Hamilton path starting at v' and ending at v (it's just the original cycle, split at the node v). Conversely, if the new graph has a Hamilton path, it must start at v' (because it has no in-edges) and it must end at v (no out-edges). Merging these nodes will yield a valid directed Hamilton cycle in the original graph. □

Fact 23. *The undirected Hamilton path problem is hard.*

[4]Note also that this is not, in itself, a proof that it is not NP hard—only a *very strong* indication.

Proof sketch. You could just use the same splitting and coloring scheme as before, reducing from the directed Hamilton path problem. Another option would be to reduce from the undirected Hamilton cycle problem. Choose some node u and add three new nodes u', v, and v', as well as the (undirected) edges (v, v') and (u, u'). Now add an edge between v and every neighbor of u. If the original graph had a Hamilton cycle, this new one will have a Hamilton path (just disconnect u from one of its neighbors in the cycle, and add u' and v' at either end). The implication works both ways: a Hamilton path in the new graph must have u' and v' as its end points. If we remove u' and v', we're left with a Hamilton path from u to a neighbor of u, and we can link them up to get a Hamilton cycle. □

Fact 24. *The longest path problem is hard.*

Proof sketch. Can be used to find a Hamilton path. □

Feel free to twiddle with the specifics of the previous fact. (For example, what about edge weights? Do you specify the start and end nodes? Does it matter?)

Fact 25. *The shortest (weighted) path problem is hard.*

Proof sketch. Just negate all edge weights, and you've got the longest path problem. □

Fact 26. *The Traveling Salesman Problem (TSP) is hard.*

Proof sketch. If you're just trying to find the shortest Hamilton cycle in a weighted graph, the reduction is obvious. If you constrain the problem to *complete* weighted graphs, you can still reduce from the Hamilton cycle problem by setting the weights of the original edges to 1, and those of the edges that don't exist in the original to some sufficiently high value. The shortest cycle would then never include them, if given a choice (that is, if there is a Hamilton cycle in the original). □

One specialized version of TSP involves weights that are *metric* (that is, $w(x, z) \leq w(x, y) + w(y, z)$, for all $x, y, z \in V$).

Fact 27. *Metric TSP is hard.*

Proof sketch. Same idea. Original edges get weight 1, new edges get a weight of 2 (or some other value greater than 1, and at most 2). □

Fact 28. *Euclidean TSP is hard.*

Proof sketch. Not including a proof for this one here. Good to know, though. □

There are efficient approximation algorithms for metric and Euclidean TSP (better ones for the Euclidean case). We can't approximate it well in the general case, though. What we'd like is to find an algorithm that finds a Hamilton cycle in the complete, weighted graph, whose total weight is at most a constant factor away from the optimum—that is, we'd at most be off by a few (or, for that matter, many) percent. No dice.

Fact 29. *Approximating TSP is hard.*

Proof sketch. Reduction from the Hamilton cycle problem. Let's say our approximation ratio is k (that is, our "best guess" is at most k times as long as the optimum TSP tour). We use the same reduction as before, giving the original edges a weight of 1, and the new ones (those in the complete graph that aren't in the original) get a huge weight. If m is the number of edges in the original graph, we set each of these weights to km. The optimum TSP tour would have a weight of at most m if the original had a Hamilton cycle, and if we included only *one* of the new edges, we'd have broken our approximation guarantee. In other words, if (and only if) there were a Hamilton cycle in the original graph, the approximation algorithm for the new one would find it. □