SI 335, Unit 6: Completeness and Complexity

Daniel S. Roche (roche@usna.edu)

Spring 2014

1 Introduction

So far in this class we've concentrated on how to compare different algorithms for the same problem. We looked at a bunch of algorithms for sorting, for integer multiplication, for all-pairs shortest paths, etc.

Now we want to get all "meta" and actually compare different *problems* to each other. This is an ambitious endeavor, but one that is really important to understand the *inherent difficulty* of problems. One reason that you should care about this is cryptography. For example, if integer factorization is inherently difficult, then we can be more confident in the security of the RSA cryptosystem.

1.1 Computational Complexity

The field of *computational complexity*, which is what we'll be getting into in this unit, is the study and classification of different problems according to their inherent difficulty. For example we would like to make statements like "sorting has roughly the same difficulty as integer multiplication" or "integer factorization is more difficult than computing shortest paths in a graph".

The first step to comparing problems is to define what the difficulty of a problem actually is:

Definition

The difficulty of a problem is the **best worst-case cost** among all possible algorithms that correctly solve that problem.

In other words, I take all the possible algorithms for that problem (including ones that haven't been invented yet!), figure out the worst-case cost (maybe running time or space, whatever we're measuring) of each of those algorithms, and then take the least of all those worst-case costs. That's the inherent difficulty of the problem.

If this seems crazy, that's because *it is*! This is the first indication that getting complete answers to our questions regarding computational complexity is going to be kind of rare. What this really means is that we will be making a lot of simplifications, and even then that there will be some fundamental questions left unanswered.

However, we actually have seen a few instances where the computational complexity is completely nailed, such as sorting in the comparison model. We know the inherent difficulty of that problem is $\Theta(n \log n)$ because there are algorithms with this worst-case cost, and there is a lower bound on the problem to say that the worst-case cost of any algorithm must be at least this much.

1.2 Comparing problems (overview)

Short of getting an exact measure of their computational complexity (which is hard to do in general), how can we compare two completely different problems? We'll spend a lot of time making this precise, but let's start with a dead-simple example: finding the minimum element in a list versus sorting the list. In complexity, we like to give problems upper-case names to make it easier to talk about them, so I'll call these two problems MIN and SORT.

Now you are already thinking in your head of the algorithms you know to solve MIN and SORT - but that's not what we're talking about here! We're not talking about MergeSort or InsertionSort or any particular way of solving these problems; we're talking about the problems themselves. How can we compare the inherent difficulty of MIN and SORT?

One way to compare these problems directly, without having to talk about *any* particular algorithm for either problem, is to figure out how to solve one problem with an algorithm for the other. For this example, MIN can obviously be solved by running any SORT algorithm once, and then you return the first thing in the sorted array. What this means is that any brilliant sorting algorithm immediately gives you an algorithm for MIN, with exactly the same cost. We say the *inherent difficulty* of MIN is less than or equal to that of SORT.

How about the other direction? We have to think of how to solve SORT using MIN. As you know, there are lots and lots of ways to SORT, but in order to compare these two problems we just want to think about how to use MIN to do it. Well, you can SORT using n calls to MIN, by finding the MIN, then changing that number to infinity, finding the MIN again (which is the second-smallest number now), changing that number to infinity, and so on until you've computed the whole sorted order.

(This is exactly what the SelectionSort algorithm does, but in a stupider way here because here we don't bother to reduce the size of the array for subsequent MINs. Notice that we are allowed - even encouraged - to do a lot of "stupid" things in the context of complexity!)

What this means is that any brilliant MIN algorithm would right away give you a SORT algorithm whose running time is n times more than the MIN algorithm's running time. We say that the *inherent difficulty* of SORT is less than or equal to n times that of MIN.

Putting these two together, and being a little more formal, we could write

$$\texttt{MIN} \leq \texttt{SORT} \leq n * \texttt{MIN}$$

This general process of figuring out how to solve one problem using an algorithm for another one is called a *reduction*, and it will be our main tool in actually comparing algorithms. What we just showed was a reduction from MIN to SORT, and a reduction from SORT to n MINs. We'll have to get more specific about the "rules" of this game of doing reductions, but this is the basic idea, and the basic tool we will use to compare problems. The important thing to recognize is that we didn't rely on any particular algorithm for either problem, or on any lower bound, in order to make this comparison.

1.3 Tractable vs intractable

The ultimate goal of our brief study of complexity is to decide what problems are tractable and what problems are intractable. So we have to define what these words mean!

Luckily, we already have a pretty good definition of what kind of problem should be solved in a reasonable amount of time (*tractable*). The Cobham-Edmonds thesis that we talked about back in Unit 3 declares that an algorithm is tractable only if it runs in polynomial-time in the size of the input.

The key task ahead of us in this unit is to try and show that some problems are actually hard, or *intractable*. Let's consider the realm of what intractable problems could look like:

- Undecidable problems. You learned in Theory class that the Halting Problem ("Does this program always terminate?") is undecidable, meaning that no computer program can always solve it in a finite amount of time. Then we definitely won't be getting any polynomial-time algorithms!
- Problems with big output. For example, consider the problem of computing *every path* between two vertices. Since a path is really just an ordering (or *permutation*) of the vertex names, there are something like $n!$ possible paths in the worst case, which means output size exponentially larger than the input size. So this is another kind of problem that is definitely not going to have a polynomial-time solution.
- Problems that seem infinitely hard. For example, consider the problem of determining whether two given regular expressions (with the Kleene star and a squaring operator) represent the same language. There are actually ways to solve this problem, but none that require less than exponential space. Problems this hard are also definitely not going to yield any polynomial-time algorithms.

These are all ridiculously hard, intractable problems. But the problems we're thinking about as being "hard" are not this hard. For example, consider integer factorization, where we are given a number and asked to find its least prime factor. It's certainly decidable. The output isn't too large either, since the factor can't be larger than the input operator. And it doesn't require an exponential amount of space; we can solve it by just checking each possible factor from 2 up to the (square root of) the input number until we find one. And yet we think factorization is a hard problem!

This is actually the class of "hard" problems that we are going to be most interested in. In a way, they're the easiest kind of hard problems. And what characterizes problems like integer factorization and minimal vertex cover is that, while computing the answer might be difficult, checking the answer is easy. Given a factor, we can see that it actually divides the input. Given a vertex cover, we can confirm that it actually covers all the edges in a graph.

So the problems we're going to focus on are essentially those that can be solved by a "guess-and-check" or "trial-and-error" strategy: repeatedly guess the answer, then check it, and return your best guess. And actually almost all the problems we've seen in this class can be categorized this way. Yet some of them (like finding the shortest path in a graph) are tractable, and some others (like finding a minimum vertex cover) seem intractable. The most important (and difficult) question we will ask in this class is whether *any* problems are so hard that they can only be solved by a brute-force trial-and-error approach. We just need a little more background in order to be able to ask that question in a precise way.

2 Complexity Basics

Any time we try to say something about how difficult a problem is, it requires us to be very precise about how the problem is defined, what kinds of operations allowed, and how the difficulty and performance will be measured. This is sort of like laying out the *rules* of a very precise game. We have to define the rules carefully so that the comparisons made between algorithms will be fair, consistent, and meaningful.

2.1 Machine models

The first thing to define is what kinds of operations are allowed by the algorithms we consider. This is like when we proved the lower bound on the sorting problem earlier, and said that only comparisons between array elements were allowed. Except now, since we are talking about all different kinds of problems, we can't be so restrictive. We want to come up with a general model of computation that is well-defined (so we can reason about it) but which also corresponds closely to the power of actual computers.

Traditionally, the preferred model of computation for theoretical work is a Turing machine, like you learned about in Theory. In this class, we've been using a model of "primitive operations", which we defined sort of loosely as anything which would take a fixed number of operations on any real computer. We could also get more specific, like counting the number of MIPS assembly instructions, or use another abstract computer model.

The good news is, it doesn't actually matter for us! Why? Well, our only goal in this brief foray into computational complexity is to understand what problems are tractable and which ones are intractable. While the models listed above are certainly not identical, they could be mapped to each other in polynomial-time. That is, given any program in one model that runs in time $\Theta(n^k)$, we could construct a program in any of the other models that runs in time $\Theta(n^\ell)$, where k and ℓ are both constants.

(I'll spare you the proof of that last claim.)

2.2 Input Size

What definitely *will* matter for is is the size of the input. In order to have fair and consistent analysis, we must measure the size of the input the same for all problems. The following list of previously-stated complexities demonstrates that we have definitely not been doing this so far:

- Least Prime Factor: $\Theta(\sqrt{n})$
- Karatsuba multiplication: $O(n^{1.59})$

- Strassen multiplication: $O(n^{2.81})$
- Dijkstra's (using adjacency lists): $\Theta((n+m)\log n)$

The key problem here is that each of these analyses is measuring the size of its input in a different way. We already know that $\Theta(\sqrt{n})$ isn't really the cost of factorization in terms of the size of the input, because n in this equation is the number itself, whereas the number of digits in that number is more like $s = \lg n$.

But even between Karatsuba's and Strassesn's algorithms, there's a discrepancy. The n in the cost of Karatsuba's is giving the number of digits in the input numbers, but in Strassen's it's the dimension of one side of a square matrix. So the actual size of the input to Strassen's is more like $s = n^2$. Using this definition of size, Strassen's algorithm actually looks faster than Karatsuba's; it's $O(n^{1.41})$. Dijkstra's is even more confusing because it introduces this second parameter m into the cost.

The issue at hand is that we need a way of comparing all different kinds of problems to each other. Measuring the difficulty in slightly different ways is fine if you're just comparing a bunch of algorithms for the same problem (for example, computing shortest paths). But when we start talking about different problems together, we need a consistent and universal definition of input size.

Bit size is the answer to this problem. Since everything is stored in a computer, every input takes up a certain number of bits. So we'll count the total number of bits as the size of the input for *every* problem from now on, and we'll always call this n. So in terms of bit size, the above algorithms actually cost

- Least Prime Factor: $\Theta(2^{n/2})$
- Karatsuba multiplication: $O(n^{1.59})$
- Strassen multiplication: $O(n^{1.41})$
- Dijkstra's (using adjacency lists): $\Theta(n \log n)$

(The last one is a little tricky. The reason it becomes $\Theta(n \log n)$ is that the original input size is $n = |V| + |E|$. Notice that having a larger input size actually makes algorithms look faster, which seems a bit counter-intuitive at first!)

2.3 Decision problems

So we've nailed down the input size — well what about the output size? Clearly the size of the output makes a difference (potentially) for the difficulty of the problem. So are we going to have to have a second parameter in the measure of difficulty of a problem?

To avoid this, we will only study a very restricted class of problems called *decision problems* that only answer yes-or-no questions. So the output will always be a single bit: YES or NO, TRUE or FALSE, ON or OFF, etc.

In fact, these are the only kind of problems that can be handled by the basic models of computation that you studied in Theory. Formally, a Turning machine can only output ACCEPT or REJECT. So we must restrict the discussion to decision problems in order to include such machines in the discussion.

Let's look at an example: factorization. The normal factorization problem (as solved by the LeastPrimeFactor algorithm) is, given a number N, to compute the smallest prime factor of N. How could we make this into a decision problem?

The first decision problem version we might try is, "Does N have any prime factors?" This indeed follows along the lines of the original problem, and clearly by solving the original problem of factorization we could answer this question. But the question has gotten too much easier, it would seem: while we suspect the original factorization problem is intractable, we know that this decision problem (primality testing) can be solved in polynomial-time. Since the whole point of this unit is to figure out which problems are tractable and intractable, this is a simplification gone too far!

So let's try a different decision problem version of factorization. In this version, we'll try to mirror the actual computation, and add a second parameter K, which is the candidate factor. So the decision problem becomes, "Does N have a factor equal to K?" This seems good because it mirrors the actual computation and doesn't just ask about primality, but we've fallen into the same trap! Our new decision problem is actually an even easier one: testing divisibility! Once again, the decision problem is clearly tractable, although we suspect the original problem is not.

OK, third time's a charm. Instead of asking for exact divisibility, we'll turn the question into an inequality: "Does N have any factor **less than or equal to** K?" This way, we concentrate on the factorization problem (not just primality), but we're also asking about a whole bunch of factors, not just one.

As it turns out, this problem is almost the same difficulty as the original factorization problem. How do I know? A reduction of course! If we wanted to factor a number N, and had access to an algorithm that solved the third decision problem version above, then this we could find the factor by using something like a gallop search, calling the decision-factorization problem $\Theta(\log N)$ times. And since the size of the input in this case is $n = \Theta(\log N + \log K)$, the cost of the reduction is just $\Theta(n)$ times the cost of the decision problem. But more on this kind of reduction later.

Here are a few more problems, stated carefully as decision problems, for us to think about and discuss.

FACT(N,K)

Input: Integers N and k

Output: Does N have a prime factor less than k?

SHORTPATH(G,u,v,k)

Input: Graph $G = (V, E)$, vertices u and v, integer k

Output: Does G have a path from u to v of length at most k?

LONGPATH(G,u,v,k)

Input: Graph $G = (V, E)$, vertices u and v, integer k

Output: Does G have a path from u to v of length at least k?

VC(G,k)

Input: Graph $G = (V, E)$, integer k

Output: Does G have a vertex cover with at most k vertices?

One thing to take note of is the difference between "at least" and "at most". It makes a big difference, every time! For example, the SHORTPATH problem can be solved quickly by Dijkstra's algorithm: just compute the shortest path, then compare its length to k. But we don't know any way of solving LONGPATH quickly. Same with vertex cover: the VC problem above seems difficult — the best thing we know how to do is approximate it using a maximal matching. But if we changed the "at most" to "at least k vertices", then it would be really easy, since we know that the set of all vertices always forms a vertex cover. This one-sidedness of problems turns out to be really important for the kinds of hard problems we're going to be thinking about.

2.4 Definition of P

Remember that the study of complexity theory is all about classifying problems based on their inherent difficulty. Formally, what this means is defining sets of problems (called *complexity classes*) whose members have some kind of common level of difficulty.

It's important to recognize that actual complexity theory consists of a ton of different complexity classes, so many in fact that there's a whole website to help researchers keep track of them. It's called the complexity zoo and currently claims to have 495 different complexity classes on display.

Fortunately, we're only going to worry about TWO different complexity classes, which will roughly correspond to our notions of "tractable" and "intractable". And the first one is what you should already be familiar with: polynomial-time:

The complexity class **P** consists of all decision problems whose inherent difficulty is $O(n^k)$ primitive operations, for some constant k, and where n is the bit-length of the input instance.

Remember that "inherent difficulty" means "the worst-case cost of the best possible algorithm for this problem". So this definition, although it corresponds to the very simple and intuitive notion of tractable problems, actually requires all our previous careful definitions.

Of the four problems just presented, only SHORTPATH is known to be in **P**. And actually most problems that we've talked about (if they can be stated as a decision problem) are in this class as well.

What exactly would it require to show a problem is in **P**? All we need to know that the "best worst case" is at most $O(n^k)$ is to have an example of a single algorithm that is polynomial-time. So for example, SHORTPATH can be solved by running Dijkstra's algorithm in $O(n \log n)$ time (remember that n is the size of the input!), and since $n \log n \in O(n^2)$, and obviously 2 is a constant, this means that the SHORTPATH problem is in **P**.

Something important to keep in mind as we go forward is that this notion of polynomial-time allows us to be **really lazy** in our analysis. For example, consider the problem of sorting. The idea of something like selection sort is not particularly brilliant: find the smallest thing, put that first, then repeat. The worst-case running time of selection sort is $\Theta(n^2)$. More sophisticated algorithms like HeapSort and RadixSort have better worst-case running times (maybe), but they're also more difficult to reason about. The point here is that, as far as polynomial-time is concerned, these algorithms are all basically the same! Any one of these sorting algorithms we've talked about is good enough to show that the sorting problem is solvable in polynomial-time.

The lesson here is that a lot of the low-level tricks we've learned to move the running time from say $\Theta(n^2)$ to $\Theta(n \log n)$, while **extremely** important in developing efficient algorithms for these problems, are not particularly useful in this unit, where we're trying to compare problems in the broad sense of polynomial-time. Some particular properties of polynomial-time that might be useful are:

- **Closed under addition**. If the running times of two algorithms are $O(n^k)$ and $O(n^e ll)$, respectively, then the worst-case cost of calling one after the other is $O(n^k + n^e ll) = O(n^{\max(k,\ell)})$.

  In other words, two polynomial-time algorithms performed in sequence is still polynomial-time.

- **Closed under multiplication**. If we call a $O(n^\ell)$-time algorithm inside a loop that runs $O(n^k)$ times, the total cost is $O(n^{k+\ell})$ — still polynomial-time!

  This means that repeating any polynomial-time algorithm, a polynomial number of times, is still polynomial-time.

- **Closed under composition**. If we all an $O(n^k)$-time algorithm on an input whose bit-length is $O(n^\ell)$, the total cost is $O(n^{\ell k})$ — still polynomial-time!

  This allows polynomial-time algorithms to be effectively *composed*, like piping the output of one to the input of another, and still be polynomial-time.

3 Certificates and NP

At this point we know what polynomial-time is, but that's really nothing new. The challenge is going to be how we can classify problems such as minimum vertex cover, longest path, and factorization. The key property that unites these kinds of problems is that, while it's hard to come up with the answer, it's easy to check the validity of an answer once we have it. This kind of reasoning seems fine for talking about *computational* problems like finding a minimum vertex cover or the least prime factor of an integer. But what about the *decision problem* versions? How can we check a "yes" answer? That's where certificates come in.

3.1 Certificates

A certificate is like a digital "proof" that the answer to a decision problem is YES. For example, if you asked me, "Does this graph have a vertex cover with at most 10 nodes?" and I said "yes", then you probably wouldn't be very satisfied. To really *convince* you the answer is "yes", I would need to demonstrate that there is such a vertex cover, by producing the vertex cover itself! Then you could check that it's an actual vertex cover, and it has less than or equal to 10 nodes, and you would be sure that the answer I gave is correct.

So even though the original decision problem may have just a yes/no answer, the justification that the answer is "yes" — what we call a *certificate* — can be anything. But usually, the certificate is pretty obvious; it's the output of whatever the computational version of the question would ask for.

Can you think of what the certificate would be for the four decision problems mentioned above?

3.2 Verifiers and NP

Since I said a certificate "can be anything", what does it actually mean? In the context of what we're talking about, it should mean that there's some easy way of actually using the certificate to check the correctness of the "yes" answer. For example, if you asked for proof that the longest path in some graph has length at least 20, and I just pointed to the graph and said, "Look, it's right there", that wouldn't be very helpful (or nice) of me. I would need to show you the actual path, so you could check that it is indeed a path and its length is indeed at least 20.

So the definition of a certificate is actually connected to some algorithm, which we will call a *verifier*, that uses the certificate to check the answer. The verifier algorithm for the LONGPATH problem is basically: "Check the nodes in the given path (the certificate). If they form a valid path, then compute the path length. If it is at least k, then accept the certificate as valid proof."

The funny thing about our second complexity class, called **NP**, is that it doesn't depend on there being any sort of algorithm for the problem itself! Instead, it just talks about there being a fast verifier for the problem — where "fast" of course means polynomial-time! Here's the definition:

> The complexity class **NP** consists of all decision problems for which there exist:
>
> - *Certificates* of length polynomial in the input, for every instance to which the answer is YES
> - *Verifier algorithm* that takes the original problem input and the certificate and runs in polynomial-time.

Notice that the verifier itself is a decision problem. It answers the question, "Does *this* certificate prove that the answer to the original problem, for *this* input, is YES? So a problem is in **NP** if it has a verifier that is in **P**.

We are going to talk about a lot of problems that are in **NP**. For every one of them, the proof of being in **NP** follows four basic steps:

1. Define what the certificates should be.
2. Show that the length of the certificate is bounded by a polynomial in the size of the original input.
3. Present the verifier algorithm based on what the certificates are.
4. Analyze the verifier algorithm and show that it is polynomial-time.

For example, here is a proof that the minimum vertex cover problem is in **NP**.

> **Theorem**: The problem VC(G,k) is in **NP**.
>
> **Proof**: First we have to define the certificates. The certificate that the answer is YES will be a vertex cover of size at most k. This will be stored as a list of node names, for the nodes that are in the vertex cover.
>
> The vertex cover contains at most $|V|$ nodes, since that's how many nodes are in the original graph. So the size of the certificate is never more than the size of the original input graph G. Therefore the size is $O(n)$ (remember n means the bit-length of the original input), which is a polynomial in n.
>
> The verifier algorithm works as follows: For every edge in the graph G, go through every node in the vertex cover (the certificate), and check that one of the endpoints of the graph is in the vertex cover. If

not, then return NO. If every edge of the graph is covered in this way, then check the size of the vertex cover. If the size is more than k, then return NO. Otherwise return YES.

The input graph is an adjacency list, so it takes $O(|V|^2) = O(n)$ time to go through every edge in the graph. For each edge, we have to compare its endpoints to every vertex in the cover, which can have size at most $|V|$. This gives a total cost of $O(|V|^3)$, which is $O(n^{1.5})$. The final loop to check the size of the cover is $O(n)$ as well, so the whole verifier algorithm is polynomial-time.

See how the four steps are followed? The only thing that's a little odd is that I described the verifier algorithm in words instead of writing out pseudocode. This is fine for such a simple one, but for more complicated **NP** proofs, you probably want to right out the verifier algorithm in pseudocode, to make sure everything is clear.

By the way, how does this compare to the class **P**? Well let's say I have a problem that's in **P**, like SHORTPATH(G,u,v,k). This means that there's some algorithm to solve this problem that runs in polynomial-time. But can we verify a YES answer in polynomial-time? The answer is, of course! The certificate can actually be *empty*, and the verifier is the original algorithm that solves the problem. Since we can already answer the problem in polynomial-time, we don't need any "help" (in the form of a certificate) to show that the problem is in **NP**. This is a bit like my "unhelpful" answer above of "just look at the graph". When the problem is in **P**, we can "verify" an answer in polynomial-time by just computing it. This proves that, mathematically, **P** is a subset of **NP**.

The big question is whether **P** is actually equal to **NP** or not. And by "big question", I mean "the biggest, baddest, most important question in the history of computer science". This is not just my opinion, and it's not an exaggeration. In fact, if you can answer this question, the Clay Institute will give you a million dollars. Seriously. The reasons why this question is so important will become clearer as we move along.

3.3 Alternate definition of NP

You know that **P** stands for "polynomial-time", but what does **NP** stand for? A common — and TOTALLY INCORRECT — misconception is that it stands for "not polynomial".

Actually what it stands for is "nondeterministic polynomial-time". This is an alternate definition of the class. Now you should recognize the term *nondeterministic* from Theory class. The difference between a DFA and an NDFA is that the NDFA can sort of take multiple computational paths in processing any given string, and as long as *at least one* of those paths ends up in an accepting state, the string is accepted.

Nondeterministic computing is like that, except applied to more powerful machines than DFAs. The original definition corresponds to a nondeterministic Turning machine, which is like a regular Turning machine, except there can be many transitions out of every state. Again as long as one computational path ends up in an accepting state, the string is accepted.

In more general computing the way we usually think of it, a "nondeterministic machine" can make yes/no "guesses" at any point and branch based on that guess, like an "if" statement whose conditional is just an arbitrary guess. The true answer is YES as long as there is *at least one* sequence of "guesses" that makes the program output YES.

And now hopefully you see the connection to our definition of **NP** using certificates: the certificate can be thought of as the series of guesses, or the sequence of Turing-machine transitions, that ends in a YES output or an accepting state. The verifier is just the non-deterministic algorithm, executed deterministically using the certificate's information.

So what the famous "P vs NP" problem really boils down to is whether having a nondeterministic machine that could magically make correct guesses all the time actually makes some problems easier to solve or not. Since such a machine doesn't actually exist, it's easy to think that the answer is "of course it makes some problems easier". In fact this is what most computer scientists suspect, that **P** is actually not equal to **NP**. But none of us can *prove* it yet, unfortunately!

4 Reductions

Great, so we know what **P** and **NP** are. And we know that everyone suspects there are some problems in **NP** that are not in **P**, but they can't prove it yet. So what *can* we say about these classes? Remember that reductions are a tool that allowed us to compare problems without having any particular algorithm for the problem. With some really clever reductions, we'll be able to say quite a bit about **P** versus **NP**. But first we need to be a little more precise about what a reduction is and how to analyze it.

> A *reduction* from problem A to problem B is an algorithm that solves problem A by using any algorithm for problem B as a subroutine.

The terminology is confusing, even to me after seeing it for a number of years. But I've been told that other people don't get confused by the reduction terminology, so maybe it will seem more natural to you. The point is, always remember:

- What you're reducing *from* because this is the problem you're actually solving — you take the input for that problem and produce the output for that problem.
- What you're reducing *to* is the problem that you use as a subroutine. You will be creating input(s) for this problem, and using output(s) from it.

Also keep in mind the core meaning of a reduction: it's comparing the difficulty of the two problems. A reduction from problem A to problem B shows that (in some sense) problem B is at least as difficult as problem A.

4.1 Example: Matrix Multiplication

Let's look at a concrete example: matrix multiplication versus squaring. The matrix multiplication problem, MMUL(A,B) is, given two square matrices, compute their matrix product AB. The matrix squaring problem, MSQR(A) is, given a single square matrix, compute its square A^2.

We want to reduce these problems to each other. That means solving each problem with the other one. Here's how that is done:

- MSQR reduces to MMUL.

  Usually one direction of a reduction will be fairly easy or straightforward. For these two problems, this is the straightforward direction, reducing squaring to multiplication.

  Remember that a reduction is an algorithm to solve one problem using the other. So we want an algorithm for the MSQR(A) problem that will use MMUL as a subroutine.

  This algorithm just has a single step: compute MMUL(A,A) and return the result. This result will be $A \cdot A = A^2$, so the reduction is correct.

- MMUL reduces to MSQR.

  This one is a little trickier, and maybe not obvious. How can we do matrix multiplication, just by squaring matrices? One way to think about this that I find to be useful is, how can I "trick" an algorithm for one problem to solve the other? That is, imagine I have a stubborn assistant who will only square matrices for me, but I want to multiply two different matrices. How can I trick my assistant into doing my job?

  The way to "trick" the assistant is to somehow *embed one problem in the other one.* This is usually where some ingenuity comes into play. For this particular problem, the reduction works in three steps. Remember that this is an algorithm to solve MMUL(A,B), so the input is the two matrices A and B, and the output should be their product.

  1. Compute matrix C as follows:

  $$C = \left[\begin{array}{c|c} 0 & A \\ \hline B & 0 \end{array}\right]$$

  2. Compute matrix D as the result of SQR(C)
  3. Return the top-left quarter of D.

The reason this works is because of the following mathematical equation:

$$\left[\begin{array}{c|c} 0 & A \\ \hline B & 0 \end{array}\right] \cdot \left[\begin{array}{c|c} 0 & A \\ \hline B & 0 \end{array}\right] = \left[\begin{array}{c|c} AB & 0 \\ \hline 0 & BA \end{array}\right]$$

Now the details of the math are I think really fascinating, but not particularly important to understand. The main point is that we managed to embed the multiplication problem into an *input* to the squaring algorithm, so that any algorithm for squaring could be used to solve multiplication.

Now the question is, so what? What have we actually shown? What this demonstrates is that, if we had an algorithm to solve MMUL in say $f(n)$ time, then we would have an algorithm to solve MSQR in $O(f(n))$ time as well. And if we had an algorithm to solve MSQR in say $g(n)$ time, then the second reduction would produce an algorithm for MMUL that uses $O(g(n))$ time. In other words, these two problems are *equivalent* up to a constant factor; the inherent difficulty of both problems is essentially the same.

Of course, this is a very restricted kind of equivalence, where any algorithm for one problem gives an algorithm for the other with exactly the same asymptotic cost. This kind of equivalence between completely different problems is actually quite rare. What kind of a definition of equivalence would make sense to answer our big tractable-versus-intractable questions? To answer this, we have to look a little more carefully at the reductions themselves.

4.2 Analyzing Reductions

Now we have a pretty good idea of what a reduction is, but can you see how the loose definitions so far can be abused? What about an algorithm that say, reduces integer factorization to integer division, by trying every possible factor and testing divisibility? This is a plausible reduction, but certainly we wouldn't conclude that factorization is easier than integer division. If it were, the RSA algorithm would be in a lot of trouble!

The issue here is how to analyze the efficiency of a reduction. There are three places we have to look at for this. Supposing that we are reducing problem A to problem B, we must examine:

- The *number* of times an algorithm for problem B is called
- The *size* of each input created for problem B
- The amount of *extra work* besides calling the problem B algorithm

If each one of these three parts is bounded by $O(n^k)$ for some constant k, and where n is the input size of the original input to problem A, then this is called a **polynomial-time reduction**. If there is such a reduction from A to B, then we write A $\leq_{\mathbf{P}}$ B.

Now observe what this means for our big classification of problems as tractable or intractable:

Theorem: If A reduces to B in polynomial-time, and B is in **P**, then A is in **P** as well.

Proof: What does it mean for A to reduce to B in polynomial-time? From the definition above, it means that there are three constants k_1, k_2, k_3 such that A calls the algorithm for B $O(n^{k_1})$ times, each on inputs whose size is $O(n^{k_2})$, and does $O(n^{k_3})$ amount of extra work.

And if B is in **P**, this means B can be solved in $O(n^\ell)$ time for some other constant ℓ. And notice that this is NOT the same n as above! This n is the size of the input to B.

So what to we get when we put this together? The cost of the reduction, using the $O(n^\ell)$ algorithm for B, is in total

$$O(n^{k_1}(n^{k_2})^\ell + n^{k_3})$$

What an ugly formula! But what can we say about it? It's just the multiplication, addition, and composition of some polynomials. Therefore it's another polynomial! And thus A is also in **P**.

Actually the contra-positive of the above theorem is even more useful in many situations:

Corollary: If A $\le_{\mathbf{P}}$ B, and A is *not* in **P**, then B is not in **P** either.

So you see, polynomial-time reductions are *exactly* the tool we need to compare programs in terms of tractable versus intractable.

4.3 A polynomial-time reduction

Ready for an example? First we need to add a couple more problems to our repertoire:

Minimum hitting set: HITSET(L,k)

Input: List L of sets $S_1, S_2, \ldots, S_m$, and an integer k

Output: Is there a set H with size at most k such that every $S_i \cap H$ is not empty?

This is like asking for a list of representatives to cover every group in a list. For example, maybe each of the sets S_i represents a certain muscle group, and the elements in each set represent some different exercises that works that muscle group. Then the question asks whether we can construct a complete workout (to work every muscle group) using only k different exercises.

Hamiltonian Cycle: **HAMCYCLE(G)**

Input: Graph $G = (V, E)$

Output: Does G have a cycle that goes through every vertex exactly once?

A cycle that goes around the entire graph, traveling through every vertex exactly once, is called a *Hamiltonian cycle* or sometimes a *Rudrata cycle*. This decision problem is just asking whether such a cycle exists in a given graph.

Now for our first "real" polynomial-time reduction, from HAMCYCLE to LONGPATH. The key thing to observe here is that *we don't have any good algorithms to solve either problem*. This is the first reduction between two problems that both seem to be difficult in general.

Theorem: HAMCYCLE(G) $\le_{\mathbf{P}}$ LONGPATH(G,u,v,k)

Proof: To do this reduction, we need to consider any input to the HAMCYCLE problem, and show how to solve HAMCYCLE by using some algorithm for LONGPATH. Here's the algorithm to do that, given a graph G:

```
def HAMCYCLE(G):
    n = G.n
    for (u,v) in G.edges():
        newE = list(G.edges())
        newE.remove((u,v))
        H = ALGraph(G.V, newE)
        # H = G with (u,v) removed
        if LONGPATH(H, v, u, n-1) == "YES":
            return "YES"
    return "NO"
```

The first thing we need to show is that this reduction is correct. First, if the reduction algorithm outputs YES, then there really is a Hamiltonian cycle in the graph. The reason is that, since the call to LONGPATH returned YES, then there must be a path with $|V| - 1$ edges that goes from v to u and doesn't include edge (u, v). So now if I add that edge from u to v that was removed, I have a path from u to v and then back to u — that is, a cycle! And this cycle must hit every vertex exactly once, since its length is $|V|$. Therefore it's a Hamiltonian cycle, and the reduction works correctly in this case.

But it's still possible that the reduction doesn't "catch" every Hamiltonian cycle. To see why it does, say there is a Hamiltonian cycle in the graph G. Then this cycle contains many edges; say one of those edges is (u, v). This edge will be examined in the **for** loop, and when it is, we will discover there is indeed a LONGPATH from v back to u, which is just the Hamiltonian cycle minus that one edge. Therefore the reduction gives the correct output in every case.

We now know that this is a correct reduction, but is it polynomial-time? Recall that there are three parts to examine for this:

- The *number* of time LONGPATH is called is exactly the number of edges in G, which is $|E|$ and is less than the size of the input.
- The *size* of each input created for the call to LONGPATH is the same as the original input G, with one edge removed. So this is also less than the size of the original input.
- The amount of *extra work* besides calling the LONGPATH algorithm. The only real "work" performed by the reduction is in removing each edge from the original graph G. This could be accomplished by just copying the whole adjacency matrix and setting that one edge to infinity. The cost would be $O(|V|^2)$ for every step, for a total cost of $O(|E| \cdot |V|^2)$. There are probably more clever ways to do this, but this is certainly polynomial-time in the size of the input, so that's good enough!

All in all, we have shown the reduction algorithm, shown why it's correct, and why it's polynomial-time. Therefore the statement of the theorem is proved. QED.

Now remember what this means: any fast algorithm for LONGPATH would immediately give us a fast algorithm for HAMCYCLE. But we don't have any algorithm for either one, so what good is this? Well what if we knew that the HAMCYCLE problem was difficult? Then the LONGPATH problem must be difficult too! Take a moment to wrap your head around this idea.

5 NP-Completeness

Imagine the Fairy Godmother of Algorithms visits you. She will give you a fast algorithm for any problem that you like; you get to pick the problem. So what problem would you pick?

You might like to have an algorithm for integer factorization, so you could crack everyone's encrypted messages. But then there are other crypto schemes that are based on other hard problems. And of course there are hard problems we want to solve that have nothing to do with security as well!

Well how about a problem that could solve *every* other problem? This is a little like wishing for more wishes. Well...

5.1 Halting problem

Recall from Theory that the Halting Problem is, given a program P and an input I for that program, determine whether P halts on input I in any finite amount of time.

I claim that an algorithm to solve the halting problem can solve any other problem that's in **NP**. How could I prove such a thing? With a reduction of course! But this will be a special kind of reduction — one that takes ANY problem in **NP** and reduces it to the halting problem.

Say we have some problem, call it PROB, that's in **NP**. We want to reduce it to the halting problem. The only thing we really know about PROB is that it is in **NP**, so we have to *use the definition* of **NP** in order to start the reduction.

This definition tells us two things about PROB: for every instance that should produce a "YES" answer, there is a polynomial-size *certificate*, and there is also a polynomial-time *verifier* that checks the validity of a given certificate for a given input.

Knowing only this information about PROB, how could you solve it? Well, for some input instance I for PROB, we know that the answer to PROB(I) is "YES" if and only if there is some certificate C such that the verifier returns YES when you give it both I and C. This gives a basic sort of approach to solving PROB:

```
for every possible certificate C:
    if verifier(I,C) == "YES":
        return "YES"
return "NO"
```

Now it seems odd to say that we could generate every possible certificate *C*, when we don't know if the certificates should be vertices, or integers, or paths, or something completely different, since we don't really know what PROB is all about. But remember that *everything is just bits*, so every certificate is just some string of bits. Therefore the **for** loop above can be implemented by simply trying every possible sequence of bits in some order: maybe something like 0, 1, 00, 01, 10, 11, 000, 001, 010, ... you get the idea.

This is basically the guess-and-check strategy that we have said categorizes all **NP** problems: try every possible answer (certificate), and check each one until you get a "hit". If the answer to PROB(I) is "YES", then this will eventually be returned by the algorithm. But what happens otherwise? An infinite loop! If the answer to PROB(I) is "NO", then the verifier will just keep returning NO for longer and longer certificates.

In summary: we have an algorithm that eventually returns "YES" if the real answer is "YES", and otherwise runs forever, and we want to figure out which is the case for some particular input. Hopefully a light bulb is going off in your head right now: This is exactly what the Halting problem can do for us!

Formally, then, the reduction from PROB to the Halting problem works by using the verifier for PROB (which must exist since PROB is in **NP**), and plugging the verifier into the algorithm above to make a program. Then we give this program to the Halting problem, with the given input, and say "Does this halt?" If so, then the answer is "YES", and otherwise the answer is "NO".

In other words, given *any* input to *any* problem that is in **NP**, we can use an algorithm for the Halting problem to answer the problem for that input. This means that `VC` $\le_{\mathbf{P}}$ `HALTING`, `LONGPATH` $\le_{\mathbf{P}}$ `HALTING`, `SHORTPATH` $\le_{\mathbf{P}}$ `HALTING`, we could go on and on.

5.2 Nomenclature

Since a reduction from A to B means that B is *at least as hard as* problem A, the result above showed that the Halting problem is at least as hard as EVERY problem in **NP**. There's actually a name for this sort of thing:

> A problem B is **NP**-hard if every problem in **NP** is polynomial-time reducible to B.

Therefore, from the previous subsection:

> **Theorem**: The Halting problem is **NP**-hard.

The awesomeness of the previous subsection notwithstanding, this isn't actually too surprising when it comes to the Halting problem. You know from theory that HALTING is actually *undecidable', meaning it can't be solved by any computer in any finite amount of time. So it should come as no surprise that it's at least as difficult as problems like minimum vertex cover and integer factorization which, while seemingly difficult, can certainly be solved if given enough time.

What would be really awesome if some problem that is in the class **NP** were also **NP**-hard. Then we could say that this is really the "hardest problem" in the class **NP**. Well there's a name for this too:

> A problem B is **NP**-complete if B is in **NP** and B is **NP**-hard.

So, any guesses on an **NP**-complete problem, the hardest problem in the land of **NP**?

5.3 CIRCUIT-SAT

Time to define a new problem:

> **Circuit Satisfiability Problem**: CIRCUIT−SAT(C)
>
> **Input**: Boolean logic circuit C with multiple inputs, one output, and any number of AND, OR, and NOT gates, connected with wires.
>
> **Output**: Is there any setting of the inputs that will make the output "true"?

We want to figure out how hard this problem is. First thing we might ask: can you think of any polynomial-time algorithm to solve it?

Hmm, I can't. But what we can do is pretty exciting: we can prove this problem is **NP**-complete. This is quite a long and complicated proof, but you should be able to follow each part. The proof will proceed as a series of "claims", and then an informal proof of each claim.

First claim: CIRCUIT-SAT is in NP. A certificate for a "YES" answer will just be a setting of each of the inputs to true or false. The size of this certificate is just the number of inputs, which is certainly less than the size of the whole circuit. Then the verifier just has to simulate the circuit and check whether the output is true. This means simulating the logic of every gate in sequence until we get to the output. Constant-time for every gate means polynomial-time in the total number of gates, which is the size of the input. Therefore CIRCUIT−SAT is in **NP**.

OK, nothing new so far. But what we want to do now is prove that CIRCUIT−SAT is **NP**-hard, like we proved for the Halting problem above. The basic approach is going to be to turn the verifier for any problem in **NP** into a polynomial-size boolean circuit, then run CIRCUIT−SAT on that verifier circuit to see if there's any certificate that verifies the given input, thereby telling us if the real answer to the problem is "YES" or "NO".

Second claim: Polynomial-time programs use polynomial-space. Suppose some program runs in polynomial-time $O(n^k)$. In the worst case, this program might do nothing at all but just access memory. Since it can only access a fixed number of memory locations for every primitive operation it performs, the total space usage of the program is also at most $O(n^k)$.

Third claim: Any YES/NO program can be modeled by a boolean circuit. Think back to your Architecture class. You learned that the CPU in every computer you have ever used is essentially the same thing: a big sequential logic circuit. It's built from a bunch of pieces that look like this:

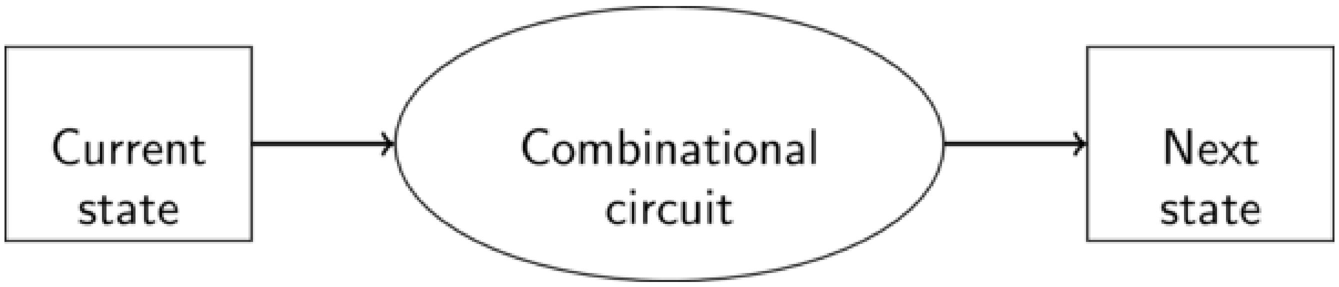


Figure 1: Sequential circuit

The combinational circuitry here is everything in the CPU that gets executed at each step: MUXes, ALUs, adders, shifters, and control logic. All these things are constructed out of simple AND, OR, and NOT gates. At each step of the program, the boolean values for the current state are run through this combinational circuit to produce the boolean values for the next state.

Each "state" consists of all the storage that is used to control the program: registers, program counter, and main memory. We can represent all these things with wires that are true or false. The initial state will be the initial settings of things like the program counter, plus the input to the program. The final state will contain a single boolean value that contains the YES/NO output of the program.

Now for a particular input, the program will proceed through some finite number of steps. So we could write out a single circuit (with no feedback loops) for the ENTIRE program just by pasting together a bunch of copies of the combinational circuit, corresponding to the total number of steps in the program. This circuit will be pretty big, yes, but its behavior will correspond exactly to the original program.

Fourth claim: Any polynomial-time YES/NO program can be modeled by a polynomial-size boolean circuit. We have to use the previous two claims. From the second claim, the total storage space used by any polynomial-time program is polynomial-time. Therefore the size of the "state" in each part of the circuit is polynomial in the size n of the actual input to the program.

Now since the combinational part of each piece of the circuit has polynomial-size in the size of the state, and since polynomials are closed under *composition*, the size of each combinatorial part is also a polynomial in the size of the original input.

Finally, the total number of parts in the circuit corresponds to the number of steps in the program, which is polynomial-time. Since polynomials are closed under *multiplication*, the total size of this circuit is STILL a (very big) polynomial in the size of the input to the program itself.

Fifth claim: CIRCUIT-SAT is NP-hard. Take any problem PROB in **NP**. It must have a polynomial-time verifier algorithm, which takes as input the original problem instance and a polynomial-size certificate, and answers "YES" if the certificate is a valid proof that the original problem's answer is "YES" on that input. From the last claim, we can take this verifier algorithm, and any input I for PROB, and construct a polynomial-size boolean circuit that will check any certificate against the input I.

Now if we hard-wire I into this circuit, and make the only "inputs" of the circuit the bits of the certificate, then we have a circuit that is *satisfiable* if and only if there is some polynomial-size certificate which verifies a "YES" answer for PROB(I). Now give this big (but still polynomial-size!) circuit to any algorithm that solves the CIRCUIT−SAT problem. If the circuit is satisfiable, then there is a certificate for I and PROB(I) is "YES"; otherwise PROB(I) is "NO".

Since this reduction is polynomial-time, CIRCUIT−SAT can be used to solve *any* problem that is in **NP**.

> **Theorem**: CIRCUIT−SAT is **NP**-Complete.
>
> **Proof**: The first and last claims give us all that we need: CIRCUIT−SAT is in **NP**, and it is also **NP**-hard.

5.4 Implications of NP-Completeness

Something truly amazing has happened here. What we have shown is that the CIRCUIT−SAT problem is, in a very precise sense, the hardest problem in **NP**. Unfortunately this doesn't resolve the **P** vs **NP** question — we aren't millionaires yet. But it does give us two really important implications:

- If $\mathbf{P} \neq \mathbf{NP}$, then CIRCUIT−SAT is not in **P**. Since CIRCUIT−SAT is the hardest problem in **NP**, if there's any problem in **NP** that can't be solved in polynomial-time, CIRCUIT-SAT must be it.
- If CIRCUIT−SAT is in **P**, then $\mathbf{P} = \mathbf{NP}$. This is just the contrapositive of the statement above.

In other words, the whole grand question of **P** vs **NP** boils down to a simple question: can you solve CIRCUIT−SAT in polynomial-time? If you come up with an answer, in *either* direction, you get a million bucks (and everlasting fame, and groupies, and job offers, etc.).

6 More NP-Complete Problems

The best part about our awesome and impressive **NP**-hard proof for CIRCUIT SAT is... we'll never have to do it again! To prove a new problem is **NP**-hard, I can just give a reduction from CIRCUIT−SAT, rather than having to have a reduction from every problem in **NP**. Since CIRCUIT−SAT is **NP**-complete, any other problem is **NP**-hard *if and only if* it can be used to solve CIRCUIT−SAT!

As we move on, we'll be proving more and more problems are **NP**-complete. So the basic task in proving some new problem is **NP**-hard is just to give a reduction from *any known NP-complete problem.* As we find more and more **NP**-complete problems, the task gets easier and easier.

6.1 3-SAT

The $3-$SAT problem is all about boolean formulas — those that can be written with variables (True or False) and the operators AND ($\wedge$), OR ($\vee$), and NOT ($\neg$).

And $3-$SAT isn't about just any boolean formula, it's about those that can be written as a "conjunction of disjunctions", a.k.a. "product of sums", a.k.a. "conjunctive normal form (CNF)". Let's break this down. Don't worry, it's not too complicated.

A *variable* in a boolean formula is just like any other variable, except that it can only stand for True or False. We'll write variables as lowercase letters like x or y.

A *literal* is either a single variable, or the negation of a single variable. So x and $\neg x$ and y and $\neg y$ are all literals.

A *clause* is a "disjunction of literals", i.e., a whole bunch of literals OR'ed with each other. So $(x \vee y \vee \neg x)$ is an example of a clause with 3 literals.

FINALLY, a CNF formula is a "conjunction of clauses", i.e., a whole bunch of clauses AND'ed with each other. For example, here is a CNF boolean formula:

$$(\neg x \vee y \vee z) \wedge (x \vee y \vee \neg x) \wedge (\neg y \vee y \vee \neg z)$$

This formula is also special because *every clause has at most 3 literals in it.* This is exactly the kind of formula that $3-$SAT is about:

> **Boolean Satisfiability Problem**: $3-$SAT(F)
>
> **Input**: CNF boolean formula F with at most three literals in every clause
>
> **Output**: Does F have a "satisfying assignment", i.e., a setting of every variable to True or False that makes the whole formula True?

For the example CNF formula above, one satisfying assignment is x=True, y=False, and z=True. You can confirm that each clause evaluates to True with these settings, which makes the whole formula True.

Now how could we prove $3-$SAT is **NP**-complete? As always, there are two steps.

First we must prove that $3-$SAT is in **NP**. This *also* involves two parts: polynomial-size certificates, and a polynomial-time verifier. Both of these should feel pretty obvious to you at this point: the certificate is a single "satisfying assignment" of the variables, and the verifier plugs in the assignment specified by the certificate, and computes the value of the entire formula, confirming that it is True.

Now the tricky part of proving $3-$SAT is **NP**-complete will be to prove that it is **NP**-hard. We will do this by showing a reduction from CIRCUIT$-$SAT, our only known **NP**-complete problem at this point. In a nutshell, this means taking any input to the CIRCUIT$-$SAT problem (a boolean circuit), and solving CIRCUIT$-$SAT for that input by using $3-$SAT as a subroutine.

The reduction will work by transforming any given boolean circuit into a CNF formula (with at most three literals per clause), such that the original circuit is satisfiable *if and only if* the formula has a satisfying assignment.

The first step to doing this is: **every wire in the circuit becomes a variable in the formula**. By "wire" I mean: the original inputs to the circuit, and the outputs of every gate in the circuit. We can just number all these wires 1, 2, 3, …, and then call the variables $x_1, x_2, x_3, \ldots$. We saw an example in class of how this looks on an example circuit.

Now the original circuit satisfiability question is just asking whether there is an assignment of True/False to all these variables such that:

- All the gates work correctly. (Output wire from each gate is correctly set according to the input wires to that gate.)
- The output wire of the whole circuit is set to "True".

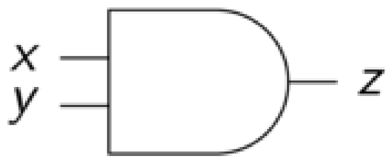
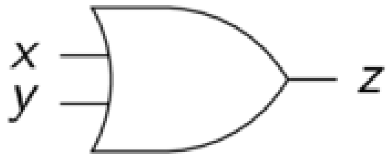
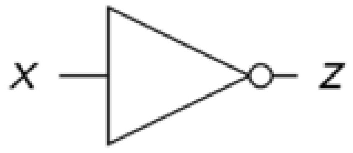
Gate	Formula
	$(\neg x \vee \neg y \vee z) \wedge (x \vee \neg z) \wedge (y \vee \neg z)$
	$(x \vee y \vee \neg z) \wedge (\neg x \vee z) \wedge (\neg y \vee z)$
	$(x \vee z) \wedge (\neg x \vee \neg z)$

Figure 2: Gate encodings

Our CNF boolean formula, then, just has to encode each of these two facts. For the first one, we will use the following table to translate every gate into two or three clauses in the formula, specifying essentially that the gate works correctly:

You can confirm that each of the formulas listed actually corresponds to the correct operation of that gate. For example, the formula for the negation gate basically says "one of the input/output wires must be true, and one of the input/output wires must be false". If this is the case, then that "NOT" gate is functioning properly.

We use these formula translations for *every* gate in the original circuit, then combine them all together with AND's to get one big CNF formula. But notice that every clause only has at most three literals, and the total number of clauses is at most 3 times the number of gates in the circuit. So the formula has *polynomial-size* in the size of the original input circuit. That's important!

The only missing piece of the formula now is the second point above: we have to ensure that the output wire from the entire circuit is set to "True". This is easy to do: just add one more clause that looks like (x), where x is the name of the wire for the output from the circuit. Then the whole formula is satisfiable if and only if (1) all the gates are functioning properly, and (2) the output from the circuit is True. Therefore this CNF formula has a satisfying assignment if and only if the original circuit is satisfiable. That's the reduction!

Here's an summary of the reduction that we just did:

1. Take any circuit C on which we want to solve CIRCUIT−SAT
2. Number all the "wires" in C, and make variables for them
3. Convert C into an equivalent CNF formula F using the rules in the table above, plus one more clause for the output wire
4. Feed F into any algorithm that solves 3−SAT
5. Return the answer to 3−SAT(F) — it's the same as the answer to CIRCUIT−SAT(C).

Since this reduction works in polynomial-time, we conclude that

Lemma: CIRCUIT−SAT $\leq_{\mathbf{P}}$ 3−SAT

Now, since we already know CIRCUIT−SAT can be used to solve any problem in **NP**, and 3−SAT can be used to solve any CIRCUIT−SAT problem, 3−SAT can also be used in a round-about way to solve any problem in **NP**! Since we also proved that 3−SAT is in **NP**, this means that

Theorem: 3−SAT is **NP**-complete.

Awesome! Hopefully you agree that this was much easier than the last reduction, from *any* **NP** problem to CIRCUIT−SAT. And now that we know *two* **NP**-complete problems, we can use either one as the basis for our next **NP**-hard proof!

6.2 Vertex Cover

Time for a really surprising reduction. We want to prove that VC is hard, but all we know about so far are these two problems about the satisfiability of circuits and boolean formulas. As it turns out, there is a connection from the seemingly unrelated problem of 3−SAT to VC. Here is the reduction:

```
3−SAT−to−VC(F)
Input: Boolean formula $F$ in CNF form with three literals per clause (input to a 3−SAT problem).
Output: Is there a satisfying assignment for the variables in $F$?

  1. Create an empty graph $G$.
  2. For each variable $x_i$ that appears in $F$, create two nodes labeled $x_i$ and $\neg x_i$, with an edge between
     them, and add the two nodes and one edge to $G$
  3. For each clause $(L_1 \vee L_2 \vee L_3)$ in $F$, create a “triangle” of three nodes $L_1$, $L_2$, and $L_3$, with three
     edges connecting them. (Remember that each “literal” $L_i$ is either a variable $x_i$ or the negation of
     a variable $\neg x_1$.) Add the three nodes and three edges to $G$.
  4. Add additional edges between every node in each triangle to the node with the same label in the
     original pairings, to $G$.
  5. Return $\texttt{VC}(G, v + 2c)$, where $v$ is the number of variables and $c$ is the number of clauses in $G$.
```

Here’s why this works. Each of the edges in the original pairings must have at least one endpoint vertex in the cover: these v vertices correspond to the satisfying assignment of the variables. Each triangle must have at least *two* of its vertices in the cover: these $2c$ vertices correspond to the 2 literals in each clause that *could* be false. The third literal in each clause — the one that must be true — corresponds to the vertex in the triangle that is not in the cover. Its edge to the vertex in the matching must be covered by that vertex in matching. This corresponds to saying that literal is true because of that variable’s setting in the satisfying assignment. So a satisfying assignment exists if and only if there is a vertex cover of size $v + 2c$. Awesome!

I acknowledge that this is a difficult reduction. Reviewing the pictures we drew on the board in class would be helpful. But the important point to remember is that *we can have reductions between seemingly unrelated problems*. And the payoff is sweet: now we know that VC is **NP**-hard! Since we already proved it was in **NP**, we actually know that it is **NP**-complete. Another “hardest problem in **NP**”!

6.3 More NP-Complete Problems

This process can, and has, continued to show a whole bunch of problems are **NP**-complete. In fact, Wikipedia claims that there are thousands of them — all of them of course being the “hardest problem in **NP**”.

And this is actually a good thing! It means that, to show a new problem is **NP**-hard, all we have to do is come up with a reduction from some known **NP**-complete problem to the new problem. This is a much easier task than the very first **NP**-hard proof for CIRCUIT−SAT, and as we learn about more and more **NP**-complete problems, the chances that any new problem we see is related to one of them should increase.

The **NP**-complete problems we have looked at so far are:

- LONGPATH
- VC
- HITSET

- HAMCYCLE
- CIRCUIT−SAT
- 3−SAT
- SPLIT−EVENLY

Notice something missing? It's FACT, the integer factorization problem that was one of our original motivations for studying the inherent difficulty of problems. The basic issue with FACT is that, unlike all the examples above, it's easy to verify both "YES" **and** "NO" answers to FACT. The certificate for either one would just be a list of the prime factors of the input integer N. This means FACT is not only in **NP** but also in another complexity class called **coNP**, and the fact that FACT is in both of these makes it extremely unlikely to be **NP**-complete. In any case, most researchers still don't believe the problem is solvable in polynomial-time, but it is classified as an "**NP**-intermediate" problem, somewhere in the ether between **P** and **NP**.

7 Traveling Salesman

Say you are a delivery truck driver and you have a bunch of stops for deliveries. You want to schedule a route that starts and ends at your warehouse and visits every stop exactly once. And of course you'd like to do this in the shortest time possible! So how do you decide the route? That's this problem:

Traveling Salesman Problem: TSP(G)

Input: Weighted graph G

Output: Least-weight cycle in G that goes through every vertex exactly once.

This is an important problem with numerous other applications, and it's also notoriously difficult to solve. In fact, as we'll see in a moment, TSP is **NP**-hard, so we don't expect there to be any polynomial-time algorithm in general. What we're going to look at is a quick overview of various strategies to *try and solve it anyway.* The approaches we look at will either have to compromise in running time (taking worst-case exponential time) or in the quality of the answer (not always giving the optimum answer).

Here is a graph for an example TSP problem that we'll use repeatedly:

The dashed lines both have length 6 (not drawn). You should convince yourself that the optimal TSP tour has total length 18.

7.1 TSP is NP-Hard

This is a "specific-to-general" reduction, where we show a reduction from a specific case to a more general problem. Hopefully the problem we will reduce from is fairly obvious: HAMCYCLE. Here is an algorithm to solve HAMCYCLE using TSP:

1. Take the input graph to HAMCYCLE, and make it weighted by setting every edge to weight 1.
2. Run TSP on the weighted graph. The TSP returns a Hamiltonian cycle, if there is one.

Therefore HAMCYCLE $\leq_{\mathbf{P}}$ TSP. Since HAMCYCLE is a known **NP**-complete problem, then TSP must be **NP**-hard.

7.2 Minimum Spanning Trees

Just like matchings were used to approximate the vertex cover problem, minimum spanning trees (MSTs) are often used to approximate TSP. Here's the connection:

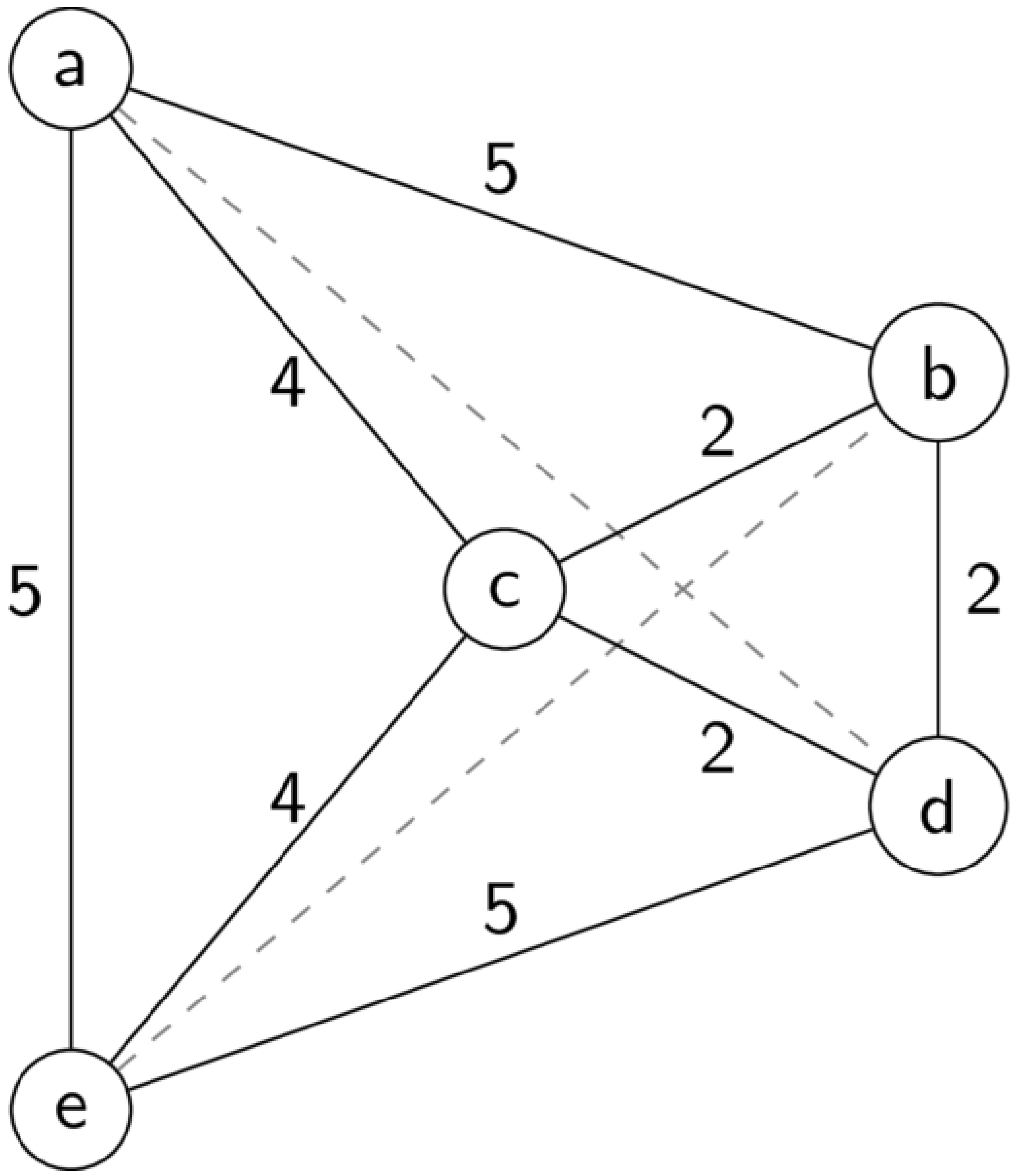


Figure 3: Example TSP problem

Theorem: The length of any travelling salesman tour in a graph G is at least as much as the total size of a minimum spanning tree in G.

Proof: Take any travelling salesman tour (cycle); call it T. Now remove the last edge from that cycle. The result is a path that touches every vertex exactly once; call this path P.

What do we know about P? Well, it connects every vertex and it has exactly $|V| - 1$ edges. Therefore… it's a tree! In fact, it's a *spanning tree* since it touches every vertex. Therefore the length of P must be greater than or equal to the size of any MST in the graph.

And finally, since P was formed by removing an edge from T, the length of the original travelling salesman tour T must be greater than the length of P, which in turn is at least as large as the size of any MST.

Since we know at least two fast (polynomial-time) algorithms to compute MSTs (Prim's and Kruskal's), this theorem provides a quick way to find a lower bound on the size of the optimal TSP solution. (This is similar to how the size of any matching gave a lower bound on the size of a minimum vertex cover in a graph.)

7.3 Branch and Bound

Here's a basic algorithm to compute the TSP:

1. Pick a starting vertex u arbitrarily
2. Explore every possible path in the graph from u, depth-first.
3. When the depth-first search gets to a leaf (can't go any further), check if the path contains every vertex. If so, add the edge back to u from the ending vertex, and that gives a Hamiltonian cycle
4. After completing the depth-first exploration, return the least-weight Hamiltonian cycle that has been found.

Now this is a very slow algorithm; it's cost in the worst case is something like $\Theta(|V|!)$, which is the total number of paths in the graph. This is not particularly surprising — since TSP is **NP**-hard, we know that there isn't going to be any algorithm to solve it that's polynomial-time in the worst case. (Unless **P**=**NP**, which very few people expect to be true.)

So what can we do? How about instead of trying to improve the worst case, we improve the *best case* instead? The idea is to compute the optimal solution as fast as we can, and in many cases it should work in polynomial-time.

"Branch and bound" is a popular and effective method for finding optimal solutions to **NP**-hard problems. What branch-and-bound requires is a quick way to estimate a *lower bound* on the cost of the remaining TSP tour, given some partial tour (i.e., a path). This lower bound estimate is usually called a "heuristic" (this corresponds to the same idea in the A* shortest-path search that you may have heard of). We will use the heuristic to speed up the search as follows:

- The order vertices are explored in is *in order of the heuristic.* If the heuristic actually gives a good estimate on the true cost of the remaining TSP tour, then this will lead us to finding the optimal tour sooner rather than later in the DFS.
- We always keep track of the best solution found so far. If the heuristic gives a *lower bound* on the remaining subproblem that is worse than the best TSP tour that we've already found, then we can **stop exploring** from that path and move on to the next option.

Basically, what the heuristic gives us is a way to (1) find the optimal solution quickly, and (2) eliminate sub-optimal solutions from the search without having to completely explore them.

Now the very best heuristic would be to just find the optimal TSP tour on that subproblem — but of course we don't want to do this because that would take a really long time! So how could we get a quick lower bound on the optimal TSP tour? We just saw it — compute the minimum spanning tree! Specifically, given a partial TSP tour (path in the graph), the heuristic adds up the length of the path, the shortest edges going out from the ends of the path to the rest of the graph, and the size of a MST in the rest of the graph. This gives a lower bound on the smallest TSP tour that includes that path.

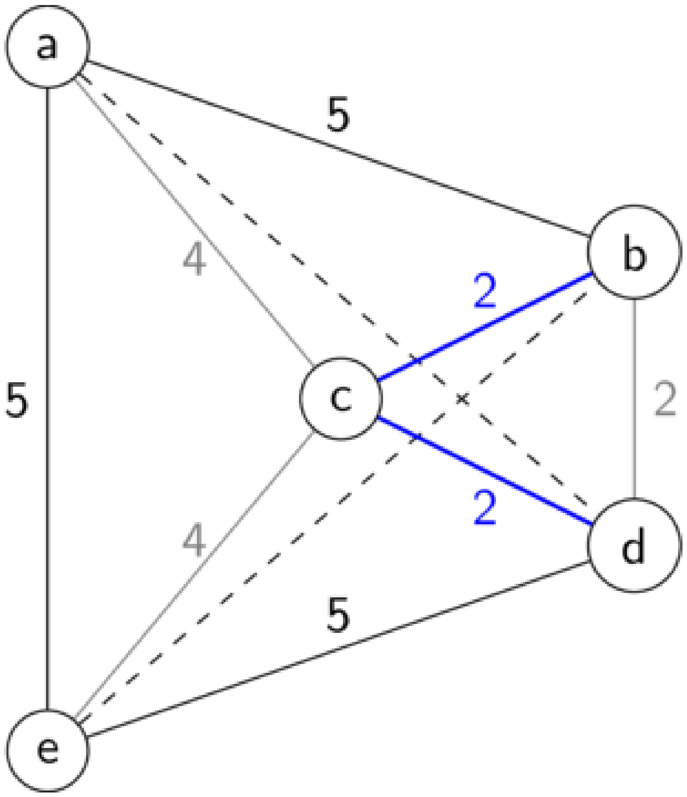


Figure 4: Partial TSP tour

Here's a concrete example. Consider the following graph, with the partial tour explored so far highlighted in blue (b-c-d), and the edges that can't possibly be part of this tour reduced to gray:

The branch-and-bound heuristic would first compute a MST in the rest of the graph not including the path so far. In this case that is just a MST on the two vertices a and e, which will consist of just the edge between them, weight 5. Then we add the length of the path so far (4), plus the shortest edges going from the ends of this path to the rest of the graph (5 and 5), plus the MST on the rest (5), for a total of 19. So if the optimal length-18 solution had already been found, this path could be eliminated without having to explore it further.

I know this seems really trivial on such a small graph, but on much larger graphs a huge savings is obtained from being able to eliminate large parts of the depth-first exploration. Many other heuristics have been developed as well, some that work only in certain situations. There is always a balance between using a very fast but inaccurate heuristic (makes the search go faster, but has to explore more), or a slower but more accurate heuristic (more time spent on the heuristic, less time spent exploring). The best choice usually depends on some intuitions about whatever the actual *meaning* of the graph is for a particular application.

7.4 Metric TSP

In many applications, the graph for TSP corresponds to some kind of actual map where the nodes have actual locations, and the distances between them correspond to the weights of the edges in the graph. This is called the *Euclidean TSP* problem because the nodes are just points on a 2-dimensional Euclidean plane.

One property that points on a plane obey is the *triangle inequality*: the distance from a to b is always less than or equal the distance of a to c plus c to b; that is, detours never make the trip faster. In other words, *the shortest path between any two vertices is the single edge between them.* Since this is a property that can be obeyed by graphs besides just the ones that are points in a plane, this is called the *Metric TSP* problem.

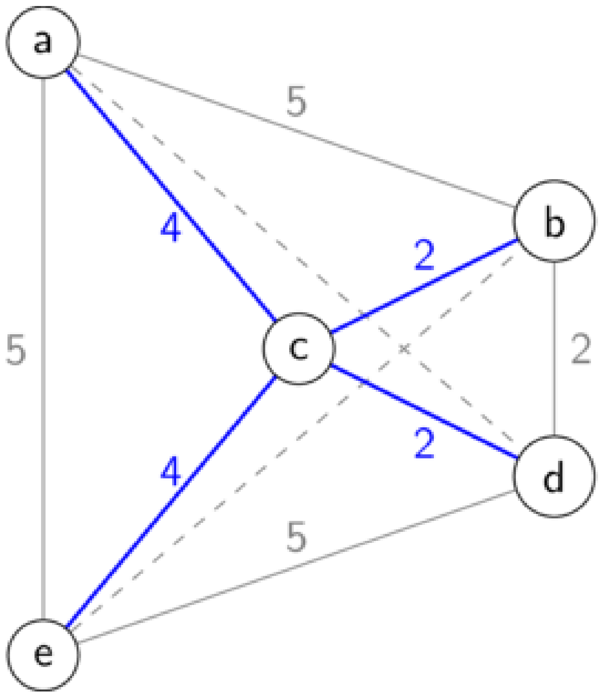
Unfortunately, Metric TSP is still **NP**-hard, so we're still not going to have a polynomial-time algorithm. Notice that the branch-and-bound approach above *always* finds the optimal solution, and *sometimes* takes polynomial time.

What we'll do here is flip that around, so that we *always* take polynomial-time, but only *sometimes* find the optimal solution.

Again, the idea boils down to minimum spanning trees! Here it is:

1. Compute a MST of the graph.
2. Duplicate every edge in the MST. This makes a cycle with *duplicate edges*.
3. Go through this duplicated-edge cycle, and just skip every vertex that gets repeated, taking the shortcut to the next *unvisited* vertex instead.
4. The result is a TSP tour!

Here's an example to see how this works. On the left is a MST of the graph, which gives the cycle $(a, c, b, c, d, c, e, c, a)$. Now we just go through this but remove every *second* occurrence of a vertex (besides revisiting a at the end); the result is (a, c, b, d, e, a) — a length-18 minumum TSP tour!



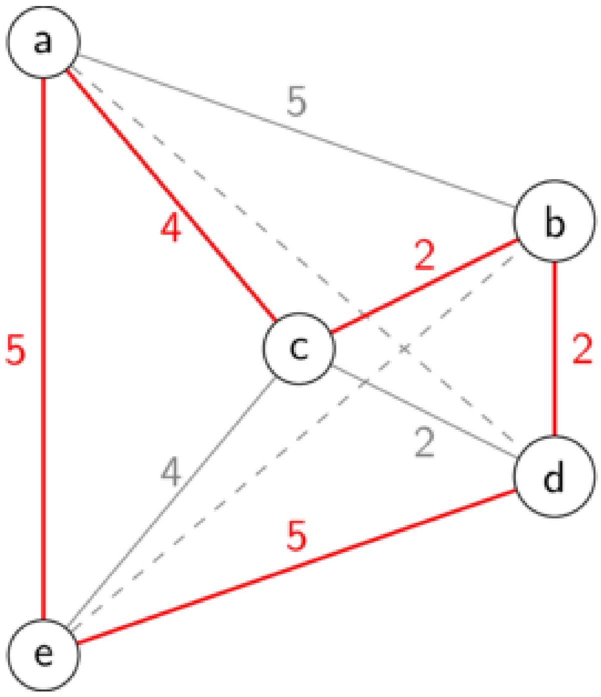


Figure 5: Approximating Metric TSP with MSTs

Of course, we won't always be quite so lucky to end up with the optimal solution, but how bad could this be? Well, we know the size of the MST is less than the optimal TSP tour, so the original cycle from step (2) could be at most two times the optimal (since every edge in the MST is repeated). And then because this is a *metric TSP*, taking the "shortcuts" on step (3) can only make the cycle shorter. Therefore this algorithm always returns a TSP tour whose length is at most 2 times the optimum length.

This is eerily similar to the factor-2 approximation algorithm we saw for vertex cover by using a maximal matching. It's kind of awesome that these two **NP**-hard problems can't be solved exactly in polynomial-time, but can be approximated very quickly. This isn't true of every **NP**-hard problem — in fact, if you could approximate the original TSP problem (without triangle inequality) to a factor of 2 in polynomial-time, you would have proven that **P**=**NP**!

7.5 Greedy TSP

Our final approach for solving TSP is by using greedy algorithms. Of course, not every greedy algorithm is created equal! Here are two ideas:

- **Nearest neighbor**: Start an any arbitrary vertex and create a path, at each step choosing the next vertex that is closest to the current endpoint and hasn't been included in the path yet. At the end, just connect the endpoints of the path to get a cycle.
- **Smallest good edge**: Repeatedly add the shortest edge that only touches at most one edge already added. This will again result in a path, and we finish by connecting its endpoints.

Notice that both of these greedy algorithms require that every possible edge in the graph exists (this is called a *complete* graph). In class we saw how each greedy algorithm works on our example graph, resulting in the following two sub-optimal solutions:

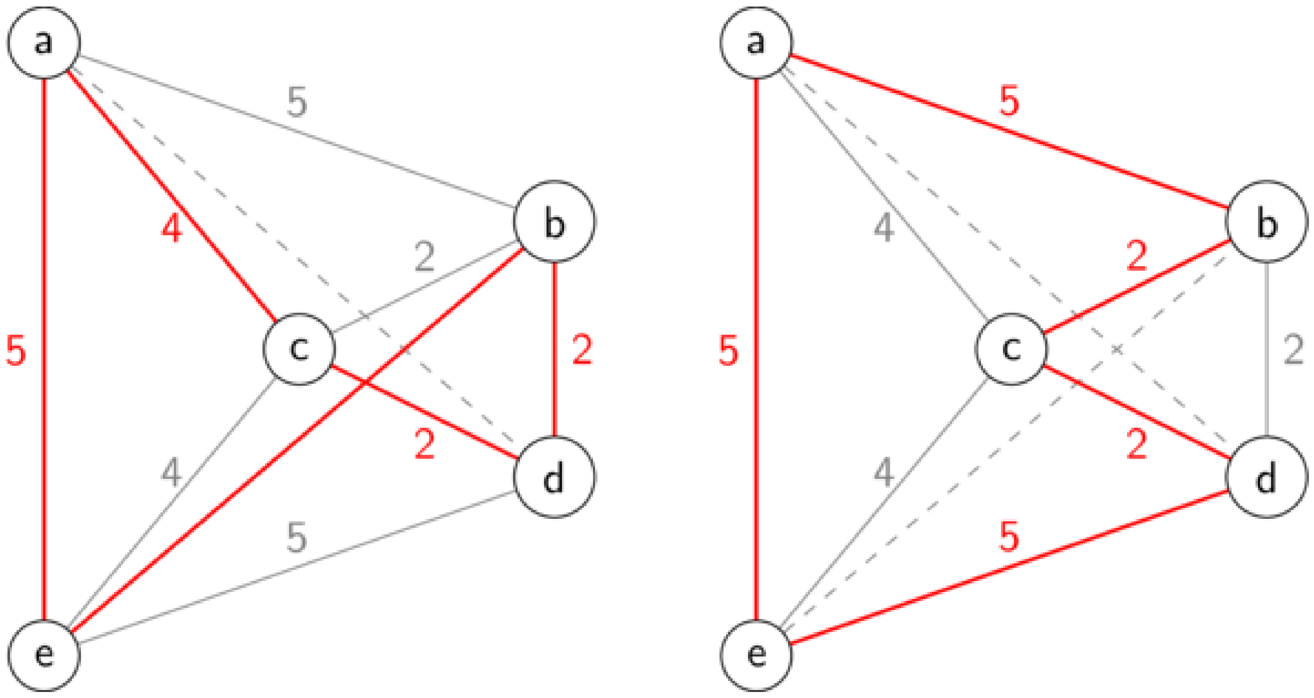


Figure 6: Greedy TSP algorithms

Of course these greedy approaches don't give optimal solutions, but they will find candidates very quickly.

7.6 Iterative Refinement

Once we have an almost-optimal solution, from the MST-based approximation or from a greedy approach, we don't have to stop at that! There's a general approach called *iterative refinement* that takes a sub-optimal solution and tries to make it better, by changing some small part.

The iterative refinement technique that gets used most commonly for TSP is called "2-OPT" and it essentially works by taking any *quadrilateral* (4-node loop) that appears in the graph, and swapping two edges of the quadrilateral that are in the current TSP tour, for two edges that are not. That is, we take two edges (w, x) and (y, z) that are in the current tour, and we replace them with (w, y) and (x, z).

To see how this fits into the whole big picture, consider the whole tour, which must look like

$(w, x, \text{PATH1}, y, z, \text{PATH2}, w)$

where PATH1 goes from x to y and PATH2 goes from z back to w. What the 2-OPT refinement does is switch those two edges, which also necessitates reversing PATH1 into PATH1-REV, which goes from y to x, so we get:

$(w, y, PATH1 - REV, x, z, PATH2, w)$

Voilà! The cool thing is, we can do this for any pair of edges in the current path. The "OPT" part of 2-OPT is that we choose the two edges to swap that will give the *most* improvement.

For example, in the TSP tour produced by the "nearest neighbor" greedy strategy above, we can swap the edges (a, c) and (b, d) (shown on the left in red) with the blue edges in the TSP tour on the right, which happens to be optimal.

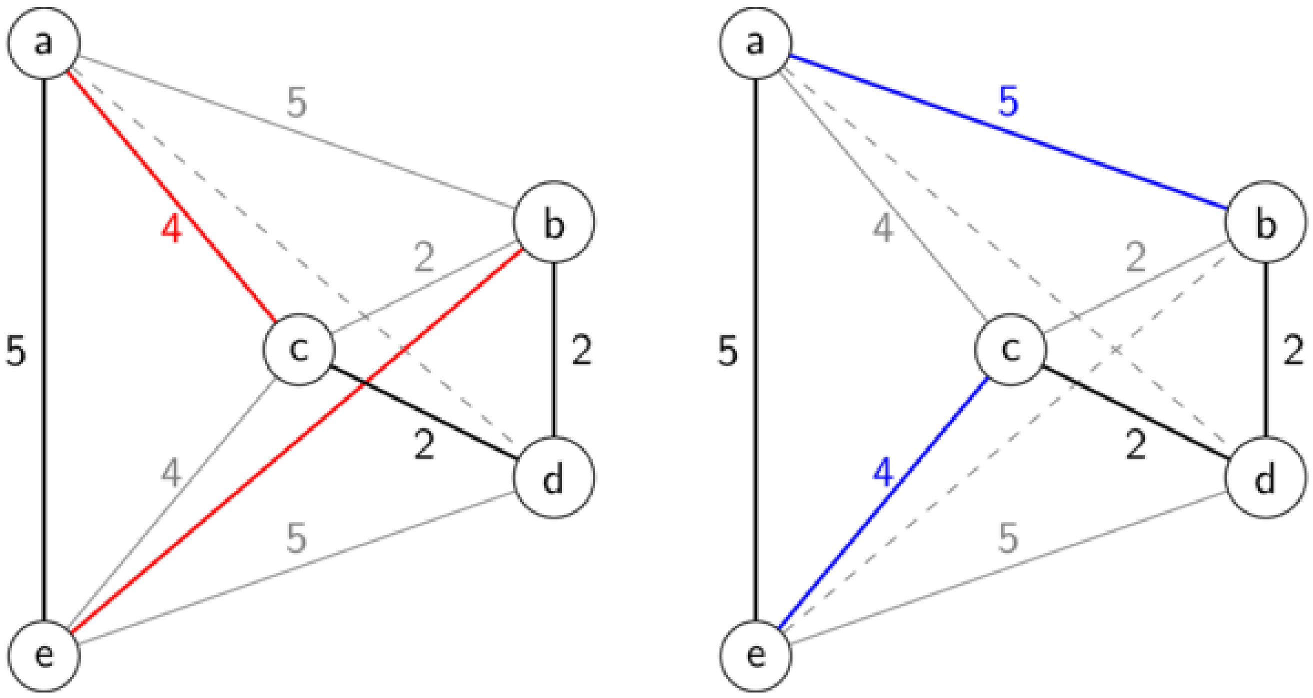


Figure 7: 2-OPT refinement

So now we have a better algorithm to solve TSP:

1. Get a TSP tour using the MST approximation or a greedy algorithm
2. Refine it with 2-OPT
3. Keep repeating (2) until no 2-OPT refinement can improve the cost

Each 2-OPT refinement only needs to examine every pair of edges in the tour to see what the best pair to swap will be. Since there are $|V|$ edges in the tour, the cost of each 2-OPT step is $\Theta(|V|^2)$ — polynomial time! Step (1) obviously costs polynomial-time too, and yet we know that this algorithm doesn't produce the optimum TSP solution in polynomial-time. So what's going on? One of two things must be true: Either

- We can have an exponential chain of 2-OPT refinements before finding the optimum, or
- It's possible to have a tour which can't be improved by any 2-OPT move, but yet which is not optimal.

In fact, one of these two properties must hold for *any* iterative refinement technique for an **NP**-hard problem (assuming of course that **P** doesn't equal **NP**). Can you figure out which one it is for 2-OPT?