Computational Complexity Theory

Nikhil Sardana

January 2020

This lecture is the next installment in our series on Turing machines. Today, we're not just going to ask "What problems can a Turing machine solve?," but "What problems can a Turing machine solve *efficiently*?" By analyzing the runtime of problems on a Turing machine, we can determine which problems our real-world computers can solve efficiently—and which ones we've got no hope of solving for large inputs. In this lecture, all problems will be decidable. We will focus on analyzing bounds on the time it takes a Turing machine to solve a problem for any input of a given length.

1 Time Complexity

I hope you remember Big-O notation. It's a measure of the runtime of an algorithm in terms of the input size, disregarding all constants and lower-order terms.

For example, an algorithm that sums over an array of length n runs in time $O(n)$.

```
Algorithm 1: SUM
  Input: An array $A$ of length $n$
  Output: The sum of the array $A$
  $s = 0$
  for $j \in \{0, \dots, n-1\}$ do
    $s+ = A[j]$
  return $s$
```

Most students first encounter Big-O notation when they learn simple sorting algorithms. On an array of length n, the following sorting algorithm takes time $O(n^2)$. You may have also seen faster algorithms, like Quicksort and Merge sort. Merge sort, for example, takes worst-case time $O(n \log n)$. $O(\log n)$ factors arise when we repeatedly split arrays (or other data structures) in half, and only consider half the information at each step.

In the algorithm below, looping over i from 0 to n and j from i to n takes only $\frac{n^2}{2}$ iterations total. However, since Big-O notation ignores constants, multiples, and lower-order terms, we say the algorithm runs in time $O(n^2)$.

Now, how does this relate to Turing machines? Well, we can also measure time complexity when we run an algorithm on a Turing machine. For some language L, we require a certain time on a Turing machine M to decide L. For an input string w, it takes M a certain number of steps to answer the question of whether or not w is in L.

Algorithm 2: SORT

```
Input: An array $A$ of length $n$
Output: A sorted array of $A$'s elements
for $i \in \{0, \dots, n-1\}$ do
    $min = i$
    for $j \in \{i, \dots, n-1\}$ do
        if $A[j] < A[min]$ then
            $min = j$
    $temp = A[i]$
    $A[i] = A[min]$
    $A[min] = temp$
return $A$
```

From now on, we're only going to consider *decidable* languages, so every language L has a Turing machine that accepts every string $s \in L$ and rejects every $s \notin L$. It doesn't make much sense to consider the runtime of undecidable languages, because for undecidable languages, every Turing machine will loop for *infinite* time on some inputs. And infinite isn't a very interesting runtime.

Definition 1.1. The time complexity of a language L is the Big-O time of the most efficient Turing machine that decides L. In other words, a language has time complexity $O(f(n))$ if there exists a Turing machine M that decides *all inputs* of length n in $O(f(n))$ steps or less. In this case, we say that $L \in TIME(f(n))$.

When we talk about "steps" of a Turing machine, we don't mean just reading a cell or writing to the tape. We consider one single "step" of a Turing machine to be the following operations:

1. Reading a cell
2. Changing head state based on the read.
3. Writing to a cell.
4. Moving the head.

Each "step" takes constant, or $O(1)$, time.

Now that we understand how to measure runtime on a Turing machine, let's calculate the time complexity of some languages.

Example 1.1. $\{0^n1^n \mid n \in \mathbb{N}\} \in TIME(n^2)$.

Let's construct a Turing machine that takes at most $O(n^2)$ steps on any input of length n. Oh wait! We already did this last lecture. Here's our algorithm from before:

1. First, check over the entire input. If we ever reach a character that is not 0 or 1, reject. If the tape is blank, accept. TIME: $O(n)$.

2. Starting at the leftmost character on the tape that is not an x. If we read 0, replace 0 with x. Otherwise, reject. TIME: $O(n)$ to travel back to the start, $O(1)$ to check the character.

3. Travel to the rightmost character on the tape that is not an x. If we do not read 1, reject. Otherwise, replace 1 with x. TIME: $O(n)$.

4. Repeat steps 2 and 3 until there are no more 0s or 1s. If, after repeating step 3, there are no zeros but there are 1s, then we know that our input was of the form 0^m1^n, $n \geq m$. So, we reject if we reach this state. If we reach a point where there are no 1s but there are 0s, then we know that our input was of the form 0^m1^n, $n \leq m$. So, we reject if we reach this state. If there are no zeros and no ones, then we know that our input was of the form $0^m1^m, m \in \mathbb{N}$. Thus, we accept. TIME: $O(n)$ repetitions, $O(1)$ work per repetition.

The above algorithm walks back and forth across our input of length n (steps 2 and 3) a total of $\frac{n}{2} = O(n)$ times (step 4), performing $O(1)$ work each time. So, the algorithm takes time $O(n^2)$ over any input, and thus $\{0^n1^n \mid n \in \mathbb{N}\} \in TIME(n^2)$.

This example illustrates why we say $L \in TIME(f(n))$ if there's a Turing machine that decides L for *any* input of length n in $O(f(n))$ steps. There are some inputs where we will trivially reject. In the above example, the input 1^{2n} will be rejected in $O(1)$ time. But saying $\{0^n1^n \mid n \in \mathbb{N}\} \in TIME(O(1))$ because we can solve the easy inputs quickly doesn't help us much. It's the hard cases, the $0^{n-1}1^{n+1}$'s of the world, that we really care about. No matter how hard the input, we want to know how fast you'll decide it, so we'll categorize languages, like the one above, by their *worst-case* time complexity.

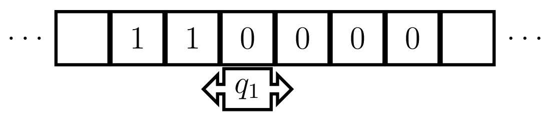
It turns out that the lower bound for $\{0^n1^n \mid n \in \mathbb{N}\}$ is $TIME(n\log(n))$, but this is a more complicated algorithm. (Like in regular algorithm runtime analysis, if an algorithm takes $O(n\log(n))$ time, it also runs in $O(n^2)$ and $O(n^3)$ and $O(2^n)$ time. Big-O is simply an upper bound on computation time. We traditionally associate the smallest possible Big-O time with an algorithm, for example, we say Quicksort takes worst-case time $O(n^2)$ because $O(n^2)$ is the tightest bound.)

Exercise 1.1. Let $L = \{ww \mid w \in \{0,1\}^*\}$. Is $L \in TIME(n^2)$?

1.1 Multi-Tape Turing Machines

Now, we aren't going to formally prove this, but multi-tape Turing machines (for example, a Turing machine with two tapes, and two different heads) are equivalently powerful to our standard single-tape Turing machines. Intuitively, you can think the equivalence as follows.

Suppose we have a 3-tape Turing machine. After some computation, the three tapes contain the following values, and the heads are in the positions below.



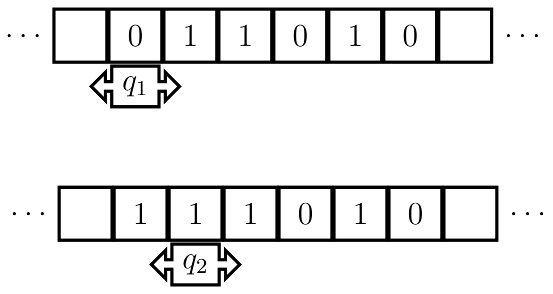


Figure 1: A 3-tape Turing Machine.

We can simulate this on a single-tape Turing machine by separating the tape states with a special character # and placing a special symbol (denoted by the underline) to remember where the three heads are.

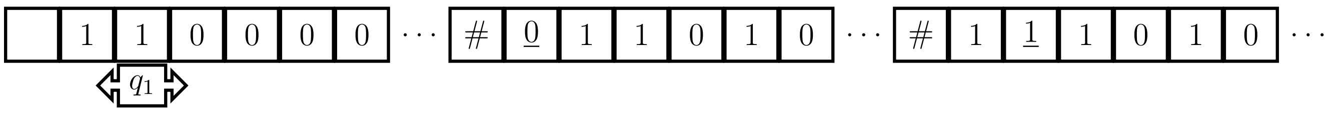


Figure 2: An equivalent single-tape Turing Machine.

Now, if on the 3-tape Turing machine we move the first head, and then move the second, and then move the third, reading and writing with each one, we can replicate this on our single-tape model with a bit more work. We move the head on the first section of our tape, and mark where we end with the special underline symbol. Then, we travel rightwards until we see the special #, and we now know we are on the "second tape". We look for the next underlined cell ($\underline{0}$), and simulate the second head, and so forth for the third head.

If we run out of space between #s on our single tape, we can always use the head to shift the second and third tape characters rightward, starting from the rightmost non-blank cell.

So, why does this all matter? Well, as you can imagine, depending on the language, it could take us quite a bit longer to decide membership using a single-tape Turing machine versus a multi-tape Turing machine.

Example 1.2. $\{0^n 1^n \mid n \in \mathbb{N}\}$ can be decided by a 2-tape Turing machine in $TIME(n)$. We simply copy the input from the first tape onto the second tape, and move the second head halfway down. Every time the second head reads a 1 on the second tape, we cross off 0 on the first tape and move one cell right. After $\frac{n}{2}$ movements by the first and second heads, the first head should reach the 1s having seen and crossed off only zeros, and the second head should reach the end of the input having seen only 1s. Otherwise, we reject. This algorithm takes time $O(n)$, and works for every input.

1.2 Polynomial Time

So then, which model do we use? Do we say that $\{0^n 1^n \mid n \in \mathbb{N}\} \in TIME(n \log n)$, because the best single-tape algorithm takes $O(n \log n)$ over all inputs at best? Or, do we say that $\{0^n 1^n \mid n \in \mathbb{N}\} \in TIME(n)$, because, after all, with only one more tape we can easily achieve this speedup?

One answer might be to say: well, which model most accurately models our real-world computers? But again, we want our answers to generalize—not just to today's computers, which have certain implementation details and architecture speedups and hardware specifications—but to most real-world computers, including future ones. And besides, no Turing machine really comes close to our real-world setups at all—our modern CPUs certainly don't operate with single tape head moving back and forth.

So, what do we do? It turns out that any algorithm can be simulated by a single-tape Turing machine in *at worst* $O(n^2 f(n))$, where $O(f(n))$ is the best time complexity achieved by a multi-tape Turing machine. For this reason, any algorithm that takes polynomial time on a single-tape Turing machine ($TIME(f(n))$, where $f(n) = n^k$ for some constant k) will also take polynomial time on a multi-tape Turing machine, with any number of tapes. **We call the set of languages that are decidable by a Turing machine in polynomial time $\boldsymbol{P}$.**

$$P = \bigcup_{k \in \mathbb{N}} TIME(n^k)$$

Problems in P correspond to problems we can solve in a reasonable amount of time on our computers. Problems outside of P correspond to ones we can't solve. Think about it: every exponential eventually dominates a polynomial. You'll struggle to run an $O(2^n)$ algorithm for $n \geq 40$. To reach the same computation on an $O(n^3)$ algorithm takes $n \geq 10,000$.

2 Nondeterministic Polynomial Time

2.1 Nondeterministic Turing Machines

Deterministic finite automata (DFAs) have a nondeterministic counterpart (NFAs). Similarly, we can imagine a nondeterministic version of a Turing machine. Instead of the Turing machine's transition function being explicitly defined (like in our diagram for deciding $\{0^n 1^n \mid n \in \mathbb{N}\}$), we allow ourselves multiple transitions from a state, each of which results in a different action but corresponds to reading the same character.

Just like in an NFA, we think of a nondeterministic Turing machine as taking all possible computation paths simultaneously whenever it has multiple options. For any given input string, if even a single one of the nondeterministic Turing machine's computation paths leads to an accepting state, the machine accepts the string.

Exercise 2.1. What is the language of the Turing machine in Figure 3?

As you might expect, nondeterministic Turing machines are no more powerful than a deterministic ones. However, a nondeterministic Turing machine can compute problems faster—after all, since it takes computation branches simultaneously, it can take all possible computation paths for the price of the longest one.

Definition 2.1. A language L is in $NTIME(f(n))$ if there exists a nondeterministic Turing machine M that decides if $x \in L$ for *all inputs* x of length n in $O(f(n))$ steps or less.

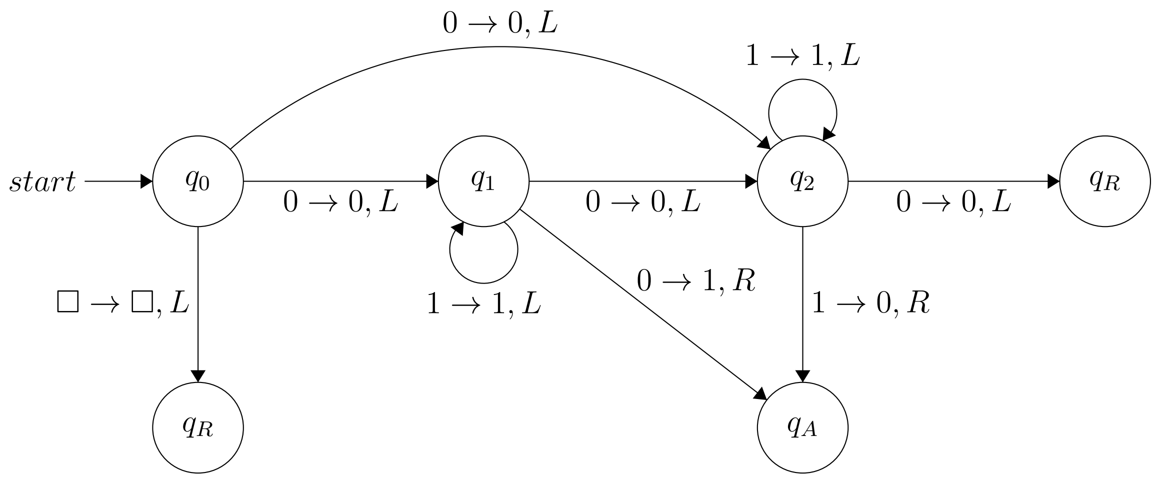


Figure 3: An example of a transition function for a nondeterministic Turing machine.

Let's look at a few languages and see how long it takes nondeterministic Turing machine to decide them. But first, we introduce graphs. There are many important problems involving graphs where nondeterminism can help naturally.

Definition 2.2. A **graph** is a set of vertices and edges connecting the vertices. We denote a graph $G = (V, E)$, where V is the vertex set and E is the edge set. We're only going to consider simple graphs—graphs which have at most a single edge between any two vertices.

You've certainly seen graphs before—and hopefully you've even written algorithms for traversing or coloring them.

We're going to consider two properties of graphs: the *clique size*, and *Hamiltonicity.*

A **clique** of a graph G is the set of vertices of a complete subgraph of G. If $G = (V, E)$ has a $k-$clique, there is some subset of V of size k such that every pair of vertices in the subset is connected by an edge. For example, the largest clique in the graph below has size 4. The four vertices that make up the clique, along with the edges between them, are highlighted in blue.

A **Hamiltonian path** is a path of edges that goes through every vertex in a graph and touches each vertex only once. A **Hamiltonian cycle** is a Hamiltonian path that starts and ends at the same vertex. The graph in Figure 4 has a Hamiltonian cycle, shown in red.

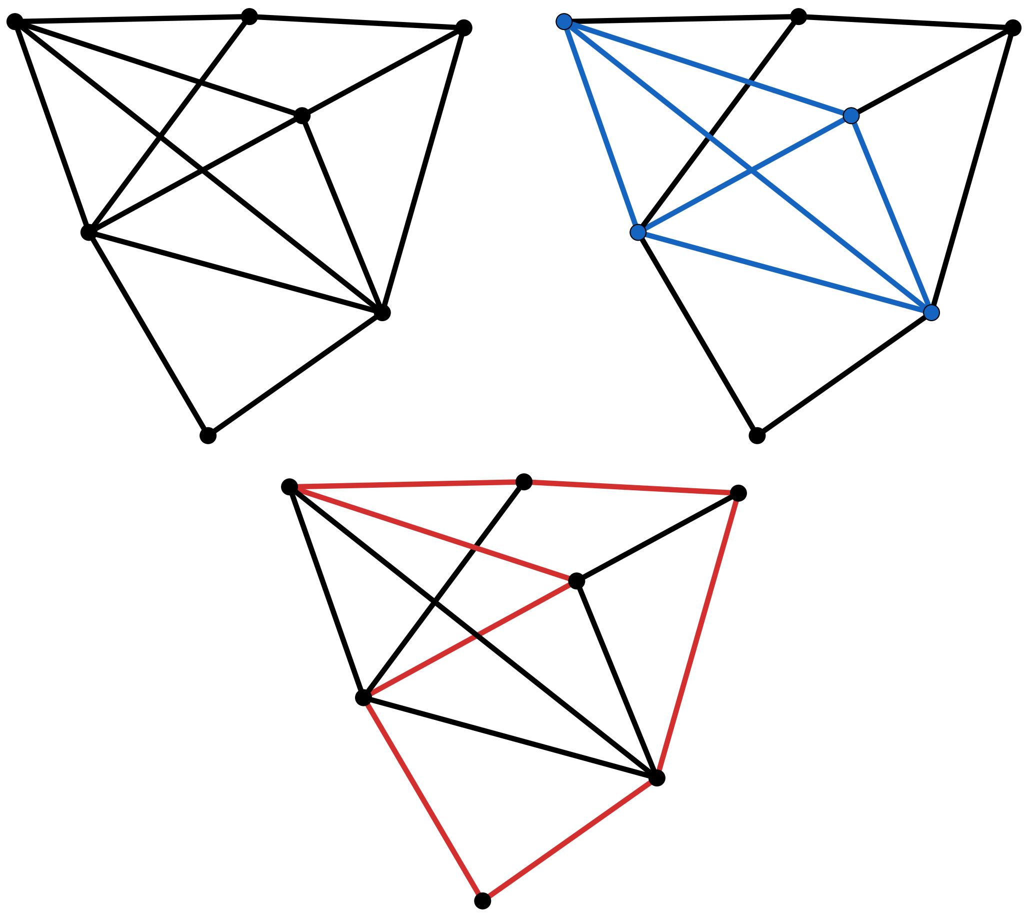


Figure 4: A graph with a 4−clique and a Hamiltonian cycle.

Example 2.1. Given a graph $G = (V, E)$ with n vertices, how can we tell if G has a clique of size k using a nondeterministic Turing machine? In other words, how can we determine if G is in the the language

$$CLIQUE(n, k) = \{G \mid G \text{ is a graph with } n \text{ vertices and contains a clique of size } k\}$$

Well, since there are n vertices, there are $\binom{n}{k}$ subsets of k vertices. For some subset $S \subset V$ of k vertices, we can quickly check if S forms a clique or not. All we need to do is check over edge set E to see if all $\binom{k}{2}$ edges connecting the elements of S are present. Since $|E|$ is at most $\binom{n}{2} = O(n^2)$, we can do this in $O(n^2)$ time.

But that's only one possible subset. However, using a nondeterminstic Turing machine, we can nondeterministically check *all* subsets of size k at the same time. If any one of them is a valid clique, then our Turing machine accepts. If none of them are, then G does not contain a k-clique—we've checked all the possibilities. So, our Turing machine rejects.

Thus, our entire algorithm correctly decides $CLIQUE(n, k)$ in $NTIME(n^2)$.

Example 2.2. Given a graph $G = (V, E)$ with $|V| = n$ vertices, how can we tell if G contains a Hamiltonian path using a nondeterminstic Turing machine?

To determine if G is in the following language

$$HAMPATH(n) = \{G \mid G \text{ is a graph with } n \text{ vertices and contains a Hamiltonian path}\}$$

we begin by noting there are $n!$ orderings of the vertices of G. So, there are at most $n!$ possible paths. If *any one* of the $n!$ orderings $(v_1, v_2, \ldots, v_n)$ has an edge between v_1 and v_2, and v_2 and v_3, etc. to $v_{n-1}v_n$, then G contains a Hamiltonian path. Given a specific ordering, we can easily check if the ordering is a valid path by simply checking the edge set E for the necessary edges. We can do this in time $O(n)$. So, we nondeterministically check all possible permutations of the vertices of G simultaneously, and our Turing machine accepts if any single permutation is accepted. So, $HAMPATH(n) \in NTIME(n)$.

Again, because of issues with multiple tapes, we're not going to care if an algorithm runs in $NTIME(n^2)$ or $NTIME(n^3)$. We only care if an algorithm can be run relatively quickly or not—i.e., whether it runs in polynomial time or not.

We call the set of languages that are decidable by a nondeterministic turing machine in polynomial time *NP*.

$$NP = \bigcup_{k \in \mathbb{N}} NTIME(n^k)$$

2.2 Verifying NP problems

If a language is in NP, it must be decidable by a nondeterministic Turing machine in polynomial time. So, every single computation path must individually take no longer than polynomial time. Since a deterministic Turing machine models a single computation path, this means that a deterministic machine can run every path individually in polynomial time.

What are the implications of this? Well, this means that if you've solved an NP problem and give me a *certificate* of your answer, I can verify you are correct in polynomial time on a deterministic Turing machine. And that's important because we can't actually build nondeterministic machines.

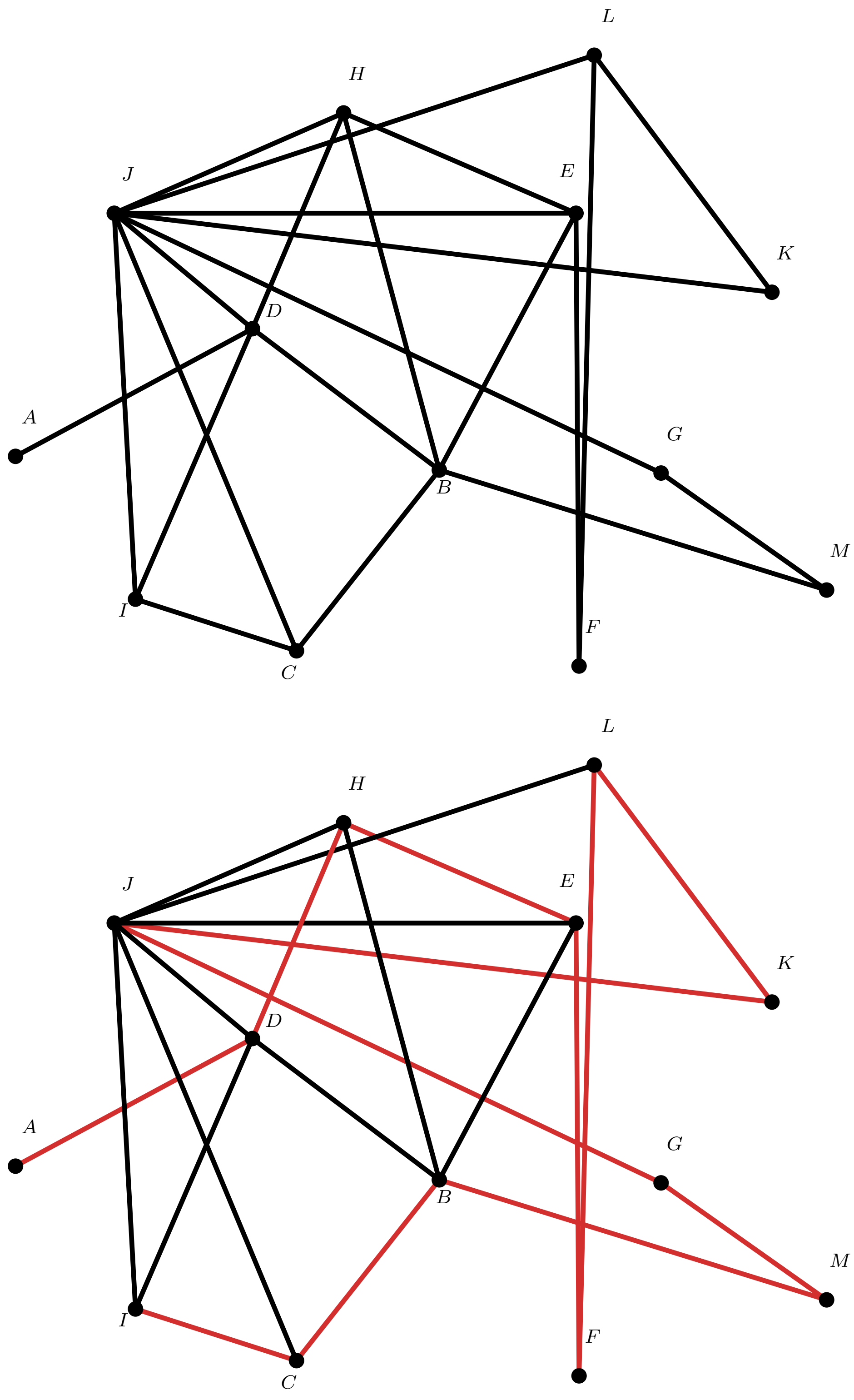


Figure 5: The alien's graph and certificate.

For example, say you are an alien and have landed here in Palo Alto from a far-off galaxy. On your home planet of Kepler-22b, you have built a real-life nondeterministic

Turing machine, and you claim the above graph has a Hamiltonian path. Me, a lowly Earthling, has no quick way to verify you are correct quickly. At best, I can use the fastest Earth algorithm on my computer, which is not in P and does not take polynomial time. However, if you say "Why don't you check the path $A \to D \to H \to E \to F \to L \to K \to J \to G \to M \to B \to C \to I$?", well then any human or ape or bumblebee can verify that this is indeed a Hamiltonian path. And, we can do so in polynomial time on our regular old deterministic Turing machines. Now I quickly know you are correct—the graph above really does contain a Hamiltonian path. And so, we say NP problems are *verifiable* in P.

By telling me a Hamiltonian path, you provided me with evidence that you were correct. This *certificate* is all I need to verify you are correct. I only need to compute that one path to verify you are correct, but you had to simultaneously try them all to find that one golden ticket.

And so, NP problems are not just defined as those which a nondeterministic Turing machine can solve in polynomial time. They are also defined as precisely those problems which we can verify using a deterministic Turing machine in polynomial time.

NP problems seem "harder" to solve than they are to verify. We can verify an NP problem in P. Can we solve every NP problem in P? Or are there some NP problems—perhaps even just one—that no matter what we do, we can't solve in polynomial time on a deterministic Turing machine?

2.3 P vs. NP

How might we approach the Hamiltonian path problem, or the clique problem, or another difficult NP problem using a deterministic Turing machine? We could check each of the $\binom{n}{k}$ cliques sequentially, but that's not efficient. Suppose we want to solve the problem efficiently, in polynomial time.

There's no obvious way to solve these problems in polynomial time on a deterministic Turing machine.

Clearly, every language decidable in polynomial time by a deterministic Turing machine ($L \in P$) is decidable in polynomial time by a nondeterministic Turing machine. After all, every deterministic Turing machine is technically a nondeterministic machine, just a degenerate one. So, if $L \in P$, then $L \in NP$. The converse, as I'm sure you're well aware, is one of the most important unsolved problems in computer science.

Conjecture 2.1 ($P \overset{?}{=} NP$)**.** Is every language decidable by a nondeterministic Turing machine in polynomial time also decidable by a deterministic Turing machine in polynomial time?

$$\bigcup_{k \in \mathbb{N}} TIME(n^k) \overset{?}{=} \bigcup_{k \in \mathbb{N}} NTIME(n^k)$$

Why is this important? We've got some really important algorithms riding on the belief that $P \neq NP$. It's an NP problem to come up with the prime factors given a large number. (Think about it: Given a *certificate* of prime factors, we can in polynomial time multiply them to come up with the original large number.) But our best algorithms to factorize large numbers do not run in polynomial time—and the entire world's cryptography relies heavily on the fact that no efficient prime factorization algorithms exist.

Every scientific field—biology, chemistry, physics, mathematics—has major problems that need to be solved efficiently, but they lie in NP and we don't have polynomial time algorithms for them. If $P = NP$, these problems are in P, so there must exist some fast algorithm to solve them. Sure, some cryptography would be broken, but every field would see major breakthroughs and scientific advancements.

However, most computer scientists strongly believe $P \neq NP$. P problems correspond to problems we can solve efficiently on our real-world computers. After all, our real-world computers are deterministic. Since we can't actually build nondeterministic machines, it's not clear that we should be able to solve all NP problems on our regular deterministic machines.

To prove $P \neq NP$, all one needs to do is show there's a single NP problem that can't be solved in polynomial time with a deterministic machine, and there are some really strong candidates, including $HAMPATH$ and $CLIQUE$.

3 NP-complete Problems

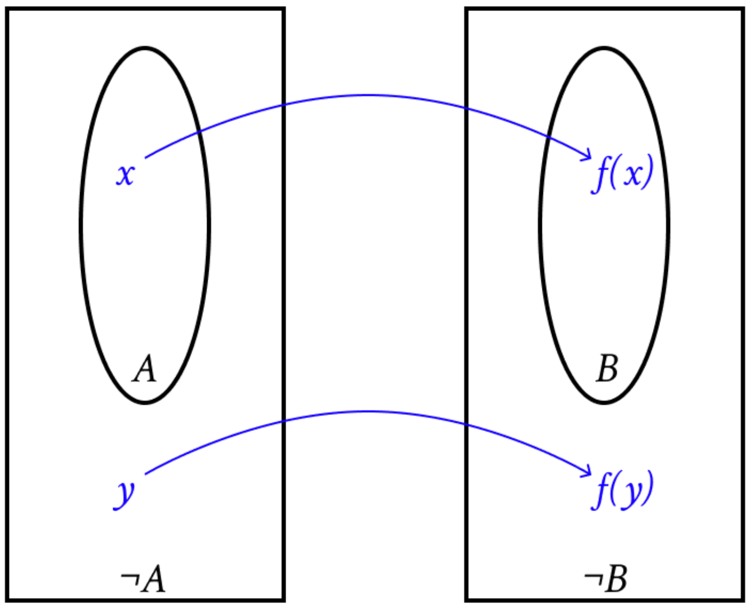
All right, you're probably thinking. We've reached the end of the road. No one has proven $P = NP$ or $P \neq NP$, and neither you nor I are going to—so what more is left to cover?

Well, let's suppose there is a problem in NP, and if we solve that problem, then we can solve every other NP problem with only polynomially more time on a deterministic TM.

"Wow," you say. "If solving that problem gives you all the rest for free, that problem must be really hard!"

"Not just hard, my friend. **NP-hard**."

Definition 3.1. A language B is NP-hard if for all languages $A \in NP$, $A \leq_p B$.

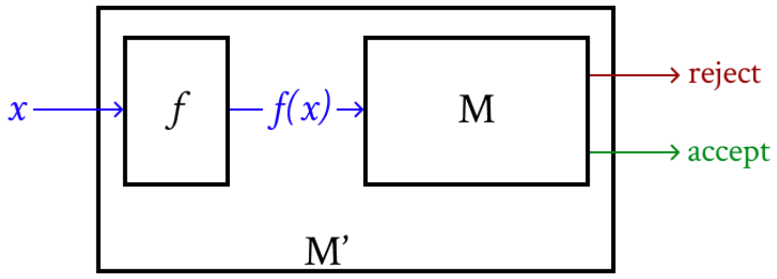


$\leq_p$ is a **polynomial-time mapping reduction**. In other words, B is NP-hard if for all $A \in NP$, there exists a computable function $f \in P$ such that $x \in A \iff f(x) \in B$. This is exactly the same as a standard reduction, but with the added constraint that f must be computable in polynomial time on a deterministic TM.

Note: From now on, we're going to write P and "polynomial time" interchangeably, implicitly meaning deterministic.

Once we've solved B, we can calculate any NP problem for "free" (in only polynomially more time) by reducing $A \in NP$ to B. This should also look very similar to our reductions from earlier.

Suppose we have a Turing machine M that decides B. Then, we can decide a language $A \in NP$ using the following Turing machine M'. For any input x, calculate $f(x)$ in polynomial time. Then, decide if $f(x) \in B$ by running $M(f(x))$. Finally, M' outputs the result of M.



If $x \in A$, then $f(x) \in B$, so M accepts $f(x)$, and thus M' accepts x.

If $x \notin A$, then $f(x) \notin B$, so M rejects $f(x)$, and thus M' rejects x.

Thus, M' correctly decides A.

In either case, the only additional work is computing f. Computing f takes polynomial time, by definition. So, M' takes polynomially more time to decide A than M takes to decide B.

We have the following theorems and corollary, which should be obvious from our Turing machine work above.

Theorem 1. *If $A \leq_p B$ and $B \in P$, $A \in P$.*

Corollary 1. *If $A \leq_p B$ and $A \notin P$, $B \notin P$.*

Theorem 2. *If $A \leq_p B$ and $B \in NP$, $A \in NP$.*

Even more interesting than NP-hard problems are NP-complete problems.

Definition 3.2. A language B is NP-complete if

- B is NP-hard.
- $B \in NP$.

This additional requirement that $B \in NP$ means that NP-complete problems are the "hardest" problems in NP. We can verify these problems in polynomial time, and yet, we don't know how to solve them efficiently. If we could solve even a single NP-complete problem in polynomial time, then we could solve every NP problem in polynomial time, proving $P = NP$.

The natural question becomes: *How do we prove a language is NP-complete?* It seems difficult—we have to somehow encode every single NP problem inside another one to show there is a polynomial-time mapping between them. We will cover this next week, in our final lecture on complexity theory.

4 Resources

Next week, we'll wrap up our lectures on Turing machines by proving SAT, the boolean satisfiability problem, is NP-complete. The proof that SAT is NP-complete is called the Cook-Levin theorem, and it is quite long and difficult. To prove other problems are NP-complete, like vertex cover, we will find reductions to SAT to show NP-completeness. In the meantime, you can check out these resources.

- Stanford's CS 154 class is a great resource for learning more about Turing machines and theory.
  `https://cs154.stanford.edu`
- "Introduction to the Theory of Computation" by Michael Sipser is a great textbook if you're looking to learn more. You can find the problems online.
  `http://www-math.mit.edu/~sipser/book.html`