CS 4510 Automata and Complexity	4/13/2022
Lecture 19	
Lecturer: Frederic Faulkner	*Scribe(s):*

1 And Now For Something Completely Different

Up to this point in the course, we have been discussing primarily *computability*, answering questions of the form "What can be done?" for each model. However, we have, in some sense, pushed computability to its limits. We have identified the most powerful model (Turing machines and equivalent) and found that some problems were too difficult even for these.

Now, however, we turn to *complexity*. We are no longer satisfied with knowing what tasks can be done; we care about how efficiently we can do them. There are two types of resources whose consumption we measure when considering an algorithm through the lens of complexity: time and space. We begin today by considering time.

1.1 What is time?

How do we measure the length of time needed for an algorithm? For some models (say, boolean circuits), this is not a straightforward question, but for Turing machines we have a very convenient measure of time, which is the number of steps taken while running the machine. Since one step of a Turing machine always consists of the same amount of work - one read, one write, and one head movement - a machine that uses more steps would take more time to simulate than one that uses less.

Definition 1 *An $f(n)$-time Turing machine is one that takes at most $f(n)$ steps before halting on inputs of size n.*

So we see that we are considering here *worst-case* time complexity. Other measures, such as *average-case* time complexity are fascinating but will not be discussed in this course.

Definition 2 *The class of languages $TIME(f(n))$ consists of all the languages that can be decided by some $O(f(n))$-time Turing machine.*

Note the use of big-O in this definition, so that $TIME(n^2)$ includes languages decided by machines with running times of n^2, $27n^2$, and $4n^2 + 11n \cdot ln(n) + n + 3$.

Recall the following decider for $w\#w$ from lecture 11, that sweeps back and forth marking characters:

1. Start by scanning the string from left to right to confirm that there is exactly one #.
2. Mark the first character of the string. (I.e after the previous step you must move the head back all the way to the left.)

3. Move to the right past the # until you come to an unmarked character. If this character is different than the one you marked in the previous step, reject. Else, mark this character, and move to the left past the #.

4. Move to the left until you get to the leftmost unmarked character. (I.e. move left until you see a marked character then move one step right.) If there were no remaining unmarked characters, go to step 5. Otherwise, return to step 2.

5. Scan to the right to check that there are no unmarked characters. If there are, reject; otherwise accept.

How many steps does this algorithm take on an input of length n?

- Well, checking the input in part 1 takes about n steps (or perhaps $2n$ if you want to move the head back to the beginning). So part 1 takes $O(n)$ steps.
- Parts 2,3,4 form a loop. How many times does the loop iterate? Well, at each step we mark two of the characters, so there will be about $\frac{n}{2}$ loop iterations
- Inside the loop, we move from the left hand side of the string to the right hand side and back. This takes about $O(n)$ steps. So the total amount of time taken by the loop is $\frac{n}{2}$ iterations $\times$ $O(n)$ inner steps $= O(n^2)$ time.
- In part 5, we scan the string one final time looking for unmarked characters, which again takes $O(n)$ work.

So the final amount of work done is $O(n) + O(n^2) + O(n) = O(n^2)$ work, so we have that $w\#w \in TIME(n^2)$. Can we do better? We cannot....if we limit ourselves to one tape. Every algorithm for ww on a single tape TM requires at least $O(n^2)$ steps.

However, the following is an algorithm for a 2-tape machine (one input tape and a second work tape) that recognizes $w\#w$:

1. Until you reach the #, read the current character, write it to the work tape, and move both heads to the right.

2. Once you pass the #, move the work tape head all the way back to the beginning.

3. Now move both heads to the left one at a time, checking that each head sees the same character at each step.

The top algorithm runs in $O(n)$ steps. So what $TIME$ class does $w\#w$ belong to, $TIME(n^2)$ or $TIME(n)$?

This is one of the differences between computability and complexity. In computability, we showed that many different models were equivalent in power; for undecidability reductions it did not matter whether you used an ordinary TM, a non-deterministic TM, a multi-tape TM, etc.

In complexity, however, as we have just illustrated above, the choice of model matters! So when discussing $TIME(f(n))$ you must specify which model you are talking about.

2 Model Independence Once Again

Definition 3 *The class P consists of all languages with deciders that run in polynomial time with respect to the input length. Formally, $P = \cup_i TIME(n^i)$.*

P is the set of all languages that can be decided in polynomial time, so it includes languages with $O(n)$ deciders as well as $O(n^{250})$ deciders.

In the same way that big O ignores constant factors, talking about P allows us to ignore, in some sense, polynomial factors. If you have a language in P, and you square its running time, you still have a language in P.

Why would we want to do that? After all, in practical applications there can be a huge difference between a running time of $O(n^2)$ and $O(n^3)$. However, the same can be said of big O; depending on the application, constant factors can play a big difference, and $1000n^2$ might be worse then $10n^3$.

And yet there is still a very meaningful sense in which $O(n^2)$ is "nicer" than $O(n^3)$. Ignoring constant factors allows us to generalize our discussion away from specific instances; ignoring polynomial factors is another instance of this generalization.

P is often treated as the set of "reasonable" problems. If a problem can be shown to be in P then we consider it to be "feasible". P also has a very nice property: it is model-independent. For every reasonable model (what does "reasonable" mean? Hmm...), if a language has a polynomial-time algorithm for one model, it has a polynomial-time algorithm for all of them. Thus we can use ordinary Turing machines, substring machines, k-tape machines, etc. when discussing whether a language is in P or not.

2.1 Unreasonable models

However, there is one model of computation which does not seem to produce the same set of languages when allocated a polynomial amount of time. That model is the non-deterministic Turing machine. The ability of an NTM to "guess" what it should do at each step is so powerful that it seems to give NTM's an unfair advantage.

Definition 4 *NP is the set of languages that are decided by some polynomial-time NTM, where we say that a NTM is polynomial-time if it halts after polynomially many steps for each branch of computation.*

For example, let $CLIQUE = \{\langle G\rangle, k \mid G \text{ contains a clique of size } k\}$. (A clique is a set of vertices such that each vertex is connected to each other vertex.) A NTM for $CLIQUE$ can first just guess some size k subset of G's vertices and then check that they form a clique. If there is at least one k-clique, we will always guess one; if there are not any, it doesn't matter what we guess.

However, although people have tried very hard for quite some time now, we have not found any deterministic TM that decides $CLIQUE$ in polynomial time. This is the single biggest open question in theoretical computer science: "Is $P = NP$?" Solving this question carries a prize of one million dollars. If the answer is "yes", then it would revolutionize several fields of computer science, including internet security. But this question remains, at the moment, too difficult for us to solve.

There is another definition of NP. We must first introduce the notion of a *verifier*.

Definition 5 *A verifier V for a language L takes in two inputs x and c, and returns True if and only if $x \in L$.*

For a normal Turing machine, if you want to know whether x is accepted, you only give the machine x as input. However, with a verifier, you are allowed to some additional information to try and convince the verifier to accept the string. This second input is called the *certificate*, and can be thought of as a proof that $x \in L$. If you have set up your verifier correctly, every string in L has at least one proof of its membership, while all of the strings not in L have no such proof. This leads to the second definition:

Definition 6 *NP is the class of languages which have associated polynomial-time, deterministic verifiers.*

Note that if L has a verifier V, then we can write L as $\{x \mid$ there is some "proof" c such that $V(x, c)$ accepts$\}$.

This definition allows for a nice characterization of NP. If P is the set of problems that can be efficiently solved, then NP is the set of problems that, while difficult to solve, are at least easy to *verify* the solution once it is found. (In other words, finding some certificate c might be difficult, but once we have it, running the verifier just takes polynomial time.)

$P = NP$ is thus the question of whether finding a solution for a problem is as easy as checking that a solution is valid. This would be a very striking fact if it were true! In some sense, mathematicians would be out of a job, because finding mathematical proofs would be as easy as reading a proof to see if it was logical. And homework would be as easy for students to find solutions for as it is for instructors to grade those solutions. Most theoretical computer scientists believe $P \neq NP$.

2.1.1 What can we say about P and NP?

Although we do not know whether $P = NP$, there is a definite, known relationship between the two sets that is almost trivially true:

Theorem 7 $P \subseteq NP$

Proof. If L is in P, then there is some deterministic polytime decider for L. But that decider is *also* a nondeterministic machine (it just never uses its power to guess anything), so $L \in NP$.

2.2 Verifiers in action

The verifier definition of NP can seem odd if you have never seen it before, so let's see some examples.

What is a verifier for $CLIQUE$? In other words, what is a short proof that G has a clique? Well, you could just write down the vertices that form the clique.

Similarly, consider $HAMCYCLE = \{\langle G \rangle \mid G$ has a Hamiltonian cycle$\}$. (A Hamiltonian cycle is a cycle that includes every vertex in the graph exactly once.) Checking whether a graph has such a cycle, is difficult; after all there are $n!$ ways to order the vertices of a

graph with n vertices. If we are clever we can do better, but not by much: the best known algorithm runs in time $O(1.657^n)$ which is clearly not polynomial.

However, there is a very short proof that a graph G has a Hamiltonian cycle: just list the vertices in the order that they appear in the cycle!

2.3 Both definitions of NP are equivalent

Technically, Definition 6 introduced a new characterization of NP, but we didn't prove that this version of NP is the same as the previous one. To do so, we must prove the following theorem:

Theorem 8 *The set of languages with polynomial-time, deterministic verifiers is exactly the set of languages with polynomial-time non-deterministic deciders.*

Proof. This proof has two directions. Let L be a language in NP. We do the easier side first. Suppose V is a verifier for L. We wish to construct a NTM N for L.

When should N accept x? Well, N should accept x if there is at least one proof c for x. But how can it find this proof? Well, N is non-deterministic, so it can just guess a proof c, write c on the tape, and then simulate V on x, c. If there is such a proof, N will be able to guess it and x will be accepted; if there is not, every guess made by N will fail and x will not be accepted. So N is an NTM for L, completing this direction of the proof.

On the other hand, suppose N is a non-deterministic decider for L, and we wish to build a deterministic verifier V. When should V accept x? Well, whenever N does. But V cannot simulate N, because V is deterministic but N is not. However, V is allowed to take in extra information besides x. How could someone prove that $x \in L$ - that is, how could someone prove that N accepts x?

Well, one proof c could be the sequence of choices that N makes on x. If we knew this, then we could simulate N, because we would no longer be non-deterministic; each time we got to a "choice", we would read c to see what N does, and deterministically choose that option as well. Then, only x which are accepted by N have valid certificates, so V is a verifier for L. This completes the proof.

3 You Complete Me

In the same way that we can use reductions to show that two languages are each undecidable (or each decidable), we can also use reductions to relate the time-complexity of one language to another.

Definition 9 *A language A is polynomial time mapping reducible to language B, written $A \leq_P B$, if there is some polynomial time computable function f such that $x \in A \rightarrow f(x) \in B$ and $x \notin A \rightarrow f(x) \notin B$.*

The definition of "polynomial time computable function" is exactly what you would expect, namely, that the Turing machine that computes f does so in polynomial time.

Theorem 10 *If $A \leq_P B$, and $B \in P$, then $A \in P$.*

Proof. Let R be a polynomial time decider for B (which exists if $B \in P$), and let f be the polynomial time reduction from A to B. Then we create a polynomial time decider S for A as follows:

> S on input x:
> 1. Create input y for R
> 2. r = $R(y)$
> 3. return r
>
> where
>
> $y = f(x)$

If you go back and check, you will see that this proof is almost identical to the proof that a mapping reduction from A to a decidable language implies that A is decidable. However, before we just needed to argue that S was decider for A, which is obvious from the definitions of f and R. But now we also must check that S is not just a decider, but a polynomial time decider.

Step 1 of S takes polynomial time because f is polynomial time computable. Step 3 clearly takes polynomial time.

Step 2 takes polynomial time, although there is a subtle argument to be made here. We mean polynomial in x, the input to S. We know that R runs in polynomial time *in the length of the input to* R, which is not x. But y *cannot* be longer than polynomial in x, so R runs in time polynomial of (polynomial of x)), which is polynomial of x. (At this point, I suspect the word "polynomial" has been repeated enough times to have lost all meaning...)

3.1 The hardest problems in NP

Definition 11 *A language L is NP-hard if, for any language $L' \in NP$, $L' \leq_P L$.*

Definition 12 *A language L is "NP-complete" if*

- $L \in NP$
- *L is NP-hard.*

NP-complete languages are thus, in some sense, the hardest problems in NP. If we can find an efficient way to solve even *one* NP-complete problem, then we will automatically find efficient ways to solve *every other problem* in NP.

NP-complete problems thus provide one avenue for trying to decide whether $P = NP$. Rather than discussing the very large class NP, which contains many different languages, we can just pick our single favorite NP-complete language and try to prove that it is in P.

After all, if a single NP-complete L language is in P, then there is an efficient (polynomial-time) decider for L. But then, since each language in NP reduces to L, we can use the construction in the proof from the last section to construct polynomial-time deciders for each other language in NP.

Theorem 13 *If $A \leq_P B$, and A is NP-hard, then B is NP-hard.*

Proof. Assume that A is NP hard, and reduces to B in polynomial time. We wish to show that B is NP-hard. In other words, given any random $L \in NP$, we wish to reduce L to B.

How can we do this? Well, we know that L reduces to A, since A is NP-hard. So there is some function f such that $x \in L \iff f(x) \in A$. But since A polytime-reduces to B, there is some function g such that $y \in A \iff g(y) \in B$. Then the composite function $g \circ f$ satisfies that $x \in L \iff g(f(x)) \in B$, completing the reduction from L to B.