CS 4510 Automata and Complexity 11/06/2023

Lecture 19: Complexity Classes

Lecturer: Abrahim Ladha *Scribe(s): Akshay Kulkarni*

1 Introduction

1.1 Motivation

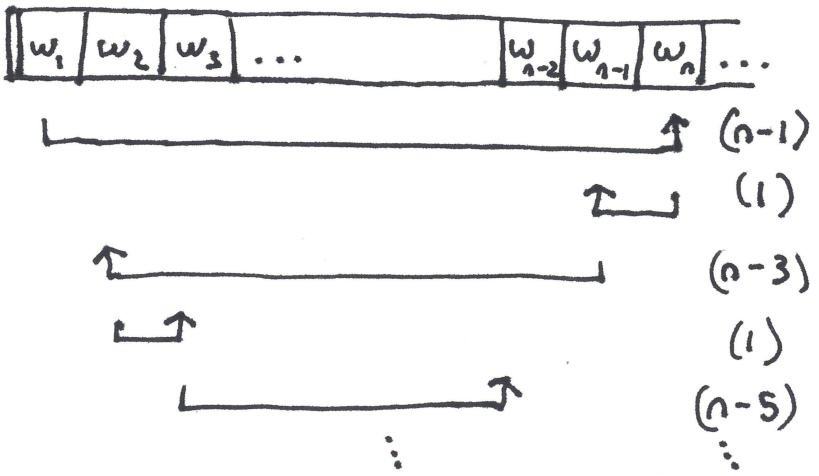
We begin our final unit entirely on computational complexity. This lecture will simply consist of some early and motivating theorems, mostly before the development of NP-completeness.

First we need a good computational model of a "hard" or "easy" computation. There is a reason we have done everything so far with Turing machines. Turing machines make an excellent model for complexity. Recall that a Turing machine performs a constant amount of work in unit time. If more work is to be done, successive steps must be taken. This is exactly what makes it an excellent model, because this is exactly how algorithms work in reality. There did exist historically some functional but Turing-complete models of computation, and they do not have this property. For example, there exists a lambda calculi for string copying. It can produce xx from x in a single step.[1] This isn't as good of a model, as to copy an arbitrarily long string should take some number of steps as a function of the length of the string. Intuitively, Longer strings should take longer to copy. It is not clear here what the "step" is in a functional model, but it certainly clear what a step is for a Turing machine.

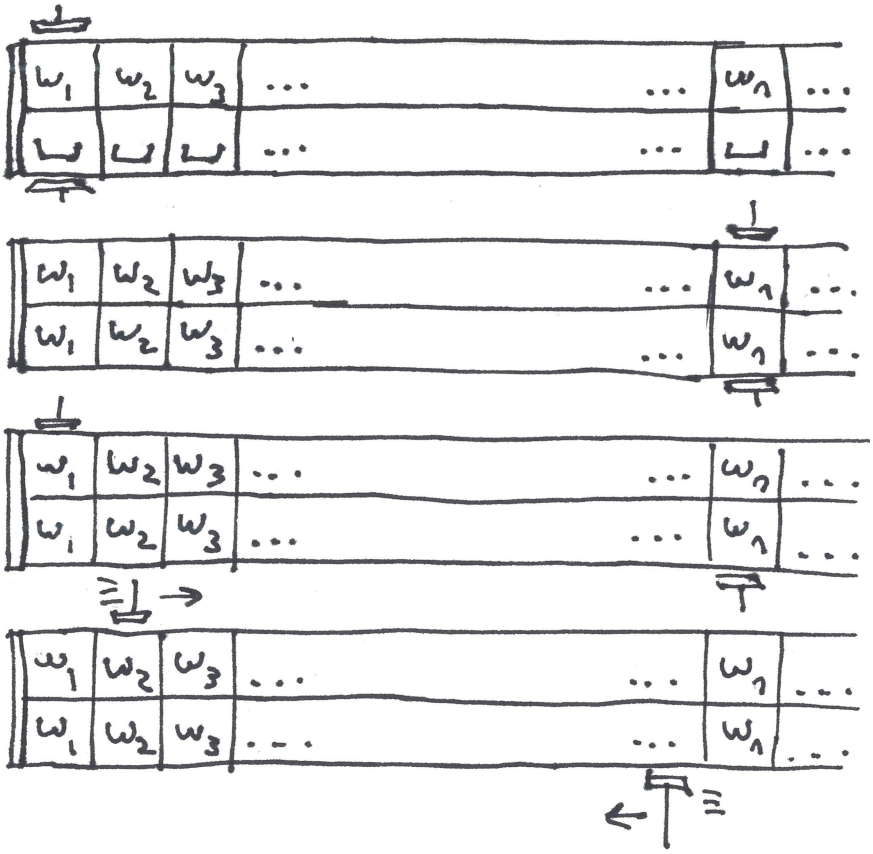
1.2 Turing Machine Variants

Does the variant and choice of Turing machine matter?. For now, we set aside the unrealizable nondeterministic Turing machine (NTM) and only consider reasonable and realizable models. Consider the language of palindromes $PAL = \{ww^R \mid w \in \Sigma^*\}$. Here is an algorithm to decide PAL on a single-tape deterministic Turing machine (DTM). Check the first symbol, then the nth symbol, then the second symbol, and so on. The limitation of this machine is that it is not random access. To read the last symbol starting from the first takes a linear number of steps because the tape head has to loop over the entire input. To decide PAL this way on a one tape DTM, this takes $n + (1) + (n-2) + 1 + \cdots = O(n^2)$.

[1] It might look like $\lambda x[xx]$.



We can give a better algorithm on a two-tape DTM as follows. Copy the input to a second tape, reset one tape head. Loop both heads in opposite directions on the tape, comparing symbols. These three routines each take linear time giving a $O(n)$ time algorithm on our two tape DTM.



Obviously, any stronger model must also take at least linear time, as to decide if $ww^R \in$

PAL must look at all the symbols for correctness[2]. Can a single-tape DTM decide PAL in $O(n)$ or even $o(n^2)$ time? Surprisingly, no. any single tape deterministic Turing machine to decide PAL must take $\Omega(n^2)$ (and so $\Theta(n^2)$) steps. This is surprising! It means on a one tape DTM, there is no way to do better for this language than the obvious way. There are two proofs, a more classic combinatorial one, and one which uses Kolmogorov complexity. Essentially, if a machine could decide PAL in $o(n^2)$ time, you could use this machine to compress an incompressible string. I recommend this proof as your final project.

2 P

Let $\text{TIME}(f(n)) \coloneqq$ be the class of languages decidable by a Turing machine in $f(n)$ steps. Similarly define $\text{NTIME}(f(n))$, $\text{SPACE}((f(n)), \text{NSPACE}(f(n))$. Let

$$\mathsf{P} = \bigcup_{k=0}^{\infty} \text{TIME}(n^k)$$

Why is P a good definition of the class of intuitively efficient algorithms? We give some arguments in favor

1. Most problems seem to have naive, trivial, brute-force solutions putting the problems at least in EXP. If there exists a polynomial time algorithm, then either the problem is trivial, ridiculous, or we have some deeper intuition about what the problem actually is (i.e. mathematical theory). An exponential time algorithm may be simple, but a polynomial time algorithm usually requires a non-trivial understanding of the structure of the problem itself. For example consider graph traversal. A bad way would be to enumerate all paths and check them this way. A better way is to notice how any sub-path of a path is itself a path. The shortest path from s to t cannot be longer than the shortest path from s to t through some v. This recurrent structure is what leads to efficient algorithms like DFS/BFS/Dijkstra's and so on.

2. Polynomials are closed under operations which our intuition of "efficient" is also closed under. If you have two algorithms A, B. If one or both is inefficient, then the composition, running both of them sequentially, should intuitively be inefficient. If both are efficient, then running both sequentially should be efficient. If $f(x), g(x)$ are polynomials, then $f(x) + g(x), f(x)g(x), f(g(x))$ are also polynomials. A combination of efficient algorithms should be efficient, and a combination of efficient and inefficient algorithms should be inefficient. Polynomials preserve these closure properties.

3. Although there exist languages with $O(n^{100})$ algorithms which require $\Omega(n^{99})$ steps, we don't have any practical examples of this. The highest polynomial run time you

[2]Most algorithms should take linear time. If an algorithm takes sublinear time, it doesn't even have time to look at the entire input, so it could only compute some toy language like a regular one. For example checking if the first symbol is a one takes constant time. Binary search is sublinear, but this is only on the random access model, not on a Turing machine. Still, binary search is efficient because it doesn't have to look at the entire input.

would see in an algorithms course might be cubic time $O(n^3)$[3]. The highest polynomial run time I know is for the LLL algorithm. In $O(n^8)$ time, it finds a short orthonormal basis of a lattice. Its poly-time, but practically infeasible. It would appear to stall on reasonably small inputs. However, the achievement of the authors was that they were able to bring the problem from far beyond P into P. Once you get a polytime algorithm at all, it seems likely in practice that it can be improved. If you can be afforded the mathematical theory to bring a problem into P at all, this knowledge can likely be grown. There were several papers on "pruning" which make the algorithm more efficient in a way thats hard to asymptotically measure. The engineers will get a hold of the problem and make it practicality efficient, even optimizing the constants. Although the vanilla algorithm may not appear to complete in reasonable time on most inputs, the version of the algorithm which is included in most libraries, it appears to halt instantly on all inputs you could test. This is one example, but the case is evident for many other algorithms. If there is enough intuition to give a polynomial time algorithm at all, its likely this intuition can be extended and the problem can be made easier and easier. The languages which are solvable in $\Omega(n^{99})$ are nonconstructive, useless. They aren't real practical problems, and are designed via diagonalization to have this property, and do nothing else.

4. All Turing machine variants appear to simulate each other with at most polynomial overhead. What a word-RAM machine does is T steps takes a one-tape DTM T^4 steps. The word-RAM model is our best computer, and the one-tape DTM might be our worst, yet the overhead is still only a polynomial. Although within P, they may take different time for different languages, A definition of P is equivalent for all these models. The extended Church-Turing Thesis says that not only are all these variants as powerful as each other. The run-time of any one model to simulate another will not have super polynomial overhead.

3 NP

Let

$$\mathsf{NP} = \bigcup_{k=0}^{\infty} \mathrm{NTIME}(n^k)$$

The definition given in your algorithms course is that NP is the class of languages verifiable in polynomial time. We now prove these definitions are equivalent. Let NP be the class of languages decidable by a non-deterministic Turing machine which halts in a polynomial number of steps. We say a language $A \in \mathsf{NP}_v$ if there exists a deterministic polynomial time verifier V for A. The verifier will take as input a word w and a witness or certificate c and $V(w, c)$ will accept or reject accordingly if $w \in A$.

Let $A \in \mathsf{NP}_v$, then there exists a polynomial time verifier V, which runs in $O(n^k)$ time for some k. We will build a NTM to decide A as follows.

[3]Like chain matrix multiplication. Maybe Bellman-Ford on a dense graph. Most things appeared to be linear or quadratic time.

Algorithm 1 N on input w

Nondeterministically guess certificate c of max length n^k
run $V(w, c)$
accept $\iff$ V accepts

Clearly, N runs in polynomial time. Since this is for all languages verifiable in polynomial time, we see $\mathsf{NP}_v \subseteq \mathsf{NP}$.

Let $A \in \mathsf{NP}$, then there exists a polynomial time NTM to decide A. We show A is a polynomial-time verifiable. Our witness c is just our nondeterministic choices. So, V is a

Algorithm 2 V on input $\langle w, c \rangle$

Simulate N deterministically on w
if faced with a nondeterministic choice **then**
    Get the next bit of c
end if
if current branch of N's computation accepts **then**
    accept
end if

deterministic polynonial-time verifier correctly for A, so $\mathsf{NP} \subseteq \mathsf{NP}_v$. Since we proved the containment both ways, we see that $\mathsf{NP} = \mathsf{NP}_v$, and may drop the subscript. From now on when we talk about NP, we can use either definition based on convenience.

4 $\mathsf{P} \subseteq \mathsf{NP}$

We prove $\mathsf{P} \subseteq \mathsf{NP}$ in both ways, using both definitions of NP.

1. First, note that by the generalization of nondeterminism, every deterministic polynomial-time Turing machine is also a nondeterministic Turing machine, so $\mathsf{P} \subseteq \mathsf{NP}$
2. If $A \in \mathsf{P}$, then there exists a polynomial time algorithm to decide A. We prove A is also verifiable in polynomial time. The verifier will simply ignore the witness and simulate the polytime decider for A. This implies $\mathsf{P} \subseteq \mathsf{NP}$.

5 More Classes

$$\mathsf{PSPACE} = \bigcup_{k=0}^{\infty} \mathrm{SPACE}(n^k)$$

$$\mathsf{L} = \mathrm{SPACE}(\log(n))$$

$$\mathsf{EXP} = \bigcup_{k=0}^{\infty} \mathrm{TIME}(2^{n^k})$$

We give some rough ideas about why the following chain holds.

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{NPSPACE} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP} \subseteq \mathsf{EXPSPACE} \subseteq \mathsf{NEXPSPACE}$$

- $\mathsf{L} \subseteq \mathsf{NL}$ follows from the generalization of non-determinism.
- $\mathsf{NL} \subseteq \mathsf{P}$ follows from the fact there exists a polytime algorithm for an NL-complete problem.
- $\mathsf{P} \subseteq \mathsf{NP}$ as proved previously.
- $\mathsf{NP} \subseteq \mathsf{PSPACE}$, since $\textsc{sat} \in \mathrm{SPACE}(n)$. Recall that $\forall L \in \mathsf{NP}$, $L \leq_p \mathrm{SAT}$.
- Later we will prove that $\mathsf{PSPACE} = \mathsf{NPSPACE}$, but this containment as shown follows obviously

Note that PSPACE contains languages we think are decidable only in exponential time, so we won't discuss too much on the right half of this chain.

6 More Questions than Answers

We have absolutely no idea how to solve $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$. We do understand to some level how hard the problem[4] actually is. The problem itself is connected via a massive web of implications to many more problems.

Consider the following two subchains.

1. $$\mathsf{L} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}$$

   We can prove $\mathsf{L} \subsetneq \mathsf{PSPACE}$. Since these are the left and right sides, one of the containments in this chain must be strict. Note that if you could prove $\mathsf{P} = \mathsf{PSPACE}$, this would imply that $\mathsf{P} = \mathsf{NP}$, so if $\mathsf{P} = \mathsf{PSPACE}$ is an open problem. If you could prove that $\mathsf{L} = \mathsf{P}$ and $\mathsf{NP} = \mathsf{PSPACE}$, then it must be the case that $\mathsf{P} \neq \mathsf{NP}$. These are open problems as well.

2. $$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{EXP}$$

   Note that $\mathsf{NP} \subseteq \mathsf{EXP}$ since you can give a deterministic exponential time algorithm to brute force all polynomial sized certificates. We can prove $\mathsf{P} \subsetneq \mathsf{EXP}$, so again there exists a strict containment in this chain. Proving $\mathsf{NP} = \mathsf{EXP}$ would imply $\mathsf{P} \neq \mathsf{NP}$, so $\mathsf{NP} = \mathsf{EXP}$ is also an open problem.

The history of complexity theory is a history of failure. Any problem which could be reasonably asked may accidentally imply something about $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ and thus becomes as hard as the problem itself. The failure to solve this one problem has shifted major directions

[4] Whenever I may refer in passing to "the problem", I could only refer to the one problem, this problem. $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$.

in research over the past fifty years. Every new research direction, every new theorem, every new foundation has been built with the motivation and direction to solve this one problem. A massive effort has been undertaken in order to try and solve the problem with zero success. We have built the shoulders of giants. Ironically, as we have understood the problem better, we are farther away from solving the problem than we were when we began.

Besides these structural connections, there are many more ways which might resolve the question. These include

- Proving existence or nonexistence of a one-way function
- Showing a super polynomial lower bound or a polynomial time algorithm for any individual NP-complete problem. There are thousands.
- Giving a polytime algorithm to convert 3SAT instances into 2SAT ones.
- Proving that random generators are indistinguishable from psuedorandom ones.
- A proof that every property expressible by a second order existential statement is also expressible in first order logic with a least fixed point operator
- There is no polynomially bounded propositional proof system.

And much much much more.