CS 221 Computational Complexity, Lecture 4 Feb 1, 2018

Nondeterminism and Circuit SAT

Instructor: Madhu Sudan *Scribe: Vinh-Kha Le*

1 Topic Overview and Announcements

Today, we will cover space complexity, nondeterminism, NP, NL, and completeness. Homework 1 is due tomorrow on 2 Feb 2018 at 8PM. For those interested in mathematics, Jacob Fox will present on arithmetic patterns and algorithms at the CMSA at 5PM in Askwith Hall, 13 Appian Way.

2 Space Complexity

When we think of computational models, we usually think of functions $f : \{0,1\}^* \to \{0,1\}^*$ that send binary strings to binary strings. Given that $f : \{0,1\}^n \to \{0,1\}^n$ has input and output of length n, how much space does it take to compute the function? This is the question we ask when we refer to the space complexity of an algorithm.

The answer to this question depends on what counts towards the space taken up by an algorithm. If we count the space it takes to hold the input and output, it is impossible for the function to take up less than $O(n)$ bits. This is not good because ideally, we want to differentiate between problems that take up a lot of space because they take on large inputs/outputs and problems that take up space because they are difficult to compute and require the machine to remember a greater number of past calculations.

The fact that there is such a thing as a log-space complexity class suggests that we tend not to count inputs and outputs towards the space complexity of an algorithm. Because we want to exclude inputs and outputs from our space complexity analysis, we should specify a model of computation that distinguishes input and output from the actual workspace of an algorithm. Our model of computation will receive an input on a read-only tape and print its output on a write-only tape. Calculations can be performed on a read-write tape called the workspace, and it is this tape that counts towards space complexity.

Our model allows us to print out bits as we calculate. This allows us to perform large calculations in small space. In this class, the smallest non-constant space complexities we will examine are $\Omega(\log n)$. For the purposes of this class, we want at the very least the ability to construct pointers and counters. Other areas of computer science, however, do indeed explore algorithms that are $o(\log n)$. For more information on HyperLogLog and other sub-logarithmic algorithms, the scribe highly suggests taking Professor Jelani Nelson's course on streaming algorithms for big data.

See Problem 4 in Homework 1 for an example of an algorithm that calculates n^2 bits in space $O(\log n)$. Examine closely the relationship between the workspace tape and the output tape. When you feel comfortable with this example, ponder the following theorem.

Theorem 1. *If some function $f_1 : \{0,1\}^n \to \{0,1\}^n$ is computable in space $s_1(n)$ and $f_2 : \{0,1\}^n \to \{0,1\}^n$ is computable in space $s_2(n)$, $f_1 \circ f_2$ is computable in space $s_1(n) + s_2(n) + \log n$.*

This statement is trivial, but not as trivial as it seems. Intuitively, you should be able to compute f_1 after f_2 in space $s_1(n) + s_2(n)$. Where does the extra $\log n$ come from? The answer lies in the way we computed $s_2(n)$. In particular, we do not count the space it takes to store the output of f_2. If we print the output to the write-only tape, we will not be able read the output as an input for f_1. We could store the output of f_2 in the workspace, but as we saw before, this output can be larger than $s_2(n)$. In fact, it can be as large as n bits. So how do we get an oracle for f_2 with no more than a logarithmic space cost?

The key is to repeat simulations of f_2 as needed to gain access to arbitrary bits in the output of f_2. The algorithm goes like this. Run f_1 until the machine reaches a call that references bit i of the output of

f_2. Then run an f_2 subroutine, but do not print to the output tape. Instead, maintain a counter j of the position where f_2 would have printed had it printed to the output tape. If f_2 writes to the output tape while $i = j$, feed the tentative output into x. When the f_2 subroutine terminates, feed x to f_1 as bit i of the output of f_2.

This algorithm is utterly useless in practice, but it proves a point. As long as you are willing to incur really large time costs, you can design algorithms with quite impressive space complexities.

3 Nondeterminism

Many of our complexity classes involve giving our models of computation a resource called *nondeterminism*. Nondeterminism is a resource in the same sense that approximation, randomness, interaction, and quantum computing are resources. It is an extra liberty or ability granted to the computational model.

Nondeterminism was first introduced in the field of automata theory. Here, we visualize algorithms as machines that move from node to node on a graph. When it reaches a halting state, it prints the output associated with that halting state.

Definition 2. *An algorithm for decision problem makes nondeterministic steps or is nondeterministic if it is allowed to exhibit different behaviors on different runs, as opposed to a deterministic algorithm, that must perform the same behavior on every run.*

We say that a nondeterministic algorithm decides the problem in time $t(n)$ and space $s(n)$ if there exists a sequence of behaviors of size $s(n)$ that leads to acceptance in time $t(n)$. In other words, $s(n)$ is the number of bits needed to enumerate every possible behavior, and $t(n)$ is the time it takes to decide the problem given the optimal behavior. Because nondeterminism only requires the ability to accept, nondeterministic complexity classes are generally not closed under complement.

The historical definition of nondeterminism reveals the underlying motivation for nondeterministic complexity classes, but it is often much easier to think of nondeterminism through more modern interpretations. The modern interpretation we will examine focuses on the existence quantifier in the definition of decision for nondeterministic algorithms. As part of this interpretation, we introduce the notion of pair languages.

Definition 3. *A pair language is a subset of* $\{0,1\}^* \times \{0,1\}^*$. *In other words, pair languages are languages that consist of pairs of strings.*

This allows us to define the complexity class NP.

Definition 4. *A language A is in* NP *if there exists a pair language $B \subseteq \{0,1\}^* \times \{0,1\}^*$ in* P *such that*

$$x \in A \Leftrightarrow \exists y : |y| \leq \text{poly}(|x|) \wedge (x, y) \in B.$$

We sometimes call B a verifier and y a certificate. This is because B simulates A running with behavior determined by y. Conversely, A is the nondeterministic algorithm whose possible paths are determined by the string y. For example, consider a polynomial-time verifier that takes an input of a graph and a 3-coloring and decides whether or not it is a valid 3-coloring:

$$\text{3-Col-Verify} = \{(G = (V, E), \chi : V \to \{0, 1, 2\}) \mid \forall (U, V) \in E : \chi(U) \neq \chi(V)\}.$$

The corresponding NP problem is to decide whether this 3-coloring exists:

$$\text{3-Col} = \{G \mid \exists \chi : (G, \chi) \in \text{3-Col-Verify}\}.$$

Later in the course, we will examine what other quantifiers besides the existential quantifier can apply to the second term in the paired language, e.g., the universal quantifier. In this lecture, we also examined and discussed NL and NEXP. **This information will be added to the final version of the notes.**

Exercise 5. *Use a padding argument to show that* P = NP *implies* EXP = NEXP.

4 NP-completeness

P problems are the easiest problems in NP and are closed under polynomial-time Karp reductions from below. Similarly, NP-complete are the hardest problems of NP and are closed under polynomial-time Karp reductions from above.

Definition 6. *A problem A is* NP*-complete if there is a polynomial-time Karp reduction from every problem in* NP *to A.*

NP-complete was first introduced in [Cook '70]. In this paper, Cook proved that SAT—or Circuit SAT as we will call it in class—is NP-complete. Circuit SAT takes as an input a Boolean circuit C and decides if there is an input x such that $C(x) = 1$. The fact that this problem is NP-complete is a well-known theorem named after Stephen Cook and Leonid Levin, who gave this proof independently on either side of the iron curtain. **A more detailed history of this theorem was explained in class and will be included in the final version of the notes.**

Theorem 7 (Cook-Levin)**.** Circuit SAT *is* NP*-complete.*

Proof. Let A be a problem in NP. There exists a polynomial-time verifier circuit B such that

$$x \in A \Leftrightarrow \exists y : |y| \leq \text{poly}(|x|) \wedge (x, y) \in B.$$

We can model B as a polynomial-sized circuit C such that $(x, y) \in B$ is equivalent to $C(x, y) = 1$. We build a circuit D that takes as input x, y, and a description of C such that

$$D(x, y, C) = C(x, y).$$

Let E be D with x and C fixed. In other words,

$$E(y) = D(x, y, C)$$

for some fixed x and C. The problem E is a restatement of Circuit SAT. □

This proof relies on the existence of a circuit D that can simulate $C(x, y)$ for any x, y, and C. We leave the construction of D as an exercise to the reader.

Exercise 8. *Design such a circuit D.*

Exercise 9. *Prove that* P = NP *implies that all problems in* P *are* NP*-complete.*

When Cook introduced NP-complete, he had to prove that all problem in NP reduce to Circuit SAT. Once this reduction has been performed, it is no longer necessary to consider all problems in NP to prove that a problem is NP-complete. Because polynomial-time Karp reductions are transitive, it suffices to prove that the problem reduces from one that is already known to be NP-complete. Cook did not have this luxury. Note that not all reductions are transitive, e.g., exponential-time reductions. **More on reductions and the decision to use polynomial-time Karp reductions to define NP-complete will be included in the final version of the notes.**

Using the foundations established in [Cook '70], Richard Karp proved that 20 other problems are NP-complete in [Karp '72]. These 20 problems with the addition of SAT are considered the canonical NP-complete problems of complexity theory and are often nicely called Karp's 21 NP-complete problems. One of these problems is 3SAT. Karp proved NP-completeness of 3SAT via reduction directly from Circuit SAT. Intuitively, 3SAT reduces to Circuit SAT, since Boolean satisfiability with at most three literals per clause is clearly a special case of Boolean satisfiability. Yet, we also have a reduction in the other direction.

Theorem 10. 3SAT *is* NP*-complete.*

Proof. We start with a circuit C and ask whether there exists an input bundle $x = x_1 x_2 \dots x_n$ such that $C(x) = 1$. Let $y_1 y_2 \dots y_m$ be the output of every gate in the circuit C. Because C is circuit, there should be no cycles. The evaluation of circuit C consists of NAND gate assignments of the form

$$\neg y_4 = x_1 \wedge y_3.$$

We can rewrite each of these assignments as 3CNFs of the form

$$(x_1 \vee y_4) \wedge (y_3 \vee y_4) \wedge (\neg x_1 \vee \neg y_3 \vee \neg y_4)$$

Convice yourself that OR gates and AND gates can be transformed similarly. Without loss of generality, let y_m be the output of C. Conjugate all the 3CNF gate translations with each other and add a $- \wedge y_m$ to ensure that the output is 1. By feeding the full 3CNF into 3SAT, we solve the Circuit SAT problem. □

5 Savitch's Theorem

Because this theorem is closely related to Problem 5 of Homework 1, it will not be included in the first version of the lecture notes.