CS 360 — Theory of Computation	March 22, 2011
Lecture 20	
Lecturer: Jalaj Upadhyay	*Scribe: (Jalaj Upadhyay)*

In this lecture, we will talk about Cook-Levin's theorem. Just to get you all motivated, following are some trivias:

- Cook won the coveted Turing award for this breakthrough result in 1982.
- This theorem is central to celebrated P vs NP problem, which is nominated by Clay Institute as one of seven millenium prize.[1]

We will start this lecture with a quick recap of some of things we have already covered in the last lecture.

1 Recap

We defined the following complexity classes:

- P: the class of language that is decidable by a deterministic Turing machine in polynomial time.
- NP: the class of language that is decidable by a non-deterministic Turing machine in polynomial time.

An alternate definition of NP uses the concept of verification. Informally speaking, NP is the class of languages L that, given an input string w and a witness/certificate c (of length polynomial in the size of x), can verify in polynomial time whether x is in L or not. For formal definition, we have to first define verifiers.

Definition 1.1 *A* verifier *for a language A is an algorithm V, where $L = \{w|V$ accepts $\langle w, c\rangle$ for some string $c\}$. We measure the time of a verifier in terms of $|w|$. c is called* certificate/witness *for w. A* polynomial time verifier *runs in polynomial time in the length of w. A language L is* polynomial time verifiable *if it has a polynomial time verifier.*

Using the above definition, the alternate definition of NP is

Definition 1.2 NP *is the class of languages that have a polynomial time verifiers.*

Therefore, the question of P vs NP is a philosophical question of *solution* vs *verification.*

Definition 1.3 (Reducibility.) *A language $L \in \{0,1\}^*$ is* polynomial time reducible *(also known as* Karp reducible*) to a language $L' \in \{0,1\}^*$, written as $L \leq_p L'$, if there exist a polynomial time computatable function f such that $x \in L \Leftrightarrow f(x) \in L'$.*

[1] If you can solve this, you will get 1 million dollars and you will be famous worldwide.

All of us uses this concept almost unknowingly. Consider, for example, you are going to visit a new city and you want to walk around the city. Moreover, you do not want to get lost in the new city. What options you have? You can either get a map or use your smartphone to get directions. Therefore, you *reduced* the problem of walking around in the city to the problem of finding a map of the city or looking up the map on your smartphone. In this case, the reduction is intuitive. However, when you are trying to build a computer program, you will want the reduction to be efficient. The notion of efficiency is captured by having the function f polynomial time computatable.

We next develop the concept of NP-hard and NP-complete. For that, recall the movie Troy. In the starting scene of Troy, the two kings decided that there is no use of blood shed; they will just nominate a fighter (presumably the best fighter in the territory.) These fighter will fight and the winner wins the battle for his king. Of course, everyone remembers Achillies won the battle. We learn a lesson from this story which we translate in the form consistent with this lecture. For P vs NP problem, we know one direction (P is in NP.) For the other direction, we have no idea. So, an approach can be to find the hardest problem in NP and show that we can solve that problem in polynomial time. We need to formalize this intuitive understanding. This lead to a formal definition of NP-hardness and NP-completeness.

Definition 1.4 *L' is* NP*-hard if for all $L \in \mathsf{NP}$, $L \leq_p L'$.*

A direct corallory is that if L is NP-hard and $L \in \mathsf{P}$, then $\mathsf{P} = \mathsf{NP}$. Just for a little brain work, try formalizing the definition of P-hard.

However, we know that Achillies was not a soldier of the king. This led to implication that works fine in one direction. A much concrete concept, if can be achieved, is a both sided implications (an *if and only if* type statement.) This leads to the definition of NP-completness. As a small exercise, trying arguing that the definition given below does achieve this purpose.

Definition 1.5 *L' is* NP*-complete if it is* NP*-hard and $L' \in \mathsf{NP}$.*

Now, we get two way implication, i.e., L is NP-complete and $L \in \mathsf{P} \Leftrightarrow \mathsf{P} = \mathsf{NP}$.

A natural question that can be raised is whether there is any language that is NP-hard (or NP-complete). In the last lecture, we saw an example of such a language, so existence is not a problem. The problem however is that the language we showed NP-hard is intimately tied to the notion of Turing machine and hence not very useful. What Cook-Levin did was to prove that many naturally occuring languages are NP-complete.

2 Cook-Levin theorem

We will prove that a natural occuring language, SAT, is NP-complete. For that, we next define SAT formally.

Definition 2.1 *A* literal *is a boolean variable or a negated variable. A* clause *is several literals connected by OR gate (denote $\vee$). A boolean formula is in* conjunctive normal form *(CNF) if it comprises several clauses connected by AND gate (denoted by $\wedge$). We define the SAT as set of satisfiable CNF formula.*

Theorem 1 *SAT is* NP*-complete.*

Proving that SAT is in NP is very simple since a satisfying assignment can act as a certificate that a formula is satisfiable; proving that SAT is NP-hard is more tricky, but not very difficult. In this lecture, we focus on this part of the proof.

Proof To prove the theorem, we need to show that we can reduce any language $L \in \mathsf{NP}$ to SAT. In other words, we need to find a polynomial time computable transformation that turns any $w \in \{0,1\}^*$ into a CNF formula φ such that φ is satisfiable iff $w \in L$. The main problem in this proof is that we do not know anything about L except that it is in NP. As it turns out, we do not need anything else.

Let N be the NTM that decides L in n^k time. We can mimic the transition of N on an input string w by a tableau whose rows are the configuration of a branch of computation of N on the string w. Thus, the first row is the start state and the tableau is accepting if any row of the tableau is accepting configuration.

Let T denotes an accepting tableau for N on w that corresponds to a computation of N on w. Therefore, in order to get a reduction, we need to produce a formula φ such that satisfying assignment to φ corresponds to an accepting tableau for N on w. We first define variables that are in φ. Let C denote the union of the set of alphabets and states of N. Then for every $1 \le i, j \le n^k$ and every $c \in C$, we define a variable $x_{i,j,c}$, which takes value 1 if $T(i,j) = c$.

We now define how to construct φ. We require φ to mimic the tableau T, therefore we require φ to satisfy following conditions:

1. Guarantee that $x_{i,j,c} = 1$ iff $T(i,j) = c$.
2. Guarantee that the start configuration is right.
3. Guarantee that the accepting configuration is right.
4. Guarantee that each row of the table entails a legal transition from the previous row.

The first three conditions are easy to follow. It is the last condition that is little tricky. For example, we can set

$$\varphi_{entry} = \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{c \in C} x_{i,j,c} \right) \wedge \left(\bigwedge_{c_1 \neq c_2 \in C} (\overline{x_{i,j,c_1}} \vee \overline{x_{i,j,c_2}}) \right) \right].$$

It is left for exercise for you to verify that this is the right expression for the first condition. (Hint: you can try proving that the first expression captures that at least one variable is set to 1 and the second expression captures the idea that at most one variable is set to 1.)

Simlarly, for the the second part, we can write a boolean expression of the following form:

$$\varphi_{start} = x_{1,1,q_{start}} \wedge x_{1,2,w_1} \wedge x_{1,3,w_2} \cdots \wedge x_{1,n+1,w_n} \wedge x_{1,n+2,\sqcup} \cdots \wedge x_{1,n^k,\sqcup}$$

and for the third condition, we can state as the following boolean expression:

$$\varphi_{accept} = \bigvee_{1 \le i,j \le n^k} x_{i,j,q_{accept}}.$$

The last condition is little tricky and for that we need to define how we construct the table from the transition function of N. There can be more than one way to translate the transition function in a tabular form, provided that it is consistent. In this lecture note, I follow the construction given in Sipser [Sipser]. The basic idea behind constructing a boolean formula corresponding to the fourth condition is the fact that a computation of any non-deterministic Turing machine is highly local, i.e., the execution on input w at the i^{th} step depends only on (a) its state in the last step, (b) the contents of the current cells of its input. Following Sipser, we need to worry about at most six table entries to verify whether any entry in the table is because of a correct transition of N or not. For completeness, we first give the Sipser's construction of a tableau. The reader who do not worry about how the table are constructed may skip it.

Construction given in Sipser [Sipser]

The Sipser construction considers the following cases.

1. For every $a, b \in \Gamma$ and $q, r \in Q$, where $q \neq q_{reject}$, if $\delta(q, a) = (r, b, R)$, put $\left[\frac{qa}{br}\right]$ in the table.
2. For every $a, b, c \in \Gamma$ and $q, r \in Q$, where $q \neq q_{reject}$, if $\delta(q, a) = (r, b, L)$, put $\left[\frac{cqa}{rcb}\right]$ in the table.
3. For every $a \in \Gamma$, put $\left[\frac{a}{a}\right]$ and $\left[\frac{aq_{accept}}{q_{accept}}\right]$, $\left[\frac{q_{accept}a}{q_{accept}}\right]$ in the table.

One can easily verify that this is a consistent construction.

Therefore, we see that to verify the $T(i, j)$ entry, we need to check that the window formed by the intersection of the above row and the column to the left and right should be consistent with one of the construction given above. So, we can write a boolean expression as,

$$\varphi_{move} = \bigwedge_{1 \leq i,j \leq n^k} \left(\bigvee_{c_1, \cdots c_6 \in C} x_{i-1,j-1,c_1} \wedge x_{i-1,j,c_2} \wedge x_{i-1,j+1,c_3} \wedge x_{i,j-1,c_4} \wedge x_{i,j,c_5} \wedge x_{i,j+1,c_6} \right).$$

Now, we can write $\varphi = \varphi_{entry} \wedge \varphi_{start} \wedge \varphi_{accept} \wedge \varphi_{move}$ as the corresponding formula. It is easy to verify that φ thus obtained corresponds to the tableau generated by the action of N on w.

We next need to show that the reduction works in polynomial time. For that, we show that the size of φ is polynomial, which in turns requires us to prove that each condition is represented by a polynomial size CNF.[2] It is easy to see that size of $\varphi_{entry}, \varphi_{start}, \varphi_{accept}$, and φ_{move} has a size bounded by $O(n^{2k}), O(n^k), O(n^{2k})$, and $O(n^{2k})$ respectively. Therefore the size of φ is $O(n^{2k})$, thus completing the proof. ■

References

[Sipser] M. Sipser. *Introduction to The Theory of Computation.* Thomson Course Technology, 2006.

[2]The size of CNF is defined as the number of AND and NOT gates.