CS 4510 Automata and Complexity	4/18/2022
Lecture 20	
Lecturer: Frederic Faulkner	*Scribe(s):*

1 Hard to Satisfy

1.1 The First NP-Complete Problem

Definition 1 *$SAT = \{\varphi \mid \varphi$ is a boolean formula with at least one satisfying assignment$\}$*

A satisfying assignment of φ is an assignment of "True" or "False" to each variable of φ so that φ evaluates to "True". For example, let $\varphi = x_1 \wedge (x_2 \vee \overline{x_1}) \wedge (\overline{x_3} \vee \overline{x_4})$. ($\vee$ is "OR" and $\wedge$ is "AND".) One possible satisfying assignment of φ is (x_1 = True, x_2 = True, x_3 = True, x_4 =False) so φ is satisfiable, i.e. $\varphi \in SAT$.

However, consider $\varphi_2 = x_1 \wedge x_2 \wedge (\overline{x_1} \vee \overline{x_2})$. There is no assignment for x_1 and x_2 that causes φ_2 to evaluate to True, so $\varphi_2 \notin SAT$.

Proposition 2 *$SAT \in NP$*

Proof. A short proof that a formula φ is in SAT is just an assignment of the variables of φ that causes φ to evaluate to True.

Theorem 3 (Cook-Levin, 1971) *SAT is NP-complete.*

Proof. Since we just showed that $SAT \in NP$, it remains to be shown that SAT is NP-hard. Normally, to show that a language B is NP-hard, we reduce some known NP-hard language to B. But then how do we know that A is NP-hard? If we reduce some other NP-hard problem Z to A, we must then prove that Z is NP-hard, and so on. The point is that there must be some method of proving NP-hardness that doesn't depend on any other NP-hard language, so that we can find our first NP-hard language on which to base the rest of the reductions.

Historically, this first language was SAT, and Cook and Levin showed how to prove SAT to be NP-hard without the help of any other NP-hard language. However, before we can start this proof, we need to introduce the notion of an *accepting tableau*.

1.2 An Accepting Tableau for M and x

An accepting tableau for some machine M and string x functions as a proof that M accepts x. It is a table where each row of the table represents one step in M's computation. If T is a valid accepting tableau, it must fulfill four requirements:

- The first row should represent what M looks like before the computation begins: M should be in the start state, the tape should contain x, and the tape head should be located at the first character of x.

- The final row should show that M accepts x. I.e. it should show that M is in the accept state.
- Each row should follow from the row before it according to some transition of M.

Suppose $x = 01\#01$ and M is the recognizer for $w\#w$. A valid accepting tableau might look something like the following:

Step 0:	q_0	0	1	#	0	1
Step 1:	~~0~~	q_1	1	#	0	1
Step 2:	~~0~~	1	q_1	#	0	1
Step 3:	~~0~~	1	#	q_2	0	1
Step 4:	~~0~~	1	q_3	#	~~0~~	1
...						
Step n:	~~0~~	~~1~~	#	~~0~~	~~1~~	q_{acc}

1.3 A decider for L

We prove that any language $L \in NP$ can be reduced to SAT by constructing a mapping reduction that maps any string w to a formula φ.

However, we have an issue. We need to design φ such that φ is satisfiable if and only if $w \in L$. Unfortunately, we don't really know anything about L; L could be a set of graphs with k-cliques, a set of subset-sum problems, a set of sudoku problems, or any other NP-set. What do we know about L?

Not much. We know that $L \in NP$, which means that there is some polytime NTM M which decides L. Perhaps surprisingly, this tiny bit of information about L is sufficient for us to complete the reduction!

We can exploit the following chain of inferences:

$$w \in L \iff M \text{ accepts } w \iff \text{ there is some accepting tableau for } M \text{ on } w$$

. We can then construct φ such that φ is satisfiable if and only if there is an accepting tableau for w. Each assignment of variables to φ will correspond to some way to fill all the cells in the tableau. But this will only result in an accepting tableau for M and w if the original variable assignment satisfied φ.

1.4 Designing φ

φ will consist of four major subformulae, which we will construct one at a time. φ_{cell} will ensure that each cell of the tableau contains exactly one symbol. φ_{start} will ensure that the first row of the table is the starting configuration of M on w. φ_{end} will ensure that M accepts at the end of its computation. φ_{step} will check that each row of the table follows from the previous row by a valid transition of M. Then, we will set $\varphi = \varphi_{cell} \wedge \varphi_{start} \wedge \varphi_{end} \wedge \varphi_{step}$.

What are the variables of φ? Often when discussing examples of boolean formulae we use variables like x_1 or x_{27}. For φ however, we will use a slightly more complicated but more useful indexing scheme. Each variable has the form $x_{i,j,a}$, and should be true if cell (i, j) of our table contains the symbol a. For example, in the tableau given above in section

1.2, $x_{0,3,\#} = True$ because the cell in column 0, row 3 contains #. However, $x_{1,1,q_4} = False$ because the cell in column 1, row 1 does not contain q_4.

Note: How many variables does φ contain? Well, each cell of the table contains either a symbol from the tape or a state, so there are $|Q \cup \Gamma|$ options for a in $x_{i,j,a}$. However, to know the bounds for i and j we need to know how tall and wide the tableau is. Fortunately, we know that M is a polytime NTM, which means that there is some polynomial $p(n)$ which bounds the runtime of M on inputs of length n. (Maybe $p(n)$ is small, like n^2, or perhaps large, such as $n^{100} + 32n^{23} + 100n$; the point is that *some* polynomial exists that is larger than or equal to the number of steps taken by M.) So, if M takes at most $p(n)$ steps, then there are at most $p(n)$ rows in the tableau. Similarly, if M runs for $p(n)$ steps it can use at most $p(n)$ tape cells. So the tableau has $p(n)$ rows and $p(n)$ columns, and there are $p(n) \times p(n) \times |Q \cup \Gamma|$ variables of the form $x_{i,j,a}$ in φ.

1.4.1 φ_{cell}

The role of φ_{cell} is to make sure that each cell contains exactly one character. This formula contains two parts. For the sake of example, let's focus on the cell, say, $(2, 5)$

We must ensure that at least one of the symbols is present in $(2, 5)$, which means that we want either $x_{2,5,\#}$ to be true or $x_{2,5,q_0}$ to be true or... Essentially, we need $x_{2,5,a}$ to be true for some a. Which logical operator corresponds to "at least one"? OR, of course. So, we create the formula $x_{2,5,0} \vee x_{2,5,1} \vee x_{2,5,\#} \vee ....$ A cleaner way to write this is $\vee_{a \in Q \cup \Gamma}\ x_{2,5,a}$.

Next, we must make sure that we do not try to write two symbols into $(2, 5)$. Note that for any two symbols picked at random, for example, # and 0, we do not know if either of those is in $(2, 5)$. However, we do know that at least one of them is NOT is (2,5)! We can represent this by $\overline{x_{2,5,\#}} \vee \overline{x_{2,5,0}}$. More generally, we can make a separate phrase for each pair of symbols like so: $\overline{x_{2,5,\#}} \vee \overline{x_{2,5,0}}$ and $\overline{x_{2,5,\#}} \vee \overline{x_{2,5,1}}$ and $\overline{x_{2,5,1}} \vee \overline{x_{2,5,q_0}}$ and $\overline{x_{2,5,\#}} \vee \overline{x_{2,5,\sqcup}}$ and ... A cleaner way to write this is $\wedge_{a,b \in Q \cup \Gamma, a \neq b}\ (\overline{x_{2,5,a}} \vee \overline{x_{2,5,b}})$

The first formula forces $(2, 5)$ to contain at least one symbol, while the second forces it to contain less then two symbols. Thus, any satisfying configuration must assign $(2, 5)$ exactly one symbol. But we must do this for each cell (i, j), not just $(2, 5)$. So our final formula is

$$\varphi_{cell} = \wedge_{i=0}^{p(n)} \wedge_{j=0}^{p(n)} [(\vee_{a \in Q \cup \Gamma}\ x_{i,j,a}) \wedge (\wedge_{a,b \in Q \cup \Gamma, a \neq b}\ (\overline{x_{i,j,a}} \vee \overline{x_{i,j,b}}))]$$

.

1.4.2 φ_{start}

Thankfully, φ_{start} is much easier than φ_{cell}. Our goal is to force the top row to be "correct", i.e. to represent the machine being in the start state, with w on the tape, and the head at the first character of w. There are no "options" for the first row, in the sense that each cell must contain exactly one specific character. Cell $(0, 0)$ must contain the start state, cells

$(0,1)$ through $(0,n)$ must contain w, and cells $(0,n+1)$ onward must contain blanks. Let $w = w_1...w_n$. Then we have

$$\varphi_{start} = x_{0,0,q_{start}} \wedge (\wedge_{i=1}^{n}\ x_{0,i,w_i}) \wedge (\wedge_{i=n+1}^{p(n)}\ x_{0,i,\sqcup})$$

1.4.3 φ_{end}

φ_{end} is even simpler. Its role is to ensure that M accepts w, which it does by checking that the machine is in the accept state at the end of the computation. This means that the final row of the tableau, row $p(n)$, must contain q_{acc} in one of its cells.

$$\varphi_{end} = \vee_{i=0}^{p(n)} x_{p(n),i,q_{acc}}$$

1.4.4 φ_{step}

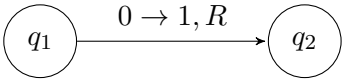
φ_{step} is a little bit trickier than the other three subformulae, but is still not too difficult. The goal of φ_{step} is to ensure that each row of the tableau follows from the previous row according to some transition of M. However, we cannot just encode every possible pair of sequential rows in φ, because there are exponentially many possible values for a given row of M.

Instead, we will focus on 2x3 subsections ("windows") of our tableau. For any machine M, there are only a small number of possibilities for a given 2x3 window. For example, the following window is never possible:

0	1	0
0	0	0

This would mean that in one step of the computation, one of the cells changed from a 1 to a 0 even though it is not currently under the tape head! (If it were under the tape head, the top left square would contain a state symbol instead.)

On the other hand, suppose our machine contains the following transition:



Then the following are some of the valid windows for M corresponding to this transition:

q_1	0	1
1	q_2	1

0	0	1
q_2	0	1

0	0	q_1
0	0	1

In addition, of course, some sections will probably not change at all, if the tape head is somewhere else on the tape, leading to windows like:

0	0	1
0	0	1

0	#	1
0	#	1

0	0	0
0	0	0

Although we will not prove it here, if every window of our tableau is a valid window, then each row of the tableau must follow from the previous row. So we just need to construct a formula that forces each window of the tableau to be valid.

Each cell (except for the cells at the bottom and right of the tableau) must be the top left corner of a valid window. Suppose that we label the valid possible windows $W_1, W_2, ...$

Then, for example, for the cell $(2, 5)$, we need a formula that expresses ($(2, 5)$ follows the pattern of W_1) or ($(2, 5)$ follows the pattern of W_2) or ...

For example, suppose we want to express that the cell $(2, 5)$ is possibly the top left corner of a window like the one shown below.

q_1	0	#
1	q_2	#

Then we would need a formula like $x_{2,5,q_1} \wedge x_{2,6,0} \wedge x_{2,7,\#} \wedge x_{3,5,1} \wedge x_{3,6,q_2} \wedge x_{3,7,\#}$.

But we must do this for every possible window, and every possible top left corner. So our final formula is

$\varphi_{step} = \wedge_{i=0}^{p(n)-1} \wedge_{j=0}^{p(n)-2} (\vee_{W \in \{\text{valid windows of M}\}}$ the window whose top left corner is cell (i, j) matches W)

1.5 Finishing up

To recap, we have created for formulae $\varphi = \varphi_{cell} \wedge \varphi_{start} \wedge \varphi_{end} \wedge \varphi_{step}$. Every assignment of variables to φ creates some table, but it is only an accepting tableau for M on w if each of the four smaller subformulae (and thus φ as a whole) is satisfied. Similarly, if you have an existing accepting tableau for M on w, you can use it to find a satisfying assignment for φ.

So $\varphi \in SAT$ exactly when there is an accepting tableau for M on w, i.e. $w \in L$. But then the decider S given in section 1.3 is in fact a decider for L!

We are almost done. Recall that for the definition of NP-complete, it is not enough for S to be a decider for L, because it must also run in polynomial time. So technically we must prove that constructing φ takes polynomial time. We will not do so here, but a proof can be found in the book. As we mentioned earlier, there are polynomially many variables in φ, and with some thought you can see that each subformula uses any given variable at most a constant number of times, so that the total amount of work needed is to construct φ is polynomial.

This completes the proof. We have successfully reduced an arbitrary langauge L to SAT in polynomial time, and thus proven that SAT is, in some sense, one of the hardest problems in NP.