Lecture 4

27 JAN 2014

We will prove that the decision problem 3SAT is NP-complete. This is the canonical NP-complete problem by which many other decision problems can be proved to be NP-complete.

Theorem: (Cook-Levin, early 1970s)
3SAT is NP-complete.

3SAT is the problem deciding whether the conjunction of m clauses consisting of 3 literals each all over n variables is satisfiable. For example, variables are x_1, x_2, x_3 & formula is

$$\overbrace{(x_1 \vee \bar{x}_2 \vee x_3)}^{\text{clause}} \wedge \overbrace{(\bar{x}_1 \vee \bar{x}_2 \vee x_3)}^{\text{clause}} \wedge (x_2 \vee x_3 \vee x_1) \wedge \cdots$$

Easy part: this is clearly in NP.

If the input is in the language, then there exists a satisfying assignment & the prover can provide it. You can then check in polytime.

If it is not in the language, then there does not exist a satisfying assignment.

- showing that this problem is NP-hard requires more cleverness.

We first begin by showing that a problem called Circuit SAT is NP-complete.

Circuit SAT: given as input is a circuit consisting of AND, OR, & NOT gates, w/ one output bit, is there an assignment to the inputs such that the output is equal to one?

As part of the proof, we will show the computational equivalence between the poly-time Turing machine model & the poly-time uniform circuit family model.

~~Again~~ Like 3SAT, circuit SAT is in NP. If there is a satisfying assignment, the prover can provide this & then it is possible to check in poly time (p-ly nomial in circuit size) whether it is actually ~~in the language~~ a satisfying assignment.

To show hardness, we need to prove that a circuit can simulate the action of any Turing machine on the input x & the witness w. (this will prove the computational equivalence of these models.)

Make the following assumptions about TM M:

1) the tape alphabet is $\triangleright, 0, 1,$ + blank
2) M runs in time at most $t(n)$
   + uses space at most $s(n)$
   where + & s are poly(n).

Recall for a Turing machine that we have tape alphabet Γ, memory register states Q, & transition function δ. $\langle q, c, q', c', s \rangle$

The simulation involves having

1) a bit for each memory state
   $\tilde{q}_s, \tilde{q}_1, \ldots, \tilde{q}_m, \tilde{q}_Y, \tilde{q}_N$
   start, accept, reject.
   Initially set $\tilde{q}_s = 1$ + all others to zero.
2) Represent each square on TM tape w/ three bits: two to identify symbol $\{\triangleright, 0, 1, b\}$
   + another as a flag to indicate if the read-write head is pointing to this symbol on the tape.

Denote the bits for the tape contents by $(u_1, v_1) \ldots (u_{s(n)}, v_{s(n)})$ & flags by $f_1, \ldots) f_{s(n)}$

Initially $f_1 = 1$ & all others are zero.

Also the first $(u_1, v_1) \ldots (u_n, v_n)$ represent x

(u_{n+1}, v_{n+1}) represent b &

$(u_{n+2}, v_{n+2}) \ldots (u_{p(n)+n+2}, v_{p(n)+n+2})$ represent w.

3) Also have a global flag register F, initially set to zero

will see the point of this later

Idea is to repeat $t(n)$ times a simulation step of the Turing machine.

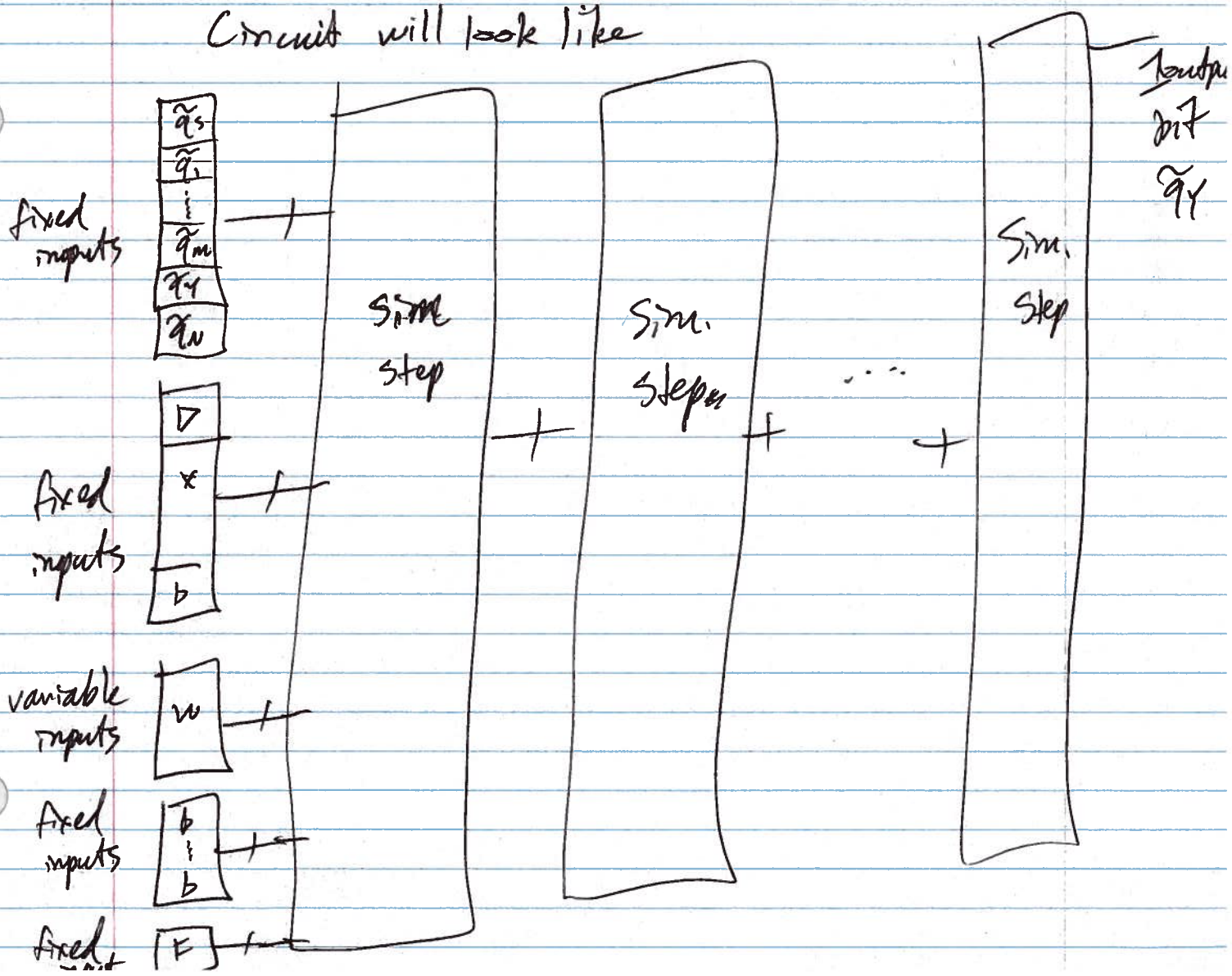
Each simulation step goes through all the program lines of the transition function making sure that they are executed if need be.

So we need to simulate a program one of the form

$\langle q_i, c, q_j, c', s \rangle$

- q_i: current mem. state
- c: tape content
- q_j: next mem.
- c': new tape content
- s: new location of tape

Circuit will look like



Procedure for executing program line is simple.

$\langle q_i, c, q_j, c', s \rangle$

(1) Check to see that $\tilde{q}_i = 1$, indicating that current state of machine is q_i.

(2) For all tape squares

a) check if the global flag bit is zero

b) check that the flag bit for this square is set to one, meaning that tape head is here.

c) check if simulated tape contents here are c.

d) if $a \wedge b \wedge c = 1$, then

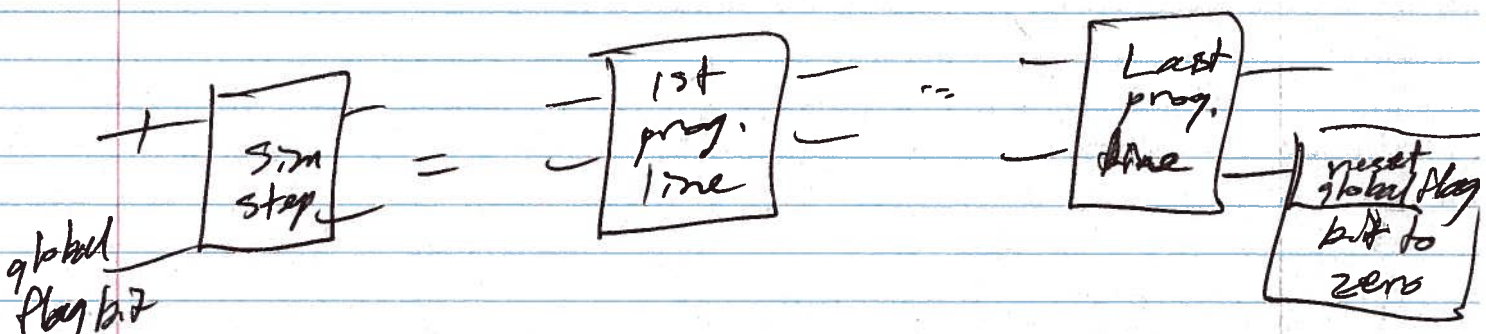
1) set $\tilde{q}_i = 0$ + $\tilde{q}_j = 1$

2) Update tape contents to c'.

3) Update flag bits for this & adjacent squares depending on where tape should be next.

4) Set the global flag bit to one, meaning that we should go to next sim. step.

Operations in 2a – 2d act on a constant number of bits. Thus by universality result for circuit, there is a circuit doing these steps which requires a constant # of gates.



each sim. step takes # of gates

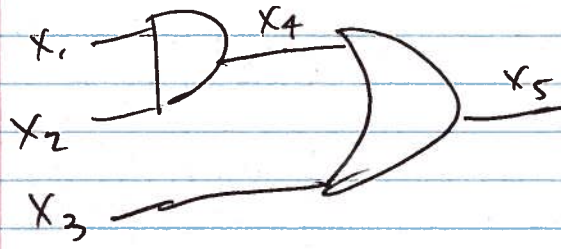
$O(s(n))$

$\Rightarrow$ total number of gates is

$O(t(n)\, s(n))$

which is poly(n).

To get a reduction from CIRCUIT SAT to 3SAT, just realize that any circuit can be written as



can write as $x_6 \wedge (x_6 \Leftrightarrow \overline{x_5}) \wedge (x_5 \Leftrightarrow x_4 \vee x_3) \wedge (x_4 \Leftrightarrow x_1 \wedge x_2)$

for every circuit element, we're establishing a relationship between input variables + output. can reexpress those relationships as satisfiability formulas.

For example, truth table for NOT ☐

x_{in}	x_{out}	sat
0	0	0
0	1	1
1	0	1
1	1	0

so the formula

$(\overline{x_{in}} \vee \overline{x_{out}}) \wedge (x_{in} \vee x_{out})$

↑ idea ☐ to penalize non satisfying instances

penalize 1 1

penalize 00

same for AND

x_1	x_2	x_{out}	sat
0	0	-	1
0	0	1	0
0	1	-	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

penalize these terms

$$(x_1 \vee x_2 \vee \overline{x_{out}}) \wedge$$
$$(x_1 \vee \overline{x_2} \vee \overline{x_{out}}) \wedge$$
$$(\overline{x_1} \vee x_2 \vee \overline{x_{out}}) \wedge$$
$$(\overline{x_1} \vee \overline{x_2} \vee x_{out})$$

for OR

x_1	x_2	x_{out}	sat
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

penalize this

$$(x_1 \vee x_2 \vee \overline{x_{out}}) \wedge (x_1 \vee \overline{x}_2 \vee x_{out}) \wedge (\overline{x}_1 \vee x_2 \vee x_{out}) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee x_{out})$$

only a constant blowup of each gate, so no more than poly overhead with this reduction.