CS 361 Meeting 30 — 5/8/20

Encoding Problems as Satisfiability Questions

(Click for video)

1. In the early 70's Stephen Cook and Leonid Levin independently showed that there are examples of problems in NP that are universal in the sense that all problems in NP are polynomial-reducible to these problems. Such problems are said to be NP-complete. That is:

   Definition: We say that a language B is *NP-Complete* if $B \in$ NP and for all $A \in$ NP, $A \leq_p B$.

2. Surprisingly (given the reductions we have looked at), Subset-sum was not the problem they showed to be NP-complete (although it is). Instead, they showed that satisfiability is NP-complete.

3. Before plunging into the proof of the Cook-Levin Theorem I want to give you one artificial but concrete illustration of how you could encode another problem as a SAT problem.
   - Not surprisingly, what I want to show you is how to "reduce" a Subset Sum problem to a satisfiability problem.
   - I don't however, want to do full-fledged Subset Sum. The construction for the general problem would be too complex.
   - Instead, I want to limit our attention to some trivial instances of Subset Sum, namely
     - The list of integers we get to work with is of size 2 (we will call them a and b),
     - Each of the integers in the set is a 1 bit binary number (so a and b are just names for bits), and
     - The target sum is a two bit binary number with s_0 referring to its low order digit and s_1 its high order digit.
   - Yes, this is silly, but I hope by showing all the details of how to convert such a small problem into a boolean formula I will enable you to imagine how much larger problems might be similarly encoded/translated/reduced.

Click here to view the slides for this class

4. Each of the names a, b, s_0 and s_1 can be thought of as a digit or as a boolean value. So, we can build a truth table for the solvability of all instances of the Subset Sum problem restricted to such small lists of small numbers.

a	b	s_1	s_0	
0	0	0	0	true
1	0	0	0	true
0	1	0	0	true
1	1	0	0	true
0	0	0	1	false
1	0	0	1	true
0	1	0	1	true
1	1	0	1	true
0	0	1	0	false
1	0	1	0	false
0	1	1	0	false
1	1	1	0	true
0	0	1	1	false
1	0	1	1	false
0	1	1	1	false
1	1	1	1	false

5. Recall that we showed how to convert a truth table into a conjunctive normal form formula by focusing on the rows of the truth table that evaluated to false. With this in mind, we can eliminate all but the false rows of our table. Better yet, we can group them into 4 subgroups that can be described by 4 formulae.

a	b	s_1	s_0	
0	0	0	1	false

$a \lor b \lor s_1 \lor \overline{s_0}$

0	0	1	0	false
1	0	1	0	false

$b \lor \overline{s_1} \lor s_0$

0	1	1	0	false

$a \lor \overline{b} \lor \overline{s_1} \lor s_0$

0	0	1	1	false
1	0	1	1	false
0	1	1	1	false
1	1	1	1	false

$\overline{s_1} \lor \overline{s_0}$

6. Combining these little formulas give us the boolean expression

$$(\overline{s_1} \lor \overline{s_0}) \land (a \lor \overline{b} \lor \overline{s_1} \lor s_0) \land (b \lor \overline{s_1} \lor s_0) \land (a \lor b \lor s_1 \lor \overline{s_0})$$

which describes whether a particular instance of our mini-subset sum problem is solvable or not.

7. Given values for a, b, s_1, and s_0 we can complete the process of turning that problem into an instance of SAT by anding the formula above with a formula that is only satisfied by the particular values of a, b, s_1, and s_0 we care about.

   For example if $a = 1$ and $b = 1$ and $s_1 s_0 = 01$, the formula

$$(\overline{s_1} \lor \overline{s_0}) \land (a \lor \overline{b} \lor \overline{s_1} \lor s_0) \land (b \lor \overline{s_1} \lor s_0) \land (a \lor b \lor s_1 \lor \overline{s_0}) \land$$

$$(a \land b \land \overline{s_1} \land s_0)$$

   is satisfiable only if the corresponding Subset sum problem is solvable.

8. This isn't meant to show how to reduce Subset sum to satisfiability. It is merely provided to illustrate the fact that once the variable are associated with relevant information, satisfiability can be used to encode interesting problems.

An Approach to Proving the Cook-Levin Theorem

(Click for video)

1. So, let's figure out how to prove the Cook-Levin Theorem.

   Theorem: SAT is NP-Complete. That is, SAT $\in$ NP and for any $A \in$ NP, $A \leq_p$ SAT.

2. Just the statement of this theorem involves a number of technical definitions we have only recently covered:

   SAT The satisfiability problem is the problem of deciding whether given a formula over a set of Boolean variables there is an assignment of true and false values to the variables that make the formula evaluate to true. Formulated as a language, this problem becomes

$$\text{SAT} = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula } \}$$

   NP is the class of languages that are decidable in polynomial time on a nondeterministic single-tape Turing machine. In other words

$$NP = \bigcup_k NTIME(n^k)$$

   or equivalently NP is the class of language that are polynomial verifiable.

   $\leq_p$ We say that A is polynomial-time reducible to B:

$$A \leq_p B$$

   if and only if there exists a polynomial time function $f : \Sigma_A^* \rightarrow \Sigma_B^*$ such that $w \in A$ if and only if $f(w) \in B$.

NP-Complete We say that a language B is *NP-Complete* if $B \in$ NP and for all $A \in$ NP, $A \leq_p B$.

3. The only thing we have to go on when we try to prove that $A \leq_p$ SAT for some A, is that $A \in$ NP. That is, we know that there is some nondeterministic TM, M_A, that decides A and that there is some polynomial $p_A(x)$ such that the length of all computation branches that can be followed by M_A on input $|w|$ is bounded by $p_A(x)$.

4. To show that $A \leq_p SAT$, we need to show how to construct a polynomial-time computable function $f : \Sigma_A^* \rightarrow \Sigma_{SAT}^*$ such that $f(w) \in SAT$ if and only if $w \in A$.

5. Since we don't really know what A is, but only can assume we have a description of M_A and p_A, what we really need is a polynomial time computable procedure

$$\mathcal{F} : TM \times \text{ polynomial} \rightarrow (\Sigma_A^* \rightarrow \Sigma_B^*)$$

such that $\mathcal{F}(M_A, p_A) = f_A$. That is, we need a (not-necessarily even computable) function that given a Turing machine description and a polynomial description produces a polynomial-time algorithm that translates strings over the alphabet of A into strings over the alphabet of SAT in such a way that exactly the members of A get mapped into into members of SAT.

Building SAT Formulae about Computation Histories

(Click for video)

1. Given a description of a TM, M, and a polynomial bounding its running time our procedure $\mathcal{F}$ has to produce a procedure that will take a possible input w to the machine M and produce a string encoding a 3-SAT formula that is satisfiable if and only if w belongs to the language of the machine.

2. The formulae we produce will encode properties of possible computation histories of M on w.

3. We have to show how to do this for any Turing machine A, but to enable us to give concrete examples of elements of the construction we will use the NTM shown in Figure 1 that decides subset sum.

4. We know that a Turing machines computation process can be described by sequences of configurations called computation histories.

   - We have previously written configurations as triples composed of the current state, the string of symbols before the tape head and the string of symbols from the tape head to the end of the tape. For example, the following could be a string encoding a computation history for the machine shown in Figure 1.

```
( S $ 1 0 1 0 $ 1 0 # 1 0 1 # 1 1 1 # )
( $ S 1 0 1 0 $ 1 0 # 1 0 1 # 1 1 1 # )
( $ 1 S 0 1 0 $ 1 0 # 1 0 1 # 1 1 1 # )
( $ 1 0 S 1 0 $ 1 0 # 1 0 1 # 1 1 1 # )
( $ 1 0 1 S 0 $ 1 0 # 1 0 1 # 1 1 1 # )
( $ 1 0 1 0 S $ 1 0 # 1 0 1 # 1 1 1 # )
( $ 1 0 1 0 $ C 1 0 # 1 0 1 # 1 1 1 # )
( $ 1 0 1 0 $ # Z 0 # 1 0 1 # 1 1 1 # )
( $ 1 0 1 0 $ # # Z # 1 0 1 # 1 1 1 # )
( $ 1 0 1 0 $ # # # C 1 0 1 # 1 1 1 # )
```

   - In this version, we have left out the commas typicially used to separate the three components of a configuration and instead simply inserted symbols representing the machine's state at the point that separates the string of symbols before the tape symbol from those at or under the read head.
   - Also, while we have typically encoded computation histories as linear strings, it should be clear that we can view the computation history as a table with the symbol in the rth row and cth column representing the cth symbol of the configuration describing the state of the machine before its rth move.

5. Given this tabular view of computation histories, we can imagine creating a set of boolean variables that completely describe a computation

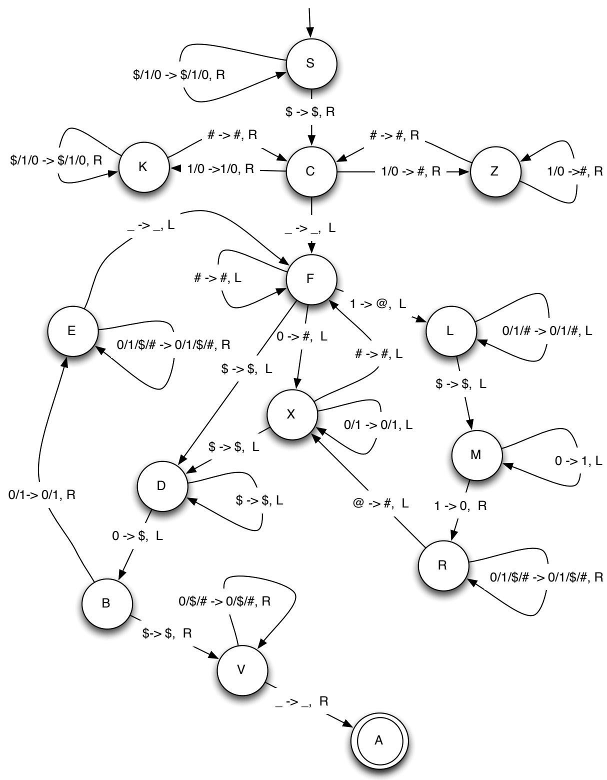

Figure 1: A non-deterministic Turing machine that decides Subset sum in polynomial time.

history. For each cell in the computation history table there would be one boolean variable for each symbol that could appear in that cell. That is, we would have a variable

$$x_{r,c,s}$$

for every row r, column c and symbol s in the union of our Turing machines tape alphabet Γ, its state set Q, and any other symbols we use to encode the configurations (like the parentheses we use to mark the beginning and end of each configuration) with $x_{r,c,s} = \text{ true } \iff$ s appeared in the cth position of the rth row of the table.

6. To create such a set of variables, we would need to know the size of the largest computation history we might have to explore to find an accepting computation.

   - Since we know that the language A which we hope to reduce to SAT is in NP, we know that some TM, M_A decides A and operates in TIME($p_A(n)$) for some polynomial $p_A(n)$. Therefore, if we include $p_A(n)$ rows and columns in our table that will be sufficient.
   - The number of variables will be the square of the running time of the TM but that is still a polynomial in the size of the input.

Details of the Conditions on 3-SAT Variables, Pt. 1

(Click for video)

1. To encode our tabular representation of a polynomial-time-bounded TM as a sequence of variable, we will have one variable of the form

$$x_{r,c,s}$$

   for every row r, column c and symbol s that can appear in the history table with $x_{r,c,s} = \text{ true } \iff$ s appeared in the cth position of the rth row of the table.

2. Given these variables, we want to construct a boolean expression that will be satisfiable iff there is an accepting computation on a given input. Our expression will need to capture four requirements:

- Only one symbol can appear in each cell of our computation history table. Therefore, for any r and c, exactly one of the $x_{r,c,s}$ should be true.
- The contents of the first row of the table must correctly describe the initial configuration.
- The computation must accept.
- For each r row $r+1$ must describe a configuration reachable in one step from the configuration described by row r.

3. We can describe a formula $happy_{r,c}$ that is true iff the variables reflect that a unique symbol is associated with cell r,c.
   - We can capture the need for some symbol to be associated with the cell with the formula:
   $$\text{assigned}_{r,c} = \bigvee_{s \in \Gamma \cup Q \cup \{(,)\}} x_{r,c,s}$$
   - We can capture the need for the assignment to be unique with with the formula:
   $$\text{unique}_{r,c} = \bigwedge_{s,t \in \Gamma \cup Q \cup \{(,)\}} (\overline{x_{r,c,s}} \vee \overline{x_{r,c,t}})$$
   - Combining these we then get:
   $$\text{happy}_{r,c} = \text{assigned}_{r,c} \wedge \text{unique}_{r,c}$$
   - Note that this formula is in CNF (so it could easily be turned into 3-CNF).
   - Note that the size of this formula depends on the size of the machine's state set and tape alphabet, but not on the input size.
4. Capturing the initial configuration is easy. Assuming the input is $w = w_1 w_2 ... w_n$, we define:
   $$start = x_{0,0,(} \wedge x_{0,1,q_0} \wedge (\bigwedge_{1 \le i \le n} x_{0,i+1,w_i}) \wedge x_{0,n+2,)} \wedge (\bigwedge_{n+2 < i \le p(n)} x_{0,i,\_})$$
   This ensures that the first row describes a configuration starting in the machine's start state with w on the tape followed by blanks.
5. Making sure that the computation accepts simply require making sure that some cell in the table contains the accept state:
   $$accept = \bigvee_{0 \le r,c \le p(n)} x_{r,c,q_{accept}}$$

Details of the Conditions on 3-SAT Variables, Pt. 2

(Click for video)

1. The tricky part is making sure that the configuration in each row of the table yields the configuration in the next row.
   - The secret to capturing this is to notice that the contents of consecutive rows in the computation history must be identical except for the symbols before and after the cell holding the state in the earlier row. Therefore, if we verify the correctness of each 2 by 3 block of cells in our table, we can be sure all transitions are handled correctly.
   - Given that the size of the tape alphabet and state set are fixed we know that there are a fixed number ($(|\Gamma| + |Q| + 2)^6$) of possible configurations of a 2 by 3 subsection of the history table. The number of invalid 2 by 3 subsection must be smaller that this.
   - For each invalid configuration, we can produce a boolean expression that verifies that the bad configuration does not appear at a particular point in our table. For example, the subsection:

0	q_4	1
1	1	q_4

     would be invalid for any machine (since the symbol 0 which was not under the tape head has changed into a 1).
   - If we name this configuration b, then the expression
   $$\text{notbad}_b(r,c) =$$
   $$\overline{x_{r,c,0}} \vee \overline{x_{r,c+1,q_4}} \vee \overline{x_{r,c+2,1}} \vee \overline{x_{r+1,c,1}} \vee \overline{x_{r+1,c+1,1}} \vee \overline{x_{r+1,c+2,q_4}}$$
   yields true only if this particular bad configuration does not appear at position r,c.

- Better yet, this formula is a disjunction. So the formula:

$$\text{notAllBad}(r,c) = \bigwedge_{b\,\in\,\text{bad configurations}} \text{notbad}_b(r,c)$$

  ensures us that everything is fine at r,c and is in CNF.

2. With these pieces, we can describe the formula we use to reduce A to SAT:

$$\phi = (\bigwedge_{0\le r,c\le p(n)} \text{happy}(r,c)) \wedge (\bigwedge_{0\le r,c\le p(n)-\epsilon} \text{notAllBad}(r,c)) \wedge start \wedge accept$$

3. This shows that we can reduce any language in NP to 3-SAT!