CS 361 Meeting 29 — 5/6/20

The Class NP

(Click for video)

1. Recall that in the previous discussion we argued that the class P:

   Definition: P is the class of language that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words

   $$P = \bigcup_k TIME(n^k)$$

   was interesting in that a) it included most practical algortihms and b) its membership was insensitive to whether our computing model was very low-level (a 1-tape deterministic Turing machine) or included more powerful features (multiple tapes, multiple stacks, a GPU!).

2. Of course, one of the biggest variation in model we can make is to switch from a deterministic model to a non-deterministic model. For DFAs and TMs with no limits on the durations of their computations, we have shown nondeterministic modes and deterministic modes are equivalent. We might well wonder whether this is the case for TMs limited by some time bound.

3. As you should already know, it is not known whether this is the case.

4. To allow us to explore this question, we consider the execution time of a nondeterministic TM to be the length of the longest path in its computation tree and say

   Definition: $NTIME(t(n)) = \{L \mid L$ is a language decided by an $O(t(n))$ time nondeterministic Turing machine $\}$.

   Definition: NP is the class of language that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words

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

Click here to view the slides for this class

5. Given that one way to understand nondeterminism is that nondeterministic machines are great guessers, another approach to defining the cost of a nondeterministic algorithm is to concentrate on the effort required to verify that the "guess" was correct.

6. We can therefore define the notion of a verifier and its running time

   Definition: A *verifier* for a language A is a Turing machine V where

   $$A = \{w \mid \text{ for some string } c, w\#c \in L(V)\}$$

   We say that V is a polynomial time verifier if it runs in polynomial time in the length of w. In this case, we say that A is polynomial verifiable.

7. In fact, a language is in NP if and only if it is polynomial time verifiable.

   - If there is a verifier V for A, then we can build a non-deterministic TM M_A that on input w decides whether $w \in A$ in polynomial time by guessing a string c (of length bounded by the appropriate polynomial) and then runs V on $w\#c$.
   - If A is decided in polynomial time by M_A, then we can build a machine V with language $\{w\#c \mid c$ is an accepting computation history of M_A on input $c\}$. Clearly, V can run in polynomial time in the length of w since the number of configurations in the c associated with w is bounded by a polynomial.

Two Sample Problems from NP

(Click for video)

1. My favorite example of a problem in NP is definitely the Subset Sum problem.

2. Given a list of numbers like:

   17, 24, 5, 9, 11, 24, 57, 4, 39, 40, 84, 11, 19

   Is there some sublist of these numbers that adds up to exactly N? (for some given N)

3. It should be clear that there is a polynomial time verifier for this problem.

4. It should also be clear how this verifier could be turned into a nondeterministic algorithm for the problem. We would first guess some numbers and then use the verifier to see if we got lucky.

5. At the same time, checking every subset would require exponential work and there is no obvious algorithm that is more efficient.

6. My second favorite problem in NP is the 3-dimensional matching problem.

   > **Definition:** Given three sets of items, X, Y, and Z, and a set of allowable triples, $A \subseteq X \times Y \times Z$, we say that $M \subseteq A$ is a *3-dimensional matching* if for any pair of distinct triples (x_1, y_1, z_1), $(x_2, y_2, z_2) \in M$, $x_1 \neq x_2$, $y_1 \neq y_2$, and $z_1 \neq z_2$.

   The problem is whether given X, Y, Z, and $A \subseteq X \times Y \times Z$, we can find an $M \subseteq A$ of a specified size that forms a valid matching.

7. Those of you who have taken 134 with me over the last few years have seen this problem in the guise of the "Awkward Diners" problem:

   > Suppose a set C of dinner companions are at a restaurant serving a set of entrees, E, and a set of desserts, D where conveniently $|C| = |E| = |D|$. None of the diners wants to order either the same entree or the same dessert as any of their companions and each diner has his or her own preferences (expressed as a subset of $E \times D$) of combinations of an entree and a dessert they are willing to eat. Given all of their preferences, is their an order for the table that satisfies everyone?

8. If we let $X = C$, $Y = E$, $Z = D$, and $A = \{(c, e, d) \mid c$ is willing to eat e as an entree with d as a dessert $\}$, then the "Awkward Diners" problem is clearly just a special case of 3-dimensional matching in disguise. In particular, it is the case where $|X| = |Y| = |Z| = |M|$.

9. For example, we might have $C = \{$ Beth, Carl, Diana, Evan, Fred, Gina, Harry $\}$, $E = \{$ Sea Scallops, Pumpkin Ravioli, Roast Chicken, Shortribs, Roast Skate, Flying Pig, Lobster$\}$, and $D = \{$ Key Lime Pie, Apple Tart, Black Forest Cake, Tangerine Sherbet, Cheesecake, Tiramisu, Creme Brule $\}$. Beth might only be willing to eat sea scallops with key lime pie, shortribs with key lime pie, roast skate with tangerine sherbet, or roast chicken with tangerine sherbet. Carl might only be willing to eat ... and so on.

10. It should be clear that we can efficiently verify a proposed solution to this problem and therefore it is in NP.

Polynomial Time Reducibility

(Click for video)

1. Hopefully, those of you who saw the subset-sum and "Awkward Diners" previously in CS 134 remember the punch line. In some sense, the subset sum problem and the 3-dimensional matching problem where $|X| = |Y| = |Z| = |M|$ are the same problem.

2. To see that this is the case, first number the elements of the sets X, Y, and Z. That is, assuming all of the sets involved are of size n we would write $X = \{x_0, x_1, x_2, ...x_{n-1}\}$, $Y = \{y_0, y_1, y_2, ...y_{n-1}\}$, and $Z = \{z_0, z_1, z_2, ...z_{n-1}\}$.

3. Now, we can associate each triple $(x_i, y_j, z_k) \in A$ with the number $b^i + b^{n+j} + b^{2n+k}$. That is, we will associate each triple with a number written in base b as $3n$ digits 000...1...000...1...000...1...000 where the last 1 appears i positions from the end, the second 1 appears $n + i$ positions from the end and the first 1 appears $2n + k$ from the end.

4. If there is a subset M of triples from A that forms a valid 3-dimensional matching, then each element of X, Y, and Z will appear in exactly 1 triple. Therefore, for any position m, exactly one of the numbers $b^i + b^{n+j} + b^{2n+k}$ associated with these triples will have a 1 at position m. Therefore the sum of the numbers corresponding to the triples will be $3n$ digits long and of the form $1111...111 = b^{3n} - 1$

5. On the other hand, given the numbers associated with the triples in A, if we can find a subset of these numbers that add up to $b^{3n} - 1$, then the corresponding set of triples must form a valid matching M. (To be careful, we should choose b to be large enough that carries between columns cannot occur during the addition.)

6. What we have shown is clearly a many-to-one reduction of 3-dimensional matching to subset sum. That is:

$$\text{3-dim matching} \leq_m \text{subset sum}$$

7. The reduction we have described, however, is more than computable. It is easy to compute. In particular, the number of numbers we must generate from the 3-dimensional matching problem and the number of digits in each number is linear in the length of the description of the original problem. As a result, we can build a Turing machine to compute the appropriate subset sum problem in some polynomial number of steps as a function of the input size (the size of the original matching problem).

8. This leads to:

    Definition: We say that A is polynomial-time reducible to B (written $A \leq_p B$) 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 $f(w) \in B$.

9. In particular, we now can say:

$$\text{3-dim matching} \leq_p \text{subset sum}$$

10. Just as many-to-one reducibility allowed us to draw conclusions about whether a language was decidable or recognizable, polynomial reducibility allows us to make statements about membership in P (and eventually NP).

    Theorem: If $A \leq_p B$ and $B \in P$ then $A \in P$.

    and conversely

    Theorem: If $A \leq_p B$ and $A \notin P$ then $B \notin P$.

11. Thus, we now know that if subset sum is in P then 3-dimensional matching (at least looking for complete matchings) is in P. Also, if 3-dimensional matching is not in P then subset sum must not be in P either.

Satisfiability (and 3-SAT)

(Click for video)

1. Another interestingly difficult problem is satisfiability for boolean expressions.

    - We restrict our attention to expressions involving boolean variables, ands ($\wedge$), ors ($\vee$), and negation ($\overline{e}$) like:

$$(x_1 \wedge ((x_2 \wedge \overline{x_3}) \vee (x_3 \wedge \overline{x_2}))) \vee (\overline{x_1} \wedge ((x_2 \wedge x_3) \vee \overline{(x_2 \vee x_3)}))$$

    - We say such a formula is satisfiable if there is an assignment of the values true and false (equivalently 0 and 1) to the variables used in the formula that causes the formula to evaluate to true (1).

    - For example,

$$(x_1 \wedge ((x_2 \wedge \overline{x_3}) \vee (x_3 \wedge \overline{x_2}))) \vee (\overline{x_1} \wedge ((x_2 \wedge x_3) \vee \overline{(x_2 \vee x_3)}))$$

      is an example of a boolean formula over three variables. If I got it right, its value is true exactly when the number of true variables is even (i.e., it is based on even parity). As a result this is an example of an easily satisfiable formula. (The truth table is shown below.)

x_1	x_2	x_3	ans
false	false	false	true
false	false	true	false
false	true	false	false
false	true	true	true
true	false	false	false
true	false	true	true
true	true	false	true
true	true	true	false

- On the other hand,

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

  is an example of a formula with no satisfying assignment.

2. The satisfiability problem is the problem of deciding membership in the language

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

3. We will also be interested in a version of the problem restricted to formulas written in a canonical form called *3-conjunctive normal form* or 3-CNF.

4. 3-CNF is a special form of a more general special form called conjunctive normal form.

   - A formula is in *conjunctive normal form* (CNF) if it is a sequence of clauses all connected by conjunctions (ands) where each clause is a series of disjunctions (ors) or variables or their negations.
   - Given any boolean formula, we can rewrite it in conjunctive normal form.
   - A simple way to see that this is true is to visualize the truth table for the formula you want to express in CNF.
     – Your CNF formula will have one clause for each row of the truth table where the value of the function is false. Each clause should be true only if the variables do not have the values corresponding to that row.
     – Your formula collectively says the result should be true if the variable values do not match any row where the value would be false.

5. In 3-CNF, all formulas are conjunctions of clauses that are disjunctions of exactly 3 variables or negations of variables.

   - Given a formula in CNF, two tricks let us convert it into 3-CNF.
     – If the formula has less than three terms, duplicate one of its terms.
     – if the formula has more than three terms, add a new variable and break the formula up into two subparts such as:

$$x_1 \vee x_2 \vee x_3 \vee x_4 = (x_1 \vee x_2 \vee x_{new}) \wedge (\overline{x_{new}} \vee x_3 \vee x_4)$$

     – Any satisfying assignment of the original problem will be a satisfying assignment of the 3-CNF form with added variables, and any assignment to the formula with added variables will reduce to a satisfying assignment of the original problem when the added variables are ignored.

6. The problem of determining if a boolean formula in 3-CNF is satisfiable is known as 3-SAT.

$$\text{3-SAT} = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable boolean formula in 3-CNF }\}$$

Reducing 3-SAT to Subset Sum

(Click for video)

1. Our ultimate goal is to show that any problem in NP can be polynomial reduced to satisfaction of a boolean formula. Before we do that, let's see that satisfaction for 3-CNF terms can be reduced to subset sum.

   - As we did for the 3-dimensional matching problem, we will generate a peculiar set of number to use as an instance of the subset sum problem based on the 3-CNF formula we are given.

- Let's assume the formula has t clauses over v variables. We will assume the variables are numbered $x_1, x_2, ...x_v$ and we will number the clauses from 1 to t. Each number in our collection will be $2k + v$ digits long (possibly with leading 0s). The digits of each number will be divided into three groups as shown below:

x_1	x_2	...	x_v	c_1	c_2	...	c_t	s_1	s_2	...	s_t

- There will be two numbers in the set for each variable x_i. One of these numbers will correspond to setting x_i to true, the other to setting x_i to false.
  - In both numbers the ith digit will be 1.
  - All of the remaining $v + t$ leading digits will be 0.
  - The remaining digits, labeled s_i in the diagram above, will differ in the two numbers for a given variable. In one number s_j will be one if x_i appears in clause j. In the other number, s_j will be one if $\overline{x_i}$ appears in clause j. Otherwise, s_j will be 0.

  For example, given the formula:

  $$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$$

  We would generate the numbers:

	x_1	x_2	x_3	c_1	c_2	c_3	c_4	s_1	s_2	s_3	s_4
x_1	1	0	0	0	0	0	0	1	1	0	0
$\overline{x_1}$	1	0	0	0	0	0	0	0	0	1	1
x_2	0	1	0	0	0	0	0	1	0	1	0
$\overline{x_2}$	0	1	0	0	0	0	0	0	1	0	1
x_3	0	0	1	0	0	0	0	0	1	1	0
$\overline{x_3}$	0	0	1	0	0	0	0	1	0	0	1

- The goal of selecting these numbers in this way is to connect the set of numbers chosen to a possible truth assignment for the variables in such a way that the sum of the numbers chosen will indicate which clauses are satisfied by the corresponding truth assignment.
  - In the final subset sum problem, we will require that the first v digits of the sum be all ones. This will imply that in any solution to the subset sum problem we have chosen one of the numbers associated with each variable x_i. The number chosen reflect whether that variable is set to true or false in our assignment.
  - If this is done, the each of the low order t digits of the sum will equal the number of literals in each clause that are true. If all of these digits are greater than 0, we know we have a satisfying assignment.

- To complete the reduction to the subset sum problem, we need more than knowing that the lower order digits must all be greater than 0. We need to be able to predict their exact values. To accomplish this we add 3 numbers to our subset sum problem for each of the clauses in the original formula.
  - In each of the numbers for clause j, the $j + v$th digit of the number (i.e., c_j)will be 1.
  - All of the remaining c_k digits will be 0.
  - The value of s_j (i.e., the $v + t + j$th digit) will be 0 in one number, 1 in the second, and 2 in the third. All of the other s_k digits will be 0.

  Given that our sample formula has 4 clauses we would generate the numbers:

	x_1	x_2	x_3	c_1	c_2	c_3	c_4	s_1	s_2	s_3	s_4
clause 1	0	0	0	1	0	0	0	0	0	0	0
clause 1	0	0	0	1	0	0	0	1	0	0	0
clause 1	0	0	0	1	0	0	0	2	0	0	0
clause 2	0	0	0	0	1	0	0	0	0	0	0
clause 2	0	0	0	0	1	0	0	0	1	0	0
clause 2	0	0	0	0	1	0	0	0	2	0	0
clause 3	0	0	0	0	0	1	0	0	0	0	0
clause 3	0	0	0	0	0	1	0	0	0	1	0
clause 3	0	0	0	0	0	1	0	0	0	2	0
clause 4	0	0	0	0	0	0	1	0	0	0	0
clause 4	0	0	0	0	0	0	1	0	0	0	1
clause 4	0	0	0	0	0	0	1	0	0	0	2

- If we have a satisfying assignment of the formula corresponding to the subset sum problem we have generated, then the number of literals that are not true in any clause of the formula must be 0, 1, or 2. By selecting the corresponding number for each clause, we can therefore ensure that the sum we obtain will be a number consisting of $t + v$ 1s followed by t 3s.
- Thus, it should be clear that if there is a satisfying assignment, then the subset sum problem we have constructed has a solution.
- Conversely, if there is a solution to the subset sum problem, it must involve choosing either the number associated with x_i ore $\overline{x_i}$ for each i and the numbers chosen in this way must correspond to an assignment that satisfies the original boolean formula.