Problem Set 10: NP completeness and time complexity
CS254, Fall 2019, Anna Rafferty

Solution: Please do not post these solutions or share them with anyone outside this class. Thanks!

You are welcome to talk to others about the problems on this problem set, but you should **read and attempt each problem prior do discussing it with others.** The write up must be your own work. That means that **you should write your answers by yourself, without looking at notes from sessions with collaborators, and you must be able to explain any answer you write down.** You should list anyone you collaborated with in your collaboration form.

Due dates: 3 problems are due Monday, November 18; the remaining problems are due on Wednesday, November 20. As for all homeworks, if a problem is due on a specific day it must be turned in by 9PM on that day. See homework handout (a copy is posted on Moodle) for more specific homework policies. Failing to name your files properly may result in a loss of points and potentially receiving no credit for that problem; please name your files properly! **No late days may be used past the last day of classes**˙

1. Sipser 7.22

Solution: To show DOUBLE-SAT = $\{\langle\phi\rangle \mid \phi$ has at least two satisfying assignments$\}$ is NP complete, we need to show that it is in NP and that it's NP hard.

First, we show DOUBLE-SAT is in NP. We can build a polynomial time verifier $V(w, y)$ for DOUBLE-SAT that interprets the extra information y as a sequence of two truth assignments. DOUBLE-SAT checks that each truth assignment satisfies w. Since verifying that a given truth assignment satisfies a formula is computable in polynomial time, it is possible to do this for two truth assignments also in polynomial time.

Second, we'll show DOUBLE-SAT is NP-hard by showing that $SAT \leq_p$ *DOUBLE-SAT*. We define the polynomial time computable function as follows:

$f(\langle\phi\rangle)$:

- Create new variables b_1, b_2 not used in ϕ, and write $(\phi) \wedge (b_1 \vee b_2)$.

Clearly, this function is computable in polynomial time: it rewrites the original ϕ (with parentheses around it) and adds a constant amount of text after that. This will take linear time (move all the way to the end of the input, move everything over one square, add the parenthesis at the beginning, and then move back to the end to add the $) \wedge (b_1 \vee b_2)$).

Now, we prove that the reduction is correct:

- If $\langle\phi\rangle \in SAT$, then there exists a truth assignment A that satisfies ϕ. Then the truth assignment A' that is equal to A and sets $b_1 = T, b_2 = F$ satisfies $f(\langle\phi\rangle)$: $(\phi) \wedge (b_1 \vee b_2)$ evaluates to $T \vee T$, which is true. Additionally, the truth assignment A'' that is equal to A and sets $b_1 = T, b_2 = T$ satisfies $f(\langle\phi\rangle)$: $(\phi) \vee b$ evaluates to $T \vee F$, which is true. Thus, $f(\langle\phi\rangle) \in$ *DOUBLE-SAT*.
- If $f(\langle\phi\rangle) \in$ *DOUBLE-SAT*, then there exist at least two truth assignments A' and A'' that satisfy $f(\langle\phi\rangle)$. Consider the truth assignment A to ϕ that sets the truth values equal to the settings in A', omitting the assignments to b_1, b_2 because these do not exist in ϕ. Since A' satisfies $f(\langle\phi\rangle)$, it must satisfy both parts of the conjunction $(\phi) \wedge (b_1 \vee b_2)$: i.e., it satisfies ϕ and $(b_1 \vee b_2)$, since a conjunction is only true if both of its operands are true. Thus, A satisfies ϕ, meaning $\langle\phi\rangle \in SAT$.

Thus, $SAT \leq_p$ *DOUBLE-SAT*, which means that *DOUBLE-SAT* is NP-hard because *SAT* is NP-hard. Since *DOUBLE-SAT* is both in NP and NP-hard, *DOUBLE-SAT* is NP-complete.

2. Consider the problem of having a large collection of different books, and a number of tags that are associated with each book. For example, one tag might be "ancient history." Each tag implicitly represents a set of books; "ancient history" corresponds to all the books that have been tagged with this label. I have some number k shelves for displaying (some or all of) my books, and I'd like to choose k of the tags and put the books associated with

that tag on the given shelf. Since I physically can't place the same book on two shelves at once and I don't want someone to be confused by a shelf not having all the books associated with its tag, this means I'll need to choose k tags that have no books in common. We can formulate this as a decision problem for whether there exists a way to choose k tags that meet my constraints. Write down this decision problem as a language, and then show that this language is NP-complete. *Hint: You might try a reduction from one of our graph problems, either from Sipser or on a worksheet. If you're stuck, the optional reading may give you some ideas.*

Solution: We define the language as:

$$L = \{\langle B, T_1, \ldots, T_n, k\rangle \mid \text{each } T_i \subseteq B \text{ and } \exists S \subseteq \{1, \ldots, n\} \text{ where } |S| = k \text{ and } \forall i, j \in S \text{ where } i \neq j, S_i \cap S_j = \varnothing\}$$

Now we'll show it's NP complete. First, we know it is in NP because we can build a polynomial time verifier for it. The verifier $V(w, y)$ will interpret the extra information y as a choice for which k tags can be chosen to form the set S. The verifier first checks if y represents a string of k integers (each encoded in binary); if k is greater than the number of sets in w, it rejects. If y does have k integers, it goes through each integer in y (call it i), and checks it against every later integer in y (call it j). If any $i = j$ (i.e., two chosen are the same), the verifier rejects. Otherwise, it checks each item in S_i against each item in S_j. If any equal each other, the verifier rejects. If it reaches the end and all pairs i, j have been checked against each other with no items found in common, it accepts.

This verifier runs in polynomial time because we know the length of w is at least as long as k (since we had to write down all the sets), and then we check $O(k^2)$ pairs of sets. Within each set, we have at most $O(|B|)$ items, and we know $|B|$ is $O(n)$, where n is the length of the input, since we have B as part of our input. Checking items against one another takes polynomial time in the size of the product of the items in both lists. Since each operation is polynomial time, the entire verifier is polynomial time.

The verifier is also correct: $\langle B, T_1, \ldots, T_n, k\rangle \in L$ if and only if there's a way to choose k tags where no books overlap, and the verifier checks if y represents such a choice of k tags. If there is such a choice, then there's a y such that $V(w, y)$ returns true, and if no such choice exists, then there will be no y such that $V(w, y)$ returns true.

Next, we show that *IndependentSet* $\leq_p L$. This will complete our proof since *IndependentSet* is NP-complete. We define the polynomial reduction as follows:

$f(\langle G, k\rangle)$:

1. Construct $B = E$, and for each $v_i \in V$, construct $T_{v_i} = \{e \mid v_i \text{ is an endpoint of edge } e\}$. That is, each T_{v_i} represents the edges that are incident to vertex v_i.
2. Output $\langle B, T_{v_1}, \ldots, T_{v_{|V|}}, k\rangle$.

First, note that this reduction is polynomial time: it requires only looping over G to copy the edges and make B, and looping over the edges $|V|$ times to create all $S_{v_1}, \ldots S_{|V|}$.

Next, we must prove it correct. If $\langle G, k\rangle \in$ *IndependentSet*, then there exists an independent set $S \subseteq V$ such that $|S| = k$; we know that if any independent sets exists, there exists one of size k because we are guaranteed that the independent set is of size at least k, and a larger independent set can always be made into a smaller independent set by simply dropping vertices. We show that S is a solution for which T_i to choose in $\langle B, T_{v_1}, \ldots, T_{v_{|V|}}, k\rangle$. Since S is an independent set, no two vertices in it are connected by an edge. That means the sets $T_{S_1}, \ldots T_{S_k}$ must be disjoint, since each consists of the edges that are connected to the vertices $S_1, \ldots, S_k$. Thus, $\langle B, T_{v_1}, \ldots, T_{v_{|V|}}, k\rangle \in L$.

For the other direction, assume $\langle B, T_{v_1}, \ldots, T_{v_{|V|}}, k\rangle \in L$. Then there is a set S such that $|S| = k$ and for each $i, j \in S$, $T_i \cap T_j = \varnothing$. That means each corresponding vertex i and j in the original graph G must not share an edge, since each T_m is the edges incident to vertex m. Thus, S is an independent set in G of size k, so $\langle G, k\rangle \in$ *IndependentSet*.

We have shown that f takes polynomial time and that $\langle G, k\rangle \in$ *IndependentSet* iff $\langle B, T_{v_1}, \ldots, T_{v_{|V|}}, k\rangle \in L$. Thus, *IndependentSet* $\leq_p L$, so L is NP-hard; since we showed above it was in NP, we know L is NP-complete.

3. Sipser 7.26

Solution: a. Assume we have a $\neq$-assignment to a 3-cnf formula ϕ. For each clause, this assignment must assign at least one literal in the clause to true, as it must satisfy the ϕ; without loss of generality, call this literal for the ith clause $c_{i,t}$. Additionally, by the definition of $\neq$-assignment, for each clause it must assign at least one literal in the clause to false; without loss of generality, call this literal for the ith clause $c_{i,f}$. The negation of an assignment assigns all literals that were true to be false and all that were false to be true. Thus, in the negation, $c_{i,t} = false$ and $c_{i,f} = true$. Thus, each clause has two literals with different truth values ($c_{i,t}$ and $c_{i,f}$), and each clause has at least one literal set to true ($c_{i,f}$). Thus, the negated assignment satisfies ϕ and is also a $\neq$-assignment.

b. We want to show that 3SAT is polynomial time reducible to $\neq$SAT. On input ϕ to 3SAT, we construct ϕ' as follows. Assume ϕ has clauses $c_1, \ldots, c_m$. Then ϕ' has clauses $c'_1, c''_1, \ldots, c'_m, c''_m$. We construct each c'_i and c''_i as follows for $c_i = y_1 \vee y_2 \vee y_3$:

$$c'_i = y_1 \vee y_2 \vee z_i \tag{1}$$
$$c''_i = y_3 \vee b \vee \neg z_i \tag{2}$$

where z_i is a new variable for each clause, and b is an additional variable that appears in each of the double prime clauses.

We claim that $\phi \in 3SAT$ iff $\phi' \in\neq SAT$. First, assume that $\phi \in 3SAT$, and let A be a satisfying truth assignment for ϕ. We construct a $\neq$-truth assignment A' that satisfies ϕ'. For all variables in ϕ, A' sets them to the same value as in A. Additionally, A' has b set to false. Finally, for each z_i, z_i is set to true only in the case where both y_1 and y_2 are false. Otherwise, z_i is set to false. A' is a satisfying $\neq$ assignment for ϕ': Because A satisfies ϕ, at least one of y_1, y_2 or y_3 is set to true in every clause. We have two cases: 1) at least one of y_1 and y_2 is true, or 2) both y_1 and y_2 are false. Assume at least one of y_1 and y_2. Then c'_i is satisfied, and $z_i = false$. Thus, c'_i has two literals with unequal truth values: one of y_1 and y_2 is true, and $z_i = false$. Additionally, since $z_i = false$, $\neg z_i = true$. This satisfies c''_i. Additionally, since $b = false$, we have two literals with unequal truth values in c''_i. Thus, in case (1), A' is a satisfying $\neq$-assignment for ϕ'. In case two, we assume both y_1 and y_2 are false. Then $z_i = true$, so c'_i is satisfied and there are two literals with unequal truth values. Because both y_1 and y_2 are false and A was a satisfying truth assignment, y_3 must be true. Thus, c''_i is satisfied, and because $b = false$, we have two literals with unequal truth assignments. Thus, A' must be a satisfying $\neq$-truth assignment for ϕ'.

To complete our proof, we must show that if $\phi' \in\neq SAT$, then $\phi \in 3SAT$. Let A' be a satisfying $\neq$-assignment for ϕ' such that $b = false$. (This is without loss of generality by part a, as the negation of any A'' satisfying $\neq$-assignment where $b = true$ is also a satisfying $\neq$-assignment that meets our condition.) We construct a satisfying truth assignment A for ϕ by copying A' for all the variables shared by ϕ and ϕ'. We claim this satisfies ϕ. In A', for each z_i, either $z_i = true$ or $z_i = false$. If any $z_i = false$, then one of y_1 and y_2 for that clause must be true, and thus c_i is satisfied in ϕ. Consider the clauses where $z_i = true$. Then $\neg z_i = false$. This means $y_3 = true$, since we have constructed A' such that $b = false$. Thus, clause c_i is satisfied in ϕ. Thus, all clauses are satisfied and A is a satisfying truth assignment for ϕ.

Thus, we have shown $\phi \in 3SAT$ iff $\phi' \in\neq SAT$. This reduction requires polynomial time to compute as we need only add a constant number of things for each clause in the original formula.

c. We've shown that $3SAT \leq_p \neq SAT$, so to show $\neq SAT$ is NP complete, we need only additionally show that $\neq SAT$ is in NP. We build a nondeterministic TM that guesses every possible truth assignment for an input ϕ by branching non-deterministically on true and false as the value for each x_i, and then verifies that this truth assignment is actually a $\neq$-assignment (polynomial time to check that each clause has one true and one false) and that it satisfies ϕ (polynomial time as we've shown previously).

4. Consider a crossword-puzzle game to be: An $m \times n$ matrix (the board), where each entry is "*" (blank - a place where one could fill in a letter) or "#" (a blocked off spot in the board where no letter goes). This can give us our jagged crossword board in which we fill in the blanks. Further, we have an alphabet. Say, $\Sigma = \{a, b, c\}$. Finally we have a language $L \subset \Sigma^*$ that is a finite list of words.

   The crossword problem is then: given a board, can we fill in each "*" with a letter from the alphabet such that

   1. Every column (read from top to bottom) is a sequence of words from L, separated by one or more #s, and
   2. Every row (read from left to right) is a sequence of words from L, separated by one or more #s.

   Show that the crossword problem is NP-complete. *Note: There are multiple ways to solve this problem - you might try a reduction related to 3SAT, although that is not the only reduction that works. Hint: Think about using the columns to enforce one kind of constraint and the rows to enforce another. In 3SAT, the sorts of "constraints" you're enforcing are that each variable is assigned to True or False, and that at least one literal in each clause is*

True. While the hashmarks allow you to make any crossword puzzle you want, you don't actually have to use them at all when reducing 3SAT to CROSSWORD.

Solution:

There are multiple ways to solve this one. All ways involve showing the problem is in NP: verify it in polynomial time by interpreting the extra information to the verifier as a solution and checking that the solution doesn't have any letters where there are #s and that all words in the solution are in L. We can argue this is in polynomial time analogous to the argument for our Sudoku verifier from the example in class.

The choice is in what problem to do a reduction from. If you want to do a reduction from 3SAT, you might try to do something like the subset sum reduction in the book. Take an input ϕ for 3SAT. (Without loss of generality, we'll assume $m \neq n+1$ in the proof below: if they're equal, we could always add an extra variable that is never used in any clause.) We need to construct a language L that will have the valid words and a board. For each clause $c_1, \ldots, c_m$ in ϕ, we'll add up to 7 words to L, each of length $n+1$. For simplicity, we'll let the first n letters of each word be T, F, and U. The letters of each word will specify which variables are in the clause: If neither x_i nor $\neg x_i$ is in clause m, then spot i in each of the seven words for clause j will be equal to U ("unused"). For the three indices in each word that are not U (corresponding to the variables that are in the clause), we will have all possible combinations of Ts and Fs except the combination that would set all literals to false if all variables equal to F were false and all that were equal to T were true. For example, if the clause was $x_1 \vee x_2 \vee \neg x_3$, we would include all 8 possible combinations of Ts and Fs except for the one where 1 and 2 are F and 3 is T, as this would correspond to $x_1 = false$, $x_2 = false$, and $x_3 = true$ (i.e., $\neg x_3 = false$), where the clause would be unsatisfied. The last letter of each word will always be the clause number ($1 - -m$; we have Σ include sufficient characters for each clause to have it's own unique symbol). This gives us (up to) $7m$ words to add to L, and we can compute each in polynomial time. (If some variable appears in multiple literals in the clause, we may have fewer than 7 words per clause.)

The words we have just created will be fit in the across rows of the crossword. We also need to make the words to go downward. The downward words will be used to encode the fact that we want to make sure there's exactly one truth value assigned to each variable. These words will be m letters long, corresponding to whether the variable is true, false, or unused in each clause. We'll include two possible words related to each variable j: both words have spots i set to U where i is the index of a clause that doesn't contain the jth variable. One word has all the other spots sets to T; the other word has all the other spots set to F. Thus, we're ensuring that the variable has the same assignment in each clause. We finally add in a word that just has each clause number in order, to ensure that the words are used in the spots we intended. This is a total of $7m + 2n + 1$ words, which is linear in the length of ϕ and thus we can construct in polynomial time. The grid size that we'll use is number of clauses plus one by the number of variables. We construct this as an input C (containing the grid, the language, and the alphabet, which is $\{T, F, U, 1, \ldots, m\}$) for the crossword puzzle.

We just need to show that this construction works: $\phi \in 3SAT \iff C \in CROSSWORD$. First, assume $\phi \in 3SAT$. Let A be a satisfying assignment for ϕ. We can make a valid crossword puzzle by placing word w in the ith row that corresponds to the setting of the truth values for the ith clause (all Us where the variable is unused and some T/F for the variables). This word must be in L since we included exactly the words that would satisfy each clause. Additionally, the down words will also be in L, since they must have Us for each row that correspond to a clause that isn't used, and either all Ts or all Fs for the others (since A is a truth assignment). Thus, $C \in CROSSWORD$.

Second, assume $C \in CROSSWORD$. Let S be a solution for C. Since it is m tall, each down word must be one of the ones created to go down in the second paragraph above. For each down word, if it contains T or F, it only contains one of them (by construction). Construct a truth assignment A where $A(x_i) = true$ if the ith column contains T and false otherwise. This assignment satisfies ϕ: it must lead to at least one thing being true in each clause by the construction of the allowed across words.

Thus, CROSSWORD is NP complete since it is in NP and $3SAT \leq_p CROSSWORD$.

5. Sipser 8.11

Solution: We need to show that if every NP-hard language is also PSPACE hard, then PSPACE = NP. We know that $SAT \in PSPACE$ (can check in polynomial space by just going through all possible formulas). SAT is NP-complete, which means it is both in NP and NP-hard. Thus, if every NP-hard language is also PSPACE-hard, SAT is PSPACE-hard. Thus, for every $L \in PSPACE$, $L \leq_p SAT$. Since $SAT \in NP$, this implies that $L \in NP$: the nondeterministic TM that decides it in polynomial time first computes the polynomial time reduction function

(no branching for this part of the machine), and then functions like the nondeterministic machine that decides SAT. Thus, $PSPACE \subseteq NP$. We know from Sipser that $NP \subseteq PSPACE$, so thus, $NP = PSPACE$.

6. **Optional suggested problems:** You don't need to do these or turn them in, but if you want more practice on reductions 7.23 and 7.33 are good additional practice on polynomial time reductions.