

CS375 Weekly Assignment 10

Due May 7th 10PM ET

A general note for CS375: When writing up your homework, please write neatly and explain your answers clearly, giving all details needed to make your answers easy to understand. Graders may not award full credit to incomplete or illegible solutions.

You are encouraged to discuss the problems with your classmates, **but your submission should be your own**. When writing your solutions up, you should not be looking at any other student's submission.

Clear communication **is the point**, on every assignment. In general in CS375, unless explicitly specified otherwise, answers should be accompanied by explanations. Answers without explanations may not receive full credit. Please feel free to ask me any questions about explanations that might come up!

Necessary acknowledgements: many of these questions are reproductions or adaptations of questions of Eric Aaron's devising. Others are from our course texts as listed on the syllabus.

Recall the following problems and definitions from class:

SUBSETSUM

Input: A list of positive integers $x_1, \dots, x_n$ and a target T .

Output: TRUE if there exists a subset S of the x_i s such that $\sum_{x \text{ in } S} x = T$, otherwise FALSE.

VERTEXCOVER

Input: A graph G with parameter k .

Output: TRUE if there exists a subset S of k vertices such that every edge in the graph has an endpoint in S , otherwise FALSE.

3SAT

Input: A boolean formula that consists of the conjunction (AND, &&) of clauses, where each clause is the disjunction (OR, ||) of three boolean variables or their negations.

Output: TRUE if there exists an assignment of TRUE/FALSE to the boolean variables that makes the whole formula evaluate to TRUE, otherwise FALSE.

P = all problems that can be solved in polynomial time

NP = all problems whose solutions can be verified in polynomial time

NP-HARD = problems B that are so hard that $A \leq_p B$ for all A in **NP**

NP-COMPLETE = problems that are both **NP-HARD** and in **NP**

The Cook-Levin Theorem = 3SAT is **NP-HARD**

1. One might notice that I often loosely switch between the 'decision' version of problems and their optimization version. For example, with VERTEXCOVER, one could instead state an optimization version:

MINVERTEXCOVER

Input: A graph G

Output: A collection S of vertices of minimum size such that every edge in the graph has an endpoint in S

- (a) Suppose I have a polynomial time algorithm for MINVERTEXCOVER. Describe a polynomial time algorithm for VERTEXCOVER (**which takes a second parameter k**). This one should be easy.

Solution: If I have an algorithm A for MINVERTEXCOVER, then to write an algorithm for VERTEXCOVER I'll just run A on the given graph G : if the size of the cover that's given as an output is at most k , output TRUE; otherwise output FALSE. The cover returned by A is a *minimal* cover: so if its size is at most k , then obviously there's a cover of at most k vertices (the one A finds is such a cover!); if A returns a cover of size greater than k , then there's no smaller cover, so we can't find one of k vertices.

- (b) Suppose I have a polynomial time algorithm for VERTEXCOVER (**which takes a second parameter k**). Describe a polynomial time algorithm for MINVERTEXCOVER. This one is not as easy.

Solution: The nice thing about the previous part was that the solution was built for us and we just had to make a decision; now, the decision is made for us, and we need to build the solution. Much more annoying.

Let A be the algorithm given for VERTEXCOVER. The first step we'll need to do is determine the minimal size k^* : to do so, we'll use good ol' BINARYSEARCH (with an initial window of $[0, n]$) to determine the minimal vertex cover: this means making $\log n$ calls of A on G with varying values of k until we find the minimal value k^* such that $A(G, k^*)$ returns TRUE but $A(G, k^* - 1)$ returns FALSE.

Now that we know the right size for our vertex cover, we'll need to determine which vertices are the actual ones we need for our vertex cover of size k^* . To do so, consider the following: let v be a vertex of G . To cover all the edges of v , we must include v or *every other endpoint of every edge that v is incident to*. That is, for every vertex v , we must include it or *all of its neighbors*. Let G_v be the graph G with v and all of its incident edges removed; let $G_{N(v)}$ be the graph G with v , all of its neighbors, and all of their incident edges removed. Then, in order for $A(G, k)$ to return TRUE, it *must* be the case that $A(G_v, k - 1)$ or $A(G_{N(v)}, k - |N(v)|)$ ¹ returns TRUE. This induces the following algorithm:

```
output = []
k = k*
while k > 0:
    pick a vertex v in G
    remove v from G # along with its edges
    if A(G, k - 1):
        add v to output
        k -= 1
    else:
        add N(v) to output
        remove N(v) from G # along with their edges
        k -= |N(v)|
return output
```

Let $f(n, m)$ be the runtime of A on a graph of n vertices and m edges. Since we never make our graph bigger over time, I'm going to upper bound the runtime of every call to A by $f(n, m)$. To determine k^* , we start but making $O(\log n)$ calls (using BINARYSEARCH) to A . We then make at most $k^* \leq n$ more calls to A over the course of our while loop as we determine output. Over the

¹I'm using a bit of set notation here - it's common in mathematical writing to write $|S|$ to denote the size of a set S

course of determining OUTPUT, we remove vertices and edges from G ; I'm going to assume that each vertex / edge removed takes constant time. As a result, the total runtime

2. Find a student in an intro-ish CS Course (like 15X or 2XX) and explain to them what it means for a problem to be **NP-HARD**. Record what you said that got them to understand it. Obviously I have no way of verifying you did this, but fundamentally you don't understand something until you're capable of explaining it to someone else.

Solution: N/A

3. Let P be some problem that is **NP-COMPLETE**.

- (a) Explain why if P is in **P** then $\mathbf{P} = \mathbf{NP}$.

Solution: In class, we've pointed out that if $A \leq_p B$ and B has a polynomial time algorithm, then so does A . Note that for P to be **NP-COMPLETE**, we have that $A \leq_p P$ for every problem in **NP**. But if P is in **P**, then it has a polynomial time algorithm, which we can use our in-class observation to immediately give us that A has a polynomial time algorithm for every problem A in **NP**. But this means that every problem in **NP** is in **P**! Since we already knew from class that every problem in **P** is in **NP**, this means that $\mathbf{P} = \mathbf{NP}$.

- (b) Explain why if P is not in **P** then $\mathbf{P} \neq \mathbf{NP}$.

Solution: Since P is **NP-COMPLETE**, we know that P is in **NP**. But if it's not in **P** then there's a problem in **NP** that's not in **P**, showing that $\mathbf{P} \neq \mathbf{NP}$.

- (c) Suppose B is some other problem and we're able to show that

- B is in **NP**
- $P \leq_p B$

Explain why this means that B is **NP-COMPLETE**.

Solution: To be **NP-COMPLETE**, you need to be:

- in **P** (this was given for B)
- harder than all problems in **NP**: that is, every problem in **NP** needs to be reducible to you. This isn't immediately obvious for B , but we do know that $P \leq_p B$. Moreover, since P is **NP-COMPLETE**, we know that $A \leq_p P$ for all A in **NP**. But we've seen in class that $\leq_p$ is a transitive relation: that $X \leq_p Y$ and $Y \leq_p Z$ means that $X \leq_p Z$. Thus, since $A \leq_p P \leq_p B$, we get that $A \leq_p B$ for all problems A in **NP**, so B is **NP-HARD**!

Thus B is **NP-COMPLETE**.

4. Following the reduction from class from 3SAT to VERTEXCOVER and the following inputs for 3SAT:

$$F_1 : \quad (x_1 || x_2 || x_3) \&\& (x_1 || x_2 || \bar{x}_3) \&\& (x_1 || \bar{x}_2 || x_3) \\
&\& (x_1 || \bar{x}_2 || \bar{x}_3) \&\& (\bar{x}_1 || x_2 || x_3) \&\& (\bar{x}_1 || x_2 || \bar{x}_3) \\
&\& (\bar{x}_1 || \bar{x}_2 || \bar{x}_3)$$

$$F_2 : \quad (x_1 || x_2 || x_3) \&\& (x_1 || x_2 || \bar{x}_3) \&\& (x_1 || \bar{x}_2 || x_3) \\
&\& (x_1 || \bar{x}_2 || \bar{x}_3) \&\& (\bar{x}_1 || x_2 || x_3) \&\& (\bar{x}_1 || x_2 || \bar{x}_3) \\
&\& (\bar{x}_1 || \bar{x}_2 || x_3) \&\& (\bar{x}_1 || \bar{x}_2 || \bar{x}_3)$$

- (a) Find a satisfying assignment for F_1

Solution: The assignment $x_1 = \text{TRUE}$, $x_2 = \text{TRUE}$, $x_3 = \text{FALSE}$ guarantees that there's at least one literal evaluating to TRUE in each clause:

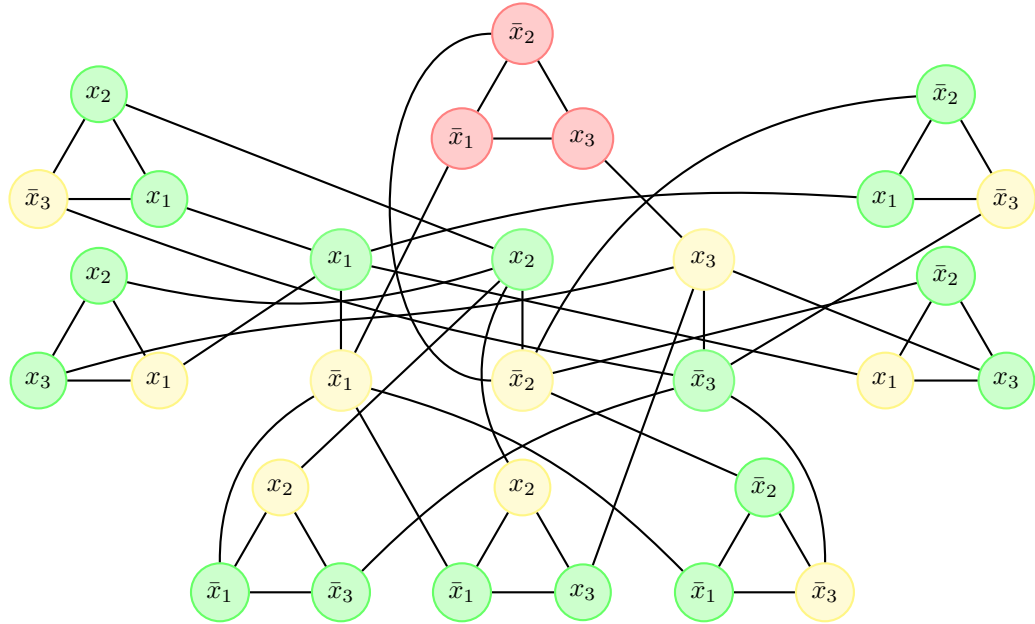
$$F_1 : \quad (x_1 || x_2 || x_3) \&\& (x_1 || x_2 || \bar{x}_3) \&\& (x_1 || \bar{x}_2 || x_3) \\
&\& (x_1 || \bar{x}_2 || \bar{x}_3) \&\& (\bar{x}_1 || x_2 || x_3) \&\& (\bar{x}_1 || x_2 || \bar{x}_3) \\
&\& (\bar{x}_1 || \bar{x}_2 || \bar{x}_3)$$

- (b) Explain why no assignment for the boolean variables of F_2 will make the formula evaluate to TRUE

Solution: First, observe that the setting from the previous part will not work here: the new clause will not have any TRUE literal in it. To see that no setting will work, notice that for 3 boolean variables there are exactly 8 clauses that we can form out of the positive and negative versions of each variable ($2^3 = 8$). So we have every clause in this formula: this means that for every setting, the clause corresponding to the opposite setting is in the formula: since this clause is the opposite setting, every literal will evaluate to FALSE, meaning that no setting over the variables can make the whole formula evaluate to TRUE.

- (c) Draw the construction of the equivalent input for VERTEXCOVER for F_1 : in a different color (or any other signifying way), include the final triangle of vertices representing the clause that is present in F_2 that isn't present in F_1 .

Solution:



- (d) Find a vertex cover of size $k = n + 2m = 3 + 2 * 7 = 17$ for the original graph (without the added clause).

Solution: You can confirm that I picked 17 green vertices in the graph above, and ignoring the red clause they form a vertex cover.

- (e) Explain why you can't pick an additional two vertices from the last triangle (the added clause) that would give a vertex cover of the whole graph (with the added clause).

Solution: Every red vertex is connected to a unique vertex I haven't picked - therefore I need to pick all 3 vertices to get a vertex cover.