CS 121 Homework 6: Fall 2020

Some policies: (See the course policy at `http://madhu.seas.harvard.edu/courses/Fall2020/policy.html` for the full policies.)

- **Collaboration:** You can collaborate with other students that are currently enrolled in this course (or, in the case of homework zero, planning to enroll in this course) in brainstorming and thinking through approaches to solutions but you should write the solutions on your own and cannot share them with other students.
- **Owning your solution:** Always make sure that you "own" your solutions to this other problem sets. That is, you should always first grapple with the problems on your own, and even if you participate in brainstorming sessions, make sure that you completely understand the ideas and details underlying the solution. This is in your interest as it ensures you have a solid understanding of the course material, and will help in the midterms and final. Getting 80% of the problem set questions right on your own will be much better to both your understanding than getting 100% of the questions through gathering hints from others without true understanding.
- **Serious violations:** Sharing questions or solutions with anyone outside this course, including posting on outside websites, is a violation of the honor code policy. Collaborating with anyone except students currently taking this course or using material from past years from this or other courses is a violation of the honor code policy.
- **Submission Format:** The submitted PDF should be typed and in the same format and pagination as ours. Please include the text of the problems and write **Solution X:** before your solution. Please mark in gradescope the pages where the solution to each question appears. Points will be deducted if you submit in a different format.
- **Late Day Policy:** To give students some flexibility to manage your schedule, you are allowed a net total of **eight** late days through the semester, but you may not take more than **two** late days on any single problem set. No exceptions to this policy.

By writing my name here I affirm that I am aware of all policies and abided by them while working on this problem set:

Your name: Todd Morrill

Collaborators: (List here names of anyone you discussed problems or ideas for solutions with)

No. of late days used on previous psets (not including Homework Zero): 1
No. of late days used after including this pset: 1

Special notes for this homework: For your convenience, here's a list of some problems you can assume are in **P**...

1. Linear Programming: given a set of linear inequalities (like $2x_1 - 3x_2 \leq 5$), is there a real solution to all of them?
2. 2-SAT: given a 2-CNF formula (an AND of clauses each an OR of at most 2 variables or their negations), is there an assignment of variables that makes it true?
3. 2-coloring: given a graph G, is there a way to color its vertices with 2 colors so that adjacent vertices have different colors?
4. Shortest Path: given a graph G, vertices s and t, and an integer k, is there a path of length at most k from s to t?
5. Min Cut: given a graph G and an integer k, is there a nonempty subset $S \subsetneq V$ with at most k edges from S to $\overline{S}$?

...and here's a list of some problems you can assume are $\mathbf{NP-complete}$:

1. Integer Programming: given a set of linear inequalities (like $2x_1 - 3x_2 \leq 5$), is there an integer solution to all of them?
2. 3-SAT: given a 3-CNF formula (an AND of clauses each an OR of at most 3 variables or their negations), is there an assignment of variables that makes it true?
3. 3-coloring: given a graph G, is there a way to color its vertices with 3 colors so that adjacent vertices have different colors?
4. Longest Path: given a graph G, vertices s and t, and an integer k, is there a path of length at least k from s to t? (A path is a sequence of *distinct* vertices with each consecutive pair adjacent.)
5. Max Cut: given a graph G and an integer k, is there a nonempty subset $S \subsetneq V$ with at least k edges from S to $\overline{S}$?

Questions

Please solve the following problems. Some of these might be harder than the others, so don't despair if they require more time to think or you can't do them all. Just do your best. Also, you should only attempt the bonus questions if you have the time to do so. If you don't have a proof for a certain statement, be upfront about it. You can always explain clearly what you are able to prove and the point at which you were stuck. You can always simply write **"I don't know"** and you will get 15 percent of the credit for the problem. If you are stuck on this problem set, you can use Piazza to send a private message to all staff.

Note on reading the textbook: If you are stuck on some of the problems, try consulting the book to 1) understand the concepts the question is referencing, and 2) review the way similar theorems are proved in the book.

Problem 0 (5 points): True or False: I have completed the midterm 2 feedback survey. (True worth 5 points, False, or I don't know worth 0 points.)

Solution 0: True

Problem 1.1 (10 points): Suppose that you are in charge of scheduling courses in computer science in University S. In University S, computer science students wake up late, and have to work on their startups in the afternoon, and take long weekends with their investors. So you only have two possible slots: you can schedule a course either Monday-Wednesday 11am-1pm or Tuesday-Thursday 11am-1pm.

Let $SCHEDULE_S : \{0,1\}^* \rightarrow \{0,1\}$ be the function that takes as input a list of courses L and a list of *conflicts* C (i.e., list of pairs of courses that cannot share the same time slot) and outputs 1 if and only if there is a "conflict free" scheduling of the courses in L to the two slots, where no pair in C is scheduled in the same time slot. (We model the course list L as just a list of strings, with the conflict lists C as a list of pairs of strings.)

Prove that $SCHEDULE_S \in \mathbf{P}$. As usual, you do not have to provide the full code to show that this is the case, and can describe operations as a high level, as well as appeal to any data structures or other results mentioned in the book or in lecture. Note that to show that a function F is in $\mathbf{P}$ you need to **(1)** present an algorithm A, **(2)** prove that A computes F in polynomial time, and **(3)** prove that A runs in polynomial time. See footnote for hint.[1]

Solution 1.1:

We can define an algorithm A that computes $SCHEDULE_S$ because it can be reduced to the decision version of the graph 2-coloring problem. We let $G = (L, C)$, where the list of courses L corresponds to the vertices of the graph and the conflicts C correspond to the edges of the graph. This transformation can be done in time $O(n + m)$ where n is the number of courses in L and m is the number of conflicts in C.

Once the input to A is converted to G, we can simply call the procedure which computes the 2-coloring function, which we know is in $\mathbf{P}$ as specified in the instructions to the problem set. NB: if a course (i.e. a node) has no conflicts (i.e. no edges) then it can be arbitrarily assigned to the schedule (i.e. a color).

We now show that $SCHEDULE_S(L, C) = 1$ iff $A(L, C) = 1$.

Completeness: If there exists a conflict-free schedule then the corresponding graph is 2-colorable. Course conflicts correspond to edges in the graph. For any arbitrary conflict, one course will be assigned Monday/Wednesday and the other will be assigned Tuesday/Thursday. But this means that nodes on either side of the corresponding edge can be assigned different colors such as red and blue.

Soundness: If the graph is 2-colorable then there exists a conflict-free schedule. Since the graph is 2-colored, nodes on each end of the edge will have different colors. These edges correspond to course conflicts. Since the nodes have different colors, the corresponding courses can be assigned to different days to create a conflict-free schedule.

Since the transformation step and the 2-coloring procedure are both computable in polynomial time and algorithm A correctly computes $SCHEDULE_S$, we conclude that $SCHEDULE_S \in \mathbf{P}$.

[1]Try to see how you model the setting mathematically, and whether it amounts to questions about objects we have looked at before.

Problem 1.2 (10 points): Consider the same question but now at university H where there is a third course slot on Monday-Wednesday from 11pm till 1am. Let $SCHEDULE_H : \{0,1\}^* \to \{0,1\}$ be the function as above that takes as input a list of courses L and a list of *conflicts* C and outputs 1 if and only if there is a "conflict free" scheduling of the courses in L to the three slots of H. Prove that $SCHEDULE_H$ is **NP**-complete.

Solution 1.2:

To show that $SCHEDULE_H$ is **NP**-complete, we must show that $SCHEDULE_H \in$ **NP** and $SCHEDULE_H \in$ **NP**-hard.

$SCHEDULE_H \in$ **NP**

Let $V : \{0,1\}^* \to \{0,1\}$ be a verification function that accepts $\rho = (L, C)$, which is an instance of the input to $SCHEDULE_H$ and x, which is a set of 3 sets - 1 for each scheduling slot, where each $l \in L$ is assigned to exactly 1 of the 3 sets in x. $V(\rho, x) = 1$ iff $SCHEDULE_H(L, C) = 1$. The procedure to compute V is as follows.

```
def compute_v(rho, x):
    # check that all courses have been scheduled
    for l in L:
        if l not in x[0], x[1], or x[2]:
            return 0

    # check that all conflicts are satisfied
    for a, b in C:
        for set in x:
            if a in set and b in set:
                return 0

    # passed verification check
    return 1
```

compute_v runs in time $O(|L|+|C|)$, which is polynomial. From this we conclude that $SCHEDULE_H \in$ **NP**.

$SCHEDULE_H \in$ **NP**-hard: $3-COLORING \leq_p SCHEDULE_H$

The decision version of the function $3-COLORING$ accepts a graph $G = (V, E)$ and returns 1 iff the graph's vertices can be colored using at most 3 colors such that no two neighboring vertices are the same color. The problem set instructions tell us that $3-COLORING$ is **NP**-complete, which tells us that $3-COLORING$ is **NP**-hard. Showing that $3-COLORING$ can be reduced to $SCHEDULE_H$ will allow us to conclude that $SCHEDULE_H$ is **NP**-hard.

$3-COLORING$ can be reduced to $SCHEDULE_H$. In particular, the set of vertices V can be mapped to courses L and the set of edges E can be mapped to course conflicts C. The 3 node colors correspond to the 3 time slots for courses.

This translation, which we denote as R, from an input to $3-COLORING$ to an input to $SCHEDULE_H$ takes $O(|V| + |E|)$ time, which is polynomial time.

We now prove that $3-COLORING(G) = SCHEDULE_H(R(G))$.

Completeness: if the graph G is 3-colorable, then there exists a conflict-free schedule with 3 time slots. Since the graph is 3-colored, all neighboring nodes are assigned different colors. Since edges correspond to conflicts and nodes correspond to courses, each course will be scheduled to a time slot according to its node's color and all conflicts will be satisfied.

Soundness: if there exists a conflict-free schedule with 3 time slots, then the graph is 3-colorable. Since the schedule is conflict-free, all conflicts in C are satisfied and each course in the pair is scheduled to a different time slot. Since each time slot corresponds to a color and conflicts correspond to edges in the graph, each neighboring node will be assigned a different color and will satisfy the 3-coloring condition.

From this we conclude that $SCHEDULE_H \in$ **NP**-hard. This shows that $SCHEDULE_H$ is **NP**-complete, which concludes the proof.

Problem 2 Recall that **P**, **NP**, and other complexity classes are sets of *Boolean* functions $f : \{0,1\}^* \to \{0,1\}$; for instance, 3SAT returns 1 iff there exists a satisfying assignment to its input formula. In this problem, we want a polynomial-time algorithm to *find* a satisfying assignment.

Problem 2.1 (5 points) If $\phi(x_0, x_1, \ldots, x_{n-1})$ is a 3-CNF formula and $i \in \{0,1\}$, let $\phi_i(x_1, \ldots, x_{n-1}) = \phi(i, x_1, \ldots, x_{n-1})$. Show that, given $3SAT(\phi_0)$ and $3SAT(\phi_1)$, you can calculate $3SAT(\phi)$.

Solution 2.1

There are 2 mutually exclusive and collectively exhaustive cases that allow us to calculate $3SAT(\phi)$.

Case 1: $((3SAT(\phi_0) = 1) \vee (3SAT(\phi_1) = 1)) \implies 3SAT(\phi) = 1$

Case 2: $((3SAT(\phi_0) = 0) \wedge (3SAT(\phi_1) = 0)) \implies 3SAT(\phi) = 0$

This shows that given $3SAT(\phi_0)$ and $3SAT(\phi_1)$, we can calculate $3SAT(\phi)$.

Problem 2.2 (15 points) Suppose you have a polynomial-time algorithm M that computes 3SAT. Give a polynomial-time algorithm M' that, on input a 3-CNF formula $\phi(x_0, x_1, \ldots, x_{n-1})$, returns a satisfying assignment if one exists, or 0 if none does.

Solution 2.2

Building off of **solution 2.1**, we can hardcode partial satisfying assignments to build up the correct solution one bit at a time. In particular, we want to build up a satisfying solution x such that $\phi(x) = 1$. The approach is as follows.

```
def embed_z(phi, z):
    """Check if z is a valid starting sequence for the solution x"""
    phi_z = phi with the first |z| bits of solution x set to z
    return M(phi_z)

def M_prime(phi):
    """Find solution x"""
    # build up solution 1 bit at a time
    z = ''
    n = number of variables in phi expression
    for _ in len(n):
        solution_0 = embed_z(phi, z + '0')
        solution_1 = embed_z(phi, z + '1')
        if solution_0 == solution_1 == 0:
            return 0
        elif solution_0 == 1:
            z = z + '0'
        else:
            z = z + '1'
    return z
```

Correctness: As demonstrated, we test partial solutions to x one bit at a time, checking if 0 or 1 can be appended to the partial solution z. We hard code ϕ with the first $|z|$ bits of x set to z. We use M to determine if adding a 0 or 1 to this partial solution z will result in a valid partial solution. We continue in this manner for n steps where $n = |x|$. If the loop finishes, then M' returns a valid solution for x. If $(M(\phi_0) = 0) \wedge (M(\phi_1) = 0)$ on the first iteration of the loop, then M' returns 0 because no satisfying assignment exists.

Running time: This solution has one main loop that executes $n = |x|$ times and on each iteration of the loop calls M, which is given to be a polynomial-time algorithm. Together this yields a polynomial-time algorithm M'.

This shows that there exists a polynomial-time algorithm M' that, on input a 3-CNF formula $\phi(x_0, x_1, \ldots, x_{n-1})$, returns a satisfying assignment if one exists, or 0 if none does.

Problem 2.3 (Bonus, 0 points (hard)) Give an algorithm M' with the following property: If it is true that 3SAT $\in P$, then, on input a satisfiable 3-CNF formula $\phi(x_0, x_1, \ldots, x_{n-1})$, M' returns a satisfying assignment in polynomial time. (If no satisfying assignment to its input formula exists, M' may not halt in polynomial time.)

Problem 2.4 (25 points) Suppose you have a polynomial-time algorithm M that computes Longest Path. Give a polynomial-time algorithm M' that, on input a graph G, vertices s and t, and an integer k, returns a path of length at least k from s to t if one exists, or 0 if none does.

Solution 2.4

The proof closely follows the pattern of **solution 2.2**, by building up a satisfying path 1 node at a time. The algorithm M' is as follows.

```
def M_prime(G, s, t, k):
    """
    Find path between s and t in graph G with length greater than or equal to k
    """
    # check if there exists a solution
    if not M(G, s, t, k):
        return 0

    # build up solution 1 node at a time
    k_working = k
    path = [s]
    while s != t:
        for u in neighbors of s:
            k_temp = max(k_working - 1, 0)
            if M(G, u, t, k_temp):
                path.append(u)
                k_working = k_temp
                s = u
                break
    return path
```

Correctness: The correctness of M' heavily relies on the correctness of M. Since we assume M to be correct, we can quickly determine if there exists a solution by simply checking if $M(G, s, t, k) = 1$. If $M(G, s, t, k) = 0$, then no such path exists and we simply return 0 as specified in the problem statement. If a path satisfying the inputs does exist, we build up a solution 1 node at a time, relying on the correctness of M to determine if a particular node is a valid addition to the path. At each iteration of the while loop, we reduce the the number k by 1 to reflect the fact that we are 1 node closer to finding a valid path. The path may be longer than k so we prevent k from being negative with max(k_working - 1, 0). The loop terminates when $s = t$ (i.e. when a complete path has been found) and M' returns a satisfying path.

Running time: This solution has one main while loop that executes up to $|E|$ times, where E is the set of edges in the graph, and one nested for loop that executes up to $|E|$ times. This results in $O(|E|^2)$ invocations of M, which is given to be a polynomial-time algorithm. Together this yields a polynomial-time algorithm M'.

This shows that there exists a polynomial-time algorithm M' that, on input a graph G, vertices s and t, and an integer k, returns a path of length at least k from s to t if one exists, or 0 if none does.

Question 3.1 (15 points): k-1S-POSITIVE-3SAT is the following function: the input is a CNF formula φ where each clause is the OR of one to three variables (*without negations*), and a number $k \in \mathbb{N}$. For example, the following is a valid input: $(\varphi = (x_5 \vee x_2 \vee x_1) \wedge (x_1 \vee x_3) \wedge (x_2 \vee x_4 \vee x_0), k = 2)$. The output is 1 if and only if there exists a satisfying assignment to φ in which exactly k of the variables get the value 1. For example, for the formula φ above, k-1S-POSITIVE-3SAT$(\varphi, 2) = 1$ since the assignment $(1, 1, 0, 0, 0, 0)$ satisfies all the clauses. However k-1S-POSITIVE-3SAT$(\varphi, 1) = 0$ since there is no single variable appearing in all clauses.

Prove that k-1S-POSITIVE-3SAT is **NP**-complete. See footnote for hint.[2]

Solution 3.1: To show that k-1S-POSITIVE-3SAT is **NP**-complete, we must show that k-1S-POSITIVE-3SAT $\in$ **NP** and k-1S-POSITIVE-3SAT $\in$ **NP**-hard.

k-1S-POSITIVE-3SAT $\in$ **NP**

Let $V : \{0,1\}^* \to \{0,1\}$ be a verification function that accepts $\rho = (\varphi, k)$, which consists of a CNF formula φ, $k \in \mathbb{N}$, and an assignment x to the variables in φ. $V(\rho, x) = 1$ iff k-1S-POSITIVE-3SAT$(\varphi, k) = 1$. The procedure to compute V is as follows.

```
def compute_v(rho, x):
    phi, k = rho

    # check that x contains exactly k 1s
    if k != sum(x):
        return 0

    for clause in phi:
        clause_sum = 0
        for literal in clause:
            # retrieve the assignment to the literal
            temp = x[literal]
            clause_sum += temp

        # check if the clause was satisfied
        if clause_sum == 0:
            return 0

    # passed verification check
    return 1
```

compute_v runs in time $O(m)$, where m is the number of clauses in φ, which runs in polynomial-time. From this we conclude that k-1S-POSITIVE-3SAT $\in$ **NP**.

[2]**Hint:** You can use a direct reduction from $3SAT$ or a reduction from a function that was shown to be **NP**-complete in either the textbook, lecture, or section. If you go via the direct reduction route, you might want to try to transform a SAT instance on n variables $x_0, \ldots, x_{n-1}$ to a k-1S-POSITIVE-3SAT instance on $2n$ variables $w_0, \ldots, w_{2n}$ so that for every $i \in [n]$, the variable w_{2i} will correspond to x_i and the variable w_{2i+1} will correspond to $\neg x_i$. You can try to find ways to constrain a weight n solution so that exactly one of the variables w_{2i} and w_{2i+1} will equal 1 for every $i \in [n]$.

k-1s-Positive-3SAT $\in$ **NP**-hard: $3SAT \leq_p k$-1s-Positive-3SAT

We know that $3SAT$ is **NP**-hard. Showing that $3SAT$ can be reduced to k-1s-Positive-3SAT will allow us to conclude that k-1s-Positive-3SAT is **NP**-hard.

Variables $x_0 \cdots x_{n-1}$ in the original 3SAT instance correspond to variables $w_0 \cdots w_{2n-1}$ in the k-1s-Positive-3SAT setting, where x_i corresponds to w_{2i} and $\overline{x}_i$ corresponds to w_{2i+1}, for all n variables.

For the reduction, we simply translate x_i to w_{2i} or $\overline{x}_i$ to w_{2i+1} in all the clauses found in original $3SAT$ instance. We set $k = n = |x|$ because a satisfying solution in the k-1s-Positive-3SAT setting should contain exactly n 1s. To enforce the constraint that exactly 1 of each pair w_{2i}, w_{2i+1} must equal 1, we add n additional clauses of form $(w_{2i} \vee w_{2i+1})$ for all $i \in n$.

This reduction R from an input to $3SAT$ to an input to k-1s-Positive-3SAT takes $O(m+n)$ time, where we must replace the variables in the m $3SAT$ clauses and add an additional n clauses of the form $(w_{2i} \vee w_{2i+1})$. This reduction runs in polynomial time.

We now prove that $3SAT(\varphi) =k$-1s-Positive-3SAT$(R(\varphi))$.

Completeness: If $3SAT(\varphi) = 1$, then k-1s-Positive-3SAT$(R(\varphi)) = 1$. If there exists a satisfying assignment for φ then each clause must have been satisfied and so by simple translation of variables the first m clauses of $(R(\varphi))$ will be satisfied. Since x_i in φ is either 0 or 1 (and $\overline{x}$ takes the opposite value) exactly 1 of w_{2i}, w_{2i+1} will equal 1 for all $i \in n$, hence the choice of $k = n = |x|$. This also satisfies the n remaining clauses of the form $(w_{2i} \vee w_{2i+1})$.

Soundness: If k-1s-Positive-3SAT$(R(\varphi)) = 1$, then $3SAT(\varphi) = 1$. We set $k = n$ where n is the number of variables in φ. The additional n clauses of the form $(w_{2i} \vee w_{2i+1})$ constrain solutions to those where exactly 1 of w_{2i}, w_{2i+1} equal 1 for all $i \in n$, hence, there will be no contradictions such as $x_i = \overline{x}_i = 1$ when translating $R(\varphi)$ back to φ. Since the first m clauses of $R(\varphi)$ are satisfiable, a simple translation of the w variables back x variables will satisfy the m clauses of φ.

From this we conclude that k-1s-Positive-3SAT$\in$ **NP**-hard. This shows that k-1s-Positive-3SAT is **NP**-complete, which concludes the proof.

Question 3.2 (15 points): In the *employee recruiting problem* we are given a list of potential employees, each of which has some subset of m potential skills, and a number k. We want a team of k employees such that for every skill there is at least one member of the team with it.

For example, if Alice has the skills "C programming", "NAND programming" and "Solving Differential Equations", Bob has the skills "C programming" and "Solving Differential Equations", and Charlie has the skills "NAND programming" and "Coffee Brewing", then if we want a team of $k = 2$ people that covers all $n = 4$ skills, we could hire Alice and Charlie.

Define the function EMP s.t. on input the skills L of all potential employees (in the form of a sequence L of n lists $L_1, \ldots, L_n$, each containing distinct numbers between 0 and m), and a number k, $EMP(L, k) = 1$ if and only if there is a subset S of k potential employees such that for every skill j in $[m]$, there is at least one employee in S that has the skill j.

Prove that EMP is **NP**-complete. You can use the result of the previous subquestion even if you didn't prove it. See also footnote.[3]

Solution 3.2: To show that EMP is **NP**-complete, we must show that $EMP \in$ **NP** and $EMP \in$ **NP**-hard.

$EMP \in$ **NP**

Let $V : \{0,1\}^* \rightarrow \{0,1\}$ be a verification function that accepts $\rho = (L, k)$, which consists of a sequence L of n lists $L_1, \ldots, L_n$ each containing distinct numbers between 0 and m, and $k \in \mathbb{N}$, and a subset S of k employees. $V(\rho, S) = 1$ iff $EMP(L, k) = 1$. The procedure to compute V is as follows.

```
def compute_v(rho, S):
    L, k = rho

    # check that S contains exactly k employees
    if k != |S|:
        return 0

    # compute complete skill set
    complete_skill_set = {}
    for skill_set in L:
        complete_skill_set = complete_skill_set.union(skill_set)

    # compute employee skill set
    employee_skill_set = {}
    for employee in S:
        # retrieve skills for employee
        temp = L[employee]
        employee_skill_set = employee_skill_set.union(temp)

    # check that all skills are covered
```

[3] You can assume that lists (some of which could potentially be empty) are represented as strings in some reasonable way, and for every skill j there is some employee with the skill j, and so in particular both n and m are smaller than the length of the input as a string of bits.

```
    if complete_skill_set != employee_skill_set:
        return 0

    # passed verification check
    return 1
```

compute_v runs in time $O(n+k) = O(n)$, where n is the number of candidate employees, k is the number of employees selected and $k \leq n$. This runs in polynomial-time. From this we conclude that $EMP \in$ **NP**.

$EMP \in$ **NP**-hard: k-1s-Positive-3SAT $\leq_p EMP$

We know that k-1s-Positive-3SAT is **NP**-hard by **solution 3.1**. Showing that k-1s-Positive-3SAT can be reduced to EMP will allow us to conclude that EMP is **NP**-hard.

Variables $x_0 \cdots x_{n-1}$ in the original k-1s-Positive-3SAT instance correspond to the n candidate employees in the EMP setting, and the m clauses in k-1s-Positive-3SAT correspond to the m skills employees may have. In particular, we number the CNF clauses $0 \cdots m$ and if an employee (i.e. x_i) is a member of CNF clause m_j, then they have skill m_j. The k ones in satisfying assignments to CNF formula φ correspond to the k employees needed to cover all the m skills.

For the reduction, we simply assign a number to each of the m clauses and then for each variable x_i in the k-1s-Positive-3SAT we create a list L_i that corresponds to the skills that employee i has, where the skills correspond to the clauses that employee i was a part of. k in the original problem simply maps to k in the reduced problem.

This reduction R from an input to k-1s-Positive-3SAT to an input to EMP takes $O(m+n)$ time, where we must iterate over the m CNF clauses and create n employee skills lists. This reduction runs in polynomial time.

We now prove that k-1s-Positive-3SAT$(\varphi) = EMP(R(\varphi))$.

Completeness: If k-1s-Positive-3SAT$(\varphi, k) = 1$, then $EMP(R(\varphi, k)) = 1$. If there exists a satisfying assignment x for φ then each clause must have been satisfied by the k ones in x. This means that there are k employees that collectively possess all the m skills.

Soundness: If $EMP(R(\varphi, k)) = 1$, then k-1s-Positive-3SAT$(\varphi, k) = 1$. We select k L_i employee skill lists that correspond to the k x_i variables that are set to 1 in the satisfying solution x. Since these k employees collectively possess all m skills, this means the k variables x_i satisfy all m clauses of φ.

From this we conclude that $EMP \in$ **NP**-hard. This shows that EMP is **NP**-complete, which concludes the proof.

Problem 4 (20 points): Prove that the "balanced variant" BMC of the maximum cut problem is **NP**-complete: given a graph G and an integer k, $BMC(G,k) = 1$ iff there is a subset $S \subsetneq V$ of size $|V|/2$ with at least k edges from S to $\overline{S}$.

Solution 4:

To show that BMC is **NP**-complete, we must show that $BMC \in$ **NP** and $BMC \in$ **NP**-hard.

$BMC \in$ **NP**

Let $V' : \{0,1\}^* \to \{0,1\}$ be a verification function that accepts $\rho = (G,k)$, which consists of a graph G and $k \in \mathbb{N}$, and a subset S of V. $V'(\rho, S) = 1$ iff $BMC(G,k) = 1$. The procedure to compute V' is as follows.

```
def compute_v_prime(rho, S):
    G, k = rho
    V, E = G

    # check that S contains exactly |V|/2 nodes
    if |S| != |V|/2:
        return 0

    # check that there are at least k edges crossing the cut
    k_check = 0
    for edge in E:
        u, v = edge
        if (u in S) and (v in V-S):
            k_check += 1
        elif (v in S) and (u in V-S):
            k_check += 1

    if k_check < k:
        return 0

    # passed verification check
    return 1
```

compute_v_prime runs in time $O(|E|)$ to check that there are at least k edges crossing the cut, which is polynomial-time. From this we conclude that $BMC \in$ **NP**.

$BMC \in$ **NP**-hard: $MAXCUT \leq_p BMC$

The problem set instructions tell us that $MAXCUT$ is **NP**-hard. Showing that $MAXCUT$ can be reduced to BMC will allow us to conclude that BMC is **NP**-hard.

For the reduction we can add $|V|$ nodes to the original graph G to achieve a balanced max cut. This results in a new graph G' with $|V'| = 2|V|$ nodes. No edges are modified. k in the original problem simply maps to k in the reduced problem.

This reduction R from an input to $MAXCUT$ to an input to BMC takes $O(|V|)$ time to simply add $|V|$ nodes. This reduction runs in polynomial time.

We now prove that $MAXCUT(G, k) = BMC(R(G, k))$.

Completeness: If $MAXCUT(G, k) = 1$, then $BMC(R(G, k)) = 1$. If there exists a satisfying subset S then this means there are at least k edges crossing the cut in G. In particular, $|V| - |S|$ nodes are added to S to create S'. The remainder of the $|V|$ nodes are added to $\overline{S}$. This guarantees that $|S'| = |V'|/2$, where S' is a satisfying subset in the BMC setting. Since the set of edges is unchanged, the number of edges crossing the cut is still at least k.

Soundness: If $BMC(R(G, k)) = 1$, then $MAXCUT(G, k) = 1$. Since S' is a satisfying subset for BMC with at least k edges crossing the cut, we can simply remove the $|V'|/2$ nodes that were added to achieve a subset S. Since no edges were modified, this subset S will have at least k edges crossing the cut in the $MAXCUT$ setting.

From this we conclude that $BMC \in$ **NP**-hard. This shows that BMC is **NP**-complete, which concludes the proof.

Problem 5.1 (20 points): The input to the CubicSAT function is m polynomials $P_0, \ldots, P_{m-1}$ on n variables $x_0, \ldots, x_{n-1}$. Each polynomial is a cubic equation in the variables, i.e., the degree of every monomial is at most 3. Also, the coefficients are integers in the range $\{-n, \ldots, n\}$. (Each such polynomial can be expressed as a list of $O(n^3)$ integers - why?). CubicSAT$(P_0, \ldots, P_{m-1}) = 1$ iff there exist integers $a_0, \ldots, a_{n-1}$ such that for every $j \in [m]$, $P_j(a_0, \ldots, a_{n-1}) = 0$. Prove that CubicSAT is NP-hard.

Solution 5.1

To show that CubicSAT is **NP**-complete, we must show that CubicSAT $\in$ **NP** and CubicSAT $\in$ **NP**-hard.

CubicSAT $\in$ **NP**

Let $V : \{0,1\}^* \to \{0,1\}$ be a verification function that accepts $\rho = P_0, \ldots, P_{m-1}$, which consists of m polynomials on n variables, and a sequence of integers $a = a_0, \ldots, a_{n-1}$. $V(\rho, a) = 1$ iff CubicSAT$(P_0, \ldots, P_{m-1}) = 1$. The procedure to compute V is as follows.

```
def compute_v(rho, a):
    n = number of unique vars in rho

    # check that a has length n
    if len(a) != n:
        return 0

    # check that each polynomial is satisfied
    for p in rho:
        if p(a) != 0:
            return 0

    # passed verification check
    return 1
```

compute_v runs in time $O(m)$, where m is the number of polynomials. This runs in polynomial-time. From this we conclude that CubicSAT $\in$ **NP**.

CubicSAT $\in$ **NP**-hard: $3SAT \leq_p$ CubicSAT

We know that $3SAT$ is **NP**-hard. Showing that $3SAT$ can be reduced to CubicSAT will allow us to conclude that CubicSAT is **NP**-hard.

For the reduction we can translate CNF clauses to polynomials. There are n variables $y_0 \ldots y_{n-1}$ in the $3SAT$ setting, which correspond to the n variables $x_0 \ldots x_{n-1}$ in the CubicSAT setting. Similarly, the m clauses in the CNF formula φ are mapped to the m polynomials. Each positive literal y_i in clause m_j in the $3SAT$ setting will be mapped to $(x_i - 1)$ in the m_j^{th} polynomial in the CubicSAT setting. Similarly, negative literals $\overline{y}_i$ in clause m_j will be mapped to x_i in the m_j^{th} polynomial in the CubicSAT setting. An example translation looks like the following: $(y_1 \vee y_1 \vee \overline{y}_3) \to (x_1 - 1)(x_1 - 1)(x_3) = x_3 x_1^2 - 2x_3 x_1 - x_3$

This reduction R from an input to $3SAT$ to an input to CubicSAT takes $O(m)$ time, where m is the number clauses in φ. This reduction runs in polynomial time.

We now prove that $3SAT(\varphi) = \text{CubicSAT}(R(\varphi))$.

Completeness: If $3SAT(\varphi) = 1$, then $\text{CubicSAT}(R(\varphi)) = 1$. Since y is a satsifying assignment to φ, this means that every clause m_j had at least one literal that equals 1. This means that at least one of the roots of the polynomial m_j will be zero. For example, if $y_i = 1$, then $x_i = 1$ and in turn $(x_i - 1) = 0$. This construction means that all m polynomials will evaluate to 0. The final checks are 1) for the constraint that the resulting polynomials are cubic equations and 2) for the constraint that coefficients of the polynomials are in the range $\{-n, \dots, n\}$. For all $n \geq 3$, this will be the case. CNF clauses of the form $(y_i \vee y_i \vee y_i)$ yield coefficients and degrees of the greatest magnitude (i.e. $(x_i - 1)(x_i - 1)(x_i - 1) = x^3 - 3x^2 - 3x - 1$). This shows that that all monomials will have degree and coefficients at most 3. There are only a small number of $3SAT$ instances with non-redundant clauses with $n < 3$ where this does not hold.

Soundness: If $\text{CubicSAT}(R(\varphi)) = 1$, then $3SAT(\varphi) = 1$. Since there exists a solution a on n variables that satisfies all the m cubic polynomial equations, we know that we can unpack the roots of these equations into m CNF clauses (max 3 variables per clause because equations are cubic) on n variables. In particular, we map roots of the form $(x_i - 1)$ in the m_j^{th} polynomial to y_i in the m_j^{th} clause of φ and similarly, we map (x_i) to $\overline{y}_i$. This shows that all m clauses of φ will be satisfied.

From this we conclude that CubicSAT $\in$ **NP**-hard. This shows that CubicSAT is **NP**-complete, which concludes the proof.