CS 170 HW 11

Due **2020-11-16, at 10:00 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write none.

In addition, we would like to share correct student solutions that are well-written with the class after each homework. Are you okay with your correct solutions being used for this purpose? Answer "Yes", "Yes but anonymously", or "No"

2 Path TSP and Cycle TSP

This is a solo question.

In the Traveling Salesman Problem (TSP), we are given an undirected graph G with non-negative weights and asked to find a minimum weight cycle that visits each vertex exactly once.

In the s-t Traveling Salesman Problem (s-t TSP), we are given an undirected graph G with non-negative weights, two vertices s and t, and are asked to find a minimum weight path that starts at s, visits all other vertices exactly once, and ends at t.

(a) Consider the following reduction from s-t TSP to TSP: Take G and add a weight 0 edge between s and t to get a new graph G'. Show that if a s-t TSP solution of weight w exists in G, then a TSP solution of weight w exists in G'.

(b) Despite what you proved in part (a), why is the reduction in part (a) not valid?

(c) Give a valid reduction from s-t TSP to TSP. Prove correctness for your reduction. No runtime analysis needed.

Solution:

(a) We can add the weight 0 edge (s, t) to any s-t TSP solution in G to get a TSP solution in G' of the same weight.

(b) A TSP solution in G' is not forced to use the new edge, which means the smallest weight of any TSP solution in G' could be smaller than the smallest weight of any s-t TSP solution in G.

For example, suppose we have a complete graph on vertices s, a, b, t, and all edges have weight 0 except (a, b). Then adding this new edge doesn't affect the graph, since we already had a weight 0 edge between s, t. However, in this graph the optimal TSP solution has weight 0 (e.g., the cycle $s \to a \to t \to b$), but the optimal s-t TSP solution has weight 1, since it is forced to use the edge (a, b).

In fact, it is possible no s-t TSP solution exists in G, but a TSP solution exists in G': e.g. consider if we just delete the edge (a, b) from the previous example.

(c) Let (G, s, t) be an instance of s-t TSP. Add an artificial vertex q to the vertex set. Connect it with s and t only, with edges of weight 0. Solve TSP on the new graph G'.

Any TSP solution in G' must contain the two edges $(q, s), (q, t)$, since these are the only two edges that can visit and leave q. Observe that removing these two edges from the solution recovers a s-t TSP solution on G of the same weight. Furthermore, adding these two edges to any s-t TSP solution in G gives a TSP solution in G' of the same weight. So, a TSP solution of weight w exists in G' if and only if a s-t TSP solution of weight w exists in G.

3 SAT and Integer Programming

Consider the 3SAT problem, where the input is a set of clauses and each one is a OR of 3 literals. For example, $(x_1 \vee \overline{x}_4 \vee \overline{x_7})$ is a clause which evaluated to true iff one of the literals is true. We say that the input is satifiable if there is an assignment to the variables such that all clauses evaluate to true. We want to decide whether the input is satifiable.

On the other hand, consider the integer linear programming feasibility problem: We are given a set of variables and constraints in terms of these variables (we are not given an objective). The constraints are either linear inequalities, or the 0-1 constraints $x_i \in \{0, 1\}$. We want to decide if it possible to assign the variables values that satisfy all the constraints.

Give a reduction from 3SAT to integer linear programming feasibility, and briefly justify its correctness. No runtime analysis needed.

Solution: We construct an explicit integer linear program given a 3SAT formula, and show that it's feasible iff the SAT formula is satifiable. Let the variables be $x_1, \cdots, x_n$ and we add the constraint $x_i \in \{0, 1\}$.

Now we have to translate boolean clauses like $(x_1 \vee \overline{x}_4 \vee \overline{x_7})$ into linear constraints. We tranform them systematically like this:

$$x_1 \vee \overline{x_4} \vee \overline{x_7} \quad \Longleftrightarrow \quad x_1 + (1 - x_4) + (1 - x_7) \geq 1$$

That is, we replace every negated variable $\overline{x}_i$ in the literal as $1 - x_i$ and replace the OR by $+$. The constraint ensures that at least one of the literals is true. We perform this tranform for each clause and get a bunch of linear constraints over binary variables.

The observation is that any satifiable assignment corresponds to a feasible solution to the integer program, and vice versa. Hence, solving the feasibility problem of integer linear programming suffices to solve 3SAT.

4 Convex Hull

Given n points in the plane such that no three points are collinear, the *convex hull* is the list of points, in counter-clockwise order, that describe the convex polygon that contains all the other points. Imagine a rubber band is stretched around all of the points: the set of points it touches is the convex hull. You can also play around with defining your own set of points and seeing what the polygon should look like at `http://cs.yazd.ac.ir/cgalg/AlgsVis/ConvexHull.html`

In this problem we'll show that the convex hull problem and sorting reduce to each other in linear time.

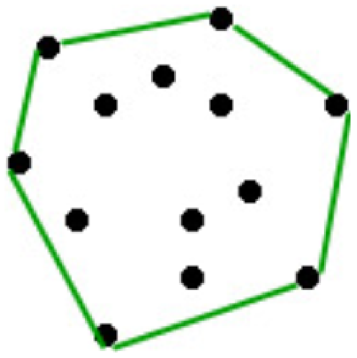


Figure 1: An instance of convex hull: the convex hull is the six points connected by green lines.

(a) Fill in the following algorithm for convex hull; you do not need to prove it correct. What is its runtime?

```
procedure ConvexHull(list of points $P[1..n]$)
    Set $low$ := the point with the minimum $y$-coordinate, breaking ties by minimum $x$-coordinate.
    Create a list $S[1..n-1]$ of the remaining points $p \in P$ sorted increasingly by the angle between the vector $p - low$ and the vector $(1, 0)$ (i.e the x-axis) .
    Initialize $Hull$ := $[low, S[1]]$
    for $p \in S[2..n-1]$ do
        <fill in the body of the loop>
    Return $Hull$
```

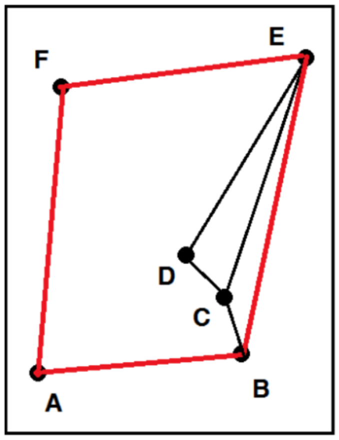
(b) Now, find a linear time reduction from sorting to convex hull. In other words, given a list of real numbers to sort, describe an algorithm that transforms the list of numbers into a list of points, feeds them into convex hull, and interprets the output to return the sorted list. Then, prove that your reduction is correct.

(Note that your reduction should not create three points that are collinear, per the definition of the convex hull problem. Hint: For each number a in the list, create a point $(a, f(a))$, where $f(a)$ is some simple function of a. Your choice of f should ensure that every point is in the convex hull.)

Solution:

(a)
```
procedure ConvexHull(list of points $P[1..n]$)
    Set $low$ := the point with the minimum $y$-coordinate, breaking ties by minimum $x$-coordinate.
    Create a list $S[1..n-1]$ of the remaining points sorted increasingly by the angle between the vector $point - low$ and the vector $(1, 0)$ (i.e the x-axis) .
    Initialize $Hull$ := $[low, S[1]]$
    for $p \in S[2..n-1]$ do
        While the angle between $Hull[-2]$, $Hull[-1]$, and $p$ is a right turn*, pop $Hull[-1]$.
        Append $p$ to the end of $Hull$.
        Return $Hull$
```

* The angle between a, b, and c is a right turn if $\overrightarrow{ab}$ is counterclockwise from $\overrightarrow{bc}$.



For example, in this diagram, $\overrightarrow{ABC}$ and $\overrightarrow{BCD}$ are left turns, but $\overrightarrow{CDE}$ is a right turn, so we pop D. Then, $\overrightarrow{BCE}$ is a right turn, so we pop C. $\overrightarrow{ABE}$ and $\overrightarrow{BEF}$ are left turns, so the complete convex hull, shown in red, is $ABEF$.

It takes $\Theta(n)$ time to compute the slope of the points from low, on which we can sort. Then, it takes $\Theta(n \log n)$ time to sort the points. The loop appends each point to the hull exactly once, and pops it from the hull at most once, and makes a constant number of operations before each push or pop. So the loop takes $\Theta(n)$ time in total. Therefore, the reduction to sorting (all the nonsorting steps) take $\Theta(n)$ time, and the algorithm as a whole takes $\Theta(n \log n)$ time.

(b) Transform the list of numbers a into points on the plane as follows: $f(a) = (a, a^2)$. Run convex hull on the list. Note that it can return the hull starting with any point, as long as the points are in counterclockwise order. So, we shift the list so that the point with the smallest a is first (we can do this by e.g. doing a linear scan to find the minimum x-coordinate of any point, and then taking the points in the list before this point and moving them to the end). Then, transform the points in the result back into numbers by removing the y-coordinate, and return the new list.

This reduction is correct because we can be certain that we transform the points into an instance of convex hull with a solution that is exactly the sorted list of points. No matter the list we're trying to sort, all the points (a, a^2) will be on the convex hull. In addition, counter-clockwise order corresponds to left-to-right order for these points.

In our reduction, transforming numbers to points (and back) takes linear time, and shifting the list takes linear time. Thus the reduction takes linear time in total.

5 Hitting Set

In the Hitting Set Problem, we are given a family of finite integer sets $\{S_1, S_2, \ldots, S_n\}$ and a budget b, and we wish to find an integer set H of size $\leq b$ which intersects every S_i, if such an H exists. In other words, we want $H \cap S_i \neq \emptyset$ for all i.

Show that the Hitting Set Problem is NP-complete. (Hint: Hitting Set generalizes one of the problems covered in Chapter 8 of the textbook).

Solution: We can see that the problem is in NP since we can quickly check that a potential hitting set covers all sets and has size less than b.

In addition, Hitting Set is a generalization of the Vertex-Cover Problem. Given a graph G, consider each edge $e = (u, v)$ as a set containing the elements u and v. Then, finding a hitting set of size at most b in this particular family of sets is the same as finding a vertex cover of size at most b for the given graph. Since Vertex Cover is NP-Hard, Hitting Set must be NP-Hard as well.

One can also reduce from the Set Cover problem. For every set S in Set Cover, we create an element e'_S in Hitting Set, and for every element e in Set Cover, we create a set S'_e in Hitting Set. Each S'_e contains all elements e'_S corresponding to sets S containing e. Again, finding a hitting set of size at most b is exactly the same as finding a set cover of size at most b in the original instance.

6 NP Basics

This is a solo question.

Assume A reduces to B in polynomial time. In each part you will be given a fact about one of the problems. What information can you derive of the other problem given each fact?

1. A is in **P**.
2. B is in **P**.
3. A is **NP**-hard.
4. B is **NP**-hard.

Solution: If A reduces to B, we know B can be used to solve A, which means B is at least as hard as A. As a result, if B is in **P**, we can say that A is in **P**, and if A is **NP**-hard, we can say that B is **NP**-hard. If A is in **P**, or if B is **NP**-hard, we cannot say anything about the complexity of B or A respectively.

7 Upper Bounds on Algorithms for NP Problems

Parts a and c of this problem are solo questions. You may collaborate on part b.

(a) Recall the 3-SAT problem: we have n variables x_i and m clauses, where each clause is the OR of at most three literals (a literal is a variable or its negation). Our goal is to find an assignment of variables that satisfies all the clauses, or report that none exists.

Give a $O(2^n m)$-time algorithm for 3-SAT. Just the algorithm description is needed.

(b) Using part (a) and the fact that 3-SAT is NP-hard, give a $O(2^{n^c})$-time algorithm for every problem in NP, where c is a constant (that can depend on the problem). Just the algorithm description and runtime analysis is needed. An informal algorithm description is fine.

(This result is known as $NP \subseteq EXP$.)

(c) Recall the halting problem from CS70: Given a program (as e.g. a .py file), determine if the program runs forever or eventually halts. Also recall that there is no finite-time algorithm for the halting problem. Let us define the input size for the halting problem to be the number of characters used to write the program.

Given an instance of 3-SAT with n variables and m clauses, we can write a size $O(n+m)$ program that halts if the instance is satisfiable and runs forever otherwise. So there is a polynomial-time reduction from 3-SAT to the halting problem.

Based on this reduction and part (b): Is the halting problem NP-hard? Is it NP-complete? Justify your answer.

Solution:

(a) Enumerate all 2^n assignments. We can check if each assignment is satisfying in $O(m)$ time. So we can find a satisfying assignment if one exists $O(2^n m)$-time.

(b) Any problem in NP can be reduced to an instance of 3-SAT in polynomial time. Since the reduction takes polynomial time, and it takes at least $O(1)$ time to write down a variable/clause, the 3-SAT instance can't have more than n^b variables and n^b clauses for some constant b (for all sufficiently large n). So our algorithm is to reduce to 3-SAT and then solve this 3-SAT instance in $O(2^{n^b} n^b) = O(2^{n^{b+1}})$ time using part a. This is $O(2^{n^c})$ for $c = b + 1$.

Alternate approach: Since the problem is in NP, if the answer to the problem is "yes", there is some polynomial-size "witness" to the problem that we can feed to a verification algorithm and have it output "yes" in polynomial time. If the answer is no, none of these witness cause it to output "yes".

So, we can enumerate over all witnesses, and feed them to this verification algorithm, and output yes if any run of the verification algorithm outputs yes. Since there are polynomially many witnesses, and the verification algorithm runs in polynomial time, this takes $O(2^{n^c})$ time for some constant c.

(c) The reduction implies the halting problem is NP-hard. We can show the halting problem isn't in NP, and thus it isn't NP-complete: By part (a), any problem in NP has a finite-time algorithm, but the halting problem doesn't have one. So the halting problem isn't NP-complete.

8 (Extra Credit) Orthogonal Vectors

In the 3-SAT problem, we have n variables and m clauses, where each clause is the OR of (at most) three of these variables or their negations. The goal of the problem is to find an assignment of variables that satisfies all the clauses, or correctly declare that none exists.

In the orthogonal vectors problem, we have two sets of vectors A, B. All vectors are in $\{0,1\}^m$, and $|A| = |B| = n$. The goal of the problem is to find two vectors $a \in A, b \in B$ whose dot product is 0, or correctly declare that none exists. The brute-force solution to this problem takes $O(n^2 m)$ time: We compute all $|A||B| = n^2$ dot products between two vectors in A, B, and each dot product takes $O(m)$ time.

Show that if there is a $O(n^c m)$-time algorithm for the orthogonal vectors problem for some $c \in [1, 2)$, then there is a $O(2^{cn/2} m)$-time algorithm for the 3-SAT problem. For simplicity, you may assume in 3-SAT that the number of variables must be even.

Solution: We use an $O(2^{n/2} m)$-time reduction from 3-SAT to orthogonal vectors. We split the variables into two groups of size $n/2$, V_1, V_2. For each group, we enumerate all $2^{n/2}$ possible assignments of these variables. For each assignment x of the variables in V_1, let v_x be the vector where the ith entry is 0 if the ith clause is satisfied by one of the variables in this assignment, and 1 otherwise. We ignore the variables in the clause that are in V_2. For example, if clause i only contains variables in V_2, then $v_x(i)$ for all x. Let A be the $2^{n/2}$ vectors produced this way.

We construct B containing $2^{n/2}$ vectors in a similar manner, except using V_2 instead of V_1.

We claim that the 3-SAT instance is satisfiable if and only if there is an orthogonal vector pair in $A \times B$. Given this claim, we can solve 3-SAT by making the orthogonal vectors instance in $O(2^{n/2} m)$ time, and then solving the instance in $O((2^{n/2})^c m) = O(2^{cn/2} m)$ time.

Suppose there is a satisfying assignment to 3-SAT. Let x_1 be the assignment of variables in V_1, and x_2 be the assignment of variables in V_2. Let v_1, v_2 be the vectors in A, B corresponding to x_1, x_2. Since every clause is satisfied, one of $v_1(i)$ and $v_2(i)$ must be 0 for every i, and so $v_1 \cdot v_2 = 0$. So there is also a pair of orthogonal vectors in the orthogonal vectors instance.

Suppose there is a pair of orthogonal vectors v_1, v_2 in the orthogonal vectors instance. Then for every i, either $v_1(i)$ or $v_2(i)$ is 0. In turn, for the corresponding assignment of variables in V_1, V_2, the combination of these assignments must satisfy every clause. In turn, the combination of these assignments is a satisfying assignment for 3-SAT.

Comment: It is widely believed that SAT has no $O(2^{.999n} m)$-time algorithm - this is called the Strong Exponential Time Hypothesis (SETH). So it is also widely believed that orthogonal vectors has no $O(n^{1.99} m)$-time algorithm, since otherwise SETH would be violated. It turns out that we can reduce orthogonal vectors to string problems such as edit distance and longest common subsequence, and so if we belive SETH then we also believe those problems also don't have $O(n^{1.99})$-time algorithms. The field of research studying reductions between problems with polynomial-time algorithms such as these is known as fine-grained complexity, and orthogonal vectors is one of the central problems in this field.