% CSC 440 % Assignment 7: P, NP, and Approximation % Due Monday, April 29th

You may work in small groups on this assignment, but the work you hand in must be your own. You may submit this in one of two ways:

- Write it up electronically (e.g. LaTeX) and upload a PDF to Gradescope.
- Scan your handwritten solution (you can use a scanning app for a smartphone, such as Evernote's free Scannable app) and upload a PDF to Gradescope.

Your full name: Marceline Kelly

1. Coffeeshops (20 pts):

TLC Coffee has decided to expand to maximize profits over all of South County, by building new TLCs on many street corners. The management turns to URI's Computer Science majors for help. The students come up with a representation of the problem:

The street network is described as an arbitrary undirected graph $G = (V, E)$, where the potential restaurant sites are the vertices of the graph. Each vertex u has a nonnegative integer value p_u, which describes the potential profit of site u. Two restaurants cannot be built on adjacent vertices (to avoid self-competition). The students further come up with an exponential-time algorithm that chooses a set $U \subseteq V$, but it runs in $O(2^V|E|)$ time. The TLC ownership is dismayed, and is convinced they should have hired UConn's CS majors instead. In order to prove that nobody else could have come up with a better algorithm, you need to prove the hardness of the coffeeshop problem.

Define coffeeshop to be the following decision problem: given an undirected graph $G = (V, E)$, given a mapping p from vertices $u \in V$ to nonnegative integer profits $p(u)$, and given a nonnegative integer k, decide whether there is a subset $U \subseteq V$ such that no two vertices in U are neighbors in G, and such that $\sum\limits_{u \in U} p(u) \geq k$.

Prove that coffeeshop is NP-hard. (Hint: Try a reduction from 3SAT.)

Also, explain why this implies that, if there is a polynomial-time algorithm to solve the original problem, i.e., to output a subset U that maximizes the total profit, then $P = NP$.

This problem can be solved similarly to the graph coloring example shown in class. Each variable represents a potential TLC location; a value of "true" means it is included in the final set and vice versa. Each clause is represented by a fully-connected triangle of three vertices, where each vertex is one of the three literals in the clause. Between distinct clauses, there are edges connecting literals and their respective negations (e.g. u and $\bar{u}$).

Let's give each vertex a profit level of 1. This means that if we select a vertex u, then the total profit increases by 1. To reach the profit level of k, we can create k clauses, from each of which we'll select one vertex. We'll only select vertices that correspond to true literals. Some variables may appear multiple times across different clauses, which is allowed.

This setup prevents us from selecting neighboring locations because we only select one vertex per clause. We also can't select both vertices connected by inter-clause edges (u and $\bar{u}$) because a literal and its negation cannot both be true.

If we manage to choose a vertex from each clause, then the resulting set will satisfy the problem. If we can't create enough clauses that satisfy the non-adjacency requirement to build such a set, then the problem has no solution, at least with the given value of k.

By proving that coffeeshop is NP-hard, we've stated that it is at least as hard as every other problem in NP. If we find a polynomial solution to the problem, then all of those other problems' hardness must be adjusted, too. Suddenly, every NP-hard problem is now polynomial.

2. Asymptotic Complexity (20 pts):

Suppose you buy a new computer that is twice as fast as the one you currently own. How much larger a problem (as a function of n) can you solve in the same amount of time on the new machine, given algorithms with time complexity:

- logarithmic base 2 ($\mathcal{O}(\lg n)$)
- linear ($\mathcal{O}(n)$)
- quadratic ($\mathcal{O}(n^2)$)
- exponential base 2 ($\mathcal{O}(2^n)$)

Let n be the problem size on the slower computer, and m be the problem size on the faster computer.

Logarithmic

$$T_{old} = \log_2(n)$$
$$T_{new} = \log_2(m)$$

$$\log_2(m) = 2\log_2(n)$$
$$\log_2(m) = \log_2(n^2)$$
$$m = n^2$$

Linear

$$T_{old} = n$$
$$T_{new} = m$$

$$m = 2n$$

Quadratic

$$T_{old} = n^2$$
$$T_{new} = m^2$$

$$m^2 = 2n^2$$
$$m = \sqrt{2n^2}$$
$$m = n\sqrt{2}$$

Exponential

$$
\begin{aligned}
T_{old} &= 2^n \\
T_{new} &= 2^m
\end{aligned}
$$

$$
\begin{aligned}
2^m &= 2 \cdot 2^n \\
2^m &= 2^{n+1} \\
m &= n + 1
\end{aligned}
$$

3. TRIPLE-SAT (20 pts):

Let TRIPLE-SAT denote the following decision problem: given a Boolean formula ψ, decide whether ψ has at least three distinct satisfying assignments. Prove that TRIPLE-SAT is NP-complete using a reduction.

NP-complete == in NP and NP-hard

To check if TRIPLE-SAT is NP-hard, we need to perform a reduction. To start, we'll take an arbitrary 3SAT formula. Any valid 3SAT formula is a valid TRIPLE-SAT formula (though not necessarily the other way around), so this reduction will be relatively straightforward. We can add new clauses to our 3SAT formula with dummy variables, like so:

[initial 3sat formula] $\wedge (d_1 \vee d_2 \vee d_3) \wedge (\neg d_1 \vee d_2 \vee d_3) \wedge ...$

If we have a satisfying assignment for the initial formula, we can choose the values of the dummy variables strategically to create additional satisfying assignments. This will allow us to reach the 3-assignment requirement. Thus, we have reduced 3SAT to TRIPLE-SAT.

After that, we need to check if TRIPLE-SAT is in NP. By definition, verifying a "yes" answer to an NP-hard problem tends to take a lot less work than a "no" answer. All we need to do to check is test three assignments and see if they satisfy the formula. If they don't, we pick new assignments and try again. If we have tried every single possible combination and still have not found a solution, then the problem is not in NP.

4. Clustering (20 pts):

Consider the following approach to grouping n distinct objects $p_1 \cdots p_n$ on which a distance function $d(p_i, p_j)$ is defined such that $d(p_i, p_i) = 0$, $d(p_i, p_j) > 0$ if $i \neq j$ (that is, p_i and p_j are distinct objects), and $d(p_i, p_j) = d(p_j, p_i)$ (distances are symmetric.) The clustering approach is as follows: divide the n points into k clusters such that we *minimize* the maximum distance between any two points in the same cluster. That is, we seek low-diameter clusters. Suppose we want to know whether, for a particular set of n objects, and a bound B, the objects can be partitioned into k sets such that no two points in the same set are further than B apart. Prove that this decision problem is NP-complete.

5. Knapsack approximations (20 pts):

In the Knapsack problem, we are given a set $A = a_1, ...a_n$ of items, where each a_i has a specified positive integer size s_i and a specified positive integer value v_i. We are also given a positive integer knapsack capacity B. Assume that $s_i \leq B$ for every item i. The problem is to find a subset of A whose total size is at most B and for which the total value is maximized. In this problem, we will consider approximation algorithms to solve the Knapsack problem. Notation: For any subset S of A, we write s_S for the total of all the sizes in S and v_S for the total of all the values in S. Let *Opt* denote an optimal solution to the problem. This problem has two parts (each 10 pts).

A: Consider the following greedy algorithm Alg_1 to solve the Knapsack problem:

Order all the items a_i in non-increasing order of their *density,* which is the ratio of value to size, $\frac{v_i}{s_i}$. Make a single pass through the list, from highest to lowest density. For each item encountered, if it still fits, include it, otherwise exclude it. **Prove** that algorithm Alg_1 does not guarantee any constant approximation ratio. That is, for any positive integer k, there is an input to the algorithm for which the total value of the set of items returned by the algorithm is at most $\frac{v_{Opt}}{k}$.

To rephrase the prompt, prove that $v_S \leq \frac{v_{Opt}}{k}$.

Let's say we've found two items to steal, a small diamond ring (D) and a 1 kg gold bar (G). k has to factor into the equation somehow, so let's assign each item the following values:

$v_D = 300; s_D = 1$

$v_G = 500k; s_G = 50k$

And let's say our bag has just enough room to fit the entire gold bar ($50k$). We know that, obviously, the gold bar is the better item to take, being worth more than the ring. However, the ring is denser in terms of value ($\frac{300}{1} > \frac{500k}{50k}$), so the algorithm chooses it, instead. Now we have no space left for the gold and have missed out big time on potential profits.

Plugging in the relevant values, we get:

$v_S \leq \frac{v_{Opt}}{k} \Rightarrow 300 \leq \frac{500k}{k}$

This statement is true given the scenario, so we have proven that the algorithm has a variable approximation ratio.

B: Consider the following algorithm Alg_2:

If the total size of all the items is $\leq B$, then include all the items. If not, then order all the items in non-increasing order of their densities. Without loss of generality, assume that this ordering is the same as the ordering of the item indices. Find the smallest index i in the ordered list such that the total size of the first i items exceeds B. In other words, $\sum_{j=1}^{i} s_j > B$, but $\sum_{j=1}^{i-1} s_j \leq B$. If $v_i > \sum_{j=1}^{i-1} v_j$, then return a_i. Otherwise, return $a_1, ..., a_i - 1$.

Prove that alg_2 always yields a 2-approximation to the optimal solution.

In other words, prove that this algorithm always produces a knapsack with at least half the total value of the optimal knapsack.

Let's say that index k is the boundary at which the sum of our items' values exceeds the knapsack's capacity. That is to say, either the algorithm will return $a_1, a_2, ..., a_{k-1}$ or a_k. We know that $s_1 + s_2 + ... + s_k$ is greater than the knapsack can hold, and thus is also greater than the optimal size. Similarly, it follows that $s_1 + s_2 + ... + s_{k-1}$ must be less than or equal to the optimal size. Depending on the knapsack's capacity and the individual item sizes, either $s_1 + s_2 + ... + s_{k-1}$ or s_k will comprise a greater share of the optimal size. If we count these as two distinct terms, then one or both will necessarily be at least half of the optimal size. Since our algorithm always chooses the greater option, it must yield a 2-approximation.

Additional Space for problem 5

Additional Space as needed