UTAH STATE UNIVERSITY, COMPUTER SCIENCE DEPARTMENT

CS 7910 Computational Complexity Assignment 6

Gopal Menon

March 6, 2016

1. **(20 points)** In this exercise, we consider the *Traveling Salesman Problem (TSP).*

   Let G be a **complete directed graph**. Each directed edge (v_i, v_j) has a weight $w(v_i, v_j)$. It is possible that $w(v_i, v_j) \neq w(v_j, v_i)$, i.e., the edge weights can be asymmetric.

   The optimization version of the TSP problem is to find a cycle C in G containing each vertex of G exactly once such that the total weight of the edges of C is minimized.

   The decision problem of TSP is as follows. Given a value W, decide whether G has a cycle C that contains each vertex of G exactly once such that the total weight of the edges of C is at most W.

   Prove that the decision problem of TSP is NP-Complete (Hint: by a reduction from the Hamiltonian Cycle problem).

   The Hamiltonian Cycle problem is known to be NP-Complete. We can reduce the Hamiltonian Cycle problem to an instance of the TSP problem. Let (G', W) be the instance of the TSP problem, where W is an integer value equal to the number of edges in the Hamiltonian cycle and the graph G' is constructed from G by

   a) creating a graph G' by incorporating every vertex and edge present in the Hamiltonian cycle of G
   b) assigning a weight of 1 both ways to every edge of G'
   c) creating an edge and assigning a weight of ∞ (infinity) to it between every node in G' that does not already have an edge

   This reduction can clearly be done in polynomial time based on the following analysis. The first two steps of the reduction shown above can be done in $O(W)$ time. In the last step, we need to create edges having a weight of ∞ between all the node pairs in G' that do not already have edges. This can be done in time:

$$\begin{aligned} T &= {}^{W-1}C_2 - W \\ &= \frac{(W-1)!}{(W-3)!2!} - W \\ &= O(W^2) \end{aligned}$$

So the reduction can be done in $O(W^2)$ time.

Given a certificate for the TSP problem, which will be an ordered list of edges, it is easy to verify in $O(N)$ time, where N is the number of edges in certificate, or polynomial time, that the certificate is correct. This means that the TSP problem is in NP.

Consider the case where a Hamiltonian cycle exists in the graph G. In this case, for the TSP instance that was constructed, there exists a cycle that contains every node in it and has a total weight of edges of W.

In the case where the instance of the TSP problem is true (there exists a cycle in graph G' that has a total weight of edges equal to W), based on the way the graph G' was constructed, we know that there must exist a cycle in graph G. This means that we can reduce the Hamiltonian Cycle problem in polynomial time to the TSP problem. It follows that the TSP problem is in NP-Hard. Since we already know that the TSP problem is in NP, it follows that it is in NP-Complete.

2. This is a "warm-up" exercise on designing approximation algorithms.

   A variation of the *knapsack problem* is defined as follows (actually it is the subset-sum problem): Given as input a knapsack of size K and n items whose sizes are $k_1, k_2, \ldots, k_n$, where K and $k_1, k_2, \ldots, k_n$ are all positive real numbers, find a full "packing" of the knapsack (i.e., choose a subset of the n items such that the total sum of the sizes of the items in the chosen subset is *exactly* equal to K).

   We have proven that this problem is NP-complete, which implies that it is very likely that efficient algorithms (i.e., polynomial-time algorithms) for this problem do not exist. Thus, people tend to look for good **approximation algorithms** for solving this problem. In this exercise, we relax the constraint of the knapsack problem as follows. We still seek a packing of the knapsack, but we need not look for a "full" packing of the knapsack; instead, we look for a packing of the knapsack (i.e., a subset of the n input items) such that the total sum of the sizes of the items in the chosen subset is at least $\frac{K}{2}$ (but no more than K). This is called a *factor of* 2 *approximation solution* for the knapsack problem. To simplify the problem, we assume that a factor of 2 approximation solution for the input always exists, i.e, there always exists a subset of items whose total size is at least $\frac{K}{2}$ and at most K.

   a) **(20 points)** Design a *polynomial-time* algorithm for computing a factor of 2 approximation solution, and analyze the running time of your algorithm (using the big-O notation).

   b) **(5 points)** Improve your algorithm to $O(n)$ time. If your algorithm for part (a) already runs in $O(n)$ time, then you will get the five points automatically.

   Note: You are required to clearly describe the main idea of your algorithm. Although the pseudo-code is not required, you may also give the pseudo-code if you feel it is helpful for you to explain your algorithm. You also need to briefly explain why your algorithm works, i.e., why your algorithm can produce a factor of 2 approximation solution. Finally, please analyze the running time of your algorithm.

If a factor 2 approximation solution is guaranteed to exist, we can go through the set of n items from beginning to end and do the following. If any item is of size $> K$, then we discard it. Any item found to be of size $\geq \frac{K}{2} \leq K$, can be returned as the solution. Else we can keep accumulating the other items (having size $< \frac{K}{2}$) and return the collection of such items as the required solution as soon as the sum of the collection crosses $\frac{K}{2}$.

The algorithm is shown in algorithm 1 where it receives the set of items I and the knapsack size K as parameters.

Algorithm 1 Factor 2 Approximation Algorithm

```
1:  procedure FACTOR 2 APPROXIMATION ALGORITHM($I, K$)
2:      create $array\_of\_items$ of size $I.length$
3:      $array\_sum = 0$
4:      for all $k_i$ in $I$ do
5:          if $k_i \geq \frac{K}{2} \leq K$ then
6:              return $k_i$
7:          else if $k_i < \frac{K}{2}$ then
8:              insert $k_i$ into $array\_of\_items$
9:              $array\_sum = array\_sum + k_i$
10:             if array_sum $\geq \frac{K}{2}$ then
11:                 return $array\_of\_items$
12:             end if
13:         end if
14:     end for
15:     return $array\_of\_items$
16: end procedure
```

Since the factor 2 approximation algorithm is able to achieve its objective by going through all the items in one pass, the running time will be $O(n)$, where n is the number of items.