CPSC 320 2023S2: Tutorial 6 Solutions

1 Exhibition guarding

In the Exhibition Guarding Problem, we are given a line L that represents a long hallway in a gallery; assume that the line has a left and right end. We are also given an unordered array X of real numbers that represent the positions of precious objects in this hallway, where a position marks the distance from the left end of the line. A guard at a position p on the line can protect objects within distance d of p, that is, objects in the range $[p - d, p + d]$.

1. Design an algorithm that finds the minimum number of guard positions needed to protect all objects in X. For this tutorial, focus on finding a promising algorithm. Assignment 3 will ask you to reason about its correctness, so you don't need to do that as part of this tutorial.

   SOLUTION: We make sure that the leftmost guard protects the object at the leftmost position, while choosing the leftmost position to be as far to the right as possible so the guard protects as many other objects as possible. Then we apply the algorithm recursively with the remaining unguarded objects. The following pseudocode captures this idea.

```
procedure PLACE-GUARDS(($X[1..n], d$))
    ▷ $X$ contains positions of $n \geq 0$ objects on a line
    ▷ $d$ is the distance (both to the left and right) that a guard can protect
    Sort $X$ in increasing order
    return Place-Guards-Helper($X[1..n], d$)

procedure PLACE-GUARDS-HELPER(($X[1..n], d$))
    if $n == 0$ then
        return the empty set ▷ There are no objects, so no guard is needed
    else
        $p \leftarrow X[1] + d$ ▷ Place a guard as far right as possible while protecting $X[1]$
        ▷ skip the other objects protected by this guard:
        $i \leftarrow 2$
        while ($i \leq n$) and ($X[i] \leq p + d$) do
            $i \leftarrow i + 1$
        return $\{p\}$ + PLACE-GUARDS-HELPER($X[i..n], d$)
```

2. What is the running time of an efficient implementation of your algorithm?

   SOLUTION: $O(n \log n)$ time is needed to sort array X. This is the dominant cost; the helper algorithm takes $O(n)$ time in total, because each array element is examined just once, taken over all the calls to the helper algorithm.

2 Knapsack with Structured Weights: Part 2

Let's continue with the **Knapsack problem** we saw in Tutorial 5. A problem instance is a set of n items, numbered $1, ..., n$. Each item i has a positive weight $w_i > 0$ and a positive value $v_i > 0$. Furthermore, we're given an overall weight capacity C. The problem is to select a subset $S \subseteq \{1, ..., n\}$ that maximizes the total value of S, but keeps the total weight below C. Formally, we want to maximize

$$V(S) = \sum_{i \in S} v_i,$$

subject to $\sum_{i \in S} w_i \leq C$. We're considering the special case of the problem where the capacity as C is 1, and restrict the weights to be either 1/2, 1/4, 1/8, or 1/16.

We showed last time that if we sort the 1/16-weight items in decreasing order of value, and then pair up the items in sequence (i.e. the 1st and 2nd items form a pair, then the 3rd and 4th items, then the 5th and 6th, etc.). then there is an optimal solution that uses that uses the c highest-valued pairs, for some c.

Now here's the key idea of our greedy algorithm. Given an instance I of the problem, create a new instance I' as follows. Initially $I' = I$. Sort the 1/16-weight items in decreasing order of value, and pair them up. Now, for each pair of items (i, j) delete the items from I' and replace them with a new item whose weight is $w_i + w_j$ and whose value is $v_i + v_j$. Instance I' has no 1/16-weight items.

There's a natural correspondence between solutions for I and I', where each "paired" item in a solution S' for I' is replaced by the two individual items in the corresponding solution S for I, and vice versa. Note also that corresponding solutions S and S' have the same total value, i.e., $V(S) = V(S')$ Use this to show that:

1. The value of an optimal solution for instance I is less than or equal to the value of an optimal solution for instance I'.

   SOLUTION: Let S be an optimal solution for instance I and let S' be the corresponding solution for I'. Then $V(S) = V(S')$, and so the optimal solution for I' must have value at least $V(S)$.

2. The value of an optimal solution for instance I' is less than or equal to the value of an optimal solution for instance I.

   SOLUTION: This follows by similar reasoning to part (a).

Given an original problem instance, we've seen how to eliminate the 1/16-weight items to create an equivalent problem without 1/16-weight instances.

3. How do we now solve the resulting problem instance that has no 1/16-weight items?

   SOLUTION: The exact same reasoning allows us to create an equivalent problem instance without 1/8-weight items! Once the 1/8-weight items are gone, we can eliminate the 1/4-weight items. Then, we solve the problem simply by picking the top two 1/2-weight items.

   This is the efficient greedy algorithm for this problem, and we've already proven that it is optimal using an exchange arguments approach!