

ECE 537 - Foundations of Computing

Prof. Sen

**Homework #8**

**Solutions**

1. Problem 16.2-2 in the Cormen et. al text.

**Solution:** This problem has optimal substructure — Let  $I$  be the highest numbered item in an optimal solution  $S$  for  $W$  pounds and items  $1..n$ . Then  $S' = S - \{i\}$  is an optimal solution for  $W - w_i$  pounds and items  $1..i - 1$ , and the value of solution  $S$  is  $v_i$  plus the value of the solution to subproblem  $S'$ .

Define  $c[i, w]$  to be the value of the solution for items  $1..i$  and maximum weight  $w$ . Then

$$c[i, w] = \begin{cases} 0, & \text{if } i = 0 \text{ or } w = 0 \\ c[i - 1, w], & \text{if } w_i > w \\ \max(v_i + c[i - 1, w - w_i], c[i - 1, w]), & \text{if } i > 0 \text{ and } w \geq w_i \end{cases}$$

That is, the value of the solution for  $i$  items either includes item  $i$ , in which case it is  $v_i$  plus a subproblem solution for  $i - 1$  items and the weight excluding  $w_i$ , or it doesn't include item  $i$ , in which case it is a subproblem solution for  $i - 1$  items and the same weight. Thus, if the thief takes item  $i$ , he takes  $v_i$  value, and he can then choose from items  $1..i - 1$  up to the weight limit  $w - w_i$  to get  $c[i - 1, w - w_i]$  additional value. If he doesn't take item  $I$ , he still has  $1..i - 1$  items to choose from, but with a weight limit of  $w$ , and he gets  $c[i - 1, w]$  value. The thief should make the better of these two choices.

Algorithm—

0-1 Knapsack( $v, w, n, W$ )

```
1 for  $w \leftarrow 0$  to  $W$  do
2    $c[0, w] \leftarrow 0$ 
3 for  $i \leftarrow 1$  to  $n$  do
4    $c[i, 0] \leftarrow 0$ 
5   for  $w \leftarrow 1$  to  $W$  do
6     if  $w_i \leq w$  then
7       if  $v_i + c[i - 1, w - w_i] > c[i - 1, w]$  then
8          $c[i, w] \leftarrow v_i + c[i - 1, w - w_i]$ 
9       else
10         $c[i, w] \leftarrow c[i - 1, w]$ 
11     else
12         $c[i, w] \leftarrow c[i - 1, w]$ 
```

Example—

Consider the following three items, and a knapsack that holds 5 pounds:

| <i>Item</i> | <i>value</i> | <i>weight</i> | <i>value/weight</i> |
|-------------|--------------|---------------|---------------------|
| 1           | 12           | 3             | 4                   |
| 2           | 10           | 2             | 5                   |
| 3           | 6            | 1             | 6                   |

The greedy strategy would take the highest value/weight items first. That is item 3, then item 2, then item 1. Note, however, that once items 3 and 2 are taken, item 1 won't fit since it weights 3 lbs., but the knapsack can only hold 2 more lbs. after storing items 3 and 2. The total value of this solution is \$16, and it leaves 2 lbs. unused.

The optimal solution, which is obtained using the dynamic programming approach takes items 1 and 2, has a value of \$22, and fully utilizes all 5 lbs.

Dynamic programming solution –  
*c* table:

| <i>i</i> | <i>w</i> |   |    |    |    |    |
|----------|----------|---|----|----|----|----|
|          | 0        | 1 | 2  | 3  | 4  | 5  |
| 0        | 0        | 0 | 0  | 0  | 0  | 0  |
| 1        | 0        | 0 | 0  | 12 | 12 | 12 |
| 2        | 0        | 0 | 10 | 12 | 12 | 22 |
| 3        | 0        | 6 | 10 | 16 | 16 | 22 |

From this table we can determine which items should be taken. We start at position  $c[n, W]$  and trace the optimal values from that point. If  $c[i, w] = c[i - 1, w]$ , item  $i$  is *not* part of the solution, and we continue tracing with  $c[i - 1, w]$ . Otherwise, item  $i$  is part of the solution, and we continue tracing with  $c[i - 1, w - w_i]$ .

The above algorithm takes  $\Theta(nW)$  time. Specifically,  $\Theta(nW)$  time is needed to fill in the *c* table (there are  $(n + 1) \times (W + 1)$  entries, and each takes  $\Theta(1)$  time to compute). Next,  $O(n)$  time is required to trace the solution, which starts at row  $n$  of the table, and moves up one row at each step.

2. Problem 16.2-4 in the Cormen et. al text.

**Solution:** The optimal strategy is the obvious greedy one: Starting with a full tank of gas, the professor should to to the farthest gas station he can get to within  $n$  miles of Newark. Fill up there. Then to the the farthest gas station he can get to within  $n$  miles of where he filled up, and fill up there, etc. Looking at this from another viewpoint, at each gas station the professor should check whether or not he can make it to the next gas station without stopping at the current one. If he can, he should skip the current one; if he cannot, he should fill up. Note that the professor does not need to know ahead of time how much gas he has or how far the next station is to implement this strategy, since at each fillup he can determine the next station that he'll need to stop at.

The problem has optimal substructure. To see this, suppose there are  $m$  possible gas stations. Consider an optimal solution with  $s$  stations and whose first stop is at the  $k$ th gas station. Then the rest of the optimal solution must be an optimal solution to the subproblem of the remaining  $m - k$  stations. Otherwise, if there were a better solution to the subproblem, i.e., one with fewer

than  $s - 1$  stops, we could use it to come up with a solution with fewer than  $s$  stops for the full problem, contradicting the assumed optimality of the overall solution.

This problem also has the greedy-choice property. That is, suppose there are  $k$  gas stations beyond the start that are within  $n$  miles of the start. The greedy solution chooses the  $k$ th station as the first stop. No station beyond the  $k$ th works as a first stop, since the professor runs out of gas before getting to those stations. If he chooses a station  $j < k$  as the first stop, then the professor could choose the  $k$ th station instead, having at least as much gas when he leaves the  $k$ th station as if he had chosen the  $j$ th station. Therefore, he would get at least as far without filling up again if he had chosen the  $k$ th station.

Since there are  $m$  gas stations on the map, the running time of this greedy algorithm is  $O(m)$ .

3. Generalize Strassen's matrix multiplication algorithm as we discussed in class to square matrices whose sizes are not a power of 2. Describe how the generalized algorithm works, and provide an analysis of its running time.

**Solution:** We have seen that the running time of Strassen's algorithm for square matrices whose sizes are a power of 2 is given by:

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2),$$

which yields a closed form of

$$T(n) = \Theta(n^{2.807}).$$

One simple way of generalizing Strassen's algorithm to non-power-of-2 matrices involves "padding" the matrix with zeros until its dimensions become that next largest power of 2. Note that this will not change the result, as the final matrix will just contain some zeros that should be ignored.

Does this change the running time of the algorithm? Well, the running time will change by the amount of time that it takes to perform the padding, along with any additional operations performed on these added elements. In the worst case, the size of the matrices are nearly doubled by the padding. Specifically, if the matrix dimensions are one greater than some power of two, i.e.,  $2^p + 1$ , then each matrix must be padded so that its dimensions become  $2^{p+1}$ . Thus, we can upper bound the running time of the generalized the algorithm by  $2T(n) = \Theta(2n^{2.807}) = \Theta(n^{2.807})$ .

4. In the traveling salesman problem (TSP) we are given a graph  $G$  that consists of a set of  $n$  vertices  $V = \{v_1, \dots, v_n\}$  that correspond to cities, along with a set of nonnegative edge weights  $E$ , where  $e_{ij}$  corresponds to the distance between city  $v_i$  and  $v_j$ . The goal is to find a tour that visits all cities, returning to the starting city, and has minimal total distance. Assume that  $e_{ii} = 0$ , and that  $e_{ij} = \infty$  if there is no edge connecting cities  $v_i$  and  $v_j$ .

- (a) Develop a brute-force algorithm that will always solve this problem optimally. What is the time complexity of your algorithm?

**Solution:** The brute force solution to this problem involves enumerating all possible tours, and calculating the length of each of these in order to find a minimal tour. Since for  $n$  cities

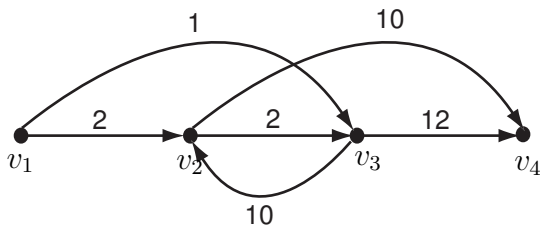
there are  $n!$  possible tours, the running time of the brute force algorithm is  $\Theta(n!)$  and the space required by this algorithm is  $\Theta(1)$ .

- (b) Show that this problem has optimal substructure, and then develop an algorithm that attempts to solve this problem by constructing a tour using a greedy strategy that always adds the next closest city to the tour. What is the time complexity of your algorithm? Give an example in which your greedy algorithm does not produce an optimal solution.

**Solution:** Consider an optimal tour, which corresponds to some permutation of the vertex set. To see that this problem has optimal substructure, assume, without loss of generality, that this optimal tour begins and ends at vertex  $v_1$ . Thus the tour includes the edge  $e_{1j}$ ,  $j \neq 1$ , followed by a path  $p^*$  from vertex  $v_j$  back to vertex  $v_1$  that passes exactly once through each vertex in  $E - \{e_{1j}\}$ . If the tour is optimal (as short as possible), as we have assumed, then it must also be the case that the path  $p^*$  is as short as possible. To see why, assume there exists shorter path  $p'$  from vertex  $v_j$  back to vertex  $v_1$  that passes exactly once through each vertex in  $E - \{e_{1j}\}$ . By appending  $e_{1j}$  to  $p'$ , we create a tour whose length is less than that of the optimal tour, which is a contradiction. This contradiction establishes the fact that the problem has optimal substructure.

For the greedy algorithm, assume we're at city  $i$ , then the algorithm must scan row  $i$  of the  $E$  matrix (ignoring location  $i$  in this row) in order to find the closest city that has not been visited yet. Since we must visit  $n$  cities, i.e.,  $i = 1, \dots, n$  and there are  $n - 1$  cities to scan on the row associated with each of these, the running time of this greedy algorithm is  $\Theta(n^2)$ .

In order to show that it does not always produce an optimal solution, consider the following graph:



An optimal tour on the graph that starts at  $v_1$  visits the cities in the order  $v_1, v_2, v_3, v_4$ , with a total cost of 16; however, the greedy algorithm will visit them in the order  $v_1, v_3, v_2, v_4$ , with a total cost of 21.

- (c) Show that this problem has overlapping subproblems, and then use dynamic programming to develop an algorithm that will always solve this problem optimally. Demonstrate how

your solution operates on the problem:

$$E = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

What are the time and space complexities of your algorithm?

**Solution:** To show that this problem has overlapping subproblems, assume without loss of generality that a tour starts at  $v_1$  and consider a subset of the vertices  $S \subseteq V - \{v_1\}$ , and a vertex  $v_j \in V - S$ , with  $j = 1$  only in the case that  $S = V - \{v_1\}$ . Next, let  $c(v_j, S)$  be the length of the shortest path from  $v_j$  to  $v_1$  that passes exactly once through each vertex in  $S$ . Given this definition,  $c(v_1, V - \{v_1\})$  is the length of an optimal tour, and according to the discussion above, this can be calculated using

$$c(v_1, V - \{v_1\}) = \min_{2 \leq j \leq n} (e_{1j} + c(v_j, V - \{v_1, v_j\}))$$

Thus, the calculation of  $c(v_1, V - \{v_1\})$  can be accomplished once we have calculated the  $n - 1$  values of  $c(v_j, V - \{v_1, v_j\})$ , and each of these will require the calculation of  $n - 2$  different  $c$  values, etc., until we reach

$$c(v_i, \emptyset) = e_{i1}, \quad i = 2, 3, \dots, n,$$

where  $\emptyset$  is the empty set. Notice that there are many repeated calculations involved in what we have just described. Specifically, if we start from the subset  $S_0 = \emptyset$  and we add one vertex at each step of the calculations until we reach  $S_{n-1} = V - \{v_1\}$ , then at the  $k$ -th step we must consider all possible paths through the vertices in all possible subsets  $S_k$ , and these paths will include the lengths of subpaths that were previously calculated when we considered all possible subsets in  $S_{k-1}$ .

Let us demonstrate this dynamic programming algorithm on the four-node graph given above. At the first step,  $k = 0$ , we initialize

$$c(2, \emptyset) = 5, \quad c(3, \emptyset) = 6, \quad c(4, \emptyset) = 8.$$

Next, at step  $S = 1$ , we obtain

$$\begin{aligned}
c(2, \{3\}) &= e_{23} + c(3, \emptyset) \\
&= 15 \\
c(2, \{4\}) &= e_{24} + c(4, \emptyset) \\
&= 18 \\
c(3, \{2\}) &= e_{32} + c(2, \emptyset) \\
&= 18 \\
c(3, \{4\}) &= e_{34} + c(4, \emptyset) \\
&= 20 \\
c(4, \{2\}) &= e_{42} + c(2, \emptyset) \\
&= 13 \\
c(4, \{3\}) &= e_{43} + c(3, \emptyset) \\
&= 15.
\end{aligned}$$

At step  $k = 2$  we have

$$\begin{aligned}
c(2, \{3, 4\}) &= \min(e_{23} + c(3, \{4\}), e_{24} + c(4, \{3\})) \\
&= \min(29, 25) \\
&= 25 \\
c(3, \{2, 4\}) &= \min(e_{32} + c(2, \{4\}), e_{34} + c(4, \{2\})) \\
&= \min(31, 25) \\
&= 25 \\
c(4, \{2, 3\}) &= \min(e_{42} + c(2, \{3\}), e_{43} + c(3, \{2\})) \\
&= \min(23, 27) \\
&= 23.
\end{aligned}$$

Finally, at step  $k = 3$  we obtain

$$\begin{aligned}
c(1, \{2, 3, 4\}) &= \min(e_{12} + c(2, \{3, 4\}), e_{13} + c(3, \{2, 4\}), e_{14} + c(4, \{2, 3\})) \\
&= \min(35, 40, 43) \\
&= 35,
\end{aligned}$$

so the minimal length tour has length 35.

The total time required by these computations can be broken up into three parts:

- i. It takes  $n - 1$  time to fill in the  $c(v_j, \emptyset)$  values, since  $j = 2, \dots, n$ .
- ii. To calculate all of the  $c(j, S_k)$  values,  $1 \leq k \leq n - 2$ , first note that each step (i.e., for each value of  $k$ ) there will always be  $(n - 1)$  ways to choose  $v_j$ . Next we have to consider all possible subsets of size  $k$  that can make up the  $S_k$  subset. There are  $\binom{n-2}{k}$

such subsets. Finally, for each of these subsets there will be  $k$  addition operations inside the minimum function. So the total number of additions is given by

$$\sum_{k=1}^{n-2} (n-1) \binom{n-2}{k} k$$

and the total space required to store the intermediate results is

$$\sum_{k=1}^{n-2} (n-1) \binom{n-2}{k} k$$

iii. There are  $n - 1$  additions required to compute  $c(1, V - \{v_1\})$ . The total computation time is bounded by the second part. That is,

$$\begin{aligned} T(n) &= \Theta\left(\sum_{k=1}^{n-2} (n-1)k \binom{n-2}{k}\right) \\ &= \Theta(n^2 2^n) \end{aligned}$$

since  $\sum_{k=1}^m m \binom{m}{k} = m 2^{m-1}$ . The space requirements are  $\Theta(n 2^n)$ .

To get an idea as to how much this saves us over the brute-force approach, consider the following table:

| $n$ | Time:<br>Brute Force<br>$n!$ | Time:<br>Dynamic Prog.<br>$n^2 2^n$ | Space:<br>Dynamic Prog.<br>$n 2^n$ |
|-----|------------------------------|-------------------------------------|------------------------------------|
| 5   | 120                          | 800                                 | 160                                |
| 10  | 3,628,800                    | 102,400                             | 10,240                             |
| 15  | $1.31 \times 10^{12}$        | 7,372,800                           | 491,520                            |
| 20  | $2.43 \times 10^{18}$        | 419,430,400                         | 20,971,520                         |