

# CS 580 HW 1

Ankur Dhoot

February 10, 2017

## Q1

The solution to this will resemble the Floyd-Warshall algorithm, except for the fact that we may allow one backward edge on any path.

Let  $b_{ij}^k$  denote the shortest path from  $i$  to  $j$  using only intermediate vertices  $\{1 \dots k\}$  with *exactly* one backward edge on the path.

As in the Floyd-Warshall algorithm, let  $d_{ij}^k$  denote the shortest path from  $i$  to  $j$  using only intermediate vertices  $\{1 \dots k\}$  (where all edges on the path go in the prescribed direction).

Finally, let  $f_{ij}^k$  denote the shortest path from  $i$  to  $j$  using only intermediate vertices  $\{1 \dots k\}$  and *at most* one backward edge.

As in the Floyd-Warshall algorithm:

$$d_{ij}^k = \begin{cases} w_{ij} & k = 0 \\ \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}) & k \geq 1 \end{cases}$$

The difference now comes when we're looking for shortest paths that contain *exactly* one backward edge on the path. If  $k$  is not an intermediate vertex on path  $p$ , then all intermediate vertices are in the set  $\{1, 2, \dots, k-1\}$ . If  $k$  is an intermediate vertex of path  $p$ , then we can decompose  $p$  into  $i \rightarrow k \rightarrow j$ . Let  $p_1$  denote the path  $i \rightarrow k$  and  $p_2$  denote the path  $k \rightarrow j$ . All intermediate vertices on  $p_1$  and  $p_2$  must be in  $\{1 \dots k-1\}$ . Since path  $p$  has exactly one backward edge, either  $p_1$  has a backward edge, or  $p_2$  has a backward edge, but not both. That is, either  $p_1$  is a shortest  $ik$  path with no backward edges and  $p_2$  is a shortest  $kj$  path with one backward edge, or  $p_1$  is a shortest  $ik$  path with one backward edge and  $p_2$  is a shortest  $kj$  path with no backward edges. The above description lends naturally to the following dynamic programming equations:

When  $k = 0$ , there is no path with exactly one backward edge between  $i$  and  $j$ . When  $k = 1$ , there is a backward path between  $i$  and  $j$  only if  $w_{ji} < \infty$ .

Thus,  $b_{ij}^k =$

$$\begin{cases} \infty & k = 0 \\ w_{ji} & k = 1 \\ \min(b_{ij}^{k-1}, d_{ik}^{k-1} + b_{kj}^{k-1}, b_{ik}^{k-1} + d_{kj}^{k-1}) & k \geq 2 \end{cases}$$

Finally, since an  $ij$  path can have *at most* one backward edge, we simply take the minimum of the shortest path without backward edges and the shortest path with *exactly* one edge. That is,

$$f_{ij}^k =$$

$$\begin{cases} \min(d_{ij}^k, b_{ij}^k) & k \geq 0 \end{cases}$$

The final result we're looking for will be  $f_{ij}^n$ . The algorithm will be identical to the Floyd-Warshall algorithm given in CLRS 25.2 except the inner for loop will have three calculations:  $d_{ij}^k$ ,  $b_{ij}^k$ , and finally,  $f_{ij}^k$ .

If we're looking to reconstruct the shortest path, we can compute predecessor matrices as in CLRS 25.2. Let  $D_{ij}^k$  denote the predecessor of  $j$  on a shortest  $ij$  path (along forward edges) with all intermediate vertices in the set  $\{1, 2, \dots, k\}$ . Let  $B_{ij}^k$  denote the predecessor of  $j$  on a shortest  $ij$  path with *exactly* one backward edge and with all intermediate vertices in the set  $\{1, 2, \dots, k\}$ . Let  $F_{ij}^k$  denote the predecessor of  $j$  on a shortest  $ij$  path with *at most* one backward edge and with all intermediate vertices in the set  $\{1, 2, \dots, k\}$ . The final matrix we're interested in is  $F^n$ .

It's easy to see how to update these predecessor matrices based on CLRS 25.2:

$$D_{ij}^0 =$$

$$\begin{cases} NIL & i = j \text{ or } w_{ij} = \infty \\ i & i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

$$D_{ij}^k =$$

$$\begin{cases} D_{ij}^{k-1} & d_{ij}^{k-1} \leq d_{ik}^{k-1} + d_{kj}^{k-1} \\ D_{kj}^{k-1} & d_{ij}^{k-1} > d_{ik}^{k-1} + d_{kj}^{k-1} \end{cases}$$

$$B_{ij}^0 = NIL$$

$$B_{ij}^1 =$$

$$\begin{cases} NIL & i = j \text{ or } w_{ji} = \infty \\ i & i \neq j \text{ and } w_{ji} < \infty \end{cases}$$

$$B_{ij}^k =$$

$$\begin{cases} B_{ij}^{k-1} & b_{ij}^{k-1} \leq d_{ik}^{k-1} + b_{kj}^{k-1} \text{ \& } b_{ij}^{k-1} \leq b_{ik}^{k-1} + d_{kj}^{k-1} \\ B_{kj}^{k-1} & d_{ik}^{k-1} + b_{kj}^{k-1} \leq b_{ij}^{k-1} \text{ \& } d_{ik}^{k-1} + b_{kj}^{k-1} < b_{ik}^{k-1} + d_{kj}^{k-1} \\ D_{kj}^{k-1} & b_{ik}^{k-1} + d_{kj}^{k-1} \leq b_{ij}^{k-1} \text{ \& } b_{ik}^{k-1} + d_{kj}^{k-1} < d_{ik}^{k-1} + b_{kj}^{k-1} \end{cases}$$

$$F_{ij}^k =$$

$$\begin{cases} D_{ij}^k & d_{ij}^k \leq b_{ij}^k \\ B_{ij}^k & d_{ij}^k > b_{ij}^k \end{cases}$$

The time complexity is clearly the same as Floyd-Warshall. We'll have three nested for loops going from 1 to n giving a time complexity of  $O(n^3)$ . We're storing 6 three dimensional matrices (3 for distance, 3 for predecessors) giving a space complexity of  $O(n^3)$ .

## Q2

(a) Essentially, we're solving a knapsack problem with two knapsacks, one of capacity  $M_1$  and the other of capacity  $M_2$  where  $m_i$  denotes the weight and  $u_i$  the profit of item i. We'll use dynamic programming to compute the information needed in a bottom up fashion.

Let  $K_1$  and  $K_2$  represent the first and second knapsacks, respectively. Let  $w(K_1)$  denote the combined weight of all items in  $K_1$ . Analogously for  $K_2$ .

We'll let  $dp[i,j,k]$  denote the maximum profit attainable using the first i items, where  $w(K_1) \leq j$  and  $w(K_2) \leq k$ .

The dynamic programming equations are clear:

$$dp[i, j, k] =$$

$$\begin{cases} 0 & i = 0 \\ dp[i-1, j, k] & w_i > j, w_i > k \\ \max(dp[i-1, j, k], dp[i-1, j-m_i, k] + u_i) & w_i \leq j, w_i > k \\ \max(dp[i-1, j, k], dp[i-1, j, k-m_i] + u_i) & w_i \leq k, w_i > j \\ \max(dp[i-1, j, k], dp[i-1, j-m_i, k] + u_i, dp[i-1, j, k-m_i] + u_i) & w_i \leq j, w_i \leq k \end{cases}$$

The maximum profit attainable is 0 if there are no items. If the weight of item i is larger than j or k, then we can't add the item to either knapsack. The next two cases deal with the weight being larger than either j or k, but not both. The last case occurs when we can add item i to  $K_1$  or  $K_2$  so we must select among the max of not adding item i, or adding it to one of  $K_1$  or  $K_2$ .

The algorithm follows naturally (u is the array of profits (utility) and m is the array of weights (memory))

The algorithm is a verbatim copy of the dynamic programming equations, except for the inclusion of the bp array, which holds the optimal choice for each (i,j,k) triple. If the optimal solution is not to add item i  $\rightarrow$   $bp[i,j,k] = 0$ . If the optimal solution is to add item i to  $K_1 \rightarrow bp[i,j,k] = 1$ . If the optimal solution is to add item i to  $K_2 \rightarrow bp[i,j,k] = 2$ .

---

**Algorithm 1** Optimize the profit of two knapsacks

---

```
1: function KNAPSACK( $M_1, M_2, u, m$ )
2:   Let  $dp[1...n, 1...M_1, 1...M_2]$ 
3:   Let  $bp[1...n, 1...M_1, 1...M_2]$ 
4:   for  $j = 1$  to  $M_1$  do
5:     for  $k = 1$  to  $M_2$  do
6:        $dp[0, j, k] = 0$ 
7:        $bp[0, j, k] = \text{NIL}$ 
8:     end for
9:   end for
10:  for  $i = 1$  to  $n$  do
11:    for  $j = 1$  to  $M_1$  do
12:      for  $k = 1$  to  $M_2$  do
13:        if  $w_i > j \ \& \ w_i > k$  then
14:           $dp[i, j, k] = dp[i-1, j, k]$   $bp[i, j, k] = 0$ 
15:        else if  $w_i \leq j \ \& \ w_i > k$  then
16:           $dp[i, j, k] = \max(dp[i-1, j, k], dp[i-1, j - m_i, k] + u_i)$ 
17:           $bp[i, j, k] = 0$  if not included, 1 if included in  $K_1$ 
18:        else if  $w_i > j \ \& \ w_i \leq k$  then
19:           $dp[i, j, k] = \max(dp[i-1, j, k], dp[i-1, j, k - m_i] + u_i)$ 
20:           $bp[i, j, k] = 0$  if not included, 2 if included in  $K_2$ 
21:        else
22:           $dp[i, j, k] = \max(dp[i-1, j, k], dp[i-1, j - m_i, k] + u_i, dp[i-1,$ 
23:             $j, k - m_i] + u_i)$ 
24:           $bp[i, j, k] = 0$  if not included, 1 if included in  $K_1$ , 2 if
25:            included in  $K_2$ 
26:        end if
27:      end for
28:    end for
29:  end for
30:  return  $dp, bp$ 
31: end function
```

---

---

**Algorithm 2** Find the items in each knapsack

---

```
1: function PACKKNAPSACK(dp, bp)
2:   Let  $K_1$  and  $K_2$  be knapsacks
3:    $k = M_1, j = M_2$ 
4:   for  $i = n$  to 1 do
5:     if  $bp[i, j, k] == 1$  then
6:       Add item  $i$  to  $K_1$ 
7:        $j = j - m_i$ 
8:     else if  $bp[i, j, k] == 2$  then
9:       Add item  $i$  to  $K_2$ 
10:       $k = k - m_i$ 
11:    end if
12:  end for
13:  return  $K_1, K_2$ 
14: end function
```

---

The knapsack construction is straightforward given the optimal solution for each  $(i, j, k)$  triple.

Runtime: The algorithm does constant amount of work inside the triply nested for loop, so the time complexity is clearly  $O(n * M_1 * M_2)$ . We keep the  $dp$  and  $bp$  arrays which use space  $O(n * M_1 * M_2)$ . Constructing the knapsacks takes  $O(n)$  time since we just do constant amount of work inside the for loop. Thus, the total runtime is  $O(n * M_1 * M_2)$ .

**(b)** Let  $|K_1|$  denote the number of items in  $K_1$ . Similarly for  $K_2$ . To satisfy the requirement that both knapsacks contain the same number of items, we'll modify the solution from (a) slightly.

We'll now have  $dp[i, j, k, l]$  denote the maximum profit using the first  $i$  items where where  $w(K_1) \leq j$  and  $w(K_2) \leq k$  and  $l = |K_2| - |K_1|$ . That is,  $l$  denote how many more items are in  $K_2$  than in  $K_1$ . Our goal is to compute  $dp[n, M_1, M_2, 0]$ .

It's clear that for a given  $i$ ,  $l$  ranges from  $-i$  to  $i$  since one knapsack can't have more than  $i$  items more than the other knapsack.

The dynamic programming equations follow:  $dp[i, j, k, l] =$

$$\begin{cases} 0 & i = 0, l = 0 \\ \max(dp[i-1, j, k, l], dp[i-1, j-m_i, k, l+1] + u_i, dp[i-1, j, k-m_i, l-1] + u_i) & \text{otherwise} \end{cases}$$

The above dynamic programming equations use the following convention: We always only take the max over  $dp[i', j', k', l']$  that are defined. If all the  $dp[i', j', k', l']$  values that we are taking the max over are undefined, then we define  $dp[i, j, k, l]$  to be undefined (not 0). We do this in order to maintain that  $l = |K_2| - |K_1|$ . If we were to let undefined values be 0, we wouldn't

be able to distinguish between the cases where we can and cannot achieve differences of 1. (e.g if we let the base case be  $dp[0,j,k,l] = 0$  instead of  $dp[0,j,k,0] = 0$ , then we wouldn't know that we can't actually achieve a difference of  $|l| > 1$  with 0 items even though the equation is well defined)

There are three options for any item  $i$ :

We either don't add it to  $K_1$  or  $K_2$ . We add it to  $K_1$  and then look at the optimal solution with capacity  $j - m_i$  and a difference of  $l + 1$  (since adding an item to  $K_1$  would then give us a difference of  $l$ ). Or, we add it to  $K_2$  and then look at the optimal solution with capacity  $k - m_i$  and a difference of  $l - 1$  (since adding an item to  $K_2$  would then give us a difference of  $l$ )

It's clear that this modification will add a for loop for  $l$  after the for loop for  $i$ . This inner  $l$  loop will range from  $-i$  to  $i$ . This will add an additional time complexity factor of  $n$  giving us a total runtime of  $O(n^2 * M_1 * M_2)$ . We still keep the arrays needed to compute the value of an optimal solution and the arrays needed in order to construct such a solution except these will also have an  $l$  index which will be  $|l| \leq n$ . Thus, the space complexity becomes  $O(n^2 * M_1 * M_2)$ .