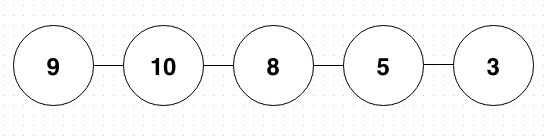
*CSE 417 HW 7*

YANG ZHANG 1030416 zhy9036@uw.edu

**Problem 1**

**(a)** Consider the graph below:

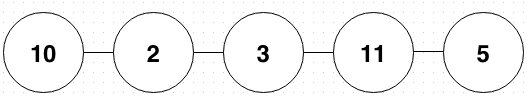
****

The gready algorithm will pick **10** at first round and then delete **9** and **8.** The 2nd round, it will pick **5.** Therefore, the gready algorithm emits the final weight of **15.**

However, the optimal solution is to select **9** first deleting **10**, and then select **8** deleting **5**, finally select 3, which has the final total weight of **20**.

Therefore, this gready algorithm is not optimal.

**(b)** Consider the graph below:



According the algorithm in (b),

S1 will select all Vi where i is odd. Thus S1 will select **10**, **3**, **5** that has total weight of **18**.

S2 will select all Vi where i is even. Thus S2 will select **2**, **11** that has total weight of **13**.

Since Max{weight(S2), weight(S1)} = weight(S1), therefore **S1** will be returned.

However, the optimal solution is to select **10** and **11**, and ends up with **21** as total weight.

Therefore, the algorithm in (b) is not optimal.

**(c) The optimal solution using DP:**

For any given path ***G = (V,E)***, and for ***i from 0 to n***, let ***Gi = (Vi , Ei )*** be the sub path of G where ***Vi = {V1 to Vi}*** and ***Ei  = {{Vj , Vj+1} | 1 <= j < i}***

This algorthim will use table ***OPT(i)*** to store the total weight of the maximum independent set for the path ***Gi***.

**Where:**

**OPT[i] = *0*, if i = 0**

**OPT[i] = *w1*, if i = 1**

**OPT[i] = Max{*OPT[i-1], OPT[i-2] + wi*}, if i > 1**

Here goes the pseudo-code:

**def buildDP(OPT,V)**

**OPT[0] 🡨 0**

**OPT[1] 🡨 V1.w**

**for i from 2 to n:**

**OPT[i] 🡨 Max{OPT[i-1], OPT[i-2] + Vi.w}**

**endfor**

**endfunc**

For traceback, there are two cases:

**if OPT[i] = OPT[i-1], which means Vi**

**else Vi is in solution set**

Here goes the pseudo-code for traceback:

**def traceback(S,OPT,V)**

**for i from n to 1:**

**if OPT[i] != OPT[i-1]:**

**S.append(Vi)**

**endif**

**endfor**

**return S**

**endfunc**

**Proof of Correctness:**

For any given path G, if the size of G is 0 then the optimal weight of indenpent set is 0, since there is no elements along the path. If the size of G is 1, then optimal weight of indenpent set is the weight of the only element, since there is one element along the path (no other option). For size of G that ***i*** is bigger than 1, there will be two cases, the independent set with element ***i*** or without ***i***, therefore we compare the two cases to select the optimal solution for current size of path. Thus, we follow the logic and keep tracking on the sub path until expending to the whole path, the solution found by this algorithm is optimal.

**RunTime:**

Since there is a n-steps for loop, and inside the for loop we are checking the already cached table (O(1)), the overall runtime of this algorithm is **O(n).**

**Problem 2**

**(a)** Consider the following case:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Week2 | Week3 | Week4 | Week5 |
| LOW | 10 | 5 | 10 | 5 |
| HIGH | 20 | 20 | 200 | 50 |

Using the algorithm from part a, it will pick ***High job for Week3*** since ***20 > 10 + 5***, and then pick ***hight job for Week 5*** since ***50 > 5 + 10***. Thus, this algorithm will emit ***70*** as the final answer.

However, the optimal plan is to ***select high job for week1*** and then select ***high job for week4*** finally ***select low job for week5***, which emits ***225*** as final answer.

Therefore the algorithm from part a is **NOT OPTIMAL**

**(b) The optimal solution using DP:**

This algorthim will use table ***OPT(i)*** to store the maximum revenue up to week ***Wi***.

**Where:**

**OPT[0] = *0*,**

**OPT[1] = Max{*W1.low*, *W1.high* },**

**OPT[i] = Max{*OPT[i-1] + Wi.low, OPT[i-2] + Wi.high*}, for n >= i > 1**

Here goes the pseudo-code:

**def buildDP(OPT,W)**

**OPT[0] 🡨 0**

**OPT[1] 🡨 Max{W1.low, W1.high}**

**for i from 2 to n:**

**OPT[i] 🡨 Max{OPT[i-1] + Wi.low, OPT[i-2] + Wi.high}**

**endfor**

**endfunc**

For traceback, there are two cases:

**if OPT[i] = OPT[i-1] + Wi.low, which means at Week i low job will be**

**selected.**

**otherwise high job will be selected and 0 for week i -1**

Here goes the pseudo-code for traceback:

**def traceback(n,OPT,W,S)**

**while n > 0:**

**if OPT[i] == OPT[i-1] + Wi.low:**

**S[i] 🡨 Wi.low**

**n--**

**else:**

**S[i] 🡨 Wi.high**

**S[i-1] 🡨 0**

**n-=2**

**endwhile**

**return S**

**endfunc**

**Proof of Correctness:**

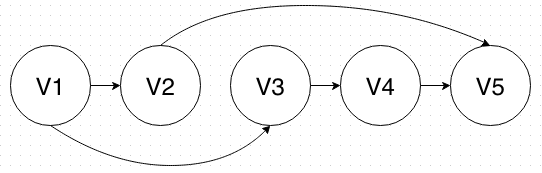
For any given schedule **S**, if the size of sub schedule **s** is 0 then the optimal plan of **s** is 0, since there is no job in the plan. If the size of **s** is 1, then optimal plan is bigger one between low job and high job, since for the first week both kinds of jobs can be selected. For **s** that size ***i*** is bigger than 1, there will be two cases, choose low job for week ***i*** or choose high job for week ***i*** and NONE for week ***i-1*** , therefore we compare the two cases to select the optimal solution for current size of plan. Thus, we follow the logic and keep tracking on the sub plan until expending to the whole path, the solution found by this algorithm is optimal.

**RunTime:**

Since there is a n-steps for loop, and inside the for loop we are checking the already cached table (O(1)), the overall runtime of this algorithm is **O(n).**

**Problem 3**

**(a)** Consider the following ordered graph:



Using the algorithm from part a, it will pick ***V2 for the first round*** since ***2 < 5***, and then pick ***V5*** since ***V5*** ***is the only option.*** Thus, this algorithm will emit ***2*** as the final answer.

However, the optimal plan is to ***V3 for the first round*** and then select ***V4 for the second round*** finally ***select V5***, which emits ***3*** as final answer.

Therefore the algorithm from part a is **NOT OPTIMAL**

**(b) The optimal solution using DP:**

This algorthim will use table ***OPT(i)*** to store the longest path from ***Vi*** to ***Vn***

Here goes the pseudo-code:

**Glode Inilization: OPT[i] 🡨 0**

**def buildDP(OPT,vi)**

**if OPT[i] != 0:**

**return OPT[i]**

**endif**

**level = []**

**for every v in neighbor(v):**

**level.append(1 + buildDP(OPT,v))**

**endfor**

**OPT[i] = level.popMax()**

**return OPT[i]**

**endfunc**

For traceback, we simply use BFS to select the neighbor with bigest OPT value, until reach goal vertax

Here goes the pseudo-code for traceback:

**def traceback(S,OPT,v)**

**if v != goal:**

**for every neighbor vi:**

**find the Vi with biggest OPT[i]**

**endfor**

**S.append(Vi)**

**return traceback(S,OPT,Vi)**

**else:**

**S.append(V)**

**return S**

**endfunc**

**Proof of Correctness:**

For any given graph **S**, we just simply apply BFS, for each level of serch we cache the result into a table, and from the table we choose the path with maxmun value. Therefore, the solution found by the algorithm is optimal.

**RunTime:**

Since there is a n-steps for loop, and inside the for loop we are checking the already cached table (O(1)), the overall runtime of this algorithm is **O(n).**

**Problem 4**

The efficient algorithm using DP:

**let slack(i,j) = L – (j-i) – sum(ci)**

**OPT(0) = 0**

**OPT(j) = min{slack(i,j)^2 + OPT(j-1)} where 1 <= I <=j**

Here goes the pseudo-code:

**def buildDP(OPT, c, slack):**

**for i from 1 to n:**

**OPT[i] 🡨 infinity**

**slack 🡨 L - ci**

**j** 🡨 **i-1**

**while slack > 0**

**OPT[i] 🡨 min(OPT[i], OPT[j] + slack2)**

**slack 🡨 slack – 1 – cj**

**j 🡨 j-1**

**endwhile**

**endfor**

**return OPT**

**endfunc**

**Proof of Correctness:**

For every line, we are testing each possible combinations of letters, and get min slack square for each line, therefore the solution found by this algorithm is optimal.

**RunTime:**

There are n iterations in the outer loop, and at most i iterations in the inner loop, so the algorithm takes at most **O(n^2)** time.

**Problem 4**

**OPT Table:**

**w1 = 5, w2 = 2, w3 = 4, w4 =3 w5 = 6**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 1 | **0** | **0** | **0** | **0** | **5** | **5** | **5** | **5** | **5** | **5** | **5** | **5** | **5** | **5** | **5** | **5** |
| 2 | **0** | **2** | **2** | **2** | **5** | **5** | **7** | **7** | **7** | **7** | **7** | **7** | **7** | **7** | **7** | **7** |
| 3 | **0** | **2** | **2** | **4** | **5** | **5** | **7** | **7** | **9** | **9** | **9** | **9** | **9** | **9** | **9** | **9** |
| 4 | **0** | **2** | **3** | **4** | **5** | **5** | **7** | **8** | **9** | **10** | **11** | **12** | **12** | **14** | **14** | **14** |
| 5 | **0** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** |

The optimal solution is to select: **{*w1, w2, w4, w5*}**, which has the total weight of **16**

**Problem 4**

*Unlimited Supply Knapsack Problem Using DP:*

There are two cases for unlimited supply KP for each item ***i*:**

**case 1: not selecting item i**

**case 2: at least selecting one item i**

Thus, the final decision depends on the **max{*case1, case2*}**

Let **OPT[w]** donate the maxmum value up to weight **w**

then:

**OPT[w] = max{*OPT[w]*, *OPT[w – W[i]] + V[i]*}**

And for traceback, if OPT[w] = ***OPT[w – W[i]] + W[i]***, then we know that item **i** has been selected. We could use matrix S to store it by setting S[w][i] **🡨** 1

Here goes the pseudo-code:

**def buildDP(OPT, V, W):**

**for i from 1 to n:**

**OPT[i] 🡨 0**

**endfor**

**for n from 1 to w:**

**for k from W[i] to m:**

**OPT[k] = max{OPT[k], OPT[k – W[n]] + V[n]}**

**endfor**

**endfor**

**return OPT**

**endfunc**

**def traceback(OPT, V, W, S):**

**for n from 1 to w:**

**for k from W[i] to m:**

**if OPT[k] == OPT[k – W[i]] + V[i]:**

**S[n][k] 🡨 1**

**endif**

**endfor**

**endfor**

**return S**

**endfunc**

**Proof of Correctness:**

For any item **i,** there will be two case, the optimal current solutoin without **i,** or the optimal solution with **i.** If it is include **i,** we are changing the state that before item **i**  appears, which is OPT[j – weight[i]], so we compare the two cases and select the bigger one, which is the optimal solution.

**RunTime:**

There are n iterations in the outer loop, and at most k iterations in the inner loop, so the algorithm takes at most **O(n\*k)** time.