COT5405 - Analys of Algorithms
Final-Exam

Qinxuan Shi
UFID: 83518162

April 20, 2023

1 Problem 1

1.1 Pseudo-code of the Algorithm

Algorithm 1 Max total value

Input:

Integer W, capacity of knapsack
Integer T, time the police will arrive
Integer n, number of the item in the store
v[n], v[i] means the value of the item with index i
w[n], w[i] means the weight of the item with index i
t[n], t[i] means the time will be grabbed of the item with index i

Output:

Define that dp[n+1][W+1][T+1] to save the maximum total value.
And dp[i][j][t] means for the first i items that can be selected, when the backpack capacity is j and the arrival time is t, the maximum value of all choices.
Initial dp[i][j][0], dp[i][0][t] and dp [0][j][t] to 0.

```
for i= 0 to n do
    for j=0 to W do
        dp[i][j][0] ← 0
    end for
end for

for i= 0 to n do
    for t=0 to T do
        dp[i][0][t] ← 0
    end for
end for

for j= 0 to W do
    for t=0 to T do
        dp[0][j][t] ← 0
    end for
end for

for t= 0 to T do
    for j= 0 to W do
        for i= 0 to n do
            if w[i]>j ∨ t[i]>t then
                dp[i][j][t] ← dp[i-1][j][t]
            else if w[i]≤ j ∧ t[i]≤ t then
                dp[i][j][t] ← max(dp[i-1][j][t], dp[i-1][j-w[i]][t-t[i]] + v[i])
            end if
        end for
    end for
end for
return dp[n+1][W+1][T+1].
```

Algorithm 2 get the items chosen(dp[][][])

Input:
get dp[n+1][W+1][T+1] from algorithm above
Output:
i=n, j=W, t=T, start from the last element of dp.
initial S to be the set of all items that are chosen.
while i >0 **do**
if dp[i-1][j][t] = dp[i][j][t] **then**
i ← i-1
else if dp[i-1][j][t] ≠ dp[i][j][t] **then**
S ← S ∪ i
i ← i-1
j ← j-w[i]
t ← t-t[i]
end if
end while
return S.

1.2 Proof of the algorithm's correctness

According to the problem, the goal is to get the max total value under the constriction of capacity of knapsack W and the arriving time of police T. so we use OPT(i, j, t) to represent for the first i items that can be selected, when the backpack capacity is j and the arrival time of police is t, the maximum value of all choices.

Compare to the kanpsack problem, we consider the following cases:

Firstly, when the weight of the item of index i is greater than the capacity of the backpack j, the item cannot be loaded into the backpack. Then for the first i items that can be selected and the first i-1 items that can be selected, their maximum total value and the way they choose are the same.

Secondly, there are two options when the item of index i can be loaded into the backpack.

1. Put it into the backpack. Under this condition, the maximum total value is $OPT(i-1, j-w[i], t-t[i]) + v[i]$. $OPT(i-1, j-w[i], t-t[i])$ represents for the first i-1 items that can be selected, when the backpack capacity is j-w[i] and the arrival time of police is t-t[i], the maximum value of all choices.

2. Not put it into the backpack. Under this condition, the maximum total value is the same as OPT(i-1, j, t).

Bellman equation:

$$\text{OPT(i, j, t)} = \begin{cases} 0, & \text{if i=0} \vee \text{j=0} \vee \text{t=0} \\ OPT(i-1, j, t), & \text{if w[i]>j or t[i]>t} \\ max\{OPT(i-1, j-w[i], t-t[i]) + v[i], OPT(i-1, j, t)\}, & \text{if w[i]}\leq \text{ j and t[i]}\leq \text{ t} \end{cases}$$

To the Algorithm 2, in order to get the items that are chosen to get the maximum total value, we start from i=n, j=W and t=T, which is the last element of dp matrix.

If dp(i, j, t) = dp(i-1, j, t), which means that the item i is not chosen, so we start from dp(i-1, j, t) to keep finding. If dp(i, j, t) ≠ dp(i-1, j, t), which means that the item i is chosen, so the next item should start from dp(i-1, j-w[i], t-t[i]). When the i = 0, it means that the value is 0, there is no item in the kanpsack, so the program end.

1.3 Algorithm's running time

According to the Algorithm Max total value, the first three for-loops are going to initialize the dp matrix which is used to memorize the maximum total value of each dp[i][j][t]. Since each of them uses two for-loops, the complexity is O(nW), O(nT) and O(WT).

To the forth for-loop, which is used to compute each dp[i][j][t]. According to the pesudocode, there are three for-loops in the forth for-loop, compare to the knapsack problem in the lecture, it can exactly form a three-dimentional table and get O(nWT) table entries.

According to the algorithm 2, the worst situation is that all of the n items are chosen, so the algorithm takes O(n) time to get the set of items that form the maximum total value.

As a result, the algorithm 1 solves the problem in O(nWT) time and O(nWT) space and the algorithm 2 solves the problem in O(n) time.

```
def generator(z):
    G_h1 = tf.nn.relu(tf.matmul(z, G_W1) + G_b1)
    G_log_prob = tf.matmul(G_h1, G_W2) + G_b2
    G_prob = tf.nn.sigmoid(G_log_prob)

    return G_prob
```

Figure 1:

2 Problem 2

2.1 Pseudo-code of the Algorithm

Algorithm 3 Minimum number of edges

Input: G(V, E) undirected graph, s,t∈V, suppose that each c(e)=1
Output:
let G'(v', E') be a directed graph, V' = V and initial E' = ∅
for each (v1, v2)∈ E **do**
E' = E'∪{v1→v2}∪{v2→v1}
end for

Using Edmonds-Karp(G')
for e∈E' **do**
f(e) ← 0
end for
G_f← residual network of G' with respect to flow f.
while there exists an s→t path in G_f **do**
P ← BFS(G_f).
f ← Augment(f, c, P).
update G_f.
end while

BFS(G_f), start from s, get the set of nodes reachable from s in G_f as V_m.

let M to save the edges that belong to the minimum cut .
for each edge e(v1,v2)∈E **do**
if v1∈ V_m and v2∉ V_m
M← M ∪ e(v1,v2)
end for

2.2 Proof of the algorithm's correctness

Refer to the edge-connectivity version of Menger's theorem, we can know that the size of the minimum edge cut for x and y is equal to the maximum number of pairwise edge-independent paths from x to y. Because the graph has no weight, so suppose that c(e)=1, and we can also get the lemma of Menger's theorem that the maximum flow flow of G equals to the maximum number of pairwise edge-independent paths from x to y.

Then we can know that even though the problem wants to get the minimum number of edges that can be disconnected to partition the graph, it exactly equals to get the edges that make the flow f to be the maximum flow flow of G, and these edges form the minimum cut.

According to the pseudocode, firstly, I turned the undirected graph to the directed graph by turning the edge (v1, v2) to two directed edges v1→ v2 and v2→ v1. Then we can get a directed graph G', and we can easily get the maximum flow, here we use the Edmonds-Karp algorithm to get the maximum flow and minimum cut.

After that, we can also get the final state graph G_f, and we do the BFS search to get all the vertex belong to the set of minimum cut.

Finally, we get set of the edges M that have vertex belong to different partitions, which is the set of edges that form the minimum cut. According to the Menger's theorem and lemma above, M equals to the minimum number of edges that can be disconnected to partition the graph in such a way that two given vertices $s, t \in V$ are in separate partitions.

2.3 Algorithm's running time

According to the algorithm, the for-loop that is used to turn undirected graph to directed graph take O(m) time.

And here since we suppose c(e)=1 for each e∈E, so to the Edmonds-Karp algorithm, we count the number of edges to get the shortest path, the Edmonds-Karp algorithm here takes O(m^2n) time.

Since we have the maximum flow and the final state graph G_f, we can get the set of nodes in O(m) time.

Finally, it also takes O(m) time to get all edges belong to the minimum number of edges.

As a result, the algorithm takes O(m^2n) to solve the problem and get the minimum number of edges.

3 Problem 3

3.1 Proof of the algorithm's correctness

According to the problem, to prove that the given problem X is NP-complete, we need to prove that:

1. X $\in$ NP.

2. prove that Subset-Sum $\leq_p$ X.

1. Firstly, proving that X $\in$ NP. We give an instance of the problem X to solve that:

Suppose that the two parts of the integers are S1 and S2. A is the set of a_i, and the length of A is n.

Verigy that except $b[i] \in A$ in S1 and $c[i] \in A$ in S2, there is no other element in A that does not belongs to these two sets. It takes O(n) time.

Compute the sum of both sets S1 and S2. It takes O(n) time.

See if the sum of S1 equals to the sum of S2. It takes O(1) time.

As a result, the certifier finishes in O(n) time, which belongs to poly-time. So the problem X $\in$ NP.

2. Secondely, we are going to prove that Subset-Sum $\leq_p$ X.

The Subset Problem provides the input as a set of numbers S and a target sum t, the problem aims at finding the subset A belonging to S with a sum equals to t. Let s be the sum of members of S. Let A' be the set of numbers equals to S-A. So the sum of A' is s-t.

According to the problem X, the set of numbers S is required to divide into two parts S1, S2 and the sum of S1 and S2 are equal.

We are going to construct problem set-partition by using Subset-Sum problem. Exactly, for all the instance of Subset-Sum problem, when the target sum t=s/2, the sum of A equals to the sum of A', it becomes the form of instance of problem set-partition.

What's more, when the instance y is a yes of problem set-partition, which means that there exists two subset S1 and S2 have the sum of s/2, make the y is a yes holds. At this time, to the subset-sum instance x, the target sum t equals to s/2, and the subset A equals to S1 or S2, that the x is a yes holds. When y is not a yes condition, there also cannot find a subset A that has a sum equals to the target t=s/2. For example

To conclude, given any instance x of Subset-Sum problem, we can construct an instance y of set-partition problem such that x is a yes instance of Subset-Sum problem if and only if y is a yes instance of set-partition problem. And y to be of size polynomial in x. So, Subset-Sum $\leq_p$ X holds.

As a result, the problem is NP-complete X P(X | Y).