COT 5405 Analysis of Algorithms Fall 2021 Final Exam

UFID: 96703101
Name: *Vyom Pathak*
Instructor: Professor Alin Dobra
Due Date: December 08, 2021

Problem 1 - Dynamic programming [35]

Solution

Algorithm
The algorithm can be devised as a 3D knapsack problem problem where the bellman equation can be formed in the following way: The current item is selected if we have enough time and enough weight available in the knapsack otherwise we do not select the item. We take the maximum of previous value or this selected item. Then we print the items which are selected by back tracking the 3D-matrix. The algorithm to find the maximum total value with the items is given in Algorithm 1.

Analysis

Proof of Correctness
The algorithm always finds the maximum total value for selected items. We prove this by induction.
Induction Hypothesis: The algorithm is correct for all $i, j, k < i', j', k'$.
Base Case: Here, base condition is that when $i > 0$, $j = 0$ or $k = 0$, $DP[i, j, k] = 0$ which is correct.
Induction Step: For induction hypothesis, let's assume without any loss of generality that the claim holds true for entries $DP[i', j', k']$ such that we have $DP[i' - 1, j', k']$ and $DP[i' - 1, j' - w_{i'}, k' - t_{i'}]$ which are always computed correctly. Then algorithm considers the optimal value for item i' in knapsack as $DP[i' - 1, j' - w_{i'}, k' - t_{i'}] + v_{i'}$ and for item i' not in knapsack as $DP[i' - 1, j', k']$. Therefore, the value at DP[i',j',k'] is correct.

Running Time
The algorithm finds the maximum total value in $O(nWT)$.
Proof: Here, we loop through each item, and for each of those items we loop through each weight W and time T. Thus, the overall time-complexity of the solution is $O(nWT)$. Infact, as the time is dependent on the total weight limit and the total time limit which can be variable, the time complexity is actually in psuedo form. We store a 3-D table of length n, width W, and height T. Thus the overall space-complexity is $O(nWT)$.

Algorithm 1 Function for finding the maximum total item value

```
**Input:** Number of values $n$, value of items $v[]$, weight of items $w[]$, time to grab an item $t[]$, max time available $T$, and max allowed weight $W$
**Output:** Prints the items index selected along with the total value.
**function** 3DKnapsack($n, v[], w[], t[], T, W$):
    Initializing 3D $DP[n+1, W+1, T+1]$ matrix to $-\infty$;
    **for** $i$ in $range(1, n+1)$ **do**
        **for** $j$ in $range(0, W+1)$ **do**
            **for** $k$ in $range(0, T+1)$ **do**
                $tp1 \leftarrow 0$;
                $tp2 \leftarrow DP[i-1, j, k]$;
                **if** $!(w[i-1] > j \ or \ t[i-1] > k)$ **then**
                    $tp1 \leftarrow DP[i-1, j-w[i-1], k-t[i-1]] + v[i-1]$;
                **end if**
                $DP[i, j, k] \leftarrow max(tp1, tp2)$;
            **end for**
        **end for**
    **end for**
    $sol \leftarrow DP[n, W, T]$;
    $print('Total\ value\ of\ the\ knapsack\ is:\ ', sol)$;
    $t\_tp \leftarrow T$;
    $w\_tp \leftarrow W$;
    **for** $i$ in $range(n, 0, -1)$ **do**
        **if** $sol \leq 0$ **then**
            break;
        **end if**
        **if** $sol \neq DP[i-1, w\_tp, t\_tp]$ **then**
            $print('Item:\ ', i-1, '\ was\ selected.')$;
            $sol \leftarrow sol - v[i-1]$;
            $w\_tp \leftarrow w\_tp - w[i-1]$;
            $t\_tp \leftarrow t\_tp - t[i-1]$;
        **end if**
    **end for**
**end function**
$3DKnapsack(5, [120, 129, 250, 130, 119], [36, 25, 50, 45, 20], [3, 10, 5, 4, 1], 5, 60)$;
```

Problem 2 - Network flow [35]

Solution

Algorithm

This problem can be formulated as finding max flow problem for directed graph and then, we can find the edge disjoint path from s to t which is essentially the value of the max flow. Now, this edge disjoint path is equal to the minimum number edges which needs to be removed to make the vertices s and t disconnected. So, we reduce it to a max flow problem by converting the undirected graph into directed graph. After that, we simply find the max flow value which will be our solution for the given problem of finding the minimum number of edges which are needed to remove in-order to make vertices s and t disconnected. We find the max-flow using Ford-Fulkerson along with the path decomposition algorithm. The algorithm is describe in Algorithm 2.

Algorithm 2 Function for finding the minimum number of edges needed to be removed to disconnect two given vertices

```
1: Input: Graph $G$, vertices $s$, and $t$
2: Output: Return the minimum number of edges which are removed from the graph to make the vertices
   $s$ and $t$ disconnected.

3: function FINDMINEDGE($G, s, t$):
4:     for $e \in E$ do
5:         $G[e] \leftarrow G[e][u \rightarrow v], G[v \rightarrow u]$;        ▷ Add directed edges from (u,v) and (v,u)
6:     end for
7:     Set capacity 1 to each edge;
8:     $maxFlowValue \leftarrow fordFulkerson(G, s, t)$;
9:     return maxFlowValue;
10: end function

11: function FORDFULKERSON($G, s, t$):
12:     for $e \in p$ do
13:         $G[e][u \rightarrow v].f \leftarrow 0$;
14:     end for
15:     $G_{res} \leftarrow ResidualNetwork(G)$;
16:     while there exists a path $p$ from $s \rightsquigarrow t$ in $G_{res}$ do
17:         $cap_f(p) \leftarrow min(cap_f(u,v)|(u,v) \in p)$;
18:         for $edgee \in p$ do
19:             if $edge\ (u,v) \in E$ then
20:                 $G[e][u \rightarrow v].f \leftarrow G[e][u \rightarrow v].f + cap_f(p)$;
21:             else
22:                 $G[e][u \rightarrow v].f \leftarrow G[e][u \rightarrow v].f - cap_f(p)$;
23:             end if
24:         end for
25:     end while
26:     return $cap_f(p)$;        ▷ Return the maximum flow value for the path $p$ from $s \rightsquigarrow t$
27: end function

28: findminEdge($G, s, t$);
```

Analysis
Proof of Correctness
The proposed algorithm correctly finds the minimum number of edges needed to remove to disconnect the vertices s and t. To prove this, we first prove that the problem can be reduced to finding the max-flow problem from source s and destination t with edge-weights as 1 for each edges. Here, the value of the max-flow gives the maximum edge disjoint path count from s to t for the given undirected graph. Next, we prove that, for an undirected graph, the count of maximum number of edge disjoint paths from s to t is equal to the minimum number of edges which when removed disconnects the vertices s and t.
Proof:

Statement 1: Given a undirected graph $G = (V, E)$ and two nodes s and t, we can find the max number of edge-disjoint $s \rightsquigarrow t$ paths.

Proof for Statement 1:
For this statement 1, we firstly do the following max-flow formulation:
Statement 2: Max-flow formulation - Replace each edge with two anti-parallel edges and assign unit capacity to every edge, we can find the max-flow of the undirected graph.
Lemma 1 for Statement 2: In any flow network, there exists a max flow f in which for each pair of anti-parallel edges e and $\hat{e}$; either $e.f = 0$ or $\hat{e}.f = 0$ or zero for both values.
Proof for Lemma 1:
We prove this using induction on no. of each such edge pairs. Suppose $e.f > 0$ and $\hat{e}.f > 0$ for a pair of anti-parallel edges e and $\hat{e}$. We set, $e.f = e.f - \Delta$ and $\hat{e}.f = \hat{e}.f - \Delta$, where $\Delta = min\{e.f, \hat{e}.f\}$. Thus, f is still a flow of the same value but has one fewer such pair.
Hence Lemma 1 is proved which proves that Statement 2 is also true.

Further, we need to connect the max-flow **Theorem 1:** value of the max flow is equal to max no. of edge disjoint $s \rightsquigarrow t$ paths.
Proof for Theorem 1: We can prove this by the following two observations:
Observation 1: There exists a max flow value $\hat{f}$ in Graph G^* that is integral. [We proved this in Lemma 1.]
Observation 2: $\hat{f}$ corresponds to max-number of edge-disjoint $s \rightsquigarrow t$ paths in Graph G using 1 to 1 correspondence.
Proof for Observation 2:
Let f be an integral flow in Graph $\hat{G}$ of value k. Consider edge (p, q) with $f(p, q) = 1$. By the flow conservation law, there exists an edge (p, r) with $f(p, r) = 1$. We continue this until we reach the final node i.e. t, always choosing a new edge. Thus, it produces k edge-disjoint paths.
Hence Observation 2 is proved which implies that Theorem 1 is true. Hence Statement 1 is proved.

Now, as we have proved that we can formulate the undirected graph as a max-flow problem and derive the max number of edge-disjoint path, we will now prove that these edge-disjoint paths are equal to the minimum number of edges required to disconnect the graph.
Menger's Theorem: The max number of edge-disjoint $s \rightsquigarrow t$ paths equals the min number of edges whose removal disconnects t from s.
Proof for Menger's Theorem:
Let's assume that max number of edge disjoints $s \rightsquigarrow t$ paths is k. Then, from Theorem 1, we know that the max flow value $= k$. Now, maximum-flow minimum-cut theorem implies that there exists a (M,N) cut of capacity k. Let $\hat{E}$, be set of edges going from M to N. $|E| = k$ and removing these edges makes vertices s and t disconnected.

Hence Menger's Theorem is proved.

Hence, the original statement is true that the algorithm finds the minimum number of edges that disconnects s and t in a undirected Graph G.
References: Lecture Slides

Running Time
The running time of the proposed algorithm is $O(VE)$, where V is the number of vertices and E is the number of edges.
Proof: Here, the first step of the algorithm to convert the undirected graph into directed which is essentially traversing all the edges at least once which takes $O(E)$ time. The second step is to set the weight of each edge to 1 which again takes $O(E)$ time. Then, we find the max flow value using the ford-fulkerson's algorithm. The running time of this algorithm is $O(Ef)$, where f is the maximum flow possible in a given graph which in our case is bounded by V. Thus, the time complexity is $O(EV)$. This, is because we find each augmented paths in $O(E)$ time which increases the max flow by 1. Thus, it is not a pseudo-time complexity. Now, this max value of the flow is essentially the number of edge-disjoint paths in the given undirected graphs. Which is in evidently the minimum number of edges required to be removed from the graph to disconnect the vertices s and t.
Therefore, the overall running time for all of these functions combined is $O(VE)$.

Problem 3 Complexity [30]

Solution

We first prove that the problem 2-set sum partitioning is in NP class. We then prove that subset-sum problem can be reduced to 2-set sum partitioning problem in polynomial time. Thus, by doing this we imply that the problem 2-set sum partitioning is NP-Hard.

2-set sum partitioning problem is in NP:
Given a set S with a subset $\hat{S}$. We can verify that the subset $\hat{S}$ and $S-\hat{S}$ have the equal sum value by taking the sum of elements of set $\hat{S}$ and $S-\hat{S}$. If the sumamtions are equal we accept the solution. Thus, this non-deterministic algorithm works in polynomial time which proves that 2-set sum partiotioning problem exists in the NP class.
2-set sum partitioning problem is NP-hard:
We do this by doing the following reduction:
Subsetsum reduces to partitioning problem: SubsetSum Problem $\leq_p$ Partition Problem
Reduction: Let's assume that there is a set S whose elements sum-up to value x. Now, let's call the $SUBSETSUM(S,d)$, where we need to find a subset from set S of sum d. Inside, we define a new set $\hat{S}$ as $\hat{S} = S \cup \{2d-x\}$. The sum of the subset $\hat{S}$ is $2d$. Now, further we call the $PARTITION(\hat{S})$. If it returns $TRUE$, we return $TRUE$ for $SUBSETSUM(S,d)$ function, otherwise we return $FALSE$.
This holds water because, there must be some partitions in set $\hat{S}$ into two sets such that each set have sum equal to d. Furthermore, only one of these sets will contain the newly added element $[\{2d-x\}]$. This implies that the other subset of $\hat{S}$ must be a subset of S with sum equal to d. Thus, the solution to the partition problem exists if and only if the solution to the subsetsum problem exists. Thus, subset-sum problem reduces to partition problem.

Furthermore, the formation of set $\hat{S}$ is done in $O(|S|)$ time by summing up the elements of set S and adding

a new element $\{2d - x\}$. This proves that our reduction is performed in polynomial time.
Now, as the subset-sum problem is NP-complete, this proves that 2-set sum partitioning problem is a NP-Hard Problem.

Thus, as Partitioning problem is in NP class and is NP-Hard, it is a NP-Complete problem.