Assignment-2 Submission

Indian Institute of Technology Delhi

COL351: Analysis and Design of Algorithms

Name: Dishant Dhiman Entry Number: 2019CS10347 Group: 03
Name: Deepanshu Entry Number: 2019CS50427 Group: 04

1 Algorithms Design book

The problem states that we have to distribute the chapters among Alice, Bob, and Charlie so that each of them gets to solve nearly an equal number of questions. Let S_1, S_2 and S_3 denote the set containing the chapter number distributed to Alice, Bob and Charlie respectively.
Define sum of exercised corresponding to set S_k as $V_k = \sum_{i \in S_k} x_i$.
We will maintain a boolean dp array valid[i][j][k], $0 \leq$ i $\leq$ n, $0 \leq$ j, k $\leq n^2$.

$$valid[i][j][k] = \begin{cases} True & \text{if } \exists S_1, S_2, S_3 \in [1, i], V_! = j, V_2 = k \\ False & otherwise \end{cases}$$

Following are the initial conditions on the dp array:

- valid[0][0][0] = True.
- valid[0][i][j] = False $\forall$ i,j $\neq 0$

After getting the complete valid array, we need to find the optimal partition. For this, we will iterate back in the array. We will travel along the path where the values are true in the array.

Algorithm 1: Distributing chapters to Alice, Bob and Charlie

```
boolean valid[n][n^2][n^2];
for j in range (0,n^2) do
    for k in range (0,n^2) do
        valid[0][j][k] = false;
    end
end
answer1 = 0;
answer2 = 0;
valid[0][0][0] = true;
for i in range (1,n) do
    for j in range (0,n^2) do
        for k in range (o,n^2) do
            temp1 ⟵ valid[i-1][j][k];
            // no one
            temp2 ⟵ false;
            temp3 ⟵ false;
            if j - x_i ≥ 0 then
                temp2 ⟵ valid[i-1][j-x_i][k];
            end
            if k - x_i ≥ 0 then
                temp3 ⟵ valid[i-1][j][k-x_i];
            end
            valid[i][j][k] ⟵ temp1 ∨ temp2 ∨ temp3;
            if valid[i][j][k] then
                answer1 = j;
                answer2 = k;
            end
        end
    end
end
S_1, S_2, S_3 ⟵ ∅ ;
i ⟵ n;
while i ≥ 0 do
    if valid[i-1][answer1][answer2] then
        S_3 ⟵ S_3 ∪ {i};
    else
        if valid[i-1][answer1 - x_i][answer2] then
            S_2 ⟵ S_2 ∪ {i};
        else
            S_1 ⟵ S_1 ∪ {i};
        end
    end
    i ⟵ i -1;
end
return S_1, S_2, S_3;
```

1.1 Algorithm correctness

The recurence relation for the algorithm is as follows:

$$valid[i][j][k] = \begin{cases} valid[i-1][j][k] \\ OR \\ valid[i-1][j-x_i][k] \\ OR \\ valid[i-1][j][k-x_i] \end{cases}$$

Here, the main idea is that if one of the previous partition without x_i is valid, then including x_i to suitable set will also be in the solution.
We will now prove by induction that the dp array valid returns correct true/false value.

Proof: Proof by induction.

Let the valid array store the boolean value as defined above.

Base case: n = 1

The best and only possibility is adding the element to any one set. Infact, upto n $\leq$ 3, adding the element to different sets is the best possibility to minimise the maximum value.

Induction hypothesis:

Assume that the valid array returns correct boolean value $\forall$ n $\leq$ k.

For n = k+1:

Now, x_{k+1} is to be added to one of the sets. If either of valid[i-1][j-x_i][k] or valid[i-1][j][k-x_i] is valid, this means that adding x_i to the set is possible and hence the OR value returns true. If valid[i-1][j][k] is valid, then we can add x_{k+1} to the third sum and in that case too, the value will become true. If x_{k+1} cannot be added to any of the sets, then every value will become false. Hence, logical or of all the values will also be false.

Hence, valid array is true for n = k +1 whenever induction hypothesis is true. Therefore, by principle of mathematical induction, valid array is true for all n.

Hence proved by induction.

- To get the subset, we are looping through the values which are true. We have maintained answer1 and answer2 to get the starting point of iteration.
- Note that checking if valid[i-1][j][k] is true is same as checking that x_i going to Charlie is true or not.
- The idea of recurrence relation is that if for i-1, sum (for any 1 set) minus x_i is true, the value valid for i is also true.
- We are using the same idea to trace back. We are including that value for which sum minus x_i is true.

- We keep including those values to the set for which valid array returns true. And at each step, we add the value to corresponding set.
- Therefore after n iterations, we will have the three sets filled with values corresponding to which the valid array is true.

1.2 Time Complexity

- In the initialisation phase, we are looping j from 0 to n^2. For each j, we are iterating k from 0 to n^2. There are some other constant time initialisations. Therefore, this process takes O(n^4) time.
- In the main running phase when we are building the array, we are looping on i n times. For each loop, we are running j from 1 to n^2 and for each i and j, we are iterating k from 1 to n^2. Therefore, the running time for the updating phase is O(n^5.
- In the part when we are building the sets S_1, S_2 and S_3, we are running the while loop O(n) times. In the loop, we are doing constant time operations. Therefore, the running time for this part is O(n).

Therefore, the total running time of the algorithm is O(n^5).

2 Course Planner

2.1 Part(a)

The problem states that we are given a set C of n courses that need to be credited for complete graduation in CSE from IIT Delhi. P(c) is a set of prerequisite courses which need to be completed before taking the course c.
We use the above representation as a directed graph G where every course $c_i \in C$ is a node of the graph and the nodes u, v contain an edge from u to v between them if u is a prerequisite of the course v. We also maintain a set O such that O(c) denotes all the nodes which have an incoming edge from the node c in the graph.

Lemma 2.1: *If the graph contains a cycle, then It is not possible to graduate from IIT Delhi.*
Proof. Assume there exists a cycle $c_i, c_{i+1}, ..., c_k, c_i$ in the graph such that graduation is possible. Since this is a cycle then it implies that

$$c_i \longrightarrow c_{i+1} \longrightarrow c_{i+2} \longrightarrow ... \longrightarrow c_k \longrightarrow c_i$$

This ordering means that to complete c_{i+1}, we need to complete c_i first, to complete c_{i+2}, we need to complete c_{i+1} and so on till c_k for which we need to complete c_{k-1} courses.
Overall, the courses we need to complete to take up the course c_k can be represented by

$$S = \{c_i, c_{i+1}.c_{i+2}, ..., c_{k-1}\}$$

Now we have an edge from c_k to c_i, which indicates that c_k is a prerequisite of c_i, and thus all courses in set S need to be completed before c_i.
This leads us to a contradiction as set S contains c_i, i.e. we need to complete c_i before taking up the course c_i which is not possible.
Hence there can be no cycle in a graph where it is possible to graduate from IITD.

lemma 2.2: *The topological ordering of the graph gives the best ordering to complete the courses.*
Proof. Let S be the optimum solution to the given problem which is not a topological ordering. There there exists a back-edge in the ordering of S indicating that the ordering has a pair of courses c_i, c_j such that c_j appears after c_i but c_j is the prerequisite of the course c_i.
Also, the ordering S cannot contain a cycle as proved in lemma 2.1. Thus, there is no path from c_i to c_j. let us assume that the set which have c_i as preequisite be given by

$$Q = \{c_l, c_{l+1}, c_{l+2}, ..., c_k\}$$

Since there is no path from c_i to c_j, there is no path from any node in the set Q because if there was a path, then that path could be used to reach c_j from c_i. Then, we can argue from the above statement that

$$Q \cap P(c_j) = \phi$$

Hence, we can bring all the courses in the set Q after c_j and get an even better ordering than the optimum solution S. This leads us to a contradiction. Hence, every optimum solution of the problem is a topological ordering of the graph.

Algorithm 2: Find order of taking n courses

```
T ⟵ Topological Ordering of the graph G;
arr ⟵ Array for storing the arrival times at each node initialized with -1;
time ⟵ 0;
for c_i ∈ T do
    arr[i] = time++;
    for c_j ∈ P(c_i) do
        if arr[j] != -1 then
            Not possible as the ordering contains a cycle;
            return {};
        end
    end
    T.insert(c_i);
end
return T;
```

Algorithm 3: Topological Ordering of the graph G

```
T ⟵ {};
visited ⟵ initialized to all False;
departureTime ⟵ Initialized to 0;
time ⟵ 0;
for c_i ∈ G do
    if visited[i] == True then
        continue ;
    end
    time++;
    DFS(c_i, time, visited, T);
    T.insert(c_i);
end
reverse(T);
return T;
```

Algorithm 4: DFS(c_i, time, visited, T)

```
for c_j ∈ P(c_i) do
    if visited[j] == True then
        continue;
    end
    visited[i] = True;
    time++;
    DFS(c_j, time, visited, T);
    T.insert(c_j);
end
```

Time Complexity of find order: The finding of order requires the topological ordering of the graph which can be obtained by running a DFS on the graph in O(m+n) time. Then, we perform the cycle check in the ordering obtained, for this every edge in the graph is traversed at most once. Thus, we get to check the cycles in O(m) time. Thus the overall time complexity of the algorithm is O(m+n).

2.2 Part(b)

lemma 2.3: *The minimum number of semesters are obtained by doing all the courses for which the prerequisites have been completed together in a semester.*

Proof. Consider the schedule S which gives the minimum number of semesters be such that the courses with all prerequisites completed are divided into semesters i and j, where i $\neq$ j, i $<$ j. Then all the courses which have the courses in semester j as prerequisite are forced to be done in an additional semester. If the courses done in semester j are done in semester i, then we can obtain an even better schedule which minimizes the number of semesters further. This leads us to the contradiction that the schedule S is optimum. Thus by contradiction, we can say that the number of semesters are minimized by completing all the courses with no prerequisites in the same semester.

lemma 2.4: *There are always some nodes with incoming degree 0, otherwise there is a cycle in the graph.*

Proof. If all the nodes have incoming degree ¿ 0, this implies that every course has a prerequisite course. If this is the case, we cannot complete any course because there is no course for which we satisfy the criteria of prerequisites. In the topological ordering of the graph, the first node must have incoming degree 0. The only way which allows the node to have incoming degree ¿ 0 is if there is a back-edge in the ordering which is not possible (from lemma 2.1).

Algorithm 5: Minimum number of semesters

```
Q ⟵ empty queue for performing BFS];
T ⟵ ;
for c ∈ C do
    if incoming_degree(c) == 0 then
        Q.push(c);
    end
end
while !Q.empty do
    c ⟵ Q.front();
    T.append(c);
    Q.pop();
    for c_j ∈ O(c) do
        incoming_degree(c_j)−−;
        if incoming_degree(c_j) == 0 then
            Q.push(c_j);
        end
    end
end
return T;
```

Time Complexity: The generation of set O takes O(m+n) time. We traverse through all P(c)'s and obtain the set. Then we perform the BFS operation in the graph by filling the initial queue with nodes of incoming degree 0. This takes O(n) time. Then we perform the BFS in O(m+n) time as the number of times the inner for loop is executed is equal to the the total number of edges in the graph and every node is inserted in the queue atmost once. The overall time complexity is O(m+n).

2.3 Part(c)

Lemma 2.5: *DFS on a node visits all the descendants of the node.*
Proof. Discussed in class

Lemma 2.6: *Calling DFS on every node and appending the given node to their prerequisites gives us a complete set of prerequisites*
Proof. Let us say a node t has an incomplete set of prerequisites and the missing prerequisite from the set is C_i. Now, we call DFS on every node, which implies we are calling DFS on C_i as well and appending it to all its descendants. Now, since the node t has C_i as prerequisite, it must be the descendant of the node C_i in its DFS tree(Using lemma 2.5). Thus, C_i must be present in the prerequisite set of t which is a contradiction. Hence, we obtain a complete set after running the DFS on every node.

Algorithm 6: Compute the set P

```
L ⟵ {};
P ⟵ {};
for c ∈ C do
    V ⟵ DFS(c);   // V is the set of vertices visited by DFS(c)
     excluding c
    for v ∈ V do
        L(v).insert(c);
    end
end
for c_i ∈ C do
    for c_j ∈ C do
        if T(c_i) ∩ T(c_j) = φ then
            P.insert(c_i, c_j);
        end
    end
end
return P;
```

Time Complexity: The first for loop calls DFS for all nodes. The Time complexity of DFS is O(m) which is O(n^2). Hence, the time complexity of the first for loop is O(n^3). For the second for loop, we have two nested for loops and we calculate the intersection of two sets is O(length of smallest set), which is O(n) as it can include a maximum of n nodes). Thus the time complexity of the second for loop is O(n^3). Hence the overall time complexity of the above algorithm is O(n^3).

3 Forex Trading

3.1 Part(a)

We need to find a cycle such that

$$R[i_1, i_2].R[i_2, i_3].R[i_{k-1}, i_k].R[i_k, i_1] > 1$$

$$\iff \frac{1}{R[i_1, i_2].R[i_2, i_3].R[i_{k-1}, i_k].R[i_k, i_1]} < 1$$

$$\iff \Sigma_2^k \log \frac{1}{R[i_{k-1}, i_k]} + \log \frac{1}{R[i_k, i_1]} < 0$$

Thus, by replacing the edge weights of the graph by $\frac{1}{R[i_k, i_{k+1}]}$, we reduce the problem to finding a negative cycle. This can be obtained using bellman-ford algorithm.

Algorithm 7: Negative cycle check using bellman ford

```
dist ⟵ array initialized with ∞;
dist[i_1] = 0;
for i in range(n-1) do
    for edge e(u,v) ∈ E do
        dist[v] = min(dist[v], dist[u] + weight(e));
    end
end
for edge e(u,v) ∈ E do
    if dist[v] > dist[u] + weight(e) then
        return True;
    end
end
return False;
```

Time Complexity: The time complexity of Bellman Ford algorithm is O(mn) for the first loop and O(m) for the second loop. The overall time complexity is O(mn). It is such that we can convert any currency to any other currency so m = O(n^2). Thus, the time complexity of the above algorithm becomes O(n^3).

3.2 Part(b)

To obtain such a cycle, we keep track of the last node which was used to visit the given node. We update the above algorithm as follows:

Algorithm 8: Negative cycle check using bellman ford

```
dist ⟵ array initialized with ∞;
dist[i_1] = 0;
parent ⟵ array initialized to NULL, used to store the previous node on minimum path.
 for i in range(n-1) do
    for edge e(u,v) ∈ E do
        dist[v] = min(dist[v], dist[u] + weight(e));
        if dist[u] + weight(e) < dist[v] then
            dist[v] = dist[u] + weight(e);
            parent[v] = u;
        end
    end
end
for edge e(u,v) ∈ E do
    if dist[v] > dist[u] + weight(e) then
        dist[v] = dist[u] + weight(e);
        parent[v] = u;
        visited ⟵ array initialized with False;
        cycle ⟵ {};
        while visited[v] == False do
            visited[v] = True;
            v = parent[v];
        end
        cycle.insert(v);
        current = parent[v];
        while current != v do
            cycle.append(v);
            v = parent[v];
        end
        cycle.append(v);
        reverse(cycle);
        return cycle;
    end
end
return ;
```

Time Complexity: The first for loop runs for O(n^3) times. The second for loop, after detecting the negative cycle, runs on all the nodes in the cycle which in worst case can be O(n). Thus, the time complexity of the second loop becomes O(n^3). Hence, the overall time complexity of the algorithm is O(n^3).

Proof of correctness: In the for loop where we update the distances for the n^{th} iteration, after finding a possible updation, we maintain a visited array and keep going back along the path until we encounter a visited node. This node is definitely a part of the cycle because it is grandparent to itself which is possible only if there is a cycle. We use the node obtained and the parent array to traverse the cycle found until we come back to the current node again and add all the nodes to our final array.

4 Coin Change

4.1 Part(a)

For any coin k, there are two possibilities:

- No coin of denomination k is used to make the sum n.
- Atleast one coin of denomination k is used to make the sum n.

The above cases are mutually exhaustive and cover all the possible combinations. Hence, the recurrence relation can be defined as

$$NumberOfWays(m, n) = NumberOfWays(m-1, n) + NumberOfWays(m, n-m)$$

Where, NumberOfWays(m,n) denotes the the number of ways we can create a sum n from the coins with maximum denomination of m.

Proof of correctness of the above recurrence relation
For the denomination m, using the above possibilities, if we don't use the m denomination to make the sum n, we get the number of ways as NumberOfWays(m-1,n). If we use m once, then the number of ways to create the remaining sum is NumberOfWays(m, n-m). Hence the total number of ways are NumberOfWays(m-1,n) + NumberOfWays(m, n-m).

The above problem can be solved efficiently by dynamic programming because the given problem has repeating sub-problems and we store the results for future use.

Algorithm 9: Number of Ways to create a sum N

```
dp ⟵ k*(N+1) array initialized with 0;
for j in range(k) do
    dp[j][0] = 1;
end
for i in range(k) do
    for j in range(1,N+1) do
        if d[i] <= j then
            dp[i][j] = dp[i][j] + dp[i][j-d[i]];
        end
        dp[i][j] = dp[i-1][j];
    end
end
return dp[k-1][N];
```

Time Complexity: The for loop runs for k*(N+1) times, so the time complexity of the above algorithm is O(kn).

4.2 Part(b)

We solve the given problem by recursive approach. Let minCoins(n) be the minimum number of coins needed to make the sum N. Then using all the coins d[1], d[2], d[3], d[4], ..., d[k] we obtain the minimum number of coins as follows:

$$minCoins(n) = min(1 + minCoins(n - d[i]))\ \ \forall i <= k$$

Lemma 4.1 *The above solution cannot be obtained greedily by using the maximum denomination coin first.*
Proof. Take the example denominations to be d = 1,3,9,10 and the sum required is 12. Then by solving the problem greedily, we take 10 and then to make a sum of 2, we use the coin 1 two times. i.e. we use 3 coins to get the sum of 12. However the minimum number of coins that can be used to create the sum 12 is 2, i.e. 9 and 3.

Lemma 4.2 *The recurrence relation results in minimum number of coins.*
Proof. We use all the denominations first and then calculate the minimum number of coins required to make the remaining sum. 1 + min(n-k) is the minimum number of coins that are required if we make the sum n if we use coin k. We take the minimum number of coins required overall by taking the minimum of these minimums.

Algorithm 10: minimum coins required to create a sum N

```
dp ⟵ array of size N+1 initialized with ∞;
table[0] = 0;
for i in range(1,N+1) do
    for j in range(k) do
        if i >= d[j] then
            table[i] = min(table[i], table[i-d[j]] + 1);
        end
    end
end
return table[N];
```

Time Complexity: The for loop runs for Nk times and all operations inside are O(1). Hence, the overall time complexity of the above algorithm is O(Nk).