COL351 : Analysis and Design of Algorithm
Assignment 2

Gaurav Jain (2019CS10349) & T Abishek (2019CS10407)

1 Algorithms Design book

Algorithm to partition the chapters into 3 sets S_1, S_2, S_3, so that maximum sum of each set in minimised.
Observation:
To minimize the objective, we can find the minimum of every possibility of a chapter being assigned to all 3 sets. We can achieve this by dynamic programming.
Algorithm:

Algorithm 1 Minimisation

```
1:  procedure DP(a,b,c,i, P, M)          ▷ a,b,c: sum of questions given to Alice, Bob and Charlie
2:      if M[a][b][c][i] ≠ -1 then
3:          return M[a][b][c][i]
4:      M[a][b][c][i] = min(DP(a+P[i],b,c,i+1), DP(a,b+P[i],c,i+1), DP(a,b,c+P[i], i+1))
5:      return M[a][b][c][i]
6:  procedure START(P, n)
7:      n ← no. of chapters
8:      P ← P is array where P[i] is no. of questions in chapter i( 1 to n)
9:      M ← M is matrix which is used to store computed totals in DP (initially -1)
10:     for i from 0 to n^2 do
11:         for j from 0 to n^2 do
12:             for k from 0 to n^2 do
13:                 M[i][j][k][n] = max(i,j,k)
14:     return DP(0,0,0,1,P,M)
```

Runtime Analysis:
This is dynamic programming problem, and in the worst case, all the elements of the matrix M will have to be filled to the final answer we are looking for. Dimensions a, b and c can be maximum n^2, because each chapter has a maximum of n questions and there are n chapters, so the maximum no. of questions is n^2, and thus each person can be assigned a maximum of n^2 questions. Dimension i can be maximum n, as there are only n chapters. So, no. of elements in M is $(n^2)^3 * n = n^7$, So, the Time Complexity is O(n^7).
Correctness of Algorithm:
The correctness of the algorithm can be proved using induction.
Base Case:
When all chapters are assigned i.e. $i = n$ in procedure DP, the max value of a,b and c is max(i,j,k) which is the optimal solution.
Induction Hypothesis:
For all $n \geq j > i$ the algorithm provides the optimal solution using chapters $P[1], P[2], ..., P[j]$.
Induction Step:
The i^{th} chapter can be assigned to any of the three sets a, b or c. Thus, the optimal solution is the minimum of the three subproblems generated by including questions of chapter i to the any one of the three sets.
By induction, the proof for all chapters. Hence, the algorithm gives an optimal solution to the problem.

□

2 Course Planner

1. Algorithm to find an order for taking the courses so that a student is able to take all the courses with the prerequisite criteria being satisfied.

   Representation:

   All courses are represented as vertices in a directed graph, with the in-edges of every vertex being the prerequisites.

   Observation: The order is required in such a way that all the prerequisites of a course is done before it, i.e, in the required order, all the edges go from left to right and there exists no edge from right to left. This is the same as the aim of topological sorting which can be achieved with a small modification to DFS.

   Algorithm:

Algorithm 2 Topological sorting

```
1:  procedure TOPSORT(Graph G, S, Node x, visited)
2:      visited[x] = grey
3:      for each vertex v in N(x) do
4:          if visited[v] = white then
5:              Topsort(G, S, v, visited)
6:          if visited[v] = grey then
7:              return Error: Not Possible
8:      S.insertFront(x)
9:      visited[x] = black
10: procedure INITIAL(Graph G)
11:     visited ← array of size of number of vertices(initially white)
12:     S ← order of taking courses(initially empty)
13:     for each vertex v in G do
14:         if visited[v] = white then
15:             Topsort(G, S, v, visited)
16: =0  return S
```

Runtime Analysis:
As discussed in class, the runtime complexity of topological sort is equivalent to DFS, i.e, O(m+n).

Correctness of Algorithm:
There can be no cycles in the graph that we are given so the given graph is a DAG and as discussed in class, topological sort guarantees a order of vertices in case of DAG where edges only go from left to right and not the other way. If we follow this order, it would satisfy the requirement that all courses have to be done only after their prerequisites are completed, as the source of edges represent the destination's prerequisites.

□

2. Algorithm to find the minimum number of semesters needed to complete all n courses.

Algorithm:

Algorithm 3 Minimum no. of semesters

```
1:  procedure MINIMUM_SEMESTER(Graph G)
2:      D ← array of buckets where vertices are arranged by their degree
3:      count = 0
4:      while G ≠ EMPTY do
5:          if D[0] = EMPTY then
6:              return Error: No order possible
7:          for v ∈ D with deg(v) = 0 do
8:              REMOVEVERTEX(G, v)
9:          count = count + 1
10:     return count
11: procedure REMOVEVERTEX(G, v)        ▷ Reduces degrees of vertices connected to v
12:     Mark v                          ▷ v is removed from invitees list
13:     D[deg(v)].REMOVE(v)
14:     for j from 1 to deg(v) do
15:         u ← adj(v)(j)
16:         if u is marked then
17:             continue
18:         deg(u) ← deg(u) − 1
19:         D[deg(u) + 1].REMOVE(u)
20:         D[deg(u)].INSERT(u)
```

Runtime Analysis:
The graph is stored in adjacency list as a HashMap mapping each vertex to an unordered set of connected vertices. The time taken to make the graph is $O(m)$. Degree of all vertices is also stored separately as a HashMap to give $O(1)$ access to degrees of each vertex and updated accordingly.

The degree of all the vertices is then sorted using bucket sort with each bucket being an unordered set(Hash) of vertices which allows us to insert and remove in $O(1)$ time. The buckets themselves are stored as an array of buckets of size N, with the index of the bucket representing the degree of the vertices stored in them. The initial sorting takes $O(m)$ as it is done through a single pass.

The inner FOR loop inside the WHILE loop in the Select procedure, is also run only $O(n)$ times as each iteration, a vertex is removed, and the maximum no. of vertices that can be removed is n.

Since the number of edges are m so the procedure REMOVEVERTEX's FOR loop runs atmost $2 * m$ over all the calls as the procedure is called for each vertex only once, and the for loop is run degree(v) times. Sum of the degrees of all the vertices is $2 * m$. So the overall running time of the algorithm is $O(m + n)$. Since $m \leq n^2$ so the time complexity is $O(n^2)$.

Correctness of Algorithm:
Claim: Let P be an array of sets, where P[i] = set of courses done by semester i. Let P' be the array representing the optimal(minimum) order of courses and let P be the array representing the above algorithm. Claim is P'[i] $\subseteq$ P[i] for all i.

Proof: Proof by induction on i

Base Case: i = 1
When i=1, the algorithm picks all the courses that don't have any prerequisites and the optimal order can't pick more courses than the given algorithm as there are no more courses left that can be taken.

Induction Hypothesis:
P'[j] $\subseteq$ P[j] for all j <= i-1

Induction Step:
for i, as P'[i-1] $\subseteq$ P[i-1], the courses that can be done in the ith semester (all prerequisites cleared) by our algorithm must be at least the same as the optimal algorithm. And as our algorithm picks all the eligible courses again, the optimal algorithm can't pick more courses than the given algorithm. Thus, P'[i] $\subseteq$ P[i].

Therefore, the claim is proved, and the algorithm provides the optimal way to finish the courses in minimum no. of semesters.

□

3. Algorithm to compute the set of all those course pairs whose prerequisites and ancestors don't intersect.

Algorithm:

Algorithm 4 Set of no intersection

```
1:  procedure NON_INTERSECTION(Graph G)
2:      D ← array of buckets where D[i] is a bucket with nodes of i indegree
3:      Root ← boolean array for each node mapping that node to its ancestor with degree 0
4:      Removed ← the nodes removed from the graph(initially empty)
5:      P ← the required set (initially empty)
6:      while G ≠ EMPTY do
7:          if D[0] = EMPTY then
8:              return Error: No order possible
9:          for v ∈ D with deg(v) = 0 do
10:             REMOVEVERTEX(G, v)
11:             for t in Removed do
12:                 if CHECK(V,T) then
13:                     P.add(v,t)
14:                     P.add(t,v)
15:             Removed.add(v)
16:     return P
17: procedure REMOVEVERTEX(G, v)          ▷ Reduces degrees of vertices connected to v
18:     D[deg(v)].REMOVE(v)
19:     for j from 1 to deg(v) do
20:         u ← adj(v)(j)
21:         if u is marked then
22:             continue
23:         deg(u) ← deg(u) − 1
24:         D[deg(u) + 1].REMOVE(u)
25:         D[deg(u)].INSERT(u)
26:         for i from 0 to size(Root(u)) do
27:             if Root(v)[i] then Root(u)[i] = True
28: procedure CHECK(v, u)          ▷ Checks whether the 2 nodes don't have any common ancestors
29:     for i from 0 to size(Root(u)) do
30:         if Root(u)[i] and Root(y)[i] then
31:             return False
32:     return True
```

Runtime Analysis:
The subroutine REMOVEVERTEX can take $O(n)$ time and the subroutine CHECK can also take $O(n)$ time for each vertex. The size of $Removed$ array can grow upto $O(n)$ and thus, for each vertex we may end up checking for all elements present in $Removed$ in $O(n^2)$ time.
Thus, the total time complexity of the algorithm is $O(n^3)$. The space complexity can be $O(n^2)$ when all nodes are independent.

Correctness of Algorithm:
We can prove the correctness of the algorithm by contradiction by assuming that there exists a pair of vertices which is not selected by the algorithm. Suppose the pair is (u, v) and at some stage of the algorithm, degree of u is 0 (as ancestors of u is finite). So, at this stage u is added to the $Removed$ array. Consider the stage at which degree of v becomes 0.
Without loss of generality, assume that u is added to $Removed$ first compared to v. Thus, the procedure CHECK will be called using (u, v). So, this pair will be checked and added to P if there is no common ancestor between them. This is a contradiction as we assumed that the algorithm will not find (u, v).

□

3 Forex Trading

1. Algorithm to verify whether or not there exists a cycle such that exchanging money over this cycle results in positive gain.

   Observation:

   A number x is strictly larger than 1 if and only if $log(1/x) > 0$.

$$\text{So, if we want } R[i_1, i_2]R[i_2, i_3]...R[i_{k-1}, i_k]R[i_k, i_1] > 1$$
$$\text{Then } log(1/R[i_1, i_2]R[i_2, i_3]...R[i_{k-1}, i_k]R[i_k, i_1]) < 0$$
$$\implies log(1/R[i_1, i_2]) + log(1/R[i_2, i_3]) + ... + log(1/R[i_{k-1}, i_k] + log(1/R[i_k, i_1]) < 0$$
$$\implies R'[i_1, i_2] + R'[i_2, i_3] + ... + R'[i_{k-1}, i_k] + R'[i_k, i_1] < 0$$

   Since this problem is related to forex trading where it is suitable to assume that every currency is exchangeable with any other currency either directly or indirectly so it is assumed that the graph generated by the currency exchange is connected.
   Thus, the problem can be reduced to finding a negative weight cycle in a (connected) graph where weight of edges is given by

$$R'[i_p, i_{p+1}] = log(1/R[i_p, i_{p+1}])$$

   A negative weight cycle can be detected easily in a (connected) graph using Bellman-Ford Algorithm.

   Algorithm:

Algorithm 5 Negative Cycle Detection Algorithm

```
1:  procedure NegativeCycleDetect(Graph G, S)
2:      distance ← array of size |V|
3:      for i = 0 to m − 1 do                          ▷ Changing edge weights
4:          R'(e_i) ← log(1/R(e_i))
5:      for each vertex v in G do
6:          distance[v] ← ∞
7:      distance[S] ← 0
8:      for i = 0 to n − 1 do
9:          for each edge (u, v) in G do
10:             if distance[u] + R'[u, v] < distance[v] then
11:                 distance[v] ← distance[u] + R'[u, v]
12:     for each edge (u, v) in G do
13:         if distance[u] + R'[u, v] < distance[v] then
14:             return True
15:     return False
        =0
```

   Runtime Analysis:
   Edge weights are changed in $O(m)$ time. After changing the edge weights, the rest part of the algorithm is identical to the standard Bellman-Ford Algorithm which has time complexity $O(mn)$ and space complexity $O(n)$.
   Thus, the total time complexity is $O(mn)$ and space complexity is $O(n)$.

   Correctness of Algorithm:
   The conversion of edge weights is explained in the observation section above and this conversion leads to a standard problem of shortest paths- Bellman-Ford Algorithm. The correctness of Bellman-Ford Algorithm is discussed in the lectures of the course. Thus, by reducing the problem to another problem, the algorithm is designed to find the solution of the problem.

   □

2. Algorithm to print out such a cyclic sequence defined in part 1, if it exists.

Algorithm:

Algorithm 6 Print Negative Cycle Algorithm

```
1:  procedure PRINTNEGATIVECYCLE(Graph G, S)
2:      distance ← array of size n
3:      parent ← array of size n
4:      for i = 0 to m − 1 do                          ▷ Changing edge weights
5:          R'(e_i) ← log(1/R(e_i))
6:      for each vertex v in G do
7:          distance[v] ← ∞
8:      distance[S] ← 0
9:      for i = 0 to n − 1 do
10:         for each edge (u, v) in G do
11:             if distance[u] + R'[u, v] < distance[v] then
12:                 distance[v] ← distance[u] + R'[u, v]
13:                 parent[v] ← u
14:     cycle ← sequence of nodes containing a cycle
15:     for each edge (u, v) in G do
16:         if distance[u] + R'[u, v] < distance[v] then   ▷ Cycle detected
17:             p ← v
18:             for i = 0 to n − 1 do
19:                 p ← parent[p]
20:             Add p to cycle
21:             while i is not p do
22:                 Add i to cycle
23:                 i ← parent[i]
24:             return cycle
25:     return False
```

Runtime Analysis:
The time complexity of this algorithm is same as the previous one because in this algorithm, we only add a $parent$ array to store the previous element of a node in the shortest path tree.
Thus, the space complexity is $O(n)$ and the time complexity is $O(mn)$.

Correctness of Algorithm:
Printing negative cycle in a graph is a standard problem based on the Bellman-Ford algorithm. Similar to the previous algorithm, we stop when an update is possible in the n^{th} iteration and using the $parent$ array, we backtrack to reach the starting point. The correctness of this algorithm is strongly related to the previous one which is based on Bellman-Ford algorithm and its correctness is discussed in the lectures.

□

4 Coin Change

1. Algorithm to count the number of ways to make change for Rs. n, given an infinite amount of coins/notes of denominations $d[1], ..., d[k]$.

   Recurrence Relation:

   Let $C(n, m)$ represent the number of ways to make change for Rs. n using coins of denomination $d[1], ..., d[m]$.

   The solution of $C(n, m)$ can be partitioned into two parts:

   - Solution containing at least one coin of denomination $d[m]$. The subproblem here is to count number of ways to make change of Rs. $n - d[m]$ using $d[1], ..., d[m]$ i.e. $C(n - d[m], m)$.
   - Solution containing no coins of denomination $d[m]$. The subproblem here is to count number of ways to make change of Rs. n using $d[1], ..., d[m - 1]$ i.e. $C(n, m - 1)$.

   Thus, the recurrence relation is formulated as:

   $$C(n, m) = C(n, m - 1) + C(n - d[m], m) \tag{1}$$

   with the base cases:

   - $C(n, m) = 1$ when $n = 0$ (No coin change)
   - $C(n, m) = 0$ when $n < 0$ (Negative money)
   - $C(n, m) = 0$ when $n > 0$ and $m \leq 0$ (No coin left)

   Algorithm:

Algorithm 7 Coin Change Algorithm

```
1:  procedure CoinChange(n, set of coins d[1], d[2], ..., d[k])
2:      count → 2D array of size (n + 1) × k
3:      for i = 0 to n do
4:          for j = 1 to k do
5:              if i = 0 then count(i, j) ← 1
6:              else if j = 0 then
7:                  if i mod d[j] = 0 then count(i, j) ← 1
8:                  else count(i, j) ← 0
9:              else if d[j] > i then count(i, j) ← count(i, j − 1)
10:             else count(i, j) ← count(i, j − 1) + count(i − d[j], j)
11:     return count(n, k)
```

   Runtime Analysis:

   Like other dynamic programming problems, this algorithm fills the complete 2D array $count$ in a single pass. So, the time complexity as well as space complexity is $O(nk)$.

   Correctness of Algorithm: Proof by Induction on i

   Base Case:
   There are three base cases:

   - When no amount is left to make change. In this case, there is only 1 way to make change.
   - When the amount is negative, there is no way to make change. Thus, $C(n, m) = 0$ is 0 in this case.
   - When there is some amount left but no coins are left to make change. Thus, in this case, number of ways to make change is 0.

   Induction Hypothesis:
   The algorithm gives the correct count of the number of ways to make change for all values of Rs. $p < i$ using coins of denominations $d[1], d[2], ..., d[j]$ and number of ways to make change of Rs. i using coins of denomination $d[1], d[2], ..., d[q]$ for all $q < j$.

   Induction Step:
   To calculate the number of ways to make change of Rs. i using coins of denomination $d[1], d[2], ..., d[j]$. We can either include the coin of denomination $d[j]$ or exclude it.
   By induction hypothesis, we have the number of ways to make change of Rs. i using $d[1], d[2], ..., d[j - 1]$. We also have the number of ways to make change of Rs. $i - d[j]$ using coins of denomination $d[1], d[2], ..., d[j]$.
   Thus, the total number of ways is calculated by using these two subproblems solutions. □

2. Algorithm to find a change of Rs. n using the minimum number of coins.

Recurrence Relation:

Let $C(n)$ represent the minimum number of coins required to make change for Rs. n using coins of denomination $d[1], ..., d[m]$.

$C(n)$ is minimum of the subproblems generated by using one coin of $d[j]$ denomination. The recurrence relation is formulated as:

$$C(n) = min(C(n), 1 + C(n - d[j])) \; \forall j \in [1, m] \tag{2}$$

with the base case: $C(n) = 0$ when $n = 0$ (No coin change)

Algorithm:

Algorithm 8 Min Coin Change Algorithm

```
1:  procedure MinCoinChange(n, set of coins d[1], d[2], ..., d[k])
2:      count → array of size n + 1
3:      previous → array of size n + 1
4:      count[0] = 0
5:      previous[0] = Null
6:      for i = 1 to n do
7:          minimum ← ∞
8:          minprevious ← Null
9:          for j = 1 to k do
10:             if i ≥ d[j] then
11:                 minimum ← min(minimum, 1 + count[i − d[j]])
12:                 minprevious ← i − d[j]
13:         count[i] ← minimum
14:         previous[i] ← minprevious
15:     change → list of coins required
16:     x ← n
17:     while previous[x] ≠ Null do
18:         change.append(x − previous[x])
19:         x ← previous[x]
20:     return change
```

Runtime Analysis:

Like other dynamic programming problems, this algorithm fills the complete array $count$ in a single pass. There are two nested for loops which run n and k times respectively. So, the time complexity is $O(nk)$. The space complexity is $O(n)$ to store the $previous$ array.

Correctness of Algorithm: Proof by Induction on i

Base Case: $i = 0$
When $i = 0$ the number of coins required to make change is zero. The algorithm is initialized with $count[i] = 0$ when $i = 0$. So, base case holds.

Induction Hypothesis:
The algorithm gives the correct count of the minimum number of coins required to make change for all values of Rs. $j < i$.

Induction Step:
Consider the i^{th} iteration of the outer for loop, the algorithm finds the minimum coins required to make change for Rs. i by using a coin of denomination of $d[j]$ and breaking the problem into various subproblems.
By induction hypothesis, the solution given by the algorithm of these subproblems is optimal and thus, we only need to add one coin of denomination $d[j]$ (which gives the minimum) to the coin sequence.

□