COL351 Assignment - 2

Aditya (2019CS50471)
Ajay (2019CS10323)

25^{th} September, 2021

Contents

1 Algorithms Design Book

Given there are n chapters [1,...,n] and for i ∈ [1,n]. x_i denotes the no of questions in chapter i. Also, $x_i \leq n$, We use dynamic programming to solve the question.

$$dp[i][u][v] = \begin{cases} True, & \text{if there are disjoint subsets A, B of [1,...,i] such that } \sum_{j\in A} x_j = u,\ \sum_{j\in B} x_j = v \\ False, & \text{otherwise} \end{cases}$$

let K = $\sum_{j=1}^{i} x_j$

This implies that we allot questions in subset A to person 1, questions in subset B to person 2, questions in subset {1,...,i}\(A ∪ B) to person 3. So, questions alloted to person 1 are u, questions alloted to person 2 are v, questions alloted to person 3 are K - (u + v)

$$dp[i][u][v] = \begin{cases} True, & \text{if dp[i - 1][u][v] = True (or) dp[i - 1][u - } x_i\text{][v] = True (or) dp[i - 1][u][v - } x_i\text{] = True} \\ False, & \text{otherwise} \end{cases}$$

Proof:

dp[i][u][v] - using i chapters we need to allot u questions to person-1 and v questions to person-2 and remaining to person-3

case-1:

We allot questions in chapter i to person 3,So using chapters 1 to i-1,we need to allot u questions to person-1 and v questions to person-2 = dp[i - 1][u][v]

case-2:

We allot questions in chapter i to person 2 so using chapters 1 to i-1,we need to allot u questions to person-1 and v-x_i questions to person-2 = dp[i - 1][u][v - x_i]

case-3:

We allot questions in chapter i to person 1 so using chapters 1 to i-1,we need to allot u-x_i questions to person-1 and v questions to person-2 = dp[i - 1][u - x_i][v]

So,if dp[i - 1][u][v] == true or dp[i - 1][u][v - x_i] == true or dp[i - 1][u - x_i][v] == true then dp[i][u][v] = true

Base case:
i)dp[1][0][0] = true, because using 1st chapter we can allot0 questions to person-1, 0 questions to person-2 , x_1 questions to person-3
ii)dp[1][0][x_1] = true, because using 1st chapter we can allot 0 questions to person-1, x_1 questions to person-2 , 0 questions to person-3
iii)Similarly, dp[1][x_1][0] = true
iv)dp[1][u][v] = false otherwise
Algorithm
let S[i] = $\sum_{j=1}^{i} x_j$

```
dp - 3D Array
dp[1][0][0] = dp[1][0][xi] = dp[1][xi][0] = true
dp[1][u][v] = false otherwise
for i from 2 to n - 1:
    for u from 0 to S[i]:
        for v from 0 to S[i]:
            if(dp[i - 1][u][v] == true or
               (u - xi >= 0 and dp[i - 1][u - xi][v] == true) or
               (v - xi >= 0 and dp[i - 1][u][v - xi] == true)) then
                dp[i][u][v] = True
```

Now we need to find u, v, S[n] - (u + v) such that max(u, v, S[n]- (u + v)) is minimized and dp[n][u][v] = true

```
min_val = INT_MIN,
q1, q2, q3 store optimal no of question to be alloted to person 1, 2, 3
respectively.
for u from 0 to S[n]:
    for v from 0 to S[n]:
        if (dp[n][u][v] == true) then:
            if (max(u, v, S[n] - (u + v)) <= min_val) then:
                q1 = u
                q2 = v
                q3 = S[n] - (u + v)
                min_val = max(u, v, S[n] - (u + v))
```

So the no of question allotted to person-1 is q1, alloted to person-2 is q2 and allotted to person-3 is q3 in optimal partition
S_1 is set of chapter alloted to person 1, S_2 is set of chapter alloted to person 2, S_3 is set of chapters alloted to person 3. Getting the partition of {1,...,n} into S_1, S_2, S_3.

```
fun getPartition(n, q1, q2, q3):
    if (dp[n - 1][q2][q3] == true) then:
        S3.append(n)
        getPartion(n - 1, q1, q2)
    else if (dp[n - 1][q1][q2 - xn] == true) then:
        S2.append(n)
        getPartition(n - 1, q1, q2 - xn)
    else if (dp[n - 1][q1 - xn}][q2] == true) then:
        S1.append(n)
        getPartition(n - 1, q1 - xn, q2)
```

Time Complexity:
1) Time Complexity to compute values of 3D array = O(n*S[n]*S[n]) = O(n^5)
2) Time complexity to find optimal u, v, S - (u + v) is O(S[n]*S[n]) = O(n^4)
3) Getting the partition takes O(n) time
So the total time taken to compute optimal solution is O(n^5).

2 Course Planner

2.a Ordering of Courses

Given a set C of courses.For each c ∈ C, set P(c) represents the set of prerequisite courses for c. Let G be a directed graph where nodes of the graph are courses. There exists a directed edge from c_i to c_j if $c_i \in P(c_j)$
Ordering of courses:
The courses should be taken in the topological ordering of graph G.
Proof:
Definition of Topological order: For every directed edge (x,y) in G, x appears before y in the ordering.
let c_i be any course. From the modelling of the graph, we can say that there exists directed edge from every course that belongs to $P(c_i)$ to c_i.Using the above definition all the courses that belong to $P(c_i)$ appear before c_i in the ordering .So in the topological ordering before taking c_i all the courses belong to $P(c_i)$ will be completed So, topological ordering satisfy the given prerequisite criteria. If there is a cycle in graph G we can't take all courses (there won't be any topological ordering).Because among the courses in the cycle, we can't find any ordering.
Algorithm
We can check whether a graph G is cyclic or not using DFS. If there is a cycle return error.

```
Stack stack
Array visited  #stores bool, whether a vertex is visited or not
fun modifiedDFS(vertex u):
    visited[u] = true
    for each v in Out[u]:
        if (!visited[v]) then:
            modifiedDFS(v)
    stack.push(u)
#Topological ordering for Graph G(the order in which courses should be taken)
fun topologicalorder():
    for each u in G:
        visited[u] = false
    for each u in G:
        if (visited[u] == false) then:
            modifiedDFS(u)
    array order  # stores the final topological ordering of G
    while (!stack.empty()) :
        order.append(stack.top())
        stack.pop()
    return order
```

Time Complexity:
Checking cycles take O(m + n) time because we use DFS. We have modified the DFS to find the topological ordering of G. So time complexity is O(m + n).

2.b Minimum Semesters

If graph G has cycles then we can't find the min no of semesters needed.Because we can't find an order in which courses should be taken.
So, let G be acyclic. So G is DAG.
Minimum no of semesters needed is the length of the longest path in DAG G.
To find the longest path in G, we visit vertices in the order they appear in topological ordering. Since G can be disconnected, we find the longest path in each component of G and the answer will be the maximum of the longest paths in each component. If we follow topological ordering, we first visit the vertex s with Indegree zero (Property of topological ordering). We compute distance to all vertices of that component from s. The max of distances will be the longest path of that component.
Algorithm:

```
let order be the array which is the topological ordering of G.
We can compute "order" using Algorithm in part(i).
In(v) is list of incoming edges to v.
fun minSemesters():
    Array dist #stores distance to each vertex, initialized to infinity.
    for each vertex v in order:
        for each u in In(v):
            dist(v) = max(dist(v), dist(u) + 1)
    return max(dist)
```

Proof
Courses in different disconnected component of G can be taken simultaneously,because they are independent of each other.In each component the course c at largest distance from vertex s(vertex with indegree 0) will be taken last,because all the courses in the path from s to c must be completed before c. Courses on different paths in same component can be done simultaneously, So longest path determines the min no of semesters required for that component. So,answer will be maximum of longest paths of all the components.
Time Complexity: Computing array order takes O(m + n) time. We visit each vertex at once to compute dist[v] using In(v). This takes atmost O(m). So time complexity is O(m + n).

2.c Pair Set

In Graph G, from course c we can reach all the courses that are dependent on c. So in reversed graph (all directed edges(u,v) are changes to (v,u)), from course c, we can reach courses on which c depends i.e., we can reach all the courses that must be completed before taking c in reversed graph. Using this we can get L(c) $\forall$ c.So we can compute pair set P using L(c).
Algorithm:

```
Step - 1: Reverse Graph G
for edge e(u,v) in G:
    (u,v) = (v,u)
Step - 2: compute L(c) for each c in G
L(c) - #stores list of courses that must be completed before c.
fun DFS(c):
    visited[c] = true
    for each c1 in Out(c):
        if (!visited[c1]) then:
            DFS(c1)
    return;
fun compute():
    for c in G:
        set visited = False for all c1 belongs to C.
        DFS(c)
        for each c1 in G:
            if (visited[c1] and c1 not equal to c) then:
                L(c).append(c1)
Step - 3: compute pairset
p = {} # stores pairs
for c in G:
    for c1 in G:
        if (L(c) intersection L(c1)) is empty then:
            P.append(c,c1)
return P
```

Time Complexity DFS for each vertex takes O(m + n) time. So, DFS for all vertices take O(m*n) time. Here m = O(n^2).Time complexity is O(n^3). Computing pairset takes O(n^3) time because intersection of two sets takes O(n) time using hash-map. So, total time complexity = O(n^3)

3 Forex Trading

Graph G:
Nodes correspond to n currencies c_1, c_2, c_3, ..., c_n
Edge weights correspond to exchange rate between these currencies.

3.a Verifying Cycle

We need to verify whether there exist a cycle C = (c_1, c_2, c_3, ..., c_k, $c_{k+1} = c_1$) such that exchanging money over this cycle results in positive gain.
$\Rightarrow R[c_1,c_2].R[c_2,c_3].....R[c_k,c_1] > 1.$

$\Rightarrow \log(\frac{1}{R[c_1,c_2].R[c_2,c_3].....R[c_k,c_1]}) < 0$

$\Rightarrow$ -log(R[c_1,c_2]) - log(R[c_2,c_3]) - log(R[c_k,c_1]) < 0

Let's define w'(c_i, c_j) = -log(R[c_i, c_j])
$\Rightarrow$ w'(c_1,c_2]) + w'(c_2,c_3) + ... + w'(c_k,c_1) < 0

We need to find a negative cycle in graph G' where nodes correspond to n currencies and weights at edge(i,j) is given by -log(R[i,j])
Terms: Weight of edge(i,j) in G = w(i,j) = R[i,j]

Algorithm Sketch:
G - original graph
Modify edge weights as w'(i,j) = -log(w(i,j))
Use Bellman Ford Algorithm to verify whether there is a negative edge cycle in modified graph

Algorithm

```
G-original graph
let s be any vertex in G
D-Distance of each vertex from arbitrary vertex s
for each edge(i,j) in G:
    w(i,j) = -log(w(i,j))
for each vertex in G:
    D[v] = INF

D[s] = 0
for i from 1 to n-1:
    for each edge (i,j) in G:
        D[j] = min(D[j], D[i] + w(i,j))
for each edge(i,j) in G:
    if (D[j] > D[i] + w(i,j)) then:
        #There is a negative cycle i.e., There is Positive gain
        return true
return false
```

Proof:
As discussed in lecture-17, Bellman ford algorithm detects negative cycle in graph.
Time Complexity:
Time to modify graph : O(m)
Time taken by Bellman ford: O(mn)
And we know that m = O(n^2) So, total Time complexity is O(n^3)

3.b Printing Cycle

We can use Bellman Ford Algorithm to print negative cycle. We maintain a parent array which stores the parent of each node in shortest path graph. In the nth iteration, if we find an edge (u,v) such that D[v] > D[u] + w(u,v), then we find the negative cycle by going backward from v using parent array . We find a vertex of the cycle and then find cycle starting from this vertex.

Algorithm:

```
G-original graph
let s be any vertex
for each edge (i,j) in G:
w(i,j) = -log(w(i,j))
for vertex v in G:
    D[v] = INF
D[s] = 0

for i from 1 to n-1:
    for each edge (i,j) in G:
        D[j] = min(D[j], D[i] + w(i,j))

vertex temp = NULL # Stores the vertex relaxed in nth iteration
for each edge (i,j) in G:
    if(D[j] > D[i] + w(i,j)) then:
        temp = j
        break

if temp == NULL then:
    return false
visited[n]  # Stores visited vertices
while (!visited[temp]) :
    visited[temp] = true
    temp = parent[temp]

cycle = []
vertex v = temp
while (true):
    cycle.append(v)
    if (v == temp && len(cycle) > 1) then:
        break
    v = parent[v]
print(cycle)
return true
```

Time Complexity:
1) Bellman ford takes O(mn).
2) Checking negative cycle takes O(m). 3) Printing cycle takes O(n).
So,overall time complexity is O(mn). m = O(n^2). So,overall time complexity is O(n^3)

Proof:
As discussed in lecture, Bellman ford identifies negative cycle.If the weight of an edge (u,v) changes in nth iteration, it means that there is a negative edge cycle in the path between s to v(if not edge weight wouldn't have changed).So,we find any vertex in the cycle and print cycle.

4 Coin Change

Given set of k denominations d[1], d[2], d[3],....,d [k]

4.a No of ways to make change of Rs.n

We use dynamic programming to solve this.
let dp[i][j] denote no of ways to make change for Rs.j using denominations d[1],d[2],.....,d[i].
Recurrence Relation: dp[i][j] = dp[i - 1][j] + dp[i][j - d[i]]. We need dp[k][n].
Proof:
case-1: Excluding d[i]
Then no of ways to make change for Rs.j = dp[i - 1][j]
case-2: Including atleast one d[i] (Since, there are infinite amount of notes)
Then no of ways to make change for Rs.j = dp[i][j - d[i]]
So, dp[i][j] = d[[i - 1][j] + dp[i][j-d[i]]
Base case: dp[i][0] = 1 , because no of ways to make change for Rs.0 is 1 (By not including any denominations)
Psuedo Code:

```
dp is 2D Array of size (k) x (n+1)
for i from 1 to k:
    dp[i][0] = 0
for j from 1 to n:
    for i from 1 to k:
        c1 = dp[i - 1][j]
        c2 = 0
        if (j - d[i] >= 0):
            c2 = dp[i][j - d[i]]
        dp[i][j] = c1 + c2
return dp[k][n]
```

Time Complexity:
Two nested loops. So, total time complexity = O(nk) (Polynomial time)

4.b Change with minimum coins

Finding a change of Rs.n using minimum no of coins. We use dynamic programming.
dp[i] stores minimum no of coins required to form a change for Rs.i. We need dp[n].
Recurrence Relation: dp[i] = $\min_{1 \leq j \leq k}$ (1 + dp[i - d[j]])
Proof:
let's say that we use d[j] in forming a change for Rs.i. So, now we need to form a change for Rs.(i - d[j]). Minimum coins required to form a change for Rs.(i - d[j]) is dp[i - d[j]].So, coins used in forming a change for Rs.i = 1 + dp[i - d[j]] (the other coin is d[j]). Since, we don't know d[j],we iterate over all possible denominations and find d[j] which result in using minimum no of coins for forming change of Rs.i.
Boundary Case: dp[0] = 0, since no of coins required to form a sum 0 is 0.
Algorithm:

```
dp is array of size n+1
dp[0] = 0
for i from 1 to n:
    dp[i] = INF
    for j from 1 to k:
        if (i - d[j] >= 0) then:
            dp[i] = min(dp[i],1 + dp[i - d[j]])
return dp[n]
```

Time Complexity:
Two nested loops.So, total time complexity = O(nk) (Polynomial time)