CS345 Theoretical Assignment 3

Ayush Agarwal, 13180

M.Arunothia, 13378

Contents

1 Shortest routes

1.1 Overview

Any path from source to vertex is a sequence of petrol pumps and non-petrol pumps in which vertices can be repeated due to petrol constraint. So for calculating shortest distance we can't simply apply Dijkstra. That is why we have to construct an equivalent graph which is free of any such constraint. For this, we calculate shortest distance from every petrol pump(considering source and distination to be also petrol pump) to every other petrol pump (shortest distance may or may not follow petrol constraint) by applying Dijkstra Algorithm from each petrol pump. For every distance between petrol pumps, we also store the path used. Then we construct a graph with only petrol pumps, and edges which are less than allowed petrol constraint. Finally we apply dikstra on this graph from source to calculate its distance to destination.

1.2 Pseudo-Code

```
procedure ShortestPath(V, E, S, T, THRESHOLD)
    G ← {V,E}
    Mark s and t as petrol pumps
    newEdgeList ← []
    pathList[$n$] ← []
    for $i$ in V.pumps do
        newEdgeList.append(Filter(Dijkstra(i), threshold))
    end for
    Construct graph from newEdgeList
    d ← Dijkstra(s)
    i ← t
    while i <> s do
        j ← i
        p ← pathList[s][i]
        while j <> p do
            print "$j < -$"
            j ← pathList[p][j]
        end while
    end while
end procedure
```

```
procedure Dijkstra(V)
    returns the list of shortest distance of v to all other petrol pumps in G by applying Dijkstra algo.
    pathList[v][i] stores the parent of i in the shortest path from v to i.
end procedure
```

```
procedure Filter(LIST, THRESHOLD)
    returns the list of elements less than threshold in the list
end procedure
```

1.3 Time-Complexity

Dijkstra Algorithm takes worst case *mlogn* time and in the above algorithm we Dijkstra is invoked O(n) times. Rest Filter and append function are O(n) and are invoked O(n) time. So complexity is:

$$O(n) * O(mlogn) + O(n^2) = O(mnlogn)$$

1.4 Space-Complexity

Space complexity is O(n^2)

1.5 Proof of Correctness

The subtlety of this problem is that vertices can be visited twice. So lets dig deeper into it and formalize the observations.

Lemma 1: Any petrol pump is visited only once in the path from source to destination.

Proof: Lets prove this lemma by contradiction. Suppose there exists a petrol pump which gets visited twice (or more than once) in the optimal path from source to destination. Lets call that petrol pump as p. So the path will be somewhat like $s --> p --> p --> t$. The sole reason for any vertex being visited twice was because of petrol constraint. We can observe that the state of the bike is same after every visit to the petrol pump. And since the edge weights are positive we can have a solution without $p --> p$ cycle which will be having weight less than the proposed optimal solution. Thus the proposed solution is not optimal solution. Hence by contradiction we can say that any petrol pump is not visited more than once.

Lemma 2: No vertex is visited twice in the shortest path(without any petrol constraints) from pump p to q.
Proof: Without any petrol constraint, the problem is like normal weighted graph problem. So to calculate the shortes path from p to q in a normal positive weighted graph, I won't be visiting any vertex twice becuase it will add an extra unnecessary cycle weight to my shortest path. Hence proved.

By Lemma2 we can say that, to calculate the shortest distance without petrol constraint from pump ṕt́o all other pumps, we can just apply Dijkstra from p. Similarly we can calculate distance of every pump with every other pump by applying Dijkstra on all of them. Then we construct a graph Gćonsisting of only petrol pumps and, s and t. In this construction the path from s to t is preserved. From lemma1 we have said that any petrol pump is visited only once in the optimal path from s to t. Hence we can safely apply Dijkstra in this newly constructed graph to calculate distance from s to t.

2 Time-Optimization

2.1 Overview

We solve this question by considering all the cases using dynamic programming. For this we use an $n \times H$ matrix for storing the already computed values. Call that matrix as M. M[i][j] stores the optimal grade value (equal to i*average grade) which can be obtained (in the set of courses less than i) in j hours. M[i][j].hours stores the number of hours devoted to course i in the optimal solution for courses 1 to i in j hours.

2.2 Pseudo-Code

```
procedure timeOptimization(f[], H, g, n)
    M[n][H] $\leftarrow$ 0
    for $i$ in $H$ do
        M[0][i] = f[0](i)
        M[0][i].hours = i
    end for
    for $i$ in 1 to n-1 do
        for $j$ in H do
            for $k$ in j do
                temp = max(f[i](k) + M[i-1][j-k], M[i][j])
                if temp $\geq$ M[i][j] then
                    M[i][j] $\leftarrow$ temp; M[i][j].hours $\leftarrow$ k;
                end if
            end for
        end for
    end for
    i $\leftarrow$ n-1; j $\leftarrow$ H
    while 0 $\leq$ i do
        print M[i][j].hours
        j = j $-$ M[i–][j].hours
    end while
end procedure
```

2.3 Time Complexity

This algorithm involves 3-Level nested loop. So its complexity will be

$$O(nH^2)$$

2.4 Space Complexity

For this algo we use a $n \times H$ matrix for storing intermediate values. So the complexity is

$$O(nH)$$

Although, it can also be otimized to O(H) because for calculating any row we just need its previous row.
So the best complexity is $O(H)$.

2.5 Proof of Correctness

Grade Value: sum of grades
Claim: M[i][j] stores the optimal grade value in subjects 1 to i by devoting j hours after (i, j) iteration.
Proof: Proof by Induction
Base Case: Only one course. If there is only one course, then the optimal grade value for j hours would simply be f(j), which is what M[0][j] stores.
Induction Hypothesis: M[i-1][k] stores the optimal grade value for all k less than equal to H.
Induction Step: The optimal solution(M[i][j]) can consist of any number of hours($\leq$ j) devoted to course i depending on the f[i] and M[i-1][]. The optimal solution will contain k hours devoted to course i, and j-k hours devoted to rest i-1 courses. To get the number of hours devoted to course i in optimal solution we have to iterate over all the possible hours, and choose the one which gives best grade value. Thus,

$$M[i][j] = \{\max(M[i][j], f[i](k) + M[i-1][j-k]) — \text{for k in } [0..j]\}$$

Now since M[i-1][j-k] is computed correctly, above loop will give correct value of M[i][j]. Hence proved.

3 Mobile Network

3.1 Overview

We take Dynamic Programming approach to solve this problem. The overview of our solution can be described using the following cost array. $cost[i]$ stores least cost($P_0, ..P_i$) of G_0 to G_i graphs. For computing cost[i+1], we will be iterating over all the graphs from 0 to i.
cost[i] = min{cost[j] + (i-j)*(shortest path from s to t in Intersection(G_{j+1} .... G_i)) + K} for j in i.
Suppose in the optimal solution for 0 to i graphs, $j + 1$ to i graphs had same value of path(let say p) and rest was optimal solution of cost[j].So $pathValue$ and $previous$ are defined in the following way.
cost[i].pathValue = p
cost[i].previous = j
Graphs are stored in $n \times n$ matrix form in order to ease merging.

3.2 PseudoCode

```
procedure MobileNetwork
    cost[n] ← infinity // array initialized to infinity
    for i in n do
        if i == 0 then
            cost[i] ← ShortestPath($G_0$)
        else
            mergedGraph ← $G_i$
            for j in i-1 to 0 do
                mergedGraph = merge($G_{j+1}$, mergedGraph)
                temp = min{cost[i], cost[j] + (i-j)*ShortestPath(mergedGraph) + K}
                if temp ≥ cost[i] then
                    cost[i] = temp
                    cost[i].pathValue = ShortestPath(mergedGraph)
                    cost[i].previous = j
                end if
            end for
            cost[i] = min{cost[i], (i+1)*ShortestPath(mergedGraph)}
        end if
    end for
    i ← n
    while i ≥ 0 do                    / print the path
        print cost[i].pathValue (i - cost[i].previous) times
        i ← cost[i].previous
    end while
end procedure
```

```
procedure Merge(mergedGraph, graphToBeMerged)
    return the matrix by taking "And" of every i-j entry of both the graphs
end procedure
```

3.3 Time Complexity

Merging two graphs takes O(V^2) where V is the number of vertices. ShortestPath between s to t in a graph can be computed in O(m+n) time. For any i, algorithm makes i iterations, and in each iteration merging is done. So total iterations are O(n^2), therefore

$$timecomplexity = O(b^2 * n^2)$$

3.4 Space Complexity

cost array takes O(n) space. All graphs are stored in matrix form. *mergedGraph* is also stored in matrix form.So space complexity is

$$O(n^2 * b)$$

3.5 Proof of Correctness

3.5.1 Claim

$cost[i]$ stores the optimal cost of graphs G_0 ... G_i.

3.5.2 Proof by Induction

3.5.3 Base Case

For i = 0, cost[i] is simply the shortest path in graph G_0.

3.5.4 Hypothesis

Let us assume that the claim is true for all j less than i.

3.5.5 Inductive Step

Any solution of graphs 0 to i will have same path in j to i graphs. So to get the optimal solution we basically have to iterate over all jś less than i. Since G_{j+1} and G_j are different, we add an extra K. They can be same as well and that case is taken care of. So
cost[i] = min{cost[j] + (i-j)*(ShortestPath(Merge(G_{j+1} .... G_i))) + K}
Now since we have correctly computed cost[j], and we are iterating over all the possible cases, therefore cost[i] is computed correctly.
Hence proved.