IIT Kanpur

CS345A - AlgorithmsII

Assignment 4

Anjani Kumar
11101

Sumedh Masulkar
11736

March 11, 2014

Contents

1 Shortest path: An adventurous drive in Thar desert

1.1 Problem

Given a graph G containing nodes $\{s, d, junctions\}$, out of which some junctions are fuel-stations, Edges: the roads connecting the nodes, and the maximum capacity c of the bike before it needs refilling. We need to find shortest feasible path from s *to* d, if it exists, otherwise notify that it does not exists.

1.2 Brief Description:

The problem can be modelled using 2 graphs:
$G(V, E)$:

- The nodes V are junctions present in the desert.
- The edges E are the roads connecting those junctions.

$G_f(V_f, E_f)$:

- The nodes V_f are the start, end, and the fuel stations.
- The edges E_f are the *feasible* roads connecting those junctions.
- Initially, G_f contains no edges.

The algorithm returns the shortest feasible path between s and d, if possible. Else, it returns no path.
We are using **Dijkstra's** algorithm, multiple times to find the feasible path.

Construction of G_f:

- insert s
- Begin Dijkstra's algorithm from s in graph G and insert the edges from s to other nodes of G_f accordingly.
- For all fuel stations in G_f, begin Dijkstra's algorithm from them in graph G, update the edges in G_f according to minimum distance between nodes.
- Delete all the edges in G_f whose weights are more than capacity of bike (c).

Functions used:

- Dijkstra's_Mod(G_f,G,u)
  Runs Dijkstra's algorithm from node u in graph G and update edges in graph G_f accordingly.
- update(G,u,v,w)
  In the input graph G, it inserts an edge between nodes u and v with weight w, if there was no edge previously. Otherwise, if $w < weight_{(u,v)}$ it updates the weight of edge (u,v) to w.

- Delete(G,c)
  Deletes any edge in input graph G whose weight is more than c.

- Pathfinder(T,u,v)
  Returns the path from u to v in tree T using BFS traversal, if it exists. Else it returns NULL.

1.3 Pseudo Code:

Algorithm 1: Pseudo-code for Dijkstra's_Mod(G_f,G,u)

```
Tree ← Dijkstra's(G,u)
for ∀ v ∈ T do
    if v ∈ {s, d, fuel − pump} then
        Insert(G_f,u,v,w)
```

Algorithm 2: Pseudo-code for Find_Path(G, s, d, c)

```
Create a graph G_f(V_f, E_f) with V_f ← {s, d, fuel − stations}
E_f ← NULL
Dijkstra's_Mod(G_f,G,s)
for ∀ u ∈ G_f - {s} do
    Dijkstra's_Mod(G_f,G,u)
Delete(G_f, c)
Path ← Dijkstra's(G_f, s)
return Pathfinder(Path, s, d)
```

1.4 Proof Of Correctness

2 cases are possible for a given input G:

- Case 1: If there is no feasible path in G.
- Case 2: If $\exists$ a feasible path from s to d in G.

Claim: *In case 1, there would be no path from s to d in graph G_f.*
Proof: Case 1 can be further divided into 2 cases:

- Case 1A: There is no path from s to d in G.
  - We are using the result of dijkstra's algorithm applied on graph G to update edges in G_f. Hence, a path from s to d in G_f in this case is not possible.
- Case 1B: There is a path from s to d but it is not feasible.

- In this case, some edges in the path have weights $> c$.
- The Delete function used by the algorithm removes all the edges E_f in G_f such that $w_{G_f} > c$. Hence, there is no path in G_f between s and d.

Claim: *In Case 2, There will always be a path from s to d in G_f.*
Proof: Case 2 can be further divided into 2 cases:

- Case 2A: The path requires no refilling.
  - In this case, $\exists$ a path from s *to* d with distance $\leq c$.
  - Since all the shortest edges in G with *weight* $\leq c$ are kept in G_f, therefore G_f will also contain that path.
- Case 2B: The path requires refilling.
  - In this case, the path has distance $> c$, and has edges with *weight* $\leq c$.
  - Using the **shortest sub-path** rule, if the given feasible path between s and d is the shortest, then all its edges will also have the minimum weight possible.
  - Since we are using **Dijkstra's** algorithm to build edges in g_f, at the end of the algorithm, all the edges in G_f will be of minimum possible weight.
  - Hence the optimal path will also be present in G_f.

1.5 Time Complexity

- **Update** take $O(1)$ time.
- **Dijkstra's_Mod** takes $O(mlog\ n)$ time.
- **Delete** takes $O(m)$ time.
- **Pathfinder** performs simple BFS traversal, hence it takes $O(m+n)$ time.
- **Dijkstra's_Mod** is called fo evey node in G_f.
- Hence the total time taken T is:

$$T = c_1 O(mlog\ n) + c_2 n O(mlog\ n) + c_3 O(m+n)$$
$$\text{T} = O(mnlog\ n)$$

2 Optimizing your time during exam days

2.1 Brief description

- In order to maximize the average, we need to maximize the sum of all the grades obtained in different courses.

- The problem can be solved using Dynamic Programming.

- The optimal solution for the grades would follow the given recursion:

$$Opt(i,h) \;=\; max_{j\ \in\ 0\ to\ h}(f_i(j) \;+\; Opt(i-1,h-j))$$

- We are using 2 matrix: *grade* and *max_time*.
- $grade[i][j]$ stores the optimum grade for i courses that are given a total of j hours.
- $max\_time[i][j]$ stores the hours required by the i^{th} course when a total of i courses are given j hours.
- For a single course, $grade[1][h]$ is simply $f_1(h)$
- In case of finding optimal solution for multiple courses, the algorithm iteratively computes the *grade* and *max_time* matrix using the formula in 2.1.
- The hours required for each course can be calculated using back tracking.

2.2 Functions used

- Optimize(n,H)
  It computes the matrix $grade[i][j]$ and $max\_time[i][j]$.
- getHours(n,H)
  It returns the amount of hours required for each course to achieve maximum grades using backtracking. Suppose the input is (i,h) Then the hours for course $i-1$ will be given by max_time$[i-1][h-max_time[i][h]]$.

2.3 Pseudo code

Algorithm 3: Pseudo code for Optimize(n, H)

```
for h ← 0 to H do
    grade[1][h] ← f_1(h)
    max_time[1][h] ← h
for i ← 2 to n do
    for h ← 0 to H do
        temp_grade ← 0
        for j ← 0 to h do
            if (f_i(j) + grade[i − 1][h − j] ≥ temp_grade) then
                temp_j ← j
                temp_grade ← f_i(j) + grade[i − 1][h − j]
        grade[i][h] ← temp_grade
        max_time[i][h] ← temp_j
```

Algorithm 4: Pseudo code for getHours(n, H)

```
h ← H
for i ← n to 1 do
    hour[i] ← max_time[i][h]
    h ← h - hour[i]
```

Algorithm 5: Pseudo code for main()

```
Optimize(n, H)
getHours(n, H)
```

2.4 Proof of Correctness

To Prove:

$$Opt(i, h) \;=\; max_{j\ \in\ 0\ to\ h}(f_i(j) \;+\; Opt(i-1, h-j))$$

Where $Opt(i, h)$ *is the optimal solution for i courses given a total of h hours.*

Proof by Induction

- Base Case:
  In case of only 1 course, entire hours will be given to that course since $f_i(h)$ is non decreasing.

- Induction Hypothesis:
  Let us assume that the theorem is true for $i-1$ courses. Hence, for $\forall\ h\ \in H$, we know the optimal solution for $i-1$ courses.

- Induction step:
  For the i^{th} course, the **Optimize** function takes the following sum for $\forall\ j\ \in h$:

  $$f_i(j)\ +\ Opt(i-1, h-j)$$

  and keeps the maximum value among those j.
  This step is repeated for $\forall\ h\ \in H$, so using optimal solution for $(i-1)^{th}$ step, we have got optimal solution for i^{th} step.

- Hence proved.

2.5 Time Complexity

As can be seen from the pseudo code, it take $O(nH^2)$ to perform **Optimize** and it takes $O(n)$ time to perform **getHours**.
Hence the overall time complexity is $O(nH^2)$.

2.6 Space Complexity

We are using 2 matrix of size n x H. Therefore total space taken is $O(nH)$.

3 Dynamic Programming again

3.1 problem

Given a set of graphs $\{G_0, G_1, ..., G_b\}$, each containing n nodes such that $\{s, t\} \in$ the vertices of all graphs.
We have to find a polynomial time algorithm to find the sequence of paths $P_0, P_1, ..., P_b$ of minimum cost given by:

$$cost(P_0, P_1, ..., P_b)\ =\ \sum_{i=0}^{b} l(P_i)\ +\ k * changes(P_0, P_1, ..., P_b)$$

where, $l(P_i)$ is the number of edges in P_i
and $changes(P_0, P_1, ..., P_b)$ is the number of indices i $(0 \leq\ i\ \leq\ b-1)$ for which $P_i \neq P_{i+1}$

3.2 Brief Description

- The problem is solved using Dynamic programming.
- Let G = $\{G_0, G_1, ..., G_b\}$ be a set of graphs with n nodes in each graph.
- We define Intersect(i, j) as:

$$Intersect(i,j) \;=\; \cap_{k=i}^{k=j}(G_k)\forall(0 \le i \le j \le b)$$

  The graph $Intersect(i, j)$ obtained contains n nodes and the common edges in graphs $G_i, G_{i+1}..., G_j$. Each $Intersect(i, j)$ is given the following parameters:

  - $shortest(i, j)$: It is the number of edges in the shortest path from s to t in $Intersect(i, j)$.
  - $sPath(i, j)$: It is the shortest path from s to t in $Intersect(i, j)$.

- Define $costMin(i)$ for a graph G_i such that $costMin(i)$ stores the minimum of $cost(P_0, P_1, ..., P_i)$ in the set of graphs $\{G_0, G_1, ..., G_i\}$. The following recursion holds for $costMin(i)$:
  $costMin(i)$=

$$min((i+1) * shortest(0,i), min_{j=0}^{j=i}(costMin(j) + (i-j) * shortest(j+1,i) + k))$$

- This computation can be done iteratively by incrementing i from $(0\ to\ b)$ as $costMin(i)$ depends on $shortest(i, j)$ and $\{costMin(j)\ \forall\ 0 \le j \le i\}$; both of which have already been computed beforehand.
- The algorithm maintains the sequence $(P_0, P_1, ...)$ in the following manner:
  - Suppose the $(i-1)^{th}$ sequence $(P_0, P_1, ..., P_{i-1})$ is computed correctly.
  - In the i^{th} iteration, $costMin(i)$ comes out to be minimum for some $(0 \le j \le i)$. In that case, the sequence $(P_{j+1}, ..., P_i)$ is updated to $sPath(j, i)$ and the rest of the sequence remains as it is.

3.3 Functions used

- $pathfinder(G, s, t)$: It performs BFS traversal on graph $Intersect(i, j) = \cap_{k=i}^{k=j}(G_k)\forall(0 \le i \le j \le b)$ and returns the shortest path, and the length of that path in $Intersect(i, j)$.

3.4 Pseudo code

Algorithm 6: Pseudo code for sequence(G,s,t,b,k)

```
pathfinder(G, shortest, sPath) for i ← 0 to b do
    min ← (i + 1) ∗ shortest(0, i)
    pos ← 0
    for j ← 0 to i − 1 do
        if min > costMin(j) + (i − j) ∗ shortest(j + 1, i) + k then
            min ← costMin(j) + (i − j) ∗ shortest(j + 1, i) + k
            pos ← j + 1
    costMin(i) ← min
    for j ← 0 to i do
        if j ≤ pos − 1 then
            minPath[i][j] ← minPath[pos − 1][j]
        else
            minPath[i][j] ← sPath(pos, i)
return minPath[b]
```

3.5 Proof of Correctness

To Prove:
$costMin(i) =$

$$min((i+1) * shortest(0,i), min_{j=0}^{j=i}(costMin(j) + (i-j) * shortest(j+1,i) + k))$$

Where $costMin(i)$ for a graph G_i such that $costMin(i)$ stores the minimum of $cost(P_0, P_1, ..., P_i)$ in the set of graphs $\{G_0, G_1, ..., G_i\}$.
Proof by Induction

- Base Case:
  In case of only 1 graph, shortest(0,0) will store the length of shortest path between s and t in G(0,0) , which is trivial to prove.
- Induction Hypothesis:
  Let us assume that the theorem is true for $i - 1$ graphs.
- Induction step:
  After inserting i^{th} graph, $\exists$ an optimal path sequences $(P_0, P_1, ...P_i)$, such that $P_i = P_{i-1} = P_{i-2}...... = P_{j+1} \neq P_j$, the cost for graphs $(G_0, G_1, ...G_i)$ will be costMin(j) + (i-j)*shortest(j+1,i) + K(penalty for $P_{j+1} \neq P_j$). Therefore, finding all such possibilities $\forall j < i$, and taking minimum cost will give the optimal solution for i.
- Hence proved.

3.6 Time complexity

- $pathfinder(G, s, t)$: pathfinder computes $Intersect(i, j)$ $\forall(0 \leq i \leq j \leq b)$. $\exists$ $O(b^2)$ pairs of (i, j). For finding the intersection, atmost b graphs are examined and each graph can have atmost $O(n^2)$ edges. Therefore each (i, j) pair takes $O(bn^2)$ time, and for $O(b^2)$ pairs, time complexity would be $O(n^2b^3)$.

- $sequence(G, s, t, b, k)$: From the pseudo code, its time complexity is $T(pathfinder)+$ $C * O(b^2)$. Hence the overall time complexity is $O(n^2b^3)$.