To find the shortest path from source to destination using dynamic programming.

Raushan Raj - IIT2018031, Rithik Seth - IIT2018032, Mrigyen Sawant - IIT2018033
Btech IT Department, Semester-4

Indian Institute Of Information Technology, Allahabad
30 March 2020

Abstract: In this paper, we have devised an algorithm to find* the shortest path from source to destination using dynamic programming. *Later we have discussed its space-time complexity through both apriori and aposteriori analysis.

Keywords: Dynamic Programming, Graphs, Shortest path, Path Finding.

1. Introduction

Pathfinding simply means finding a route between two points in a network/graph based on certain criteria such as cheapest, shortest, fastest. The most obvious applications arise in transportation or communication, such as finding the best route from Allahabad to Mumbai or figuring how to direct packets to a destination across a network.

Here we start our approach for finding the minimum cost path after considering the following assumptions.

The graph we are considering here is weighted directed graph which may contain negative weighted edges but should not contain any negative cycles.

In this paper, we have devised an algorithm where we try to look for a path from source to destination in botton-up fashion using dynamic programming. It first calculates the shortest paths using at-most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. The idea is the path from the source to destination can have at max (V - 1) edges. We repeat this process for (V -1) times over all edges since the path length can't exceed (V-1) edges. Hence, we get the shortest path from source to destination in this way. The mentioned algorithm is not in-place and uses a cost and parent arrays for above calculations.

2. Algorithm Proposed

Assumptions: Let the list of edges be EdgeList[] of type 'struct Edge' whose structure is mentioned below. Also, we are assuming the input graph doesn't contain any negative weight cycle although it may contain negative weights.

```
struct Edge{
        int from_vertex;
        int to_vertex;
        float weight;
};
```

Algorithm: To find the shortest path from source to destination through dynamic programming.

Input: The first line will contain two positive number V (the number of vertices) and E (the number of edges) such that V > 0 and E > 0. The next E lines will contain three numbers each separated by space let's say F T W denoting edge from F to T with weight W. Last line will contain source (S) and destination (D) vertices each separated by space.

Output: Print minimum cost path (if exists) with the cost itself otherwise print "No path exists".

Step 1: Start
Step 2: Read V and E.

Step 3: Initialize
i ← 0
EdgeList[E] ← []

Step 4: Repeat steps until i = E.
Read EdgeList[i].from_vertex
Read EdgeList[i].to_vertex
Read EdgeList[i].weight
Increment i ← i + 1

Step 5: Read S and D (source and destination).

Step 6: Initialize
Cost[V] ← []
Parent[V] ← []
j ← 0

Step 7: Repeat steps until j = V.
Cost[j] ← INFINITY
Parent[j] ← -1
Increment j ← j + 1

Step 8: Initialize
Cost[S] ← 0
Parent[S] ← S
I ← 0

Step 9: Repeat step 9.1 until I = V-1.

Step 9.1: Initialize J ← 0

Step 9.2: Repeat steps until J = E
F ← EdgeList[J].from_vertex
T ← EdgeList[J].to_vertex
W ← EdgeList[J].weight

If Cost[F] + W < Cost[T] then
Cost[T] ← Cost[F] + W
Parent[T] ← F

Increment J ← J+1

Step 9.2: Increment I ← I + 1

Step 10: Check whether path exists or not.

If Cost[D] == INFINITY then
Print "No path exists"
else
Print path recursively by
calling printPath(S,D,Parent).

Print "Cost = "+ Cost[D]

Step 11: Stop

Algorithm : Pseudo code for printing path recursively from source to destination .

```
printPath(S, D, parent[])
{
    If S == D then,
        Print S
        return
    Else
        printPath(S,D,parent[D])
        Print D
}
```

3. Analysis and Experimental Results

3a. Apriori Analysis

Time complexity analysis:

	Time	Best F	Freq
Step 1	1	1	1
Step 2	2	1	1
Step 3	2	1	1
Step 4	5E + 1	1	1
Step 5	2	1	1
Step 6	3	1	1
Step 7	4V + 1	1	1
Step 8	3	1	1
Step 9	1	V	V
Step 9.1	1	V-1	V-1
Step 9.2	8	(V-1)E	(V-1)E
Step 9.3	2	V	V
Step 10	V + 1	1	1
Step 11	1	1	1

Worst case complexity:

T $\propto$ 1×1 + 2×1 + 2×1 + (5E + 1)×1 + 2×1 + 3×1 + (4V + 1)×1 + 3×1 + 1×V + 1*(V - 1) + 8×(V - 1)E + 2×V + (V+1)*1 + 1×1

T $\propto$ 5 + 5E + 1 + 5 + 4V + 1 + 3 + V + V - 1 + 8VE - 8E + 2V + V + 1 + 1

T $\propto$ 8VE + 9V - 3E + 16
T $\propto$ VE

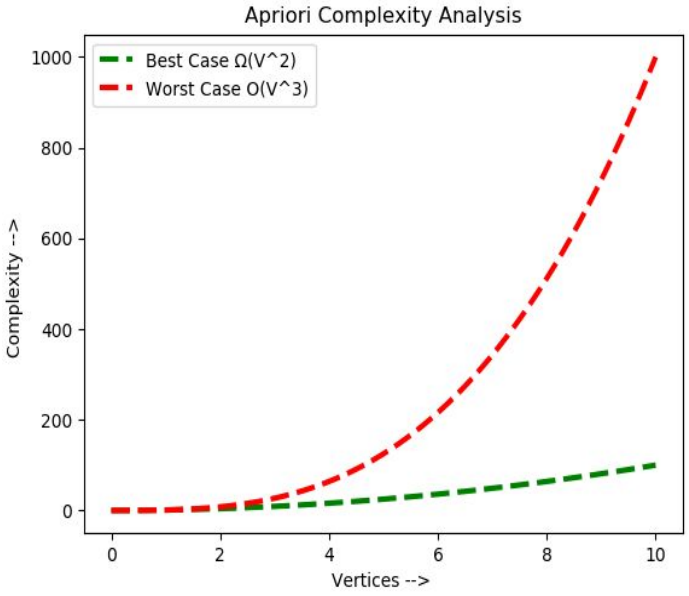
The following algorithm will show worst case performance when the input graph is complete.
E = V(V - 1)/2

$\mathbf{T_W} \propto V^2(V - 1)/2$
$\mathbf{T_W} \propto V^3$
$\mathbf{O(V^3)}$

Best case complexity:

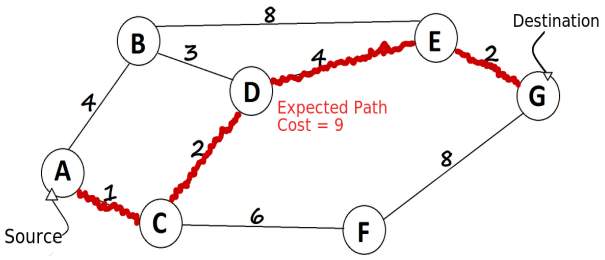
Best case complexity can be observed when the order of edges is comparable to the number of vertices i.e. E ≈ V then,

$\mathbf{T_B} \propto VE$
$\mathbf{T_B} \propto V^2$
$\mathbf{\Omega(V^2)}$



The above line graph shows V vs complexity according to apriori analysis **(Theoretical)**. The actual complexity will lie between those two curves.

Sample Test case analysis :



We are considering the above graph for dry run analysis where we count the number of steps it took for execution and compare it with the theoretical data. Also, the vertices are considered alphabets unlike numbers for readability/simplicity purposes.

In this analysis, we are only considering the execution steps (6-9) of the algorithm i.e. input, printing results and other minor operations are ignored.

Given V = 7 and E = 9, Source = A, Dest. = G.

Initialization process:

	A	**B**	**C**	**D**	**E**	**F**	**G**
Cost	0	4	1	3	7	7	14
Parent	A	A	A	C	D	C	E

Steps Executed = 2V + 2 = 2(V + 1) = 16

Computational process :

1st Iteration : Paths containing at-most 1 edge.

	A	**B**	**C**	**D**	**E**	**F**	**G**
Cost	0	4	1	3	7	7	14
Parent	A	A	A	C	D	C	E

Steps Executed = 2×6 + 5×4 = 32

2nd Iteration : Paths containing at-most 2 edges.

	A	**B**	**C**	**D**	**E**	**F**	**G**
Cost	0	4	1	3	7	7	14
Parent	A	A	A	C	D	C	E

Steps Executed = 3×6 + 4×4 = 18 + 16 = 34

3rd Iteration : Paths containing at-most 3 edges.

	A	**B**	**C**	**D**	**E**	**F**	**G**
Cost	0	4	1	3	7	7	14
Parent	A	A	A	C	D	C	E

Steps Executed = 2×6 + 5×4 = 32

4th Iteration : Paths containing at-most 4 edges.

	A	**B**	**C**	**D**	**E**	**F**	**G**
Cost	0	4	1	3	7	7	14
Parent	A	A	A	C	D	C	E

Steps Executed = 1×6 + 6×4 = 30

Rest iterations will not change the above arrays still the loop will be executed till the end.

Steps Executed = 7×4×2 = 56

Total steps (TS) = 16 + 32 + 32 + 34 + 30 = 144 or 12^2

Expected Steps = Within order of V^2 - V^3

Result - Obeys the above expectation.

$$7^2 < TS < 7^3$$

$$49 < 144 < 343$$

Hence, the above analysis obeys the calculated theoretical results of apriori analysis.

Space Time Complexity:

The above algorithm consumes 2 units of V sized arrays for cost and parent arrays. Therefore, the overall space-time complexity will be :

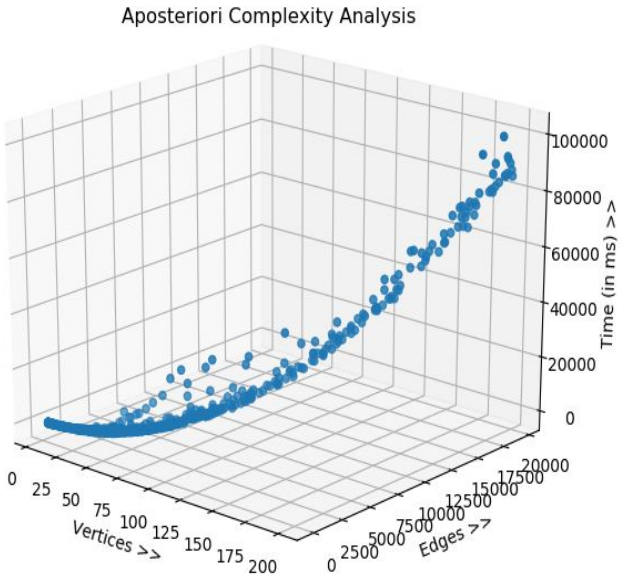
M ∝ 2V
O(V)

3b. Aposteriori Analysis

In the below 3d scatter graph (Experimental), the x-axis represents V (number of vertices), y-axis represents E (number of edges) and z-axis represents time taken in microseconds. The following graph shows similar results as that of apriori analysis up to certain extent with slight deviations at certain regions.

Time complexity analysis:

V	E	Time (in ms)
40	775	437
26	331	350
93	4160	9120
92	4102	10575
101	4949	10769
198	18849	100153
163	12934	46810
194	18297	81788
200	19520	85605



4. Conclusion

From the above analysis, the mentioned DP based single source pathfinding algorithm shows worst case and best case complexities of **O(V^3)** and **Ω(V^2)** respectively. **Although this algorithm becomes less efficient for complete graphs but more versatile as it can operate on negative weighted graphs. Therefore, the actual running time will lie within Ω(V^2) - O(V^3).**