Introduction to Algorithms

Project Report

Hina Garg (A20342375)

12/6/2015

Table of Contents

Project Report

Introduction:

Problem statement:

To find the shortest path between a source and target node in a given graph G(u ,v) by using three different algorithms as mentioned below:

Dijkstra's Algorithm: It is a single source shortest path algorithm that finds the shortest path from a starting vertex u that belongs to a set of vertices V in a graph G to all other vertices in the graph. It has a limitation of not giving correct shortest paths for graphs having negative edge weights or negative edge weight cycles.

Bellman Ford Algorithm: This algorithm also computes shortest paths from a single source to all other vertices in a graph but it is slower than Dijkstra's algorithm. However, it has an advantage of computing shortest paths in a graph having negative edge weights.

Floyd-Warshall Algorithm: This algorithm finds the shortest paths in a graph having positive or negative edge weights. However it does not work for graphs having negative edge cycles.

All three algorithms stated above are explained and implemented in this project. Comparison of the three algorithms and their time complexity analysis is done only for graphs having positive edge weights.

Design of the Algorithm & Pseudo Code

Dijkstra's Algorithm

1. We start this algorithm by selecting a source node and then we go along the graph till all the vertices are covered to calculate shortest path distance for each other vertex from the source vertex.
2. Initially we assume that all other vertices are very far from the source vertex.
3. So we assign every node a temporary distance value. The source node is assigned a distance value of 0. We also maintain a priority dictionary to keep track of the visited nodes and an estimate of their distances from the source node.
4. For every current node, we calculate the value of estimated distances of adjacent nodes. If this value is smaller than the current assigned value to these adjacent nodes in the priority dictionary, we update their value in the dictionary.
5. When all the nodes have been covered in the graph i.e. we have final distances of all the nodes in the priority dictionary or we have no smaller estimate of distances than the one already listed in priority dictionary. We stop the execution of the algorithm.
6. END

Pseudo Code

```
function dijkstra_algorithm(graph,start,target=None):
    final_dist_dict = {} // a dictionary to  maintain final distances
    dict_predecessor = {} // a dictionary to maintain predessor of nodes
    PD = priorityDictionary() // a priority dictionary using binary heaps
    PD[start] ← 0 //initialize 0 to the start vertex
    for each vertex u in PD:
        final_dist_dict[u] ← PD[u] // vertex u in PD with minimum distance
        check if u = target: break
        // Compare the value of tentative distances with the assigned values and
    evaluate the shortest path
        for each vertex v in graph[u]:
            uvLength ← final_dist_dict[u] + graph[u][v]['weight']
            check if v in final_dist_dict:                    // check if a shorter path
        exist for an already finalized node and trigger an error
                check if uvLength  < final_dist_dict[v]:
                    raise ValueError
            else check if v not in PD or uvLength < PD[v]: // if found a better
        path update the distance value in PD
                PD[v] ← uvLength
                dict_predecessor[v] ← u
    return (final_dist_dict,dict_predecessor)
```

Bellman-Ford Algorithm

1. Initialize the source node to be at a distance 0 and all other nodes to be at infinity as we assume them to be very far away from source node.
2. We execute a couple of iterations until our graph converges
3. In each iteration we look at the neighbor of the current node and for every neighbor we calculate their tentative distances from the current node.
4. If calculated distance for a node is smaller than the one currently assigned to it, then we update the node.
5. Also, update the predecessor dictionary defined to update the predecessor of the node.
6. We also check if there is a negative-weight cycle in the graph and trigger an error if there exist any such cycle.

7. End

<u>Pseudo code</u>

```
function bellmon_ford_algorithm(graph, start, target=None)
        destination_dict = {} // destination disctionary to maintain final distances
        predecessor_dict = {} // predecessor disctionary to maintain final distances
//  Section 1:  Below for loop takes O(|V|) time to complete
        for each node in graph:
                destination_dict[node]← float('inf') // initialize all nodes to have a distance of infinity
                predecessor_dict[node]← None // predecessor_dict is assigned null
                destination_dict[start] ← 0 //assign 0 distance value to the source node
// Section 2:  Below for loop takes O(|V||E|) time to complete
        for each iteration in graph:     // we run this until graph converges
                for each u in graph:
                        for each v in graph[u]:   //run for each neighbor of u
                                if destination_dict[v]>destination_dict[u] + graph(u,v→weight):
                                        destination_dict[v]=destination_dict[u] + graph(u,v→weight):
                                        predecessor_dict[v] = u


// Section 3:  Below for loop takes O(|E|) time to complete
        // now check for negative-weight cycles and trigger and error if there is one such cycle
        for u in graph:
                for v in graph[u]:
                        trigger if error destination_dict[v] <= destination_dict[u] + graph(u,v→weight)
        return destination_dict, predecessor_dict
```

Floyd-Warshall algorithm

1. Initialize a final distance dictionary and a predecessor dictionary to store predecessors.
2. In the final distance dictionary, first assign the nodes distances having direct path between them in the graph. Add infinity to the distances of the nodes which are not reachable from other nodes and zero to the distances of the nodes from itself.
3. Iterate until the graph converges.
4. In each iteration, calculate the shortest path from the current node u to the destination node v and evaluate if distance from node u to i (node in between u and v) when added to the distance from node i to v, is smaller than the current assigned value to the node v.
5. Update the tentative distance value to the nodes found in step 6.
6. End

<u>Pseudo code</u>

```
function floydWarshall_algorithm(graph, start, target = None):
    final_distance_dict = {}  //final destination dictionary storing the final distance values
    pred_dict = {}  //
    for each vertex u in graph:
        final_distance_dict[u] = {} // initialization
        pred_dict[u] = {}
        for each vertex v in graph:
            final_distance_dict[u][v]['weight'] ← 1000  //assign very very large value
        to non reachable nodes(infinity)
            pred_dict[u][v] ← -1
        final_distance_dict[u][u]['weight'] ← 0  //assign 0 distance values to the nodes
    from themselves
        for each adjacent node in graph[u]:
            final_distance_dict[u][adjacent_node]['weight'] ←
        graph[u][adjacent_node]['weight']
            pred_dict[u][child_node]  ←u  // map direct node distances from graph to
        dict
//  Below for loop takes O(|V^3|) time to complete
    // Compare the distances and evaluate shortest path
    for each iteration i in graph:
        for each vertex u in graph:
            for each vertex v in graph:
                updatedist ← final_distance_dict[u][i]['weight'] +
            final_distance_dict[i][v]['weight']
                check if updatedist < final_distance_dict[u][v]['weight']:
                    final_distance_dict[u][v]['weight'] ← updatedist
                    pred_dict[u][v] ← pred_dict[i][v] // update predecessor to
                route through i
    return final_distance_dict[start] , pred_dict[start]
```

Program Implementation:

Programming language used = Python

Data structures used = Dictinaory, List, Set, Priority Dictionary

Package = Available package of Python called as “NetworkX” is used to generate and manipulate the auto generation of graphs.

Experiment Methodology:

Several experiments are performed by generating many random graphs as the input to the programs implemented for all the three algorithms.

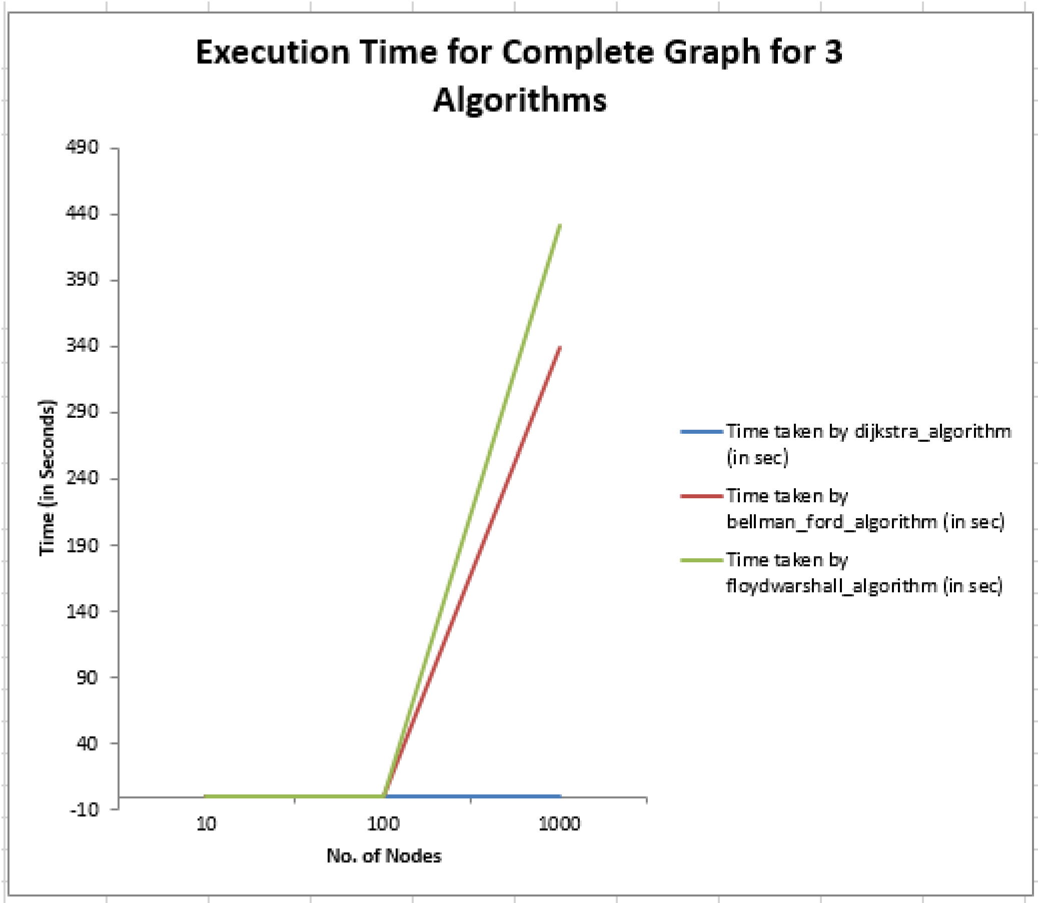
a. Giving different types of graphs (like Complete, Star, Random, Path and Cycle) as input for each algorithm

b. Changing the number of nodes/edges (like 10, 100, 1000) for each graph

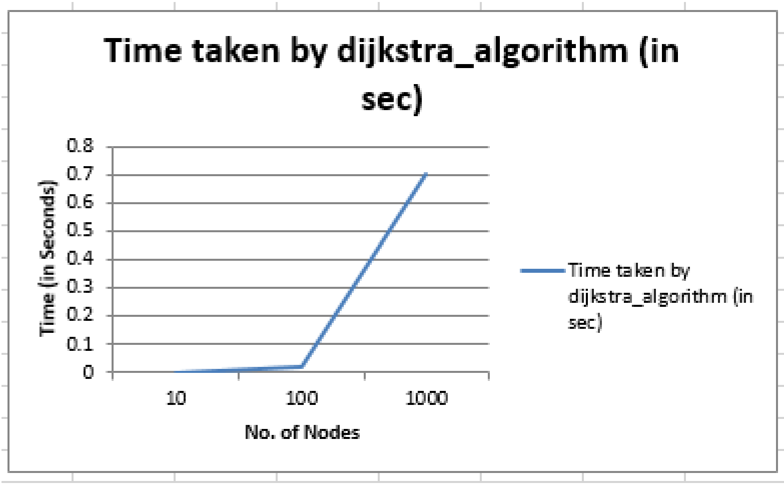
d. Measuring the time taken to complete a certain task for a certain input graph.

Experiment Observations:

Graph 1: Execution time for "Complete Graph" for 3 different algorithms



Graph 2: Execution time for "Complete Graph" for Dijkstra Algorithm



Note: Detailed graph for all the experiments are mentioned in the Appendix section of the report

Observations from the five experiments:

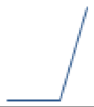
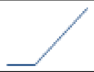
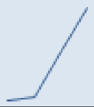
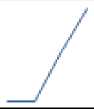
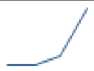
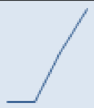
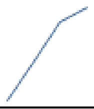
a. As the no. of nodes increase from 10 to 1000, the execution time taken by each of the three algorithms increases for each algorithm.

b. For same number of nodes, the difference in execution time for various algorithms - is very small when the number of nodes are less (i.e. 10 nodes) but it is very large with higher number of nodes (i.e. 1000 nodes). For e.g. The execution time difference for 10 nodes is in milliseconds between individual algorithms, but for 1000 nodes the time difference is 92 seconds between Bellman_ford and Floyd_Warshall algorthms.

c. It shows that Dijkstra algorithm gives the best results.

d. A sharp rise in execution time is observed from Dijkstra to Floyd_warshall algorithm. It can be explained by the time complexity observed in the pseudocode of each algorithm (Dijkstra = $O(|E| + |V| \log |V|)$, Bellman = $O(|V||E|)$ and Floyd_warshall = $O(|V|^3)$.

Comparison between different input graphs (for the same number of nodes)

Input Graph Name	Cycle Graph	Star Graph	Random Graph	Complete Graph	Graph
number of nodes are	10	10	10	10	
Time taken by dijkstra_algorithm (in sec)	0	0	0	0	
Time taken by bellman_ford_algorithm (in sec)	0	0	0	0	
Time taken by floydwarshall_algorithm (in sec)	0	0	0	0.01	
Input Graph Name	Cycle Graph	Star Graph	Random Graph	Complete Graph	
number of nodes are	100	100	100	100	
Time taken by dijkstra_algorithm (in sec)	0	0	0.01	0.02	
Time taken by bellman_ford_algorithm (in sec)	0.01	0.02	0.13	0.25	
Time taken by floydwarshall_algorithm (in sec)	0.48	0.48	0.49	0.5	
Input Graph Name	Cycle Graph	Star Graph	Random Graph	Complete Graph	
number of nodes are	1000	1000	1000	1000	
Time taken by dijkstra_algorithm (in sec)	0	0.02	0.13	0.7	
Time taken by bellman_ford_algorithm (in sec)	0.74	0.67	178.735	339.545	
Time taken by floydwarshall_algorithm (in sec)	426.49	428.627	430.729	431.599	

Observations:

a. The execution time for same number of nodes is:

Cycle Graph < Star Graph < Random Graph < Complete Graph

b. There is very less difference in the execution time between Cycle and Star graphs

c. The execution time of Random graph is almost average of the execution time of Cycle and Complete graphs

Conclusion:

It can be concluded that:

a. Djikstra algorithm gives least execution time for different types of graphs, among the three algorithms (Djikstra, Floyd Warshall, Bellman Ford)

b. As the number of nodes increases the execution time increases

c. As priority dictionary is being used for Djikstra, implementing binary heap, the worst case performance is observed as O(|E| + |V| log |V|).

d. As explained in the pseudo code of Bellman Ford (having 3 sections above), the worst case performance of this algorithm is O(|E| |V|).

e. As explained in the pseudo code of Floyd Warshall, the worst case performance of this algorithm is O(|V^3|).

References:

Dijskatras:

https://www.youtube.com/watch?v=mv4r7F82doA

http://code.activestate.com/recipes/117228-priority-dictionary/

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Bellman Ford:

https://www.youtube.com/watch?v=obWXjtg0L64

https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm

Floyd Warshal:

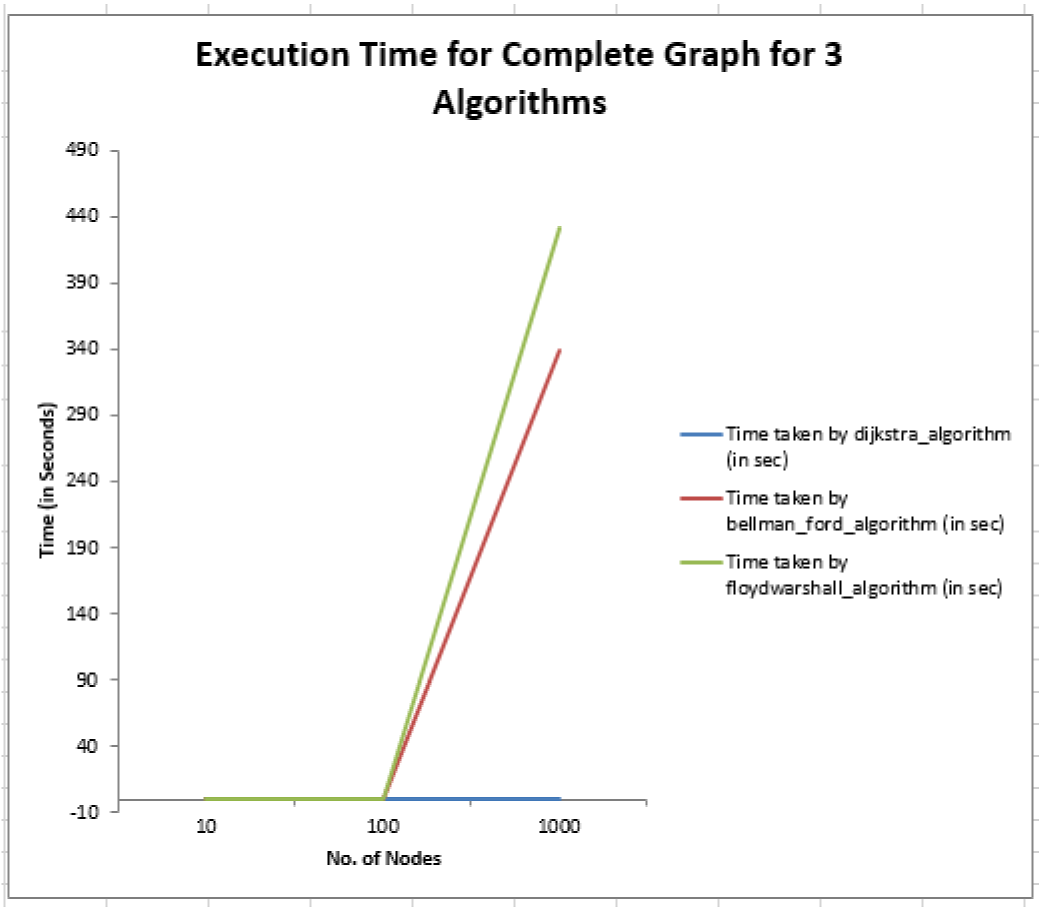
https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

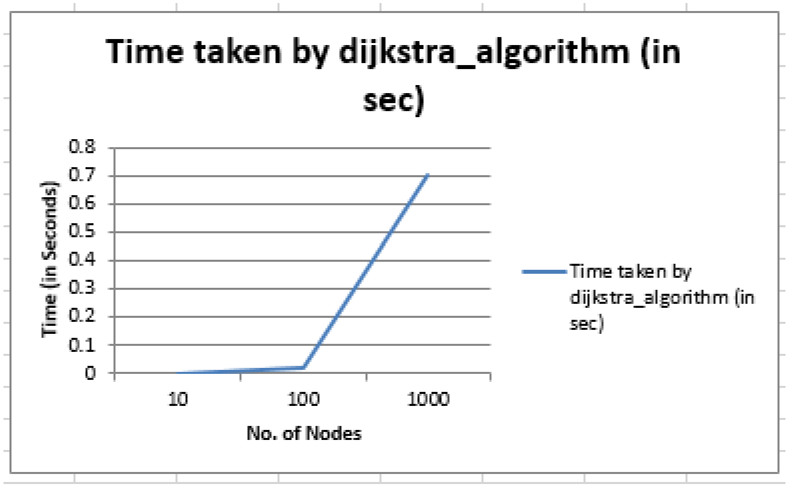
https://www.youtube.com/watch?v=KQ9zlKZ5Rzc

Appendix:

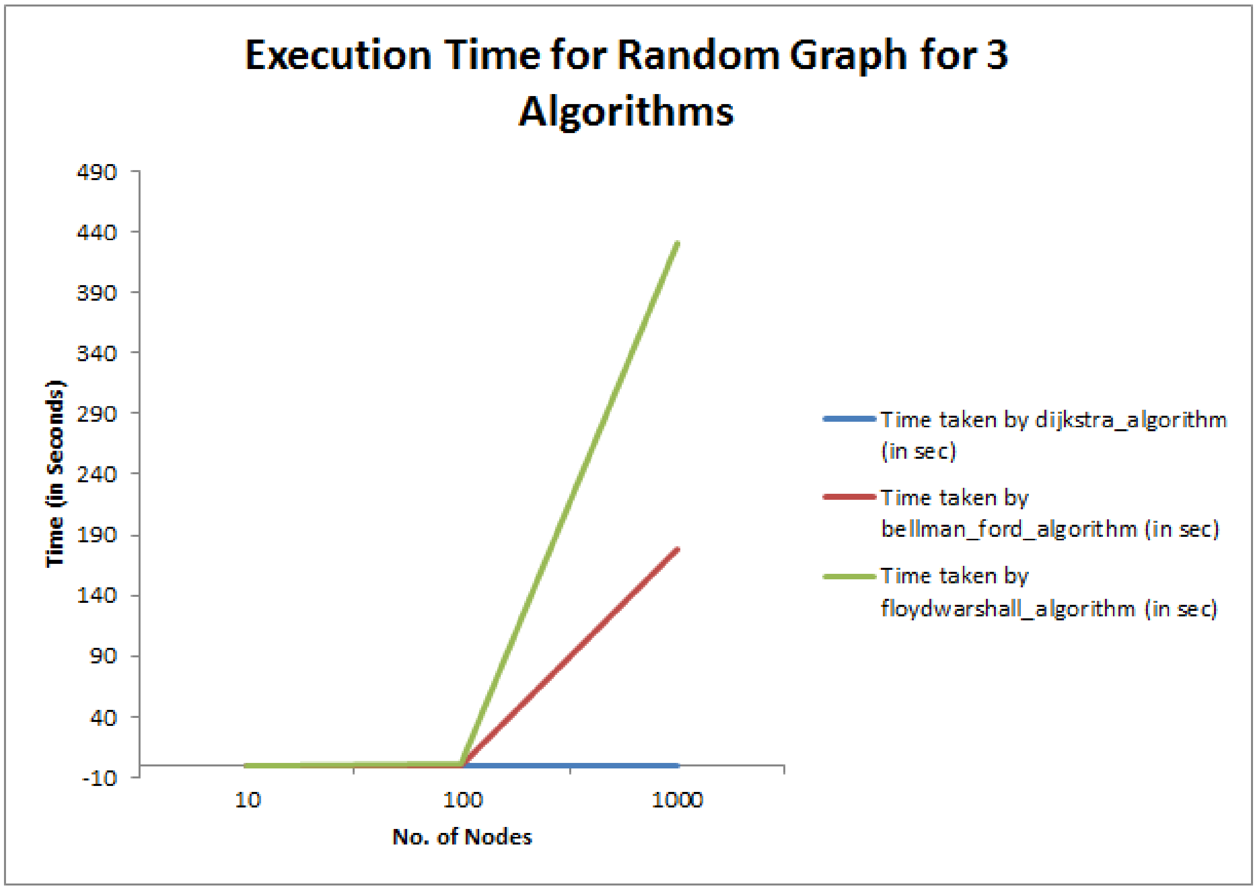
Experiments and Observations:

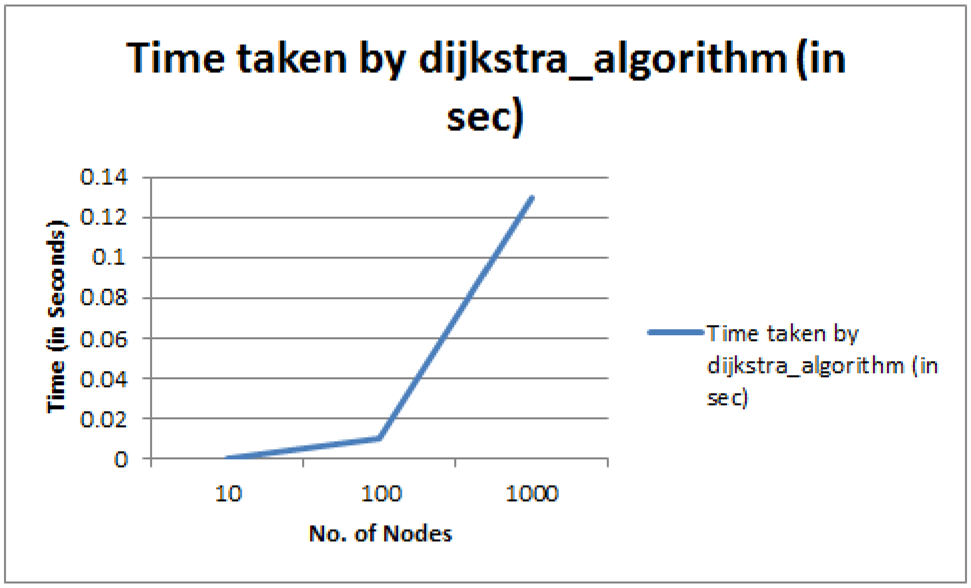
Experiment 1			
Input Graph Name	Complete Graph		
Output			
number of nodes are	10	100	1000
number of edges are	45	4950	499500
Time taken by dijkstra_algorithm (in sec)	0	0.02	0.7
Shortest path by dijkstra_algorithm is	[0, 4, 3, 8, 5, 2, 7, 9]	[0, 16, 58, 99]	[0, 406, 430, 433, 730, 999]
Time taken by bellman_ford_algorithm (in sec)	0	0.25	339.545
Shortest path by bellman_ford_algorithm is	[0, 4, 3, 8, 5, 2, 7, 9]	[0, 16, 58, 99]	[0, 406, 430, 433, 730, 999]
Time taken by floydwarshall_algorithm (in sec)	0.01	0.5	431.599
Shortest path by floydwarshall_algorithm is	[0, 4, 3, 8, 5, 2, 7, 9]	[0, 16, 58, 99]	[0, 406, 430, 433, 730, 999]


Execution Time for Complete Graph for 3 Algorithms
490
440
390
340
290
240
190
140
90
40
-10
Time (in Seconds)
10
100
1000
No. of Nodes
Time taken by dijkstra_algorithm (in sec)
Time taken by bellman_ford_algorithm (in sec)
Time taken by floydwarshall_algorithm (in sec)

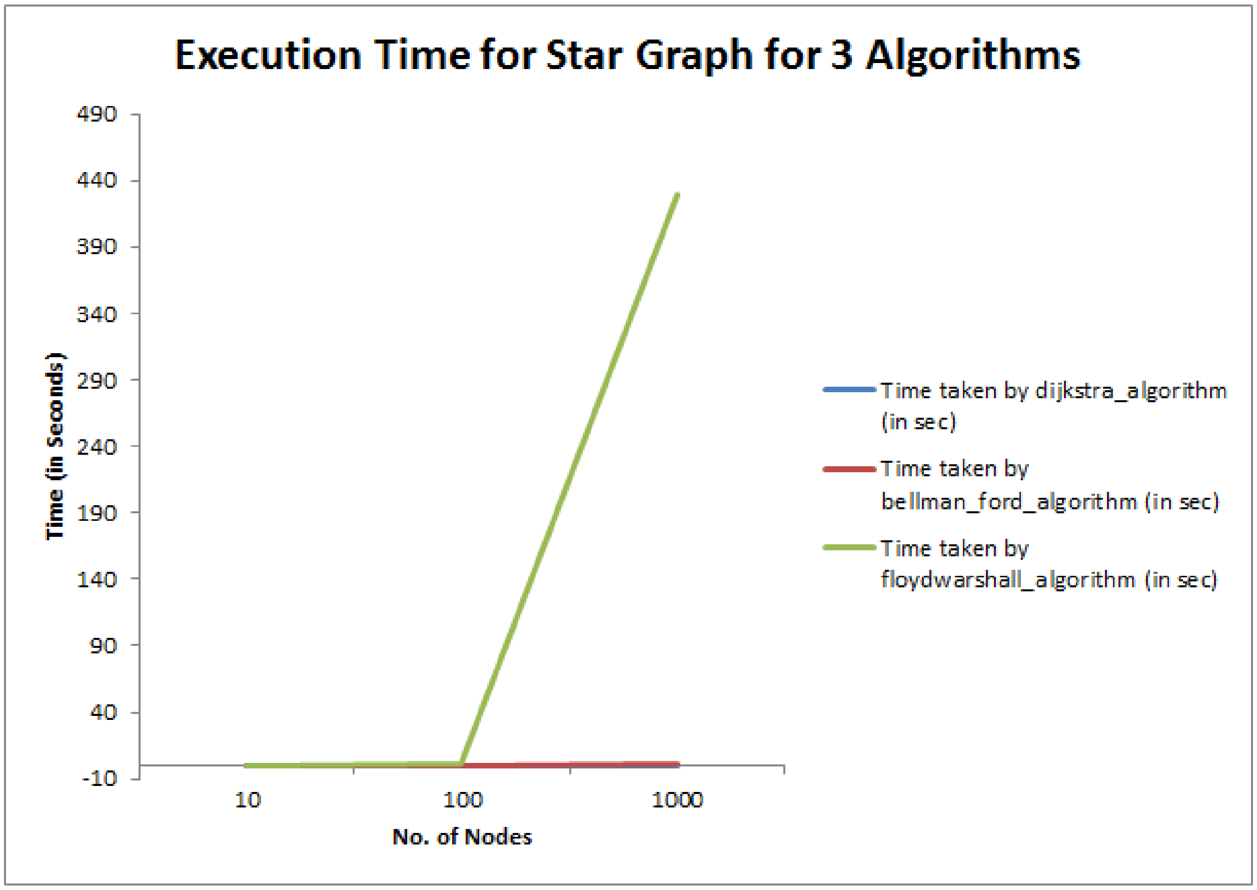

Time taken by dijkstra_algorithm (in sec)
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Time (in Seconds)
10
100
1000
No. of Nodes
Time taken by dijkstra_algorithm (in sec)

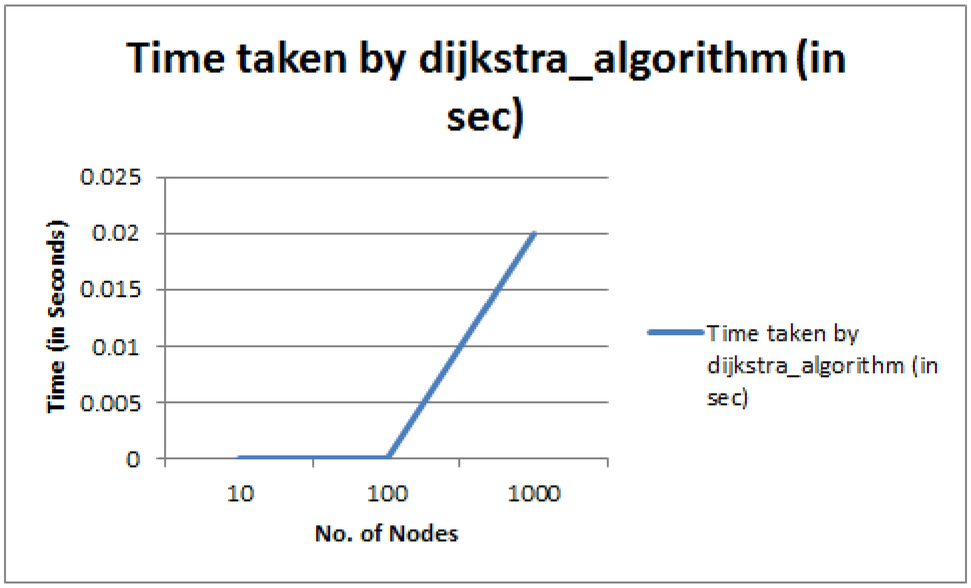
Experiment 2			
Input Graph Name	gnp_random_graph (Erdős-Rényi graph or a binomial graph.)		
P = 0.5			
Output			
number of nodes are	10	100	1000
number of edges are	24	2496	249415
Time taken by dijkstra_algorithm (in sec)	0	0.01	0.13
Shortest path by dijkstra_algorithm is	[0, 2, 6, 9]	[0, 61, 66, 99]	[0, 836, 369, 734, 515, 807, 999]
Time taken by bellman_ford_algorithm (in sec)	0	0.13	178.735
Shortest path by bellman_ford_algorithm is	[0, 2, 6, 9]	[0, 61, 66, 99]	[0, 836, 369, 734, 515, 807, 999]
Time taken by floydwarshall_algorithm (in sec)	0	0.49	430.729
Shortest path by floydwarshall_algorithm is	[0, 2, 6, 9]	[0, 61, 66, 99]	[0, 836, 369, 734, 515, 807, 999]


Execution Time for Random Graph for 3 Algorithms
Time (in Seconds)
490
440
390
340
290
240
190
140
90
40
-10
10
100
1000
No. of Nodes
Time taken by dijkstra_algorithm (in sec)
Time taken by bellman_ford_algorithm (in sec)
Time taken by floydwarshall_algorithm (in sec)

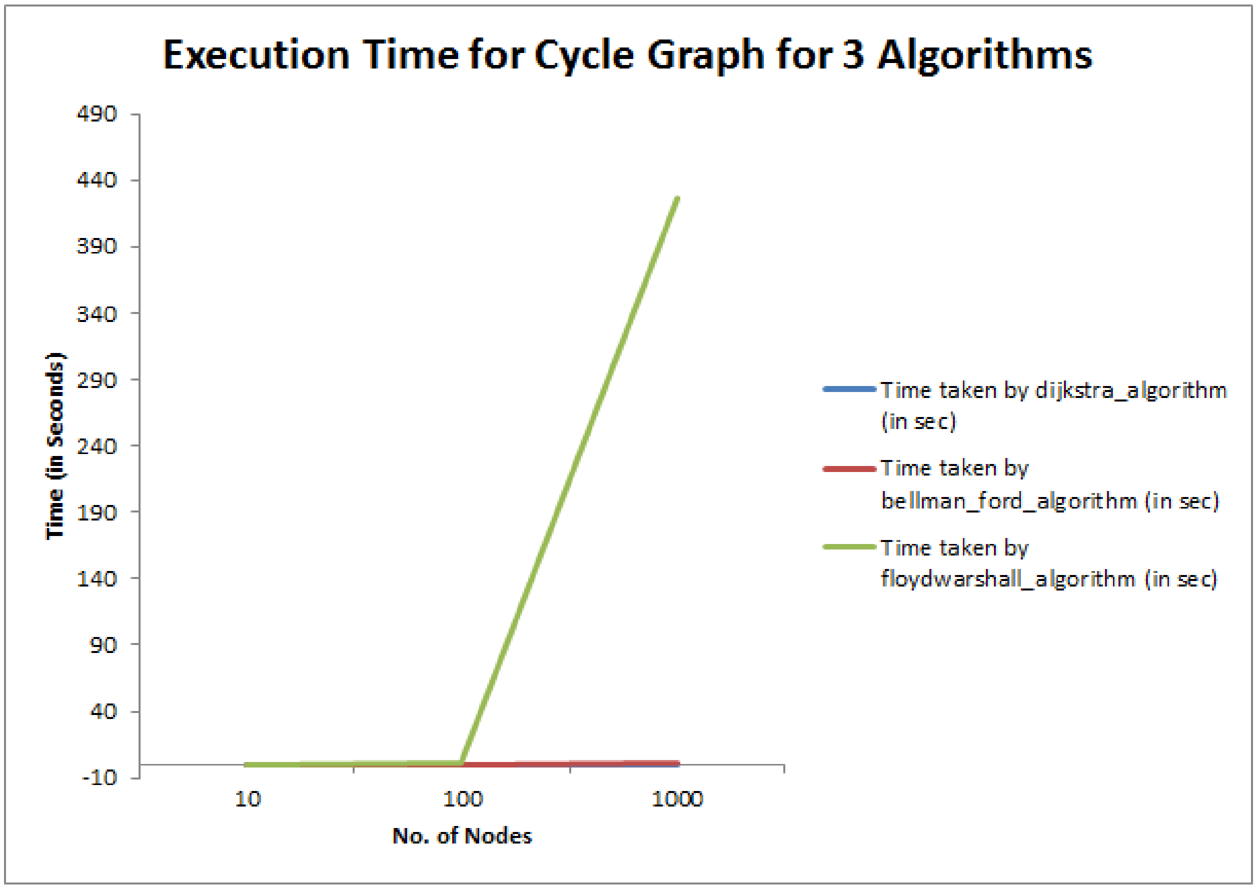

Time taken by dijkstra_algorithm (in sec)
Time (in Seconds)
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
10
100
1000
No. of Nodes
Time taken by dijkstra_algorithm (in sec)

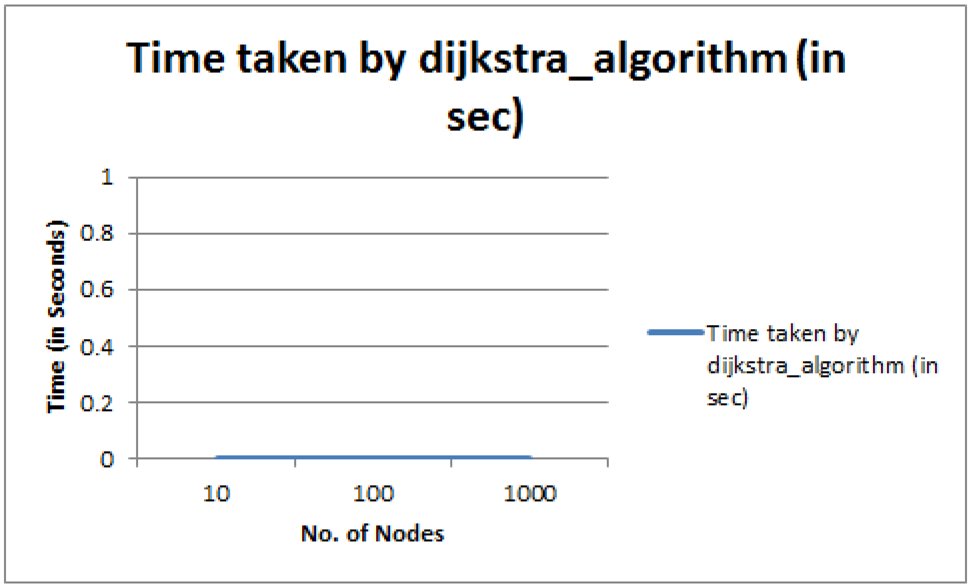
Experiment 3			
Input Graph Name	Star Graph		
Output			
number of nodes are	10	100	1000
number of edges are	10	100	1000
Time taken by dijkstra_algorithm (in sec)	0	0	0.02
Shortest path by dijkstra_algorithm is	[0, 9]	[0, 99]	[0, 999]
Time taken by bellman_ford_algorithm (in sec)	0	0.02	0.67
Shortest path by bellman_ford_algorithm is	[0, 9]	[0, 99]	[0, 999]
Time taken by floydwarshall_algorithm (in sec)	0	0.48	428.627
Shortest path by floydwarshall_algorithm is	[0, 9]	[0, 99]	[0, 999]





Experiment 4			
Input Graph Name	Cycle Graph		
Output			
number of nodes are	10	100	1000
number of edges are	10	100	1000
Time taken by dijkstra_algorithm (in sec)	0	0	0
Shortest path by dijkstra_algorithm is	[0, 9]	[0, 99]	[0, 999]
Time taken by bellman_ford_algorithm (in sec)	0	0.01	0.74
Shortest path by bellman_ford_algorithm is	[0, 9]	[0, 99]	[0, 999]
Time taken by floydwarshall_algorithm (in sec)	0	0.48	426.49
Shortest path by floydwarshall_algorithm is	[0, 9]	[0, 99]	[0, 999]


Execution Time for Cycle Graph for 3 Algorithms
490
440
390
340
290
240
190
140
90
40
-10
Time (in Seconds)
10
100
1000
No. of Nodes
Time taken by dijkstra_algorithm (in sec)
Time taken by bellman_ford_algorithm (in sec)
Time taken by floydwarshall_algorithm (in sec)


Time taken by dijkstra_algorithm (in sec)
1
0.8
0.6
0.4
0.2
0
Time (in Seconds)
10
100
1000
No. of Nodes
Time taken by dijkstra_algorithm (in sec)

Experiment 5		
Input Graph Name	Path Graph	
Output		
number of nodes are	10	100
number of edges are	9	99
Time taken by dijkstra_algorithm (in sec)	0	0
Shortest path by dijkstra_algorithm is	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 1 2, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 3 2, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 5 2, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 7 2, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 9 2, 93, 94, 95, 96, 97, 98, 99]
Time taken by bellman_ford_algorithm (in sec)	0	0.01
Shortest path by bellman_ford_algorithm is	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,

		63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
Time taken by floydwarshall_algorithm (in sec)	0	0.46
Shortest path by floydwarshall_algorithm is	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]