PROGRAMMING ASSIGNMENT II: SINGLE SOURCE SHORTEST PATHS

CS 456
Design and Analysis of Algorithms
SIUE

VENKATESH BOLLINENI - 800792618
vbollin@siue.edu

I. SINGLE SOURCE SHORTEST PATH ALGORITHMS

Directed Acyclic Graph Shortest Path (DAG SP) Algorithm:

The DAG shortest paths tree algorithm efficiently finds the shortest paths from a starting vertex to all other vertices in a directed acyclic graph (DAG). The algorithm starts by computing a topological ordering of the vertices which ensures that for any directed edge, the start vertex precedes the end vertex. This ordering allows the algorithm to process each vertex sequentially, respecting edge directions and ensures accurate shortest paths. After creating the topological order, the algorithm sets the distance for start vertex to zero and all other vertices as infinity. The algorithm then iterates over each vertex in topological order, inspects outgoing edges and updates adjacent vertex distances if a shorter path is found. This edge relaxation process is repeated for each vertex, and slowly builds the shortest path distances. At the end of the algorithm, the distance labels show the shortest paths from the start vertex to all others in the DAG. This algorithm works very well because it uses the topological order to process each vertex once, ensures all the edges are relaxed correctly and the shortest paths are calculated quickly, which utilizes the acyclic nature of the graph.

Types of graphs applicable: This algorithm is applicable to directed acyclic graphs (DAGs). These are special types of graphs where all the edges have a direction, and there are no cycles, that means we cannot start at one vertex and follow the edges around to get back to the same vertex.

Time complexity: The theoretical time complexity of the DAG Shortest Paths algorithm is O(n+m), where n is the number of vertices and m is the number of edges in the graph. The time it takes to run the algorithm depends on the total number of vertices and edges in the graph. Initially, the algorithm needs to compute the topological ordering of the vertices, which can be done in O(n+m) time. After that, it processes each vertex and its outgoing edges, and for each edge, it checks if it can update the shortest path. Since it processes each vertex and edge exactly once, this part also takes O(n+m) time. So, combining both parts, the overall time complexity still remains **O(n+m)**. This shows that the algorithm is quite efficient because it only needs the time that is proportional to the size of the graph.

Dijkstra Algorithm:

Dijkstra's algorithm is used to find the shortest paths from a starting vertex to all other vertices in a graph with non-negative edge weights. The process on graph starts by setting the distance to the starting vertex to zero and all other vertices to infinity. The algorithm then uses a priority queue to repeatedly select the vertex with the smallest known distance, updating the distances to its adjacent vertices if a shorter path is found through this vertex. This will continue until all vertices are processed completely and results in the shortest path from the start vertex to every other vertex in the graph.

Types of graphs applicable: Dijkstra's algorithm is applicable to graphs with only non-negative edge weights. It works for both directed and undirected graphs, with or without cycles, if there are no negative weights on the edges. If the graph consists of negative edge weights, then the algorithm might not work correctly because it relies on the property that

once a vertex is processed, its shortest distance is finalized but in the case of negative edge weights this procedure doesn't work properly.

Time complexity: The theoretical time complexity of Dijkstra's algorithm depends on the data structure used for the priority queue. Here binary heap is being used, the time complexity is **O((n+m)logn)**, where n is the number of vertices and m is the number of edges. This is because each vertex is inserted into the priority queue once and each edge is relaxed once, and both these operations take logarithmic time.

Bellman-Ford Algorithm:

The Bellman-Ford algorithm is used to find the shortest paths from a starting vertex to all other vertices in a directed graph which includes both positive and negative edge weights. The algorithm starts by allocating the distance to the starting vertex to zero and all other vertices to infinity. It then performs continuous edge relaxations, where each edge is checked, and the distance to the adjacent vertex is updated if a shorter path is found through the current vertex. This relaxation step will be repeated n−1 times, where n is the number of vertices. After the n−1 iterations, the algorithm performs one additional pass over all edges to check for negative-weight cycles. If any edge can still be relaxed, it indicates a negative-weight cycle exists in the graph.

Types of graphs applicable: The Bellman-Ford algorithm is applicable to directed graphs that consists of both negative and positive edge weights. It can handle graphs with negative weights and is particularly useful for detecting negative-weight cycles. If the graph has negative-weight cycles, the algorithm will identify their presence but it won't provide the correct shortest paths.

Time Complexity: The theoretical time complexity of the Bellman-Ford algorithm is **O(n.m)**, where n is the number of vertices and m is the number of edges in the graph. Firstly, initializing the distances to infinity and setting the start vertex's distance to zero which takes O(n) time. During the relaxation step, the algorithm performs n−1 iterations over all edges. Each iteration involves checking and updating the distance for each edge, taking O(m) time per iteration. So, the relaxation step takes O(n·m) time in total. After the n−1 iterations, the algorithm performs one extra pass over all the edges to check for negative-weight cycles which takes O(m) time. After Combining all these steps, the overall time complexity of Bellman-Ford algorithm is O (n·m), this algorithm can handle larger graphs but takes more time compared to other algorithms because of quadratic time in nature.

II. Experimental time complexity results:

a. DAG SPT algorithm's timings table and graph

Type A graph	**Sparse: d = 4**	**Intermediate: d = sqrt(n)**	**Dense: d = n/2**
n = 200	0.2	0.3	0.5
n = 800	0.6	1.2	44.0
n = 1400	1.1	9.6	71.1

Table 1: Timings(milliseconds) recorded for Type A graph with varying n and d values on DAG SPT algorithm

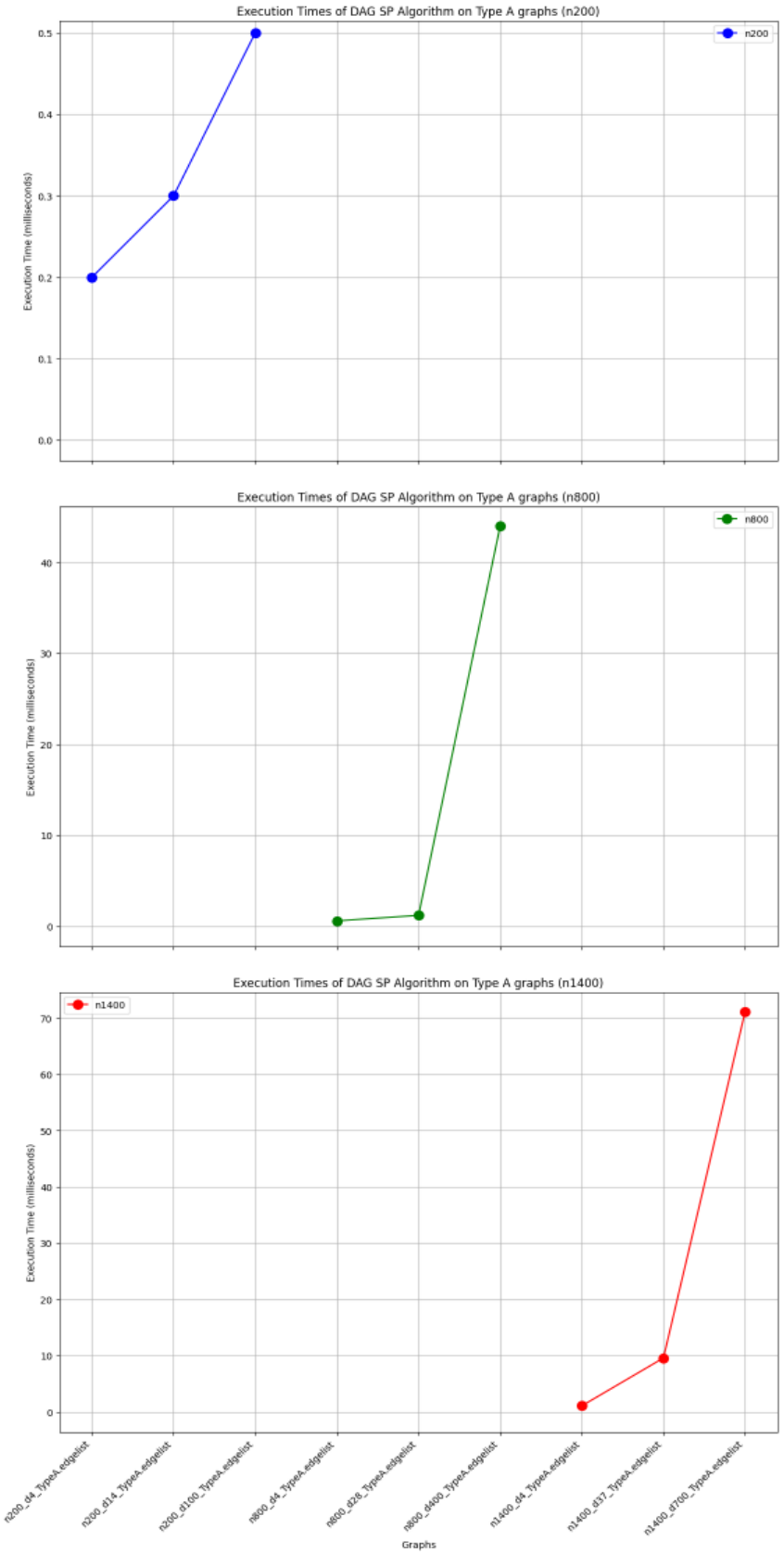

Figure 1: Execution times graph of DAG SPT algorithm

b. Dijkstra's algorithm's timings table and graph

Type B graph	Sparse: d = 4	Intermediate: d = sqrt(n)	Dense: d = n/2
n = 200	0.3	0.6	2.0
n = 800	1.4	4.3	29.5
n = 1400	2.6	8.7	87.1

Table 2: Timings(milliseconds) recorded for Type B graph with varying n and d values on Dijkstra's algorithm

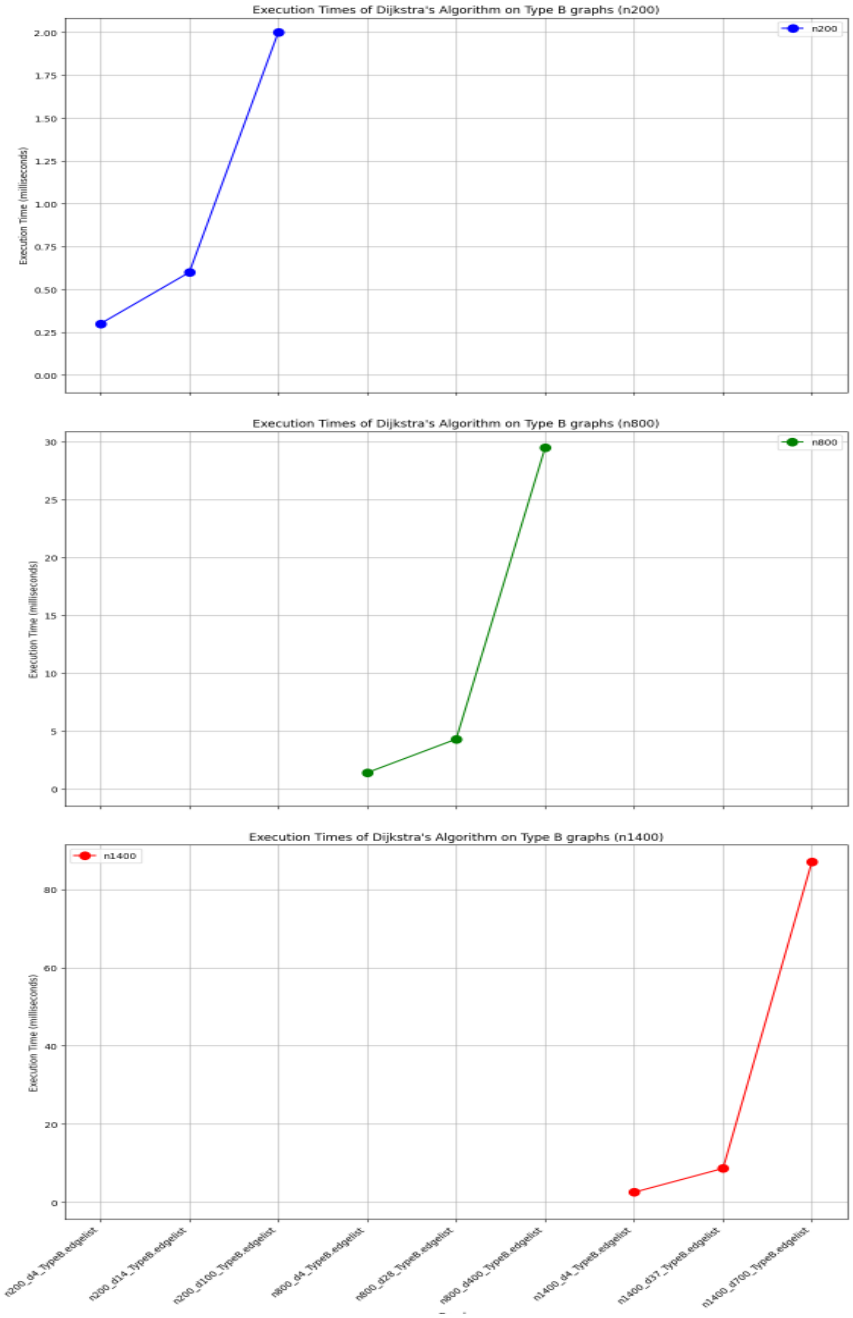

Figure 2: Execution times graph of Dijkstra's algorithm

c. Bellman ford algorithm's timings table and graph

Type C graph	Sparse: d = 4	Intermediate: d = sqrt(n)	Dense: d = n/2
n = 200	54.9	192.5	1413.2
n = 800	1064.3	7384.3	107459.0
n = 1400	3322.6	30312.8	584795.5

Table 3: Timings (milliseconds) recorded for Type C graph with varying n and d values on Bellman ford algorithm

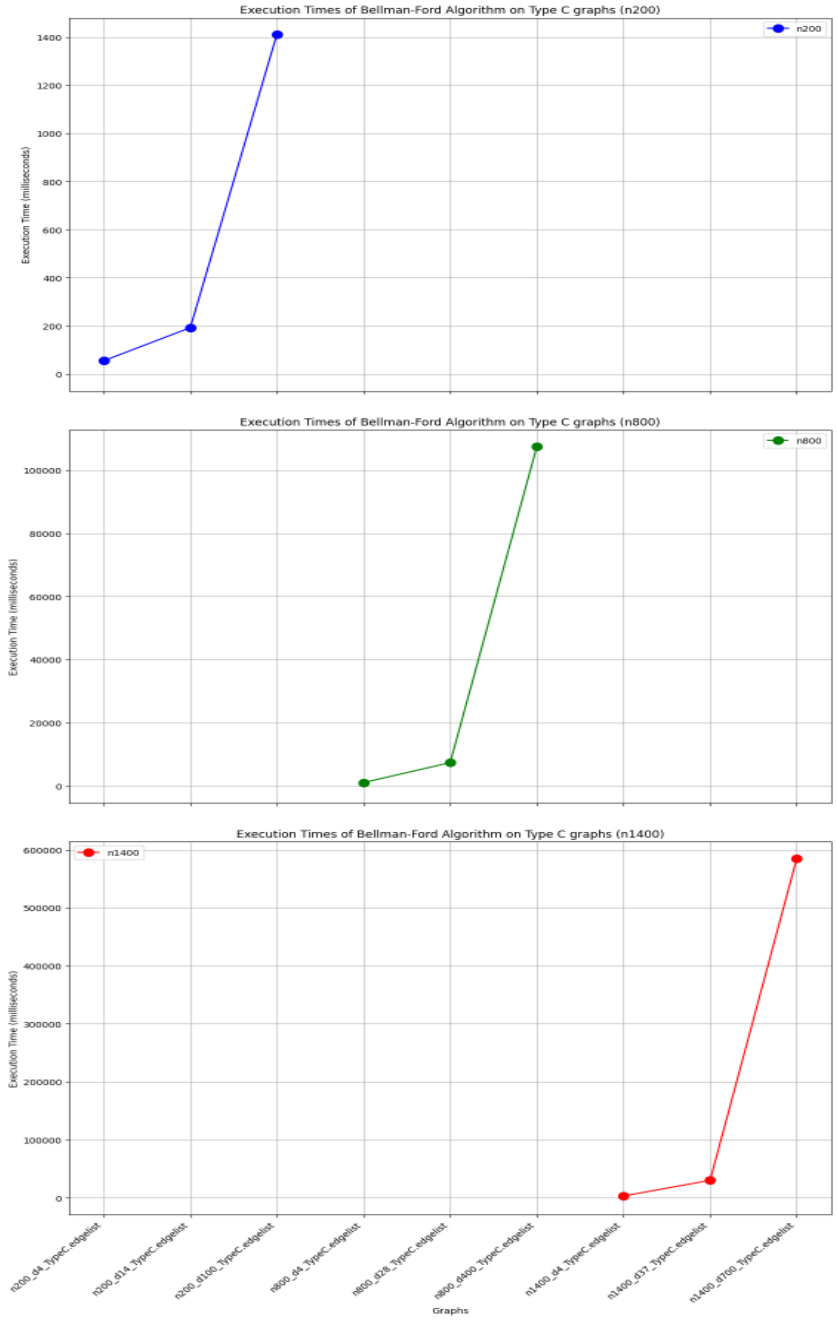

Figure 3: Execution times graph of Bellman ford algorithm

d. Comparative analysis charts of three algorithms with various n and d values

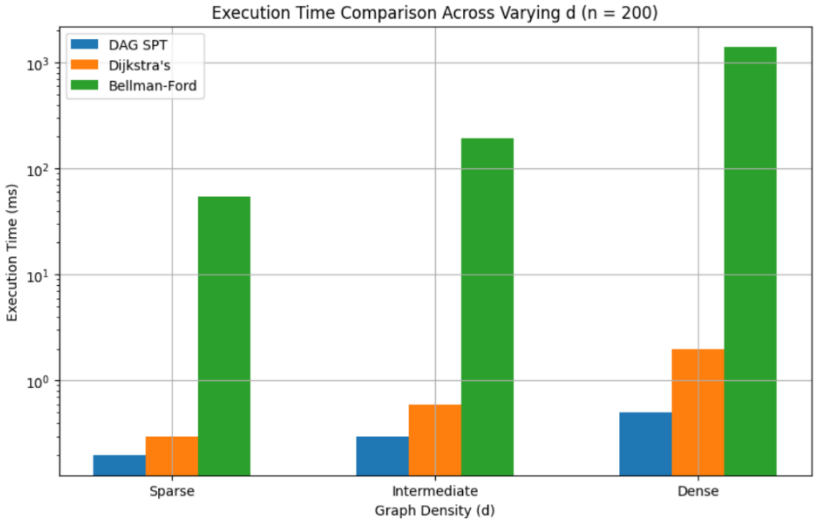

Figure 4: Execution times comparison across three algorithms with n = 200 and varying densities (d)

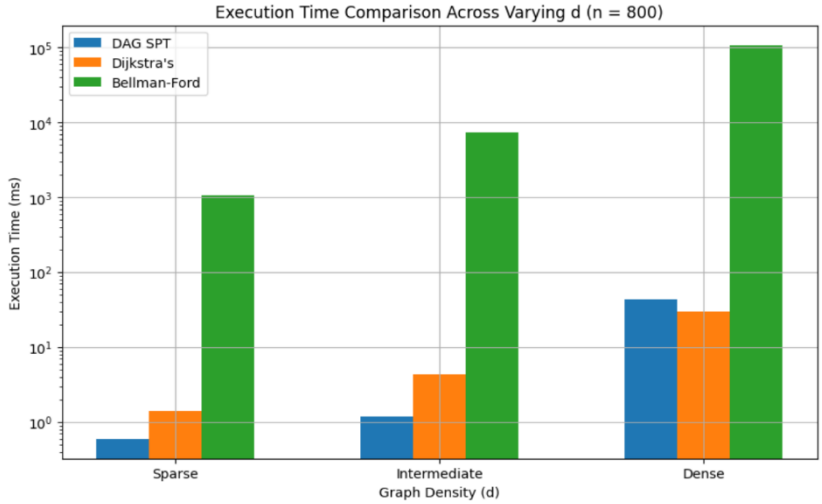

Figure 5: Execution times comparison across three algorithms with n = 800 and varying densities (d)

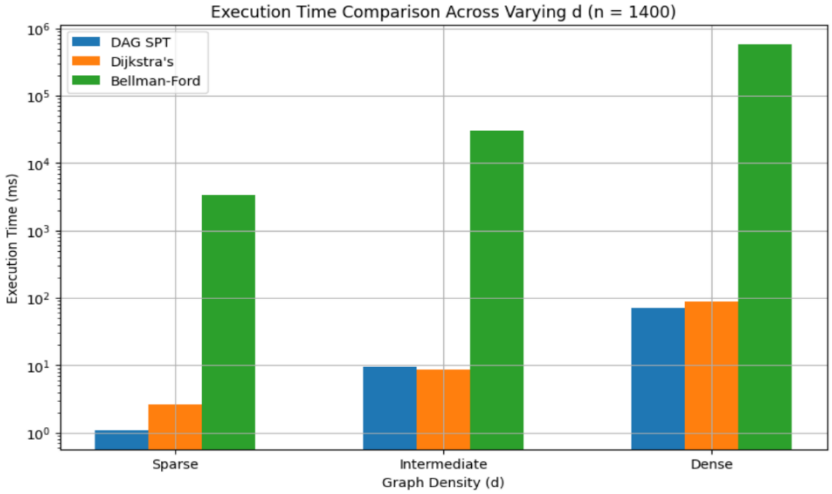

Figure 6: Execution times comparison across three algorithms with n = 1400 and varying densities (d)

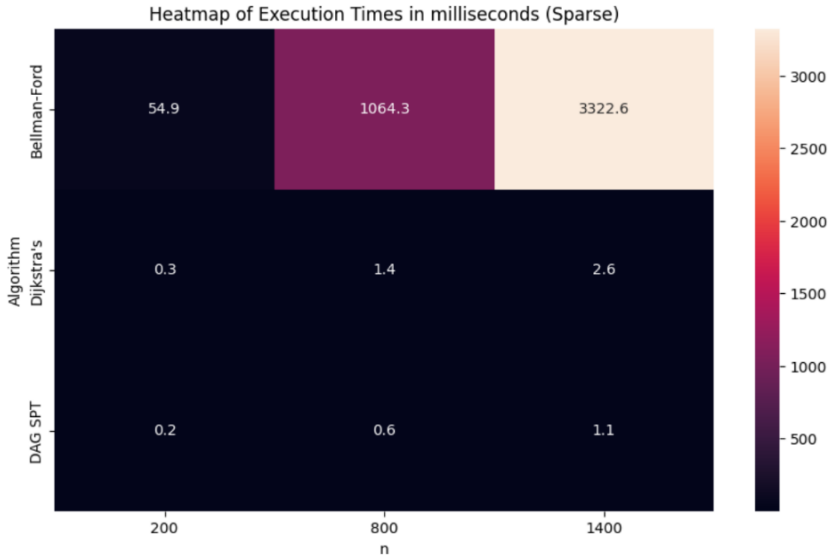

Figure 7: Execution times (milliseconds) comparison across three algorithms on sparse d values with varying n values

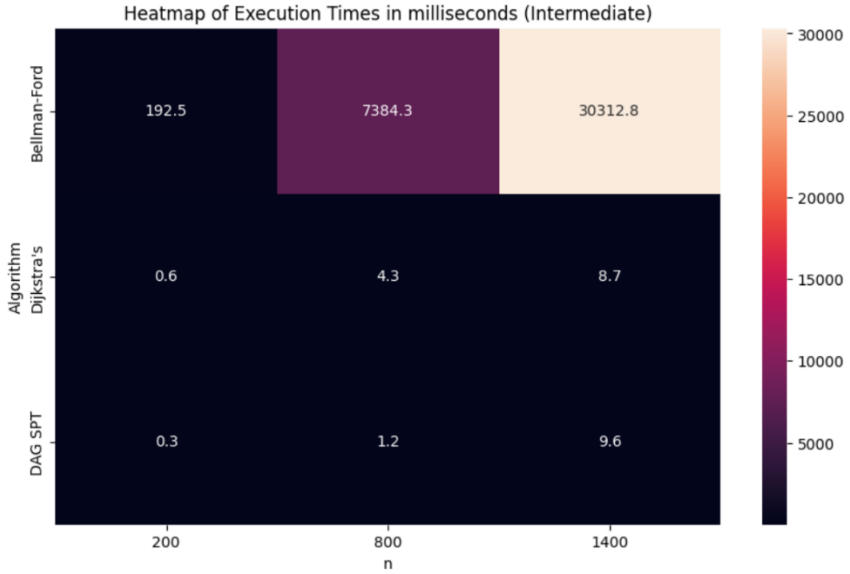

Figure 8: Execution times (milliseconds) comparison across three algorithms on intermediate d values with varying n values

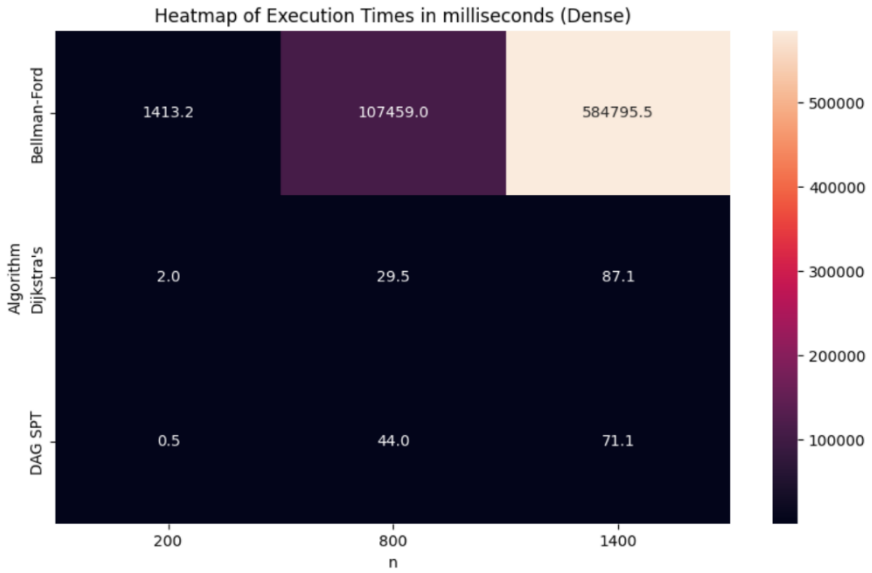

Figure 9: Execution times (milliseconds) comparison across three algorithms on dense d values with varying n values

e. Empirical times vs Theoretical times

Comparative analysis between three algorithms (DAG, Dijstra's, Bellman-Ford):

Analysis for n = 200 with varying densities

For DAG SP algorithm (on type A graph), the execution times increases slightly with increasing density values. The main reason for this is the time complexity of O(n+m), which means that whenever the number of edges (m) increases, the execution time increases linearly. Since the algorithm processes each edge once, the increase is small. From table 1 and figure 1, the DAG SP algorithm remains efficient even for denser graphs, showing only a moderate increase from 0.2 milliseconds (ms) to 0.5 ms. This indicates that the algorithm handles additional edges very well because of its linear complexity.

For Dijkstra's algorithm (type B graph), the execution times increase more observably with density. The main reason for this is the time complexity of O((n+m)logn), which includes a log factor due to the priority queue operations. As the graph becomes denser (more edges), the increase in m leads to a higher increase in execution time compared to DAG SP. From table 2, and figure 2, from sparse to dense, the time increases from 0.3 ms to 2.0 ms. The increase is very large compared to DAG SP due to the additional log factor. This indicates the overhead of managing a priority queue, which grows with the number of edges.

For Bellman-Ford algorithm (type C graph), the execution times increased drastically with density. The time complexity of O(nm) makes the Bellman-Ford algorithm very responsive to the number of edges. As edges (m) increases, the execution time grows quadratically. From table 3 and figure 3, execution times moved significantly from 54.9 ms to 1413.2 ms as the graph goes from sparse to dense. This sharp increase shows the inefficiency of Bellman-Ford for denser graphs. The quadratic relationship with edge count results in exponential growth in execution times.

Analysis for n = 800 with Varying Densities

For the DAG SP algorithm (type A graph), the execution times are 0.6 ms for sparse, 1.2 ms for intermediate, and 44.0 ms for dense graphs. The main reason for this is the time complexity of O(n+m), which means that as the number of edges (m) increases, the execution time increases linearly. Since the algorithm processes each edge once, the increase is moderate. From table 1 and figure 1, we can see that the DAG SP algorithm remains efficient even for denser graphs, showing a observable increase from 0.6 ms to 44.0 ms. This indicates that the algorithm handles additional edges well, but the increase is more noticeable with higher densities due to the larger number of edges.

For Dijkstra's algorithm (type B graph), the execution times are 1.4 ms for sparse, 4.3 ms for intermediate, and 29.5 ms for dense graphs. The main reason for this is the time complexity of O((n+m)logn), which includes a log factor because of the priority queue operations. As the graph becomes denser (with more edges), the increase in m leads to a higher increase in execution time compared to DAG SP. From table 2 and figure 2, from sparse to dense, the time increases from 1.4 ms to 29.5 ms. The increase is more substantial because of the additional log factor.

The Bellman-Ford algorithm (type C graph) shows execution times of 1064.3 ms for sparse, 7384.3 ms for intermediate, and 107459.0 ms for dense graphs. The main reason for this is the time complexity of O(nm), which makes the Bellman-Ford algorithm very responsive to the number of edges. As edges (m) value increases, the execution time grows quadratically. From table 3 and figure 3, execution times increases significantly from 1064.3 ms to 107459.0 ms as the graph goes from sparse to dense. This rapid increase highlights the inefficiency of Bellman-Ford for denser graphs.

Analysis for n = 1400 with Varying Densities

For the DAG SP algorithm (type A graph), the execution times are 1.1 ms for sparse, 9.6 ms for intermediate, and 71.1 ms for dense graphs. The main reason for this is the time complexity of O(n+m), which means that as the number of edges (m) increases, the execution time increases linearly. Since the algorithm processes each edge once, the increase is noticeable but moderate. From the table 1 and figure 1, we can see that the DAG SP algorithm remains efficient even for denser graphs, showing an increase from 1.1 ms to 71.1 ms.

For Dijkstra's algorithm (type B graph), the execution times are 2.6 ms for sparse, 8.7 ms for intermediate, and 87.1 ms for dense graphs. The main reason for this is the time complexity of O((n+m)logn), which includes a log factor due to the priority queue operations. As the graph becomes denser (with more edges), the increase in m leads to a higher increase in execution time compared to DAG SP. From table 2 and figure 2, from sparse to dense, the time increases from 2.6 ms to 87.1 ms.

The Bellman-Ford algorithm (type C graph) shows execution times of 3322.6 ms for sparse, 30312.8 ms for intermediate, and 584795.5 ms for dense graphs. The main reason for this is the time complexity of O(nm), which makes the Bellman-Ford algorithm very sensitive to the number of edges. As m size increases, the execution time grows quadratically. From table 3 and figure 3, execution times increased significantly from 3322.6 ms to 584795.5 ms as the graph goes from sparse type to dense type.

Overall, in conclusion, across all three scenarios (n=200, n=800, and n=1400), the DAG SP algorithm is consistently the most efficient. It handles extra edges very well because of its

linear complexity, making it suitable for different graph densities. Dijkstra's Algorithm performs well for sparse and intermediate graphs but has a noticeable increase in execution time for denser graphs. This is due to the log factor involved with priority queue operations, making it moderately efficient. The Bellman-Ford Algorithm performs very poorly on denser graphs. This is mainly due to its quadratic complexity leads to exponential growth in execution time as the number of edges increases, making it the least efficient for larger and denser graphs. Overall, DAG SP is the best choice among the three algorithms, Dijkstra's is good for moderate densities, and Bellman-Ford is mainly useful for smaller graphs or when handling negative weights and to detect the negative weight cycle.

Yes, resultant time complexities patterns we have got for all the algorithms are compliance to the earlier discussed theoretical time complexities.

Comparing timing results within algorithm:

For DAG SP algorithm, with varying n and d values shows a clear pattern. For n=200, the execution times are 0.2 ms for sparse, 0.3 ms for intermediate, and 0.5 ms for dense graphs. As n increases to 800, the times are 0.6 ms for sparse, 1.2 ms for intermediate, and 44.0 ms for dense graphs. For n=1400, the times are 1.1 ms for sparse, 9.6 ms for intermediate, and 71.1 ms for dense graphs. This pattern shows that the execution time increases with both the number of nodes and the number of edges. Sparse graphs show a small, linear increase in execution time, whereas the intermediate graphs increased moderately. For dense graphs, there is a significant rise in execution time due to the larger number of edges. The linear time complexity O(n+m) confirms the algorithm is efficient but the impact of the increasing edge sizes becomes more increase in time complexity. This pattern shows that it is aligned with the theoretical time complexity.

For Dijkstra's algorithm, the execution times vary with different values of n and d. For n=200, the times are 0.3 ms for sparse graphs, 0.6 ms for intermediate graphs, and 2.0 ms for dense graphs. When n increases to 800, the times are 1.4 ms for sparse, 4.3 ms for intermediate, and 29.5 ms for dense graphs. For n=1400, the times are 2.6 ms for sparse, 8.7 ms for intermediate, and 87.1 ms for dense graphs. The time complexity of O((n+m)logn) explains these patterns. Sparse graphs have fewer edges, so the increase in execution time is moderate as n grows. Intermediate graphs have more edges which leads to a significant increase in time due to the log factor in the complexity. Dense graphs, with many edges, shows a rise in execution time. These patterns shows that the resultant times are expected and follows as theoretical times.

For Bellman-Ford algorithm, the analysis with varying values of n and d shows that the execution times are consistent with the expected time complexity of O(nm). For n=200, the times increase from 54.9 ms for sparse graphs to 192.5 ms for intermediate graphs, and 1413.2 ms for dense graphs. As n increases to 800, the times rise from 1064.3 ms for sparse to 7384.3 ms for intermediate, and 107459.0 ms for dense graphs. For n=1400, the times further increase to 3322.6 ms for sparse, 30312.8 ms for intermediate, and 584795.5 ms for dense graphs. These results are consistent with the quadratic time complexity O(nm), which predicts significant increases in execution time as the number of edges (m) and nodes (n) increase. The empirical execution times align with the theoretically expected time complexity of the Bellman-Ford algorithm.

Unexpected results:

The only unexpected result which have observed is in the figure 6, even though the intermediate d (14) value for Dijkstra's is bigger than DAG SP d value (2) algorithm the time for processing the Dijkstra's algorithm took little shorter than DAG SP. The main reason for this is that where Dijkstra's algorithm took less time than the DAG SP algorithm for intermediate density graphs might be due to specific graph structures and edge distributions. Also, the log factor in Dijkstra's algorithm, which comes from managing the priority queue, usually makes it slower than the DAG SP algorithm. But in this case the impact of the log factor might be less because of the specific distribution of edges and weights in the graph. If the graph's structure allows Dijkstra's to quickly find the shortest path without updating the priority queue a lot, the log factor's impact could be less important. This could lead to faster execution times for Dijkstra's algorithm compared to the DAG SP algorithm.

III. Inappropriate choice of SPT algorithm rendered incorrect shortest path results.

a. Cyclic graph running on DAG SPT algorithm which gave incorrect results

Case : DAG SPT algorithm run on Type B graph n200_d14_TypeB.edgelist (which is having cycles and only positive weights)

Given Source node: 2, Destination node: 54

Applied DAG SPT algorithm on Type B graph with n = 200 and density value of 14, then the algorithm resulted the path and certain amount of weight as followed, which is copied from the console:

Resulted path weight - 18

```
Enter the source node:  2
Enter the destination node:  54
DIJInputGraphs/n200_d14_TypeB.edgelist - DAG SP - 0.0010s - Success
  Path: 2 -> 117 -> 178 -> 54 with total weight: 18
  Weights: 2 -> 12 -> 4
```

To compare the DAG SPT produced results, applied Dijkstra's algorithm (this algorithm works for cyclic with non-negative weights) and Bellman ford (this graph works for cyclic, positive and negative weights) on the same graph and got the following results from the console.

Dijkstra's results:

Resulted path weight – 10

```
Enter the source node:  2
Enter the destination node:  54
DIJInputGraphs/n200_d14_TypeB.edgelist - Dijkstra SP - 0.0006s - Success
  Path: 2 -> 86 -> 54 with total weight: 10
  Weights: 9 -> 1
```

Bellman ford results:

Resulted path weight – 10

```
Enter the source node:  2
Enter the destination node:  54
DIJInputGraphs/n200_d14_TypeB.edgelist - Bellman-Ford SP - 0.1446s - Success
  Path: 2 -> 86 -> 54 with total weight: 10
  Weights: 9 -> 1
```

Comparison: From the above results that DAG SPT algorithm has produced incorrect path and incorrect path length from source node: 2 to destination node: 54 when compared with the other two algorithms (Dijkstras and Bellman ford) results which are having same path and same path length as well. We can say that Dijkstras and Bellman ford algorithms have given correct result and well handled the cyclic graph.

To further explain about the discrepancy, DAG computed path weight is 18 but the actual path weight is 10, the main reason for DAG SPT algorithm to produce the wrong result is, it completely depends on topological order. If there is any cycle exists in the graph, then the topological sort creates wrong order, so which results in producing incorrect path and incorrect path length. Also, the DAG SPT algorithm is only designed for acyclic graphs so this graph cannot handle cyclic graphs.

b. Dijkstra's algorithm correctly computed all the shortest paths for graphs which included negative weight edges

Case 1: Dijkstra's algorithm run on Type A graph n800_d400_TypeA.edgelist (which is having positive and negative weights)

Given Source node: 0, Destination node: 83

Dijkstra's algorithm on Type A graph with n = 800 and density value of 400, the results are followed are copied from the console

Resulted Path weight: -714

```
Enter the source node:  0
Enter the destination node:  83
DIJInputGraphs/n800_d400_TypeA.edgelist - Dijkstra SP - 3.8823s - Success
  Path: 0 -> 493 -> 664 -> 698 -> 769 -> 770 -> 295 -> 193 -> 205 -> 95 -> 26 -> 349 -> 452 -> 640 -> 185 -> 225 -> 357 -> 123 -> 400 -> 553 -> 781 -> 90 -> 115 -> 8
4 -> 709 -> 171 -> 375 -> 729 -> 515 -> 625 -> 715 -> 327 -> 130 -> 617 -> 148 -> 9 -> 314 -> 251 -> 702 -> 660 -> 576 -> 141 -> 154 -> 167 -> 131 -> 451 -> 704 -> 1
44 -> 271 -> 21 -> 150 -> 772 -> 445 -> 173 -> 116 -> 631 -> 158 -> 330 -> 83 with total weight: -714
  Weights: -19 -> -12 -> 9 -> -15 -> -15 -> -11 -> -11 -> -15 -> -20 -> -16 -> 4 -> -7 -> -15 -> -13 -> -19 -> -16 -> -18 -> -20 -> -19 -> 3 -> -18 -> -15 -> -10 ->
-4 -> -8 -> -15 -> -20 -> -4 -> -20 -> -18 -> -9 -> -9 -> -4 -> -20 -> -13 -> -14 -> -17 -> -9 -> -11 -> -7 -> -5 -> -9 -> -18 -> -20 -> -11 -> -19 -> 8 -> -12 -> -1
4 -> -19 -> -9 -> -15 -> -14 -> -20 -> -18 -> -15 -> -9 -> -5
```

To compare the Dijkstra's results, I have employed DAG SPT and Bellman-Ford algorithm on the same graph (n800_d400_TypeA.edgelist) used for Dijkstra's algorithm and the results are copied from the console and followed,

DAG SPT results:

Resulted Path weight: -714

```
Enter the source node:  0
Enter the destination node:  83
DIJInputGraphs/n800_d400_TypeA.edgelist - DAG SP - 0.0480s - Success
  Path: 0 -> 493 -> 664 -> 698 -> 769 -> 770 -> 295 -> 193 -> 205 -> 95 -> 26 -> 349 -> 452 -> 640 -> 185 -> 225 -> 357 -> 123 -> 258 -> 111 -> 781 -> 90 -> 115 -> 8
4 -> 709 -> 171 -> 375 -> 729 -> 515 -> 625 -> 715 -> 327 -> 130 -> 617 -> 148 -> 9 -> 791 -> 402 -> 68 -> 696 -> 757 -> 154 -> 167 -> 131 -> 451 -> 704 -> 144 -> 27
1 -> 21 -> 150 -> 772 -> 445 -> 173 -> 116 -> 631 -> 158 -> 330 -> 83 with total weight: -714
  Weights: -19 -> -12 -> 9 -> -15 -> -15 -> -11 -> -11 -> -15 -> -20 -> -16 -> 4 -> -7 -> -15 -> -13 -> -19 -> -16 -> -18 -> -3 -> -17 -> -16 -> -18 -> -15 -> -10 ->
-4 -> -8 -> -15 -> -20 -> -4 -> -20 -> -18 -> -9 -> -9 -> -4 -> -20 -> -13 -> -14 -> -4 -> -6 -> -19 -> -11 -> -18 -> -18 -> -20 -> -11 -> -19 -> 8 -> -12 -> -14 ->
-19 -> -9 -> -15 -> -14 -> -20 -> -18 -> -15 -> -9 -> -5
```

Bellman ford results:

Resulted Path weight: -714

```
Enter the source node:  0
Enter the destination node:  83
DIJInputGraphs/n800_d400_TypeA.edgelist - Bellman-Ford SP - 34.2522s - Success
  Path: 0 -> 493 -> 664 -> 698 -> 769 -> 770 -> 295 -> 193 -> 205 -> 95 -> 26 -> 349 -> 452 -> 640 -> 185 -> 225 -> 357 -> 123 -> 400 -> 553 -> 781 -> 90 -> 115 -> 8
4 -> 709 -> 171 -> 375 -> 729 -> 515 -> 625 -> 715 -> 327 -> 130 -> 617 -> 148 -> 9 -> 791 -> 402 -> 68 -> 696 -> 757 -> 154 -> 167 -> 131 -> 451 -> 704 -> 144 -> 27
1 -> 21 -> 150 -> 772 -> 445 -> 173 -> 116 -> 631 -> 158 -> 330 -> 83 with total weight: -714
  Weights: -19 -> -12 -> 9 -> -15 -> -15 -> -11 -> -11 -> -15 -> -20 -> -16 -> 4 -> -7 -> -15 -> -13 -> -19 -> -16 -> -18 -> -20 -> -19 -> 3 -> -18 -> -15 -> -10 ->
-4 -> -8 -> -15 -> -20 -> -4 -> -20 -> -18 -> -9 -> -9 -> -4 -> -20 -> -13 -> -14 -> -4 -> -6 -> -19 -> -11 -> -18 -> -18 -> -20 -> -11 -> -19 -> 8 -> -12 -> -14 ->
-19 -> -9 -> -15 -> -14 -> -20 -> -18 -> -15 -> -9 -> -5
```

Case 2: Dijkstra's algorithm run on Type A graph n1400_d700_TypeA.edgelist (which is having positive and negative weights)

Given source node: 0, destination node: 184

Dijkstra's algorithm on Type A graph with n = 1400 and density value of 700, the results are followed are copied from the console

Dijkstra's results:

Resulted Path weight: -494

```
Enter the source node:  0
Enter the destination node:  184
DIJInputGraphs/n1400_d700_TypeA.edgelist - Dijkstra SP - 7.2735s - Success
  Path: 0 -> 1378 -> 169 -> 340 -> 205 -> 992 -> 146 -> 346 -> 1384 -> 656 -> 1170 -> 1207 -> 1045 -> 1105 -> 811 -> 1188 -> 49 -> 1234 -> 305 -> 233 -> 489 -> 1355
-> 53 -> 684 -> 856 -> 795 -> 1096 -> 955 -> 268 -> 1168 -> 1356 -> 1087 -> 835 -> 261 -> 777 -> 1213 -> 1119 -> 1284 -> 501 -> 470 -> 184 with total weight: -494
  Weights: -13 -> -11 -> -4 -> -18 -> -19 -> 3 -> -14 -> -10 -> -18 -> -15 -> -13 -> -16 -> -9 -> -1 -> -15 -> -19 -> -12 -> -20 -> -11 -> -6 -> -4 -> -14 -> -11 ->
-9 -> -17 -> -7 -> -17 -> -6 -> -16 -> -16 -> -8 -> -18 -> -13 -> -7 -> -19 -> -10 -> -12 -> -16 -> -17 -> -16
```

DAG results**:**

Resulted Path weight: -494

```
Enter the source node:  0
Enter the destination node:  184
DIJInputGraphs/n1400_d700_TypeA.edgelist - DAG SP - 0.0749s - Success
  Path: 0 -> 1378 -> 169 -> 340 -> 205 -> 992 -> 146 -> 346 -> 1384 -> 656 -> 1170 -> 1207 -> 1045 -> 1105 -> 811 -> 1188 -> 49 -> 1234 -> 305 -> 233 -> 489 -> 1355
-> 53 -> 684 -> 856 -> 795 -> 1096 -> 955 -> 268 -> 1168 -> 1356 -> 1087 -> 835 -> 261 -> 777 -> 1213 -> 1119 -> 1284 -> 501 -> 470 -> 184 with total weight: -494
  Weights: -13 -> -11 -> -4 -> -18 -> -19 -> 3 -> -14 -> -10 -> -18 -> -15 -> -13 -> -16 -> -9 -> -1 -> -15 -> -19 -> -12 -> -20 -> -11 -> -6 -> -4 -> -14 -> -11 ->
-9 -> -17 -> -7 -> -17 -> -6 -> -16 -> -16 -> -8 -> -18 -> -13 -> -7 -> -19 -> -10 -> -12 -> -16 -> -17 -> -16
```

Bellman Ford results:

Resulted Path weight: -494

```
Enter the source node:  0
Enter the destination node:  184
DIJInputGraphs/n1400_d700_TypeA.edgelist - Bellman-Ford SP - 150.9414s - Success
  Path: 0 -> 1378 -> 169 -> 340 -> 205 -> 992 -> 146 -> 346 -> 1384 -> 656 -> 1170 -> 1207 -> 1045 -> 1105 -> 811 -> 1188 -> 49 -> 1234 -> 305 -> 233 -> 489 -> 1355
-> 53 -> 684 -> 856 -> 795 -> 1096 -> 955 -> 268 -> 1168 -> 1356 -> 1087 -> 835 -> 261 -> 777 -> 1213 -> 1119 -> 1284 -> 501 -> 470 -> 184 with total weight: -494
  Weights: -13 -> -11 -> -4 -> -18 -> -19 -> 3 -> -14 -> -10 -> -18 -> -15 -> -13 -> -16 -> -9 -> -1 -> -15 -> -19 -> -12 -> -20 -> -11 -> -6 -> -4 -> -14 -> -11 ->
-9 -> -17 -> -7 -> -17 -> -6 -> -16 -> -16 -> -8 -> -18 -> -13 -> -7 -> -19 -> -10 -> -12 -> -16 -> -17 -> -16
```

Case 3: Dijkstra's algorithm run on Type A graph n1400_d37_TypeA.edgelist (which is having positive and negative weights)

Given source node: 0, destination node: 176

The following results are generated from the three algorithms:

Dijkstra's results:

Resulted path weight: -240

```
Enter the source node:  0
Enter the destination node:  176
DIJInputGraphs/n1400_d37_TypeA.edgelist - Dijkstra SP - 0.0353s - Success
  Path: 0 -> 622 -> 119 -> 304 -> 1328 -> 7 -> 314 -> 1203 -> 1207 -> 847 -> 142 -> 303 -> 115 -> 660 -> 821 -> 1141 -> 5 -> 796 -> 1291 -> 675 -> 870 -> 1321 -> 234
-> 10 -> 176 with total weight: -240
  Weights: -17 -> 15 -> -13 -> 1 -> -19 -> -7 -> 3 -> -15 -> -14 -> -20 -> -12 -> -20 -> -10 -> -17 -> -15 -> -13 -> -19 -> -12 -> 8 -> -15 -> -7 -> -19 -> 3 -> -6
```

DAG results:

Resulted path weight: -240

```
Enter the source node:  0
Enter the destination node:  176
DIJInputGraphs/n1400_d37_TypeA.edgelist - DAG SP - 0.0100s - Success
  Path: 0 -> 622 -> 119 -> 304 -> 1328 -> 7 -> 314 -> 1203 -> 1207 -> 847 -> 142 -> 303 -> 115 -> 660 -> 821 -> 1141 -> 5 -> 796 -> 1291 -> 675 -> 870 -> 1321 -> 234
-> 10 -> 176 with total weight: -240
  Weights: -17 -> 15 -> -13 -> 1 -> -19 -> -7 -> 3 -> -15 -> -14 -> -20 -> -12 -> -20 -> -10 -> -17 -> -15 -> -13 -> -19 -> -12 -> 8 -> -15 -> -7 -> -19 -> 3 -> -6
```

Bellman ford results:

Resulted path weight: -240

```
Enter the source node:  0
Enter the destination node:  176
DIJInputGraphs/n1400_d37_TypeA.edgelist - Bellman-Ford SP - 10.4514s - Success
  Path: 0 -> 622 -> 119 -> 304 -> 1328 -> 7 -> 314 -> 1203 -> 1207 -> 847 -> 142 -> 303 -> 115 -> 660 -> 821 -> 1141 -> 5 -> 796 -> 1291 -> 675 -> 870 -> 1321 -> 234
-> 10 -> 176 with total weight: -240
  Weights: -17 -> 15 -> -13 -> 1 -> -19 -> -7 -> 3 -> -15 -> -14 -> -20 -> -12 -> -20 -> -10 -> -17 -> -15 -> -13 -> -19 -> -12 -> 8 -> -15 -> -7 -> -19 -> 3 -> -6
```

Results comparison: From the above results, Dijkstra's have produced similar results as DAG and Bellman ford means Dijkstra's has given correct results. In the case 1 it was given

source node: 0 and destination node: 83, then the algorithm correctly found the path and even the path weight also found correctly same as DAG and Bellman ford which is -714. Similarly, for case 2 graph it was given source node: 0 and destination node: 184, then the algorithm correctly found the path and even the path weight also found correctly same as DAG and Bellman ford which is -494, and for case 3 graph it was given source node: 0 and destination node: 176, then the algorithm correctly found the path and even the path weight also found correctly same DAG and Bellman ford which is -240.

In fact, as per the design of the Dijkstra's algorithm, it can handle only the non-negative edges correctly and computes the correct results, but here in this case there might be only one path exist even it includes negative weights from the given source to the destination, so we have got correct results even the given input graph has negative weights. The correct results from Dijkstra's algorithm can be a coincidence because of the specific structure of generated graphs. The algorithm correctly produced the results because the negative weights did not disrupt the assumptions of Dijkstra's algorithm in these cases. The rationale in choosing DAG SPT and Bellman ford algorithm for comparison purpose is, since the chosen graph is Type A (positive and negative weights) and these algorithms correctly produces the output.