Data Structures 2

Assignment 3

Implementing Shortest path algorithms

Name	**ID**
Ranime Ahmed Elsayed Shehata.	21010531
Youssef Tarek Hussein Yousry.	21011595
Osama Hosny Hashem Elsayed.	21010238
Youssef Alaa Ahmed Haridy.	21011603
Mohamed Mohamed Mohamed Abdelmeneim Eldesouky.	21011213

Time & Space Complexity Analysis:

1. Dijkstra Algorithm:

Time Complexity:

- Best Case: **O((V+E) log V)**

When a priority queue is used(binary heap).

- Average Case: **O((V+E) log V)**

It’s the same as the best case scenario because it preforms well on most real world graphs as they’re extremely sparse nor fully connected.

- Worst Case: **O(V²)**

Happens when using an array-based implementation or a simple priority queue.

This occurs when the graph is dense, with many edges, and the priority queue operations become less efficient due to the lack of optimization.

- **Space Complexity:**

O(V)

The space complexity of Dijkstra's Algorithm is O(|V|) since it requires storing the tentative distances for each vertex.

Advantages:

- Highly efficient for graphs with non-negative weights.
- Applicable to both directed and undirected graphs.
- Faster in practical scenarios with advanced priority queues

Disadvantages:

- Cannot work with negative edges.
- Complexity increases with the use of more efficient priority queues like Fibonacci heaps.

2. <u>Floyd-Warshall algorithm</u>

Time Complexity:

It's **$O(V^3)$** for all the cases as it involves three nested loops , each iterating over all vertices which accounts for its cubic time complexity.

- Best Case: **$O(V^3)$**
- Average Case: **$O(V^3)$**
- Worst Case: **$O(V^3)$**

Space Complexity:

$O(V^2)$

The space complexity is $O(V^2)$ as it requires storing the distance between all pairs of vertices in a matrix and it's updated Iteratively .

Advantages:

- Straightforward implementation.
- Capable of handling negative weights.
- Useful for dense graphs and applications requiring frequent path queries as it takes the same $O(V^3)$.

Disadvantages:

- Its time complexity as it takes $O(V^3)$ time making it not suitable for very large graphs.
- Space complexity $O(V^2)$ can be prohibitive.
- The algorithm is complex and can be challenging to understand.

3. Bellman-Ford Algorithm

Time Complexity:

V: number of vertices.

E: the number of edges.

- **Best Case: O(E):** occurs when no relaxation is required during the Bellman-Ford algorithm execution.

  As the Algorithm terminates after a single pass through all edges without updating any distances.

- **Average Case: O(V*E):** It depends on the number of vertices and edges In the graph

- **Worst Case: O(V*E):** this scenario occurs when the algorithm needs to iterate through all vertices and edges for V-1 passes followed by one more pass for detecting negative weight cycles.

Space Complexity:

O(V)

as it requires to store distances from the source vertex to all other vertices in a distance array .

Advantages:

- Handles negative weight edges effectively
- Suitable for distributed systems.

Disadvantages:

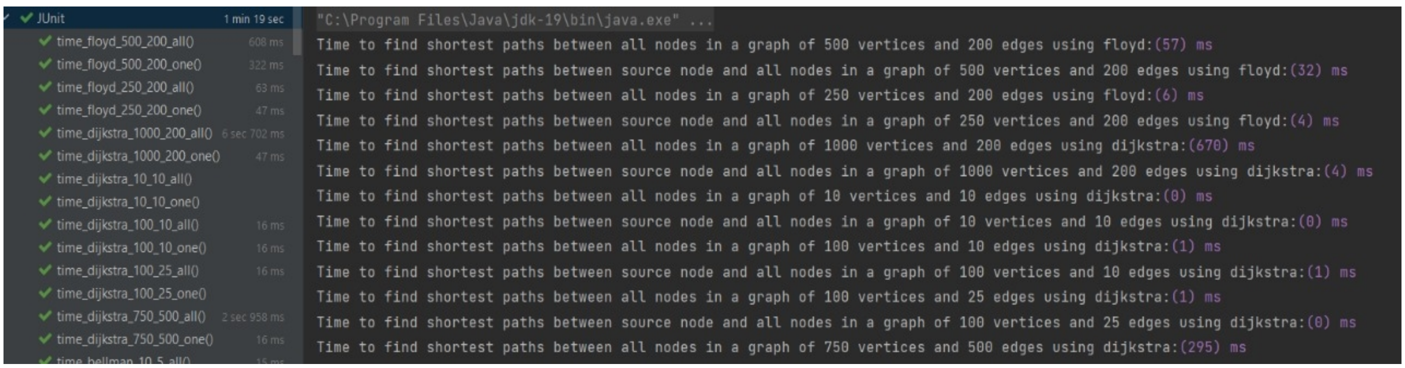
- Higher time complexity than Dijkstra's if it doesn't contain negative edges.
- Generally slower due to its **O(V*E)** complexity making it inefficient for large dense graphs**.**

- **Comparison between the 3 algorithms w.r.t time factor:-**

1. **Shortest paths between all nodes in a graph:**

No. of nodes	**No. of edges**	**Dijkstra**	**Bellman-Ford**	**Floyd-Warshall**
10	5	3 ms	1 ms	3 ms
10	10	1 ms	3 ms	1 ms
100	5	4 ms	3 ms	4 ms
100	10	6 ms	3 ms	4 ms
100	25	4 ms	3 ms	3 ms
250	50	22 ms	7 ms	16 ms
250	100	22 ms	9 ms	15 ms
250	200	22 ms	20 ms	19 ms
500	50	107 ms	9 ms	54 ms
500	100	105 ms	14 ms	55 ms
500	200	110 ms	23 ms	62 ms
750	200	318 ms	25 ms	162 ms
750	300	295 ms	33 ms	159 ms
750	400	331 ms	53 ms	162 ms
750	500	357 ms	64 ms	163 ms

1000	200	742 ms	17 ms	236 ms
1000	300	704 ms	25 ms	247 ms
1000	400	711 ms	36 ms	236 ms
1000	500	708 ms	80 ms	238 ms



2. Shortest path between a source node and all nodes:

No. of nodes	No. of edges	Dijkstra	Bellman-Ford	Floyd-Warshall
10	5	0 ms	0 ms	1 ms
10	10	0 ms	0 ms	1 ms
100	5	0 ms	0 ms	0 ms
100	10	0 ms	0 ms	0 ms
100	25	1 ms	0 ms	0 ms
250	50	0 ms	0 ms	4 ms
250	100	0 ms	0 ms	4 ms
250	200	1 ms	1 ms	6 ms
500	50	1 ms	1 ms	34 ms
500	100	1 ms	0 ms	33 ms
500	200	1 ms	1 ms	62 ms
750	200	3 ms	3 ms	109 ms
750	300	3 ms	3 ms	104 ms
750	400	3 ms	3 ms	112 ms
750	500	3 ms	3 ms	108 ms
1000	200	7 ms	6 ms	250 ms
1000	300	4 ms	4 ms	239 ms
100	400	6 ms	6 ms	245

0				ms
1000	500	6 ms	7 ms	251 ms

JUnit Test: