

Session 6

Single Source Shortest Path Algorithm

This session introduces shortest path problems in weighted graphs. The main idea is to start from one source vertex and compute the minimum distance to every other vertex. This is useful in routing, maps, network design, and any problem where we need the cheapest or shortest route from one point to many destinations.

Objectives

× Do not copy. Read for understanding and the viva

- Understand single-source shortest path problems.
- Apply Dijkstra's algorithm when all edge weights are non-negative.
- Identify why negative edge weights require a different algorithm such as Bellman-Ford.

Concept

× Do not copy. Read for understanding and the viva

A single-source shortest path algorithm finds the minimum distance from one starting vertex to every other vertex in a weighted graph.

Algorithm	Suitable When
Dijkstra	All edge weights are zero or positive
Bellman-Ford	Negative edge weights may exist

Standard Dijkstra Approach

× Do not copy. Read for understanding and the viva

1. Set the source distance to 0 and all other distances to infinity.

2. Pick the unvisited vertex with the smallest known distance.
3. Relax all outgoing edges from that vertex.
4. Repeat until all vertices are processed.

Standard Bellman-Ford Approach

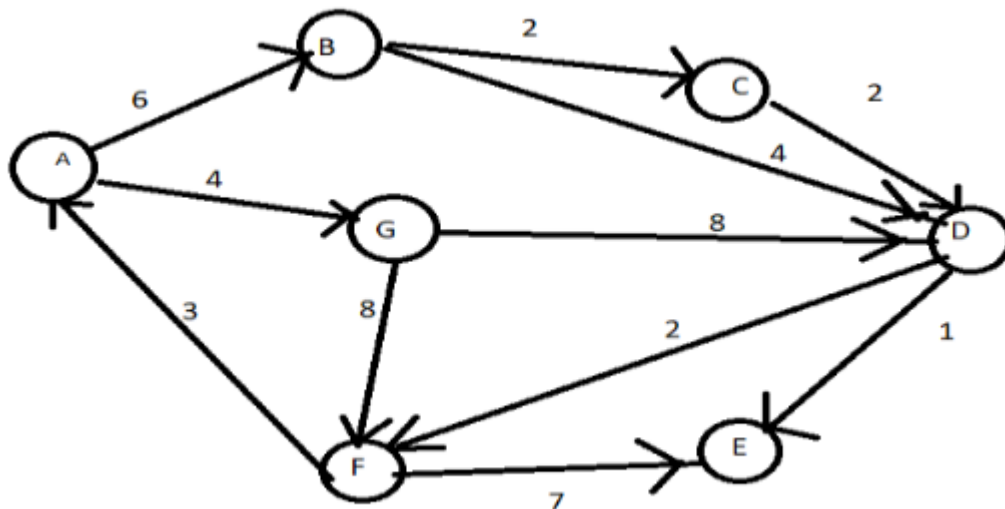
✗ Do not copy. Read for understanding and the viva

1. Set the source distance to 0 and all other distances to infinity.
2. Relax every edge $V - 1$ times.
3. Run one more relaxation pass to detect a negative cycle.

Common statement

It's for all three questions in this sessions.

Implement Dijkstra's algorithm to find the single source shortest path algorithm from different sources to the rest of nodes in the following graph and show all the intermediate processes:



Dijkstra's Algorithm Implementation

This code also includes solutions for all the three question as code and text part is explained standalone. This is done because of implementation requirement for code.



Lab record: write one language only. Pick yours once and every page opens on it; the other tabs are the same solution for comparison.

Python

dijkstra-algo.py

```
1  """
2  Dijkstra's Algorithm Implementation
3  Finds shortest path from source to all other vertices
4  """
5
6  import heapq
7  from collections import defaultdict
8
9
10 class DijkstraAlgorithm:
11     def __init__(self):
12         # Graph represented as adjacency list
13         self.graph = defaultdict(list)
14
15     def add_edge(self, u, v, weight):
16         """Add directed edge from u to v with given weight"""
17         self.graph[u].append((v, weight))
18
19     def dijkstra(self, source, vertices):
20         """Dijkstra's algorithm to find shortest paths from source
21
22         Args:
23             source: Starting vertex
24             vertices: List of all vertices
25
26         Returns:
27             distances: Shortest distance from source to each vertex
28             previous: Previous vertex in shortest path
29             steps: Intermediate steps for visualization
30         """
31         # Initialize distances to infinity
32         distances = {v: float("inf") for v in vertices}
33         distances[source] = 0
34
35         # Track previous vertex for path reconstruction
36         previous = {v: None for v in vertices}
37
38         # Track visited vertices
39         visited = set()
40
41         # Priority queue: (distance, vertex)
42         pq = [(0, source)]
43
```

```
44     # Store intermediate steps
45     steps = []
46     iteration = 0
47
48     while pq:
49         iteration += 1
50         current_dist, current_vertex = heapq.heappop(pq)
51
52         # Skip if already visited
53         if current_vertex in visited:
54             continue
55
56         # Mark as visited
57         visited.add(current_vertex)
58
59         # Record this step
60         step = {
61             "iteration": iteration,
62             "current_vertex": current_vertex,
63             "current_distance": current_dist,
64             "distances": distances.copy(),
65             "visited": visited.copy(),
66         }
67         steps.append(step)
68
69         # Explore neighbors
70         for neighbor, weight in self.graph[current_vertex]:
71             if neighbor not in visited:
72                 new_distance = current_dist + weight
73
74                 # If shorter path found
75                 if new_distance < distances[neighbor]:
76                     distances[neighbor] = new_distance
77                     previous[neighbor] = current_vertex
78                     heapq.heappush(pq, (new_distance, neighbor))
79
80     return distances, previous, steps
81
82     def get_path(self, previous, source, target):
83         """Reconstruct path from source to target"""
84         path = []
85         current = target
86         while current is not None:
87             path.append(current)
```

```

88         current = previous[current]
89     path.reverse()
90     return path if path[0] == source else []
91
92
93 def print_dijkstra_steps(steps, distances, previous, source, vertices):
94     """Print detailed step-by-step execution"""
95     print("\n" + "=" * 80)
96     print(f"DIJKSTRA'S ALGORITHM - Source: {source}")
97     print("=" * 80)
98
99     for step in steps:
100
101         print(f"\nIteration {step['iteration']}:")
102
103         print(f"    Current Vertex: {step['current_vertex']}")
104
105         print(f"    Distance to {step['current_vertex']}: {step['current_distance']}")
106
107         print(f"    Visited: {sorted(step['visited'])}")
108
109         print("    Current Distances: ", end="")
110
111         for v in sorted(vertices):
112
113             dist = step["distances"][v]
114
115             if dist == float("inf"):
116
117                 print(f"{v}=∞ ", end="")
118
119             else:
120
121                 print(f"{v}={dist} ", end="")
122
123         print()
124
125     print("\n" + "=" * 80)
126
127     print("FINAL RESULTS:")
128
129     print("=" * 80)

```

```
11
6   for v in sorted(vertices):
11
7       dist = distances[v]
11
8       if dist == float("inf"):
11
9           print(f"Distance from {source} to {v}: ∞ (unreachable)")
12
0       else:
12
1           path = []
12
2           current = v
12
3           while current is not None:
12
4               path.append(current)
12
5               current = previous[current]
12
6               path.reverse()
12
7               path_str = " → ".join(path)
12
8               print(f"Distance from {source} to {v}: {dist} | Path: {path_str}")
12
9
13
0
13
1   if __name__ == "__main__":
13
2       # Create graph instance
13
3       dijkstra = DijkstraAlgorithm()
13
4
13
5       # Define all vertices
13
6       vertices = ["A", "B", "C", "D", "E", "F", "G"]
13
7
```

```
13
8   # Add edges (original graph)
13
9   dijkstra.add_edge("A", "B", 6)
14
0   dijkstra.add_edge("A", "G", 4)
14
1   dijkstra.add_edge("B", "C", 2)
14
2   dijkstra.add_edge("B", "D", 4)
14
3   dijkstra.add_edge("C", "D", 2)
14
4   dijkstra.add_edge("G", "D", 8)
14
5   dijkstra.add_edge("G", "F", 8)
14
6   dijkstra.add_edge("D", "F", 2)
14
7   dijkstra.add_edge("D", "E", 1)
14
8   dijkstra.add_edge("F", "A", 3)
14
9   dijkstra.add_edge("F", "E", 7)
15
0
15
1   # =====
15
2   # Q1: Shortest path from A to rest of vertices
15
3   # =====
15
4   print("\n" + "#" * 80)
15
5   print("# Q1: SHORTEST PATH FROM A TO ALL VERTICES")
15
6   print("#" * 80)
15
7
15
8   distances_A, previous_A, steps_A = dijkstra.dijkstra("A", vertices)
15
9   print_dijkstra_steps(steps_A, distances_A, previous_A, "A", vertices)
```

```
16
0
16
1 # =====
16
2 # Q2: Shortest path from B to rest of vertices
16
3 # =====
16
4 print("\n\n" + "#" * 80)
16
5 print("# Q2: SHORTEST PATH FROM B TO ALL VERTICES")
16
6 print("#" * 80)
16
7
16
8 # Create new graph for source B
16
9 dijkstra_B = DijkstraAlgorithm()
17
10 dijkstra_B.add_edge("A", "B", 6)
17
11 dijkstra_B.add_edge("A", "G", 4)
17
12 dijkstra_B.add_edge("B", "C", 2)
17
13 dijkstra_B.add_edge("B", "D", 4)
17
14 dijkstra_B.add_edge("C", "D", 2)
17
15 dijkstra_B.add_edge("G", "D", 8)
17
16 dijkstra_B.add_edge("G", "F", 8)
17
17 dijkstra_B.add_edge("D", "F", 2)
17
18 dijkstra_B.add_edge("D", "E", 1)
17
19 dijkstra_B.add_edge("F", "A", 3)
18
20 dijkstra_B.add_edge("F", "E", 7)
18
1
```

```
18
2    distances_B, previous_B, steps_B = dijkstra_B.dijkstra("B", vertices)
18
3    print_dijkstra_steps(steps_B, distances_B, previous_B, "B", vertices)
18
4
18
5    # =====
18
6    # Q3: Graph with negative weights (G→F = -8, F→A = -3)
18
7    # =====
18
8    print("\n\n" + "#" * 80)
18
9    print("# Q3: SHORTEST PATH FROM A WITH NEGATIVE WEIGHTS")
19
0    print("# Note: Dijkstra's algorithm DOES NOT WORK with negative weights!")
19
1    print("# We'll show why it fails and use Bellman-Ford instead")
19
2    print("#" * 80)
19
3
19
4    # Show Dijkstra failure
19
5    dijkstra_neg = DijkstraAlgorithm()
19
6    dijkstra_neg.add_edge("A", "B", 6)
19
7    dijkstra_neg.add_edge("A", "G", 4)
19
8    dijkstra_neg.add_edge("B", "C", 2)
19
9    dijkstra_neg.add_edge("B", "D", 4)
20
0    dijkstra_neg.add_edge("C", "D", 2)
20
1    dijkstra_neg.add_edge("G", "D", 8)
20
2    dijkstra_neg.add_edge("G", "F", -8) # NEGATIVE WEIGHT
20
3    dijkstra_neg.add_edge("D", "F", 2)
```

```

20
4     dijkstra_neg.add_edge("D", "E", 1)
20
5     dijkstra_neg.add_edge("F", "A", -3) # NEGATIVE WEIGHT
20
6     dijkstra_neg.add_edge("F", "E", 7)
20
7
20
8     print("\n! PROBLEM: Dijkstra's algorithm fails with negative weights!")
20
9     print("    Reason: Once a vertex is marked 'visited', Dijkstra assumes")
21
0     print("    the shortest path is found. But negative edges can create")
21
1     print("    shorter paths later, which Dijkstra will miss.")
21
2
21
3     # Implement Bellman-Ford for negative weights
21
4     def bellman_ford(graph, source, vertices):
21
5         """Bellman-Ford algorithm that handles negative weights"""
21
6         distances = {v: float("inf") for v in vertices}
21
7         distances[source] = 0
21
8         previous = {v: None for v in vertices}
21
9
22
0         # Relax all edges |V| - 1 times
22
1         for i in range(len(vertices) - 1):
22
2             updated = False
22
3             for u in vertices:
22
4                 # Only check paths if u itself is reachable
22
5                 if u in graph and distances[u] != float("inf"):

```

```

22
6         for v, weight in graph[u]:
22
7             if distances[u] + weight < distances[v]:
22
8                 distances[v] = distances[u] + weight
22
9                 previous[v] = u
23
0                 updated = True
23
1         if not updated:
23
2             break
23
3
23
4         # Check for negative weight cycles
23
5         for u in vertices:
23
6             if u in graph and distances[u] != float("inf"):
23
7                 for v, weight in graph[u]:
23
8                     if distances[u] + weight < distances[v]:
23
9                         print(
24
0                             "\n⚠️ CRITICAL ERROR: Graph contains a negative
24
1                             )
24
2                         print(
24
3                             "    Shortest paths cannot be accurately found be
24
4                             )
24
5                         return None, None
24
6
24
7         return distances, previous

```

```
24
8
24
9     # Build graph for Bellman-Ford
25
0     graph_neg = defaultdict(list)
25
1     graph_neg["A"] = [("B", 6), ("G", 4)]
25
2     graph_neg["B"] = [("C", 2), ("D", 4)]
25
3     graph_neg["C"] = [("D", 2)]
25
4     graph_neg["G"] = [("D", 8), ("F", -8)]
25
5     graph_neg["D"] = [("F", 2), ("E", 1)]
25
6     graph_neg["F"] = [("A", -3), ("E", 7)]
25
7
25
8     distances_neg, previous_neg = bellman_ford(graph_neg, "A", vertices)
25
9
26
0     print("\n" + "=" * 80)
26
1     print("BELLMAN-FORD ALGORITHM (Handles Negative Weights)")
26
2     print("=" * 80)
26
3
26
4     if distances_neg is None:
26
5         print("Could not calculate final paths due to the negative weight cy
26
6     else:
26
7         for v in sorted(vertices):
26
8             dist = distances_neg[v]
26
9             if dist == float("inf"):
```

```
27
0         print(f"Distance from A to {v}: ∞ (unreachable)")
27
1     else:
27
2         path = []
27
3         current = v
27
4         visited_in_path = set()
27
5         while current is not None and current not in visited_in_path
27
6             path.append(current)
27
7             visited_in_path.add(current)
27
8             current = previous_neg[current]
27
9             if len(path) > len(vertices): # Safety check
28
10                break
28
11        path.reverse()
28
12        if path and path[0] == "A":
28
13            path_str = " → ".join(path)
28
14            print(f"Distance from A to {v}: {dist} | Path: {path_str}
28
15        else:
28
16            print(f"Distance from A to {v}: {dist} | Path: (cycle de
```

Question 1

Problem Statement

■ Write in lab record

Find the shortest path from **A** to the rest of vertices.

Explanation / Approach

✖ Do not copy. Read for understanding and the viva

Use Dijkstra's algorithm if all edge weights in the graph are non-negative. Start with distance of **A = 0**, mark all other vertices as infinity, and repeatedly relax the nearest unvisited vertex.

Answer

■ Write in lab record

Initial State

```
1 Distances: A=0, B=∞, C=∞, D=∞, E=∞, F=∞, G=∞
2 Visited: {}
```

JS

Iteration 1: Process A (dist=0)

■ Write in lab record

```
1 Neighbors: B(6), G(4)
2 Update: B=0+6=6, G=0+4=4
3 Distances: A=0, B=6, C=∞, D=∞, E=∞, F=∞, G=4
4 Visited: {A}
```

JS

Iteration 2: Process G (dist=4) ← Minimum unvisited

■ Write in lab record

```
1 Neighbors: D(8), F(8)
2 Update: D=4+8=12, F=4+8=12
3 Distances: A=0, B=6, C=∞, D=12, E=∞, F=12, G=4
4 Visited: {A, G}
```

JS

Iteration 3: Process B (dist=6) ← Minimum unvisited

■ Write in lab record

```
1 Neighbors: C(2), D(4)
2 Update: C=6+2=8, D=min(12, 6+4)=10
3 Distances: A=0, B=6, C=8, D=10, E=∞, F=12, G=4
4 Visited: {A, G, B}
```

JS

Iteration 4: Process C (dist=8) ← Minimum unvisited

■ Write in lab record

```
1 Neighbors: D(2)
2 Update: D=min(10, 8+2)=10 (no change)
3 Distances: A=0, B=6, C=8, D=10, E=∞, F=12, G=4
4 Visited: {A, G, B, C}
```

JS

Iteration 5: Process D (dist=10) ← Minimum unvisited

■ Write in lab record

```
1 Neighbors: F(2), E(1)
2 Update: F=min(12, 10+2)=12 (no change), E=10+1=11
3 Distances: A=0, B=6, C=8, D=10, E=11, F=12, G=4
4 Visited: {A, G, B, C, D}
```

JS

Iteration 6: Process E (dist=11) ← Minimum unvisited

■ Write in lab record

```
1 Neighbors: (none that improve distances)
2 Distances: A=0, B=6, C=8, D=10, E=11, F=12, G=4
3 Visited: {A, G, B, C, D, E}
```

JS

Iteration 7: Process F (dist=12) ← Last unvisited

■ Write in lab record

```

1 Neighbors: A(3), E(7)
2 Both already visited, no updates
3 Final: A=0, B=6, C=8, D=10, E=11, F=12, G=4

```

JS

Final Answer

```

1 A→A: 0 (Path: A)
2 A→B: 6 (Path: A→B)
3 A→C: 8 (Path: A→B→C)
4 A→D: 10 (Path: A→B→D)
5 A→E: 11 (Path: A→B→D→E)
6 A→F: 12 (Path: A→G→F or A→B→D→F)
7 A→G: 4 (Path: A→G)

```

JS

Question 2

Problem Statement

■ Write in lab record

Find the shortest path from B to the rest of nodes.

Explanation / Approach

✗ Do not copy. Read for understanding and the viva

The method is the same as Question 1, but the source vertex changes to B. The initial distance table should therefore start with $B = 0$ and all other vertices as infinity.

Answer

■ Write in lab record

Initial State:

```

1 Distances: A=∞, B=0, C=∞, D=∞, E=∞, F=∞, G=∞

```

JS

Iteration 1: Process B (dist=0)

■ Write in lab record

```
1 Update: C=2, D=4
2 Distances: A=∞, B=0, C=2, D=4, E=∞, F=∞, G=∞
```

JS

Iteration 2: Process C (dist=2)

■ Write in lab record

```
1 Update: D=min(4, 2+2)=4
2 Distances: A=∞, B=0, C=2, D=4, E=∞, F=∞, G=∞
```

JS

Iteration 3: Process D (dist=4)

■ Write in lab record

```
1 Update: F=4+2=6, E=4+1=5
2 Distances: A=∞, B=0, C=2, D=4, E=5, F=6, G=∞
```

JS

Iteration 4: Process E (dist=5)

■ Write in lab record

```
1 No updates
```

JS

Iteration 5: Process F (dist=6)

■ Write in lab record

```
1 Update: A=6+3=9, E=min(5, 6+7)=5
2 Distances: A=9, B=0, C=2, D=4, E=5, F=6, G=∞
```

JS

Iteration 6: Process A (dist=9)

■ Write in lab record

- 1 Update: $B = \min(0, 9+6) = 0$, $G = 9+4 = 13$
- 2 Distances: $A=9$, $B=0$, $C=2$, $D=4$, $E=5$, $F=6$, $G=13$

JS

Iteration 7: Process G (dist=13)

■ Write in lab record

- 1 Update: $D = \min(4, 13+8) = 4$, $F = \min(6, 13+8) = 6$
- 2 Final: $A=9$, $B=0$, $C=2$, $D=4$, $E=5$, $F=6$, $G=13$

JS

Final Answer

- 1 $B \rightarrow A$: 9 (Path: $B \rightarrow D \rightarrow F \rightarrow A$)
- 2 $B \rightarrow B$: 0 (Path: B)
- 3 $B \rightarrow C$: 2 (Path: $B \rightarrow C$)
- 4 $B \rightarrow D$: 4 (Path: $B \rightarrow D$)
- 5 $B \rightarrow E$: 5 (Path: $B \rightarrow D \rightarrow E$)
- 6 $B \rightarrow F$: 6 (Path: $B \rightarrow D \rightarrow F$)
- 7 $B \rightarrow G$: 13 (Path: $B \rightarrow D \rightarrow F \rightarrow A \rightarrow G$)

JS

Question 3

Problem Statement

■ Write in lab record

Find the shortest path from **A** to the rest of vertices when there are negative weights **-8** between **G** and **F** and **-3** between **F** and **A**.

Explanation / Approach

✗ Do not copy. Read for understanding and the viva

Dijkstra's algorithm should not be used when negative edge weights exist. For this case, use Bellman-Ford reasoning because it can handle negative edges and can also detect negative cycles.

Answer

■ Write in lab record

CRITICAL ISSUE: Dijkstra's algorithm FAILS with negative weights

Why it fails:

✗ Do not copy. Read for understanding and the viva

- Dijkstra marks vertices as "visited" permanently
- It assumes once visited, the shortest path is found
- Negative edges can create shorter paths through already-visited vertices
- This violates Dijkstra's greedy assumption

Example of failure:

```
1 With G→F=-8 and F→A=-3:
2 Path A→G→F→A creates a cycle: 4 + (-8) + (-3) = -7
3 This creates a NEGATIVE CYCLE!
4 You can loop infinitely to reduce distance.
```

JS

Viva Questions

✗ Do not copy. Read for understanding and the viva

- Why does Dijkstra fail with negative edge weights?
- What is edge relaxation?
- How does Bellman-Ford detect a negative cycle?

✎ Edit this page

< Previous
Session 5

Next >
Session 7