Dijkstra Algo

Dijkstra's Algorithm

```
import heapq

def dijkstra(graph, start):
    # Priority queue to store (distance, vertex) tuples
    priority_queue = [(0, start)]
    # Dictionary to store the shortest distance from start to each vertex
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0

    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)

        # If the distance of the current vertex is greater than the known shortest
distance
        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight

            # If a shorter path to the neighbor is found
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

# Example graph
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}

# Example usage
start_vertex = 'A'
```

```
shortest_distances = dijkstra(graph, start_vertex)
print(f"Shortest distances from {start_vertex}: {shortest_distances}")
```

Dry Run Explanation

Consider the graph and starting vertex 'A':

```
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}
```

1. **Initialization:**
   - `priority_queue = [(0, 'A')]`
   - `distances = {'A': 0, 'B': inf, 'C': inf, 'D': inf}`
2. **First Iteration:**
   - `current_distance, current_vertex = heapq.heappop(priority_queue)` -> `(0, 'A')`
   - For each neighbor of 'A':
     - `neighbor = 'B', weight = 1`
       - `distance = 0 + 1 = 1`
       - Compare: `1 < inf` (True)
       - Update: `distances['B'] = 1`
       - Push: `heapq.heappush(priority_queue, (1, 'B'))`
     - `neighbor = 'C', weight = 4`
       - `distance = 0 + 4 = 4`
       - Compare: `4 < inf` (True)
       - Update: `distances['C'] = 4`
       - Push: `heapq.heappush(priority_queue, (4, 'C'))`
   - `priority_queue = [(1, 'B'), (4, 'C')]`
   - `distances = {'A': 0, 'B': 1, 'C': 4, 'D': inf}`
3. **Second Iteration:**
   - `current_distance, current_vertex = heapq.heappop(priority_queue)` -> `(1, 'B')`
   - For each neighbor of 'B':
     - `neighbor = 'A', weight = 1`
       - `distance = 1 + 1 = 2`

        - Compare: `2 < 0` (False) - skip
    - `neighbor = 'C', weight = 2`
        - `distance = 1 + 2 = 3`
        - Compare: `3 < 4` (True)
        - Update: `distances['C'] = 3`
        - Push: `heapq.heappush(priority_queue, (3, 'C'))`
    - `neighbor = 'D', weight = 5`
        - `distance = 1 + 5 = 6`
        - Compare: `6 < inf` (True)
        - Update: `distances['D'] = 6`
        - Push: `heapq.heappush(priority_queue, (6, 'D'))`
- `priority_queue = [(3, 'C'), (4, 'C'), (6, 'D')]`
- `distances = {'A': 0, 'B': 1, 'C': 3, 'D': 6}`

4. **Third Iteration:**
    - `current_distance, current_vertex = heapq.heappop(priority_queue)` -> `(3, 'C')`
    - For each neighbor of 'C':
        - `neighbor = 'A', weight = 4`
            - `distance = 3 + 4 = 7`
            - Compare: `7 < 0` (False) - skip
        - `neighbor = 'B', weight = 2`
            - `distance = 3 + 2 = 5`
            - Compare: `5 < 1` (False) - skip
        - `neighbor = 'D', weight = 1`
            - `distance = 3 + 1 = 4`
            - Compare: `4 < 6` (True)
            - Update: `distances['D'] = 4`
            - Push: `heapq.heappush(priority_queue, (4, 'D'))`
    - `priority_queue = [(4, 'C'), (6, 'D'), (4, 'D')]`
    - `distances = {'A': 0, 'B': 1, 'C': 3, 'D': 4}`
5. **Fourth Iteration:**
    - `current_distance, current_vertex = heapq.heappop(priority_queue)` -> `(4, 'C')`
    - For each neighbor of 'C':
        - `neighbor = 'A', weight = 4`
            - `distance = 4 + 4 = 8`
            - Compare: `8 < 0` (False) - skip
        - `neighbor = 'B', weight = 2`
            - `distance = 4 + 2 = 6`

- Compare: `6 < 1` (False) - skip
    - `neighbor = 'D', weight = 1`
        - `distance = 4 + 1 = 5`
        - Compare: `5 < 4` (False) - skip
- `priority_queue = [(4, 'D'), (6, 'D')]`
- `distances = {'A': 0, 'B': 1, 'C': 3, 'D': 4}`

6. **Fifth Iteration:**
    - `current_distance, current_vertex = heapq.heappop(priority_queue)` -> `(4, 'D')`
    - For each neighbor of 'D':
        - `neighbor = 'B', weight = 5`
            - `distance = 4 + 5 = 9`
            - Compare: `9 < 1` (False) - skip
        - `neighbor = 'C', weight = 1`
            - `distance = 4 + 1 = 5`
            - Compare: `5 < 3` (False) - skip
    - `priority_queue = [(6, 'D')]`
    - `distances = {'A': 0, 'B': 1, 'C': 3, 'D': 4}`
7. **Sixth Iteration:**
    - `current_distance, current_vertex = heapq.heappop(priority_queue)` -> `(6, 'D')`
    - For each neighbor of 'D':
        - `neighbor = 'B', weight = 5`
            - `distance = 6 + 5 = 11`
            - Compare: `11 < 1` (False) - skip
        - `neighbor = 'C', weight = 1`
            - `distance = 6 + 1 = 7`
            - Compare: `7 < 3` (False) - skip
    - `priority_queue = []`
    - `distances = {'A': 0, 'B': 1, 'C': 3, 'D': 4}`

After all iterations, the shortest distances from the start vertex 'A' are:

```
distances = {'A': 0, 'B': 1, 'C': 3, 'D': 4}
```

This dry run includes detailed conditions for each comparison to help understand how the algorithm determines the shortest path from the starting vertex to all other vertices.