A Mathematical Perspective on Dijkstra's Algorithm

Priyanshu Pandey[1], Parsh Jain[1], and Karan Kumar Anand[1]

Indraprastha Institute of Information Technology Delhi,
Govind Puri, New Delhi 110020

Keywords: Shortest path, MST, Graph Theory, Greedy Algorithm, Dijkstra Algorithm, Bellman-Ford Algorithm, Floyd-Warshall Algorithm

Abstract. Dijkstra's algorithm, introduced by Edger W. Dijkstra, is a widely used method for finding the shortest path in a graph. It determines the shortest paths from a starting node to all other nodes in the graph, thereby solving the single-source shortest paths problem.
The algorithm works by prioritizing the node with the smallest known distance from the source, visiting it, and updating the distances of its neighbouring nodes. This process is repeated until the shortest paths to all reachable nodes are found. A key requirement is that all edge weights must be non-negative. The time complexity of Dijkstra's algorithm is $O((V+E)\log V)$, where V is the number of vertices and E is the number of edges. This paper also discusses similar algorithms such as Bellman-Ford and A* search.

1 History

Dijkstra's algorithm was developed in 1956 by Edsger W. Dijkstra, a renowned Dutch computer scientist. He conceived the idea while thinking about the shortest path from Rotterdam to Groningen. The algorithm was first published in 1959 in his paper titled "A Note on Two Problems in Connection with Graphs." This work also introduced an efficient solution to the minimum spanning tree problem.

1.1 What is Spanning Tree?

A spanning tree is a tree-like subgraph of a connected, undirected graph that connects to all the vertices of the graph, i.e every node of the graph is a part of the tree.

1.2 What is Spanning Tree?

A MST is a spanning tree which has minimum possible weight amongst all different types of spanning trees Some algorithms used to find Minimum Spanning Tree:

- Kruskal's Algorithm
- Prim's Algorithm
- Borůvka's Algorithm

1.3 Related Work

i. **Graph Theory Foundations:**
· Leonhard Euler (1736): Euler's work on the Königsberg Bridge Problem established the foundations of graph theory. Although focused on traversal rather than shortest paths, this work introduced the idea of representing real-world problems using graphs.
· Gustav Kirchhoff (1847): Kirchhoff applied graph theory to electrical circuits, introducing concepts like spanning trees, which are closely related to shortest-path problems.
ii. **Optimization (1950s):**
· Bellman's Principle of Optimality (1957): It formed the basis for solving problems that can be broken into smaller, interdependent subproblems.
· Linear Programming and Network Optimization: This contributed to solving resource allocation and routing problems which provided mathematical tools for tackling graph-related challenges enabling formation of Dijkstra's Algorithm.

2 Data Structures used in Dijkstra's Algorithm

2.1 Graph

Graphs are non-linear data structures representing the "connections" between the elements. These elements are known as the Vertices, and the lines or arcs that connect any two vertices in the graph are known as the Edges. More formally, a Graph comprises a set of Vertices (V) and a set of Edges (E). The Graph is denoted by G(V, E)

2.2 Component of Graphs

i. **Vertices**: Vertices are basic units of the graph representing objects, entities, or people and are also called nodes.
ii. **Edges**: Edges are connections between vertices, representing relationships, also called arcs.

2.3 Graph Types

The Graphs can be categorized into two types:
Undirected Graphs: : A graph where edges have no direction, indicating a two-way relationship. Each edge can be traversed in either direction. The figure shows an example of an undirected graph with 5 nodes and 7 edges.

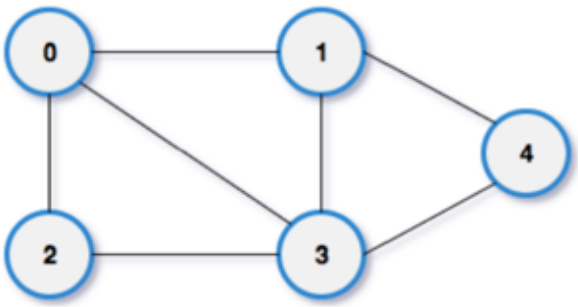

Fig. 1. Example of an undirected graph.

]

Directed Graphs: · A graph where edges have a specific direction, representing one-way relationships. Each edge can only be traversed in its assigned direction. The figure shows a directed graph with 5 nodes and 7 edges.

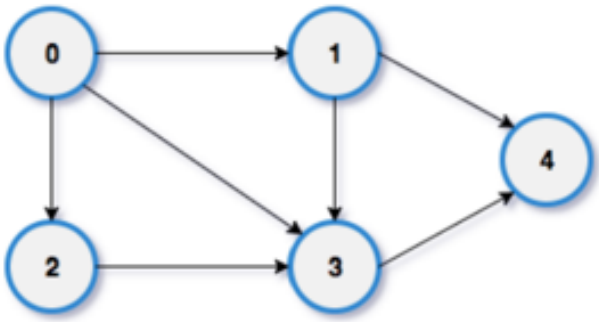

Fig. 2. Example of a directed graph.

The length, position, or orientation of edges in a graph illustration typically doesn't matter. The same graph can be visualized in various ways by rearranging the vertices or reshaping the edges, as long as the graph's underlying structure remains the same.

2.4 Weighted Graphs

A Graph is said to be Weighted if each edge is assigned a 'weight'. The weight of an edge can denote distance, time, or anything that shows the connection between the pair of vertices it connects. The blue number in the adjacent figure shows the weight of each edge.

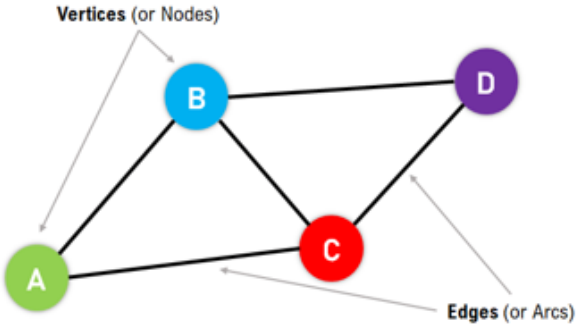

Fig. 3. Example of a weighted graph.

3 In-depth Study of the Dijkstra Algorithm

The algorithm operates by keeping two sets, one for visited vertices and another for unvisited ones. Starting from the source vertex, it repeatedly picks the unvisited vertex with the smallest tentative distance to the source. For each selected vertex, it examines its neighbours, updating their tentative distances if a shorter path is identified. This process continues until the destination vertex is reached or all vertices that can be accessed have been visited. Dijkstra's algorithm can work on both directed graphs and undirected graphs as this algorithm is designed to work on any type of graph as long as it meets the requirements of having non-negative edge weights and being connected.

- In a directed graph, the algorithm follows the direction of the edges when searching for the shortest path.
- In an undirected graph, the edges have no direction, and the algorithm can traverse both forward and backward along the edges when searching for the shortest path.

The algorithm determines the shortest path from node 0 to all other nodes in the graph.In this graph, the weights of the edges represent the distances between connected nodes.

As a result, the algorithm calculates the shortest paths from:

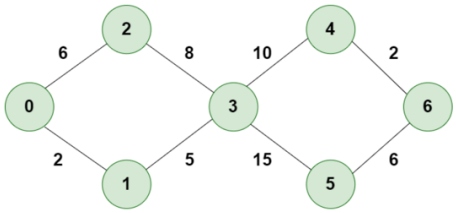

Fig. 4. Original graph.

Initial Point	Final Point
Node 0	Node 1
Node 0	Node 2
Node 0	Node 3
Node 0	Node 4
Node 0	Node 5
Node 0	Node 6

Table 1. Shortest path calculation between nodes.

Initially, the following resources are established:
1. The distance from the source node to itself is set to 0. In this example, the source node is 0.
2. The distances from the source node to all other nodes are initially unknown, so they are marked as infinity
Example: $0 \to 0, 1 \to \infty, 2 \to \infty, 3 \to \infty, 4 \to \infty, 5 \to \infty, 6 \to \infty$
An array of unvisited nodes is maintained to track nodes that have not yet been processed.
The algorithm concludes once all nodes have been visited and their shortest distances have been calculated.
Unvisited nodes at the start: 0, 1, 2, 3, 4, 5, 6

3.1 Steps of the Algorithm

1. **Step 1:** *Start from Node 0 and mark Node as visited as you can check in below image visited Node is marked red.*
2. · **Step 2***: Check for adjacent Nodes, Now we have to choices (Either choose Node1 with distance 2 or either choose Node 2 with distance 6) and choose Node with minimum distance. In this step* ***Node 1*** *is Minimum distance adjacent Node, so marked it as visited and add up the distance.*
   Distance: Node 0 → Node 1 = 2

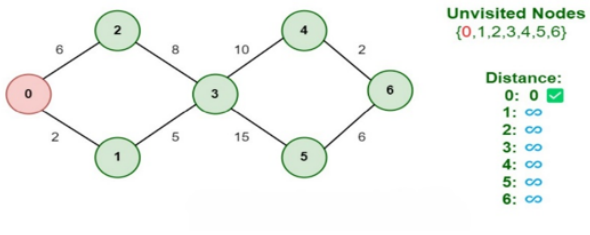

Fig. 5. Step 1 of Original graph.

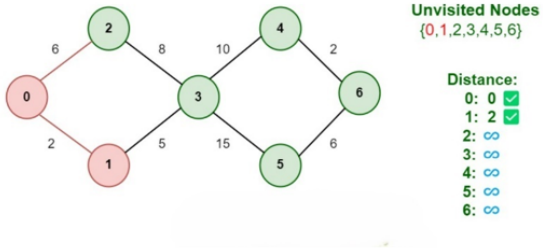

Fig. 6. Step 2 of Original graph.

3. · **Step 3:** *Then Move Forward and check for adjacent Node which is Node 3, so marked it as visited and add up the distance, Now the distance will be:* **Distance: Node 0** $\rightarrow$ **Node 1** $\rightarrow$ **Node 3** $= 2 + 5 = 7$

4. · **Step 4***: Again we have two choices for adjacent Nodes (Either we can choose Node 4 with distance 10 or either we can choose Node 5 with distance 15) so choose Node with minimum distance. In this step* ***Node 4*** *is Minimum distance adjacent Node, so marked it as visited and add up the distance.* **Distance: Node 0** $\rightarrow$ **Node 1** $\rightarrow$ **Node 3**$\rightarrow$ **Node 4** $= 2 + 5 + 10 = 17$

5. · **Step 5***:Again, Move Forward and check for adjacent Node which is Node 6, so marked it as visited and add up the distance, Now the distance will be* **Distance: Node 0** $\rightarrow$ **Node 1** $\rightarrow$ **Node 3**$\rightarrow$ **Node 4** $= 2 + 5 + 10 = 17$

6. So, the Shortest Distance from the Source Vertex is 19 which is optimal one

3.2 Pseudocode

```
function Dijkstra(Graph, source):
```

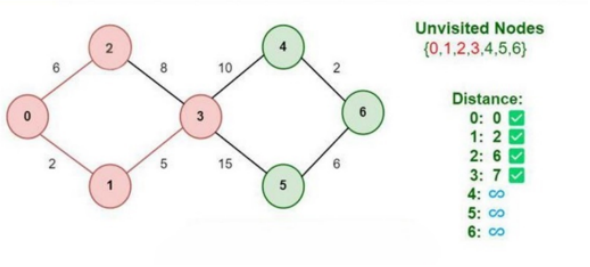

Fig. 7. Step 3 of Original graph.

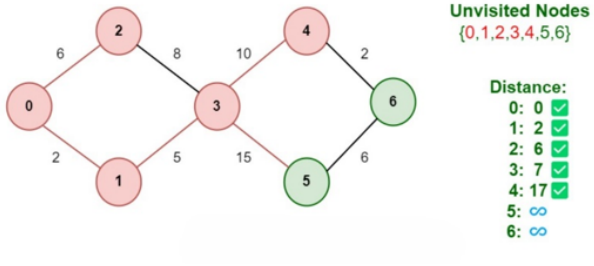

Fig. 8. Step 4 of Original graph.

```
dist[source] ← 0
for each vertex v in Graph.Vertices:
    if v  source:
        dist[v] ←
Q ← Graph.Vertices
while Q is not empty:
    u ← Extract-Min(Q)
    for each neighbor v of u:
        alt ← dist[u] + weight(u, v)
        if alt < dist[v]:
            dist[v] ← alt
```

3.3 Implementation with C++ code

```
#include <limits.h>
#include <stdio.h>

// Number of vertices in the graph
#define V 9
```

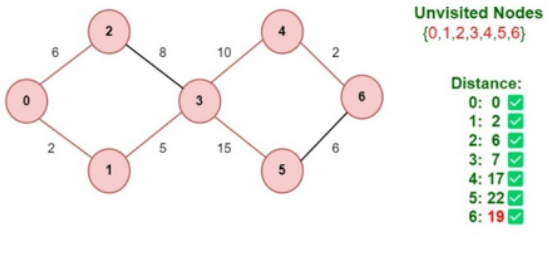

Fig. 9. Step 5 of Original graph.

```
// A utility function to find the vertex with minimum
// distance value, from the set of vertices not yet included
// in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
 // Initialize min value
 int min = INT_MAX, min_index;

 for (int v = 0; v < V; v++)
  if (sptSet[v] == false && dist[v] <= min)
   min = dist[v], min_index = v;

 return min_index;
}

// A utility function to print the constructed distance
// array
void printSolution(int dist[], int n)
{
 printf("Vertex Distance from Source\n");
 for (int i = 0; i < V; i++)
  printf("\t%d \t\t\t\t %d\n", i, dist[i]);
}

// Function that implements Dijkstra's single source
// shortest path algorithm for a graph represented using
// adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
 int dist[V]; // The output array. dist[i] will hold the
    // shortest
 // distance from src to i
```

```
 bool sptSet[V]; // sptSet[i] will be true if vertex i is
     // included in shortest
 // path tree or shortest distance from src to i is
 // finalized

 // Initialize all distances as INFINITE and stpSet[] as
 // false
 for (int i = 0; i < V; i++)
  dist[i] = INT_MAX, sptSet[i] = false;

 // Distance of source vertex from itself is always 0
 dist[src] = 0;

 // Find shortest path for all vertices
 for (int count = 0; count < V - 1; count++) {
  // Pick the minimum distance vertex from the set of
  // vertices not yet processed. u is always equal to
  // src in the first iteration.
  int u = minDistance(dist, sptSet);

  // Mark the picked vertex as processed
  sptSet[u] = true;

  // Update dist value of the adjacent vertices of the
  // picked vertex.
  for (int v = 0; v < V; v++)

   // Update dist[v] only if is not in sptSet,
   // there is an edge from u to v, and total
   // weight of path from src to v through u is
   // smaller than current value of dist[v]
   if (!sptSet[v] && graph[u][v]
    && dist[u] != INT_MAX
    && dist[u] + graph[u][v] < dist[v])
    dist[v] = dist[u] + graph[u][v];
 }

 // print the constructed distance array
 printSolution(dist, V);
}

// driver program to test above function
int main()
{
 /* Let us create the example graph discussed above */
```

```
int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
     { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
     { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
     { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
     { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
     { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
     { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
     { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
     { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

 dijkstra(graph, 0);

 return 0;
}
```

```
Vertex Distance from Source
        0                       0
        1                       4
        2                       12
        3                       19
        4                       21
        5                       11
        6                       9
        7                       8
        8                       14
```

Fig. 10. Output for the above code for Dijsktra's Algorithm.

4 Different approaches to solving Dijkstra's Algorithm:

There are mainly 3 implementations to Dijkstra's Algorithm, which are formed using the following data structures:

1. Unsorted Array
2. Binary Heap and Priority Queue Implementation
3. Fibonacci Heap + Priority queue Implementation

4.1 Detailed discussion on each approach:

Basic Implementation using unsorted array and sets: This is the simplest and least efficient implementation. In this approach, we use an unsorted array to search through all vertices to find the closest one in the graph. As a result, the search has an initial time complexity of O(V). This brings

the total time complexity to O(V$\hat{2}$), where V is the number of vertices in the graph. The space complexity is O(V), which represents the worst-case scenario, such as in a complete graph where every edge must be visited.
Here's how the algorithm works:

1. Initialize the Unvisited Set: Create a set containing all vertices, initially marked as unvisited.
2. Assign Distance Costs: Set the distance cost of the root node to 0 and all other vertices to infinity. Start with the root node as the current node.
3. Update Neighbors' Costs: For the current node (i), examine all its neighbors. Calculate the cost for each neighbor (j) by adding the edge weight (from i→j) to the current node's cost.
4. Mark Node as Visited: After checking all neighbors, mark the current node as visited and move it to the visited set. This set will not be revisited.
5. Repeat: Continue with the next closest unvisited node. Repeat steps 3–5 until all vertices are in the visited set.

When all nodes have been visited, the algorithm is complete.

Time complexities:

- Worst case: O(V$\hat{2}$)
- Average case: O(V$\hat{2}$)
- Best case: O(V$\hat{2}$)

Explanation: As mentioned earlier, this implementation represents the worst-case time complexity for Dijkstra's algorithm when implemented using an unsorted array without a priority queue. This happens because:

1. Vertex Selection: In every iteration, the algorithm must select the vertex with the smallest tentative distance. Since we are using a normal array, this requires a linear scan over all vertices, which takes O(V) time. This scan happens once for each vertex, leading toV such scans, thus O(V$\hat{2}$) in total.
2. Edge Relaxation: For each selected vertex, the algorithm relaxes all its adjacent edges, which again leads to O(V$\hat{2}$) work.

In case of worst, average, and best case, the same method must be applied leading to the same time complexity.

4.2 Pseudocode

```
COST = 0
PREVIOUS = 1

FUNCTION dijkstras(graph, start_node)

    unvisited = {}
    visited = {}
```

```
 FOR key IN graph

     unvisited[key] = [infinity, NULL]
NEXT key

unvisited[start_node][COST] = 0

finished = False
WHILE finished == False
    IF LEN(unvisited) == 0
        finished = True
    ELSE
        current_node = minimum(unvisited)

        FOR neighbour IN graph[current_node]
            IF neighbour NOT IN visited
                cost = unvisited[current_node][COST] + graph[current_node][neighbour]
                IF cost < unvisited[neighbour][COST]
                    unvisited[neighbour][COST] = cost
                    unvisited[neighbour][PREVIOUS] = current_node
                ENDIF
            ENDIF
       NEXT neighbour

       visited[current_node] = unvisited[current_node]
       DEL(unvisited[current_node])
     ENDIF
ENDWHILE
RETURN visited

ENDFUNCTION
```

Implementation using Binary Heap and Priority Queue: To improve the initial implementation of the algorithm, we can replace the unsorted array with a priority queue implemented using a binary heap. Additionally, an adjacency list can be incorporated into the heap for efficient indexing. The steps for this approach are as follows:

1. Initialize the Binary Heap: Create a binary heap with V entries, where V is the number of vertices in the graph. The heap will represent the adjacency list, containing each vertex and its associated distance cost.
2. Set Initial Distances: Set the distance of the root vertex to 0, and all other vertices to infinity.
3. Iterate While Heap is Not Empty: Use a loop that continues until the binary heap is empty.
   - Extract the vertex with the smallest distance cost (i) from the binary heap.

- For each adjacent vertex (j) of the extracted vertex (i), check if it exists in the heap.
- If j is in the heap and its current distance value is greater than i→j+i, update j's distance in the heap.
- Here are common heap operations and their time complexities using binary heap:

Operation	**Time Complexity**
Insert	$O(\log N)$
ReturnMin	$O(1)$
DeleteMin	$O(\log N)$
Delete	$O(\log N)$
DecreaseKey	$O(\log N)$
Merge	$O(1)$

Table 2. Time complexity of operations

4.3 Analyzing Time complexities:

Worst Case Time Complexity:

- The inner loop processes O(V+E), where V is the number of vertices, and E is the number of edges.
- The decrease-key operation takes O(logV) time for each relaxation.
- This results in an overall time complexity of O((V+E) logV). For dense graphs where E V2̂, this simplifies to O(E logV)

Total Complexity: O((V+E) logV)

Average Case Time Complexity:

- On average, the algorithm runs in O((V+E) logV), as the inner loop still executes O(V+E), with decrease-key operations taking O(logV).
- The difference is that the actual number of decrease-key operations depends on the graph's density and distribution of edge weights. However, this is bounded by O(E).

Total Complexity: O((V+E) logV)

Best Case Time Complexity

- The best case occurs when the number of decrease-key operations is minimized (e.g., when most vertices are directly reachable with no need for repeated updates).
- Even in this case, the algorithm's time complexity remains O((V+E) logV) due to the structural need to process all vertices and edges.

Total Complexity: O((V+E) logV)

Pseudocode

1. Vertex Selection: In every iteration, the algorithm must select the vertex with the smallest tentative distance. Since we are using a normal array, this requires a linear scan over all vertices, which takes O(V) time. This scan happens once for each vertex, leading toV such scans, thus O(V$\hat{2}$) in total.
2. Edge Relaxation: For each selected vertex, the algorithm relaxes all its adjacent edges, which again leads to O(V$\hat{2}$) work.

In case of worst, average, and best case, the same method must be applied leading to the same time complexity.

4.4 Pseudocode

```
function binHeapDijkstras(times: vector<vector<int>>, x: int, y: int) -> int {
    // Create graph as adjacency list
    map<int, map<int, int>> graph;
    for each (src, dest, cost) in times {
        graph[src][dest] = cost;
    }

    // Initialize distance costs and priority queue
    map<int, int> distanceCosts;
    for (i = 1; i <= x; i++) {
        distanceCosts[i] = INT_MAX;
    }
    distanceCosts[y] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> minD
    minDist.push({0, y});

    set<int> visited;

    // While there are elements in the priority queue
    while (!minDist.empty()) {
        auto [currentDist, currentNode] = minDist.top();
        minDist.pop();

        if (visited.find(currentNode) != visited.end()) {
            continue;
        }

        visited.insert(currentNode);

        for (auto [neighbor, weight] : graph[currentNode]) {
            if (visited.find(neighbor) != visited.end()) {
                continue;
```

```
            }

            int newDist = currentDist + weight;
            if (newDist < distanceCosts[neighbor]) {
                distanceCosts[neighbor] = newDist;
                minDist.push({newDist, neighbor});
            }
        }
    }

    // Check if all nodes are visited
    if (visited.size() != distanceCosts.size()) {
        return -1;
    }

    // Find the maximum distance
    int maxDistance = *max_element(distanceCosts.begin(), distanceCosts.end(),
                                   [](auto &a, auto &b) { return a.second < b.second; })
    return maxDistance;
}
```

4.5 Fibonacci Heap + Priority queue Implementation:

Our final implementation of Dijkstra's algorithm is the most efficient. A Fibonacci heap is a lazy data structure, meaning it prioritizes speed during common operations by deferring some of the internal reorganization. This results in faster execution for many operations compared to a binary heap. Here are the time complexities for the same common operations when using a Fibonacci heap:
The reason Fibonacci heaps achieve O(1) time for many operations, compared to binary heaps, lies in their structure. Each node maintains a pointer to its parent and one of its children. Additionally, the children of a node are connected through a circular linked list. This design allows the heap to always have a direct pointer to the node with the minimum value, enabling highly efficient execution of several operations.

Implementation Details

Graph Representation: The graph is represented using an adjacency list to store vertices and their corresponding edges.

Initialization of Fibonacci Heap:

- Insert all vertices into the heap with initial distances:
  - Set the source vertex distance to 0.
  - Set all other vertex distances to infinity (∞).

Main Algorithm Loop:

- While the heap is not empty:
  - Extract the vertex u with the smallest distance from the heap.
  - For each neighbor v of u:
    - Calculate the potential new distance: $d[u] + c(u, v)$.
    - If the new distance is smaller:
    - Update $d[v]$.
    - Perform a decrease-key operation in the heap.

Termination: The algorithm terminates when all vertices have been processed. The distance array $d[]$ will then hold the shortest distances from the source vertex.
Main Algorithm Loop

- While the heap is not empty:
  - Extract the vertex u with the smallest distance from the heap.
  - For each neighbor v of u:
    - Calculate potential new distance: d[u]+c(u,v).
    - If the new distance is smaller, update d[v] and perform decrease-key in the heap.

Steps to Calculate Time Complexity

*Initialization

- Initialize the Fibonacci Heap with V vertices.
- Each vertex starts with a distance of infinity, except the source vertex, which is set to 0.

Time Complexity of this step: $\mathcal{O}(V)$.

Extract-Min Operations

- Each vertex is extracted from the heap exactly once.
- Extracting the minimum element in a Fibonacci Heap takes $\mathcal{O}(\log V)$ time.

Relaxation of Edges

- For each edge, a decrease-key operation is performed if a shorter path is found.
- The decrease-key operation takes $\mathcal{O}(1)$ time in a Fibonacci Heap, and each edge is relaxed at most once.

Time Complexity: $\mathcal{O}(E)$.

5 Limitations and Edge Cases of Dijkstra's Algorithm

1. Dijkstra's algorithm fails when the shortest paths in graphs that contain negative edge weights are calculated. This limitation arises because the algorithm assumes that once a node's shortest distance is finalized, it will not need to be revisited. However, in graphs with negative edge weights, it is possible for shorter paths to emerge after visiting a node, violating this assumption.
2. An edge case is duplicate node in priority queue
3. Another edge case are disconnected graphs or graphs with a single vertex

5.1 Why Dijkstra's Fails?

The primary issue is that the algorithm does not consider the possibility of negative edges reducing the total path length after the initial calculation. For example, if the algorithm finalizes a path to a node using a non-optimal route due to a negative edge later, it won't revisit that node, leading to incorrect results. This causes us to come up with new algorithms which can handle this issue. We will have a brief look at 3 different algorithms which are similar to Dijkstra's Algorithm. They are:

- Bellman-Ford Algorithm
- Floyd Warshall Algorithm
- A* algorithm

5.2 The Bellman-Ford Algorithm

The Bellman-Ford algorithm is specifically designed to handle graphs with negative edge weights. It works by iterating through all the edges in the graph multiple times (up to V1 iterations, where V is the number of vertices). Each iteration attempts to "relax" the edges by updating the shortest distance to each vertex.

– Advantages of Bellman-Ford:

1. Handles Negative Edge Weights: Accurately calculates shortest paths in graphs with negative edges.
2. Detects Negative Weight Cycles: Identifies if a graph contains cycles where the total weight is negative, which makes finding the shortest path undefined.

– Time complexity: O(V*E), where V is the number of vertices and E is the number of edges

5.3 Floyd Warshall Algorithm

This is a dynamic programming approach used to find the shortest paths between all pairs of vertices in a weighted graph. It works by iteratively updating the shortest path matrix, starting with direct edge weights and progressively considering whether each vertex v provides a shorter path between any pair of vertices i and j. The algorithm uses a distance matrix dist[i] [j] (which represents the shortest path from vertex i to vertex j). Initially, the direct edges are set in the matrix, and then for each intermediate vertex v, the algorithm updates the shortest path using the relation:

dist[i] [j] = min(dist[i] [j],dist[i] [v]+dist[v] [j])

This process is repeated for all vertices v, and at the end, the matrix contains the shortest paths between all pairs of vertices.

– Time complexity: V is the number of vertices, then O(V^3).

5.4 Floyd Warshall Algorithm

The A algorithm* is a pathfinding and graph traversal algorithm used to find the shortest path between a start and a goal node in a weighted graph. It combines the benefits of Dijkstra's algorithm and greedy best-first search by using both the actual cost to reach a node and a heuristic estimate of the cost from that node to the goal. The algorithm maintains a priority queue of nodes to explore, where each node has a score f(n)=g(n)+h(n), with g(n) being the cost from the start node to node n and h(n) being the heuristic estimate of the cost from n to the goal. A* selects the node with the lowest f(n) value, expanding it while exploring the graph.

– Time complexity: Depends on the heuristic function used but in worst case it is O(E) where E is the number of edges.

6 Our Proposed Contribution (Bonus Part)

Dijkstra's algorithm, developed by Edsger Dijkstra in 1959, is a classic solution to this problem when all edge weights are non-negative. The traditional approach is to find the shortest path from a single source node to all other nodes in the graph. However, in certain applications, the problem arises where one needs to compute the shortest paths from all nodes in the graph to a specific destination node. This part explores an efficient method for solving this problem using a modified version of Dijkstra's algorithm.

6.1 Problem Overview

The problem is to compute the shortest paths from all nodes in the graph to a particular destination node. Given a directed, weighted graph G=(V,E), where

V is the set of vertices (nodes) and E is the set of edges with associated weights, the goal is to find the shortest path from every node vVto a specified destination node tV. The traditional method of finding the shortest paths from each node to the destination individually would involve running Dijkstra's algorithm multiple times, once for each source node. This can be computationally expensive, especially for large graphs.

6.2 Reversing the Graph

An intuitive solution to avoid running Dijkstra multiple times is to reverse the direction of all edges in the graph. The idea behind this approach is to transform the problem of finding the shortest paths from all nodes to a specific destination into the standard problem of finding the shortest paths from a single source to all other nodes. Reversing the graph means that if there is an edge from node u to node v in the original graph, we reverse it to create an edge from v to u. In this reversed graph, the problem becomes one where we need to find the shortest paths from the destination node N to all other nodes in the graph.

6.3 Applying Dijkstra's Algorithm

1. Initialize the distance of the destination node N to zero and the distances of all other nodes to infinity.
2. Place the destination node in a priority queue.
3. Repeatedly extract the node with the smallest tentative distance from the priority queue, update the distances of its neighbors, and insert those neighbors into the priority queue.
4. Continue the process until all nodes have been processed.

At the end of this process, we will have the shortest path distances from every node to the destination node in the reversed graph, corresponding to the shortest path distances from every node to the destination in the original graph.

6.4 Pseudocode

```
#FUNCTION reverse_graph(graph):
    INITIALIZE reversed_graph as an empty map
    FOR each node in graph:
        FOR each neighbor, weight in graph[node]:
            ADD (node, weight) to reversed_graph[neighbor]
    RETURN reversed_graph

FUNCTION dijkstra(graph, source):
    INITIALIZE distance as a map where all nodes have value infinity
    SET distance[source] = 0
    INITIALIZE a priority queue with (0, source)  # (distance, node)
```

```
    WHILE priority_queue is not empty:
        EXTRACT (current_distance, current_node) from priority_queue

        IF current_distance > distance[current_node]:
            CONTINUE  # Ignore outdated entry

        FOR each neighbor, weight in graph[current_node]:
            SET distance_to_neighbor = current_distance + weight
            IF distance_to_neighbor < distance[neighbor]:
                SET distance[neighbor] = distance_to_neighbor
                ADD (distance_to_neighbor, neighbor) to priority_queue

    RETURN distance

FUNCTION shortest_paths_to_destination(graph, destination):
    # Step 1: Reverse the graph
    reversed_graph = reverse_graph(graph)

    # Step 2: Run Dijkstra's algorithm from the destination node in the reversed graph
    distances = dijkstra(reversed_graph, destination)

    RETURN distances

# Example usage:
graph = {
    'A': [('B', 1), ('C', 4)],
    'B': [('C', 2), ('D', 5)],
    'C': [('D', 1)],
    'D': []
}

destination = 'D'
shortest_paths = shortest_paths_to_destination(graph, destination)

# Output the shortest paths from all nodes to the destination node
FOR each node, distance in shortest_paths:
    PRINT "Shortest path from", node, "to", destination, ":", distance
```

6.5 C++ code with output:

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
#include <climits>
```

```
using namespace std;

// Function to reverse the graph
unordered_map<char, vector<pair<char, int>>> reverse_graph(unordered_map<char, vector<pa
    unordered_map<char, vector<pair<char, int>>> reversed_graph;

    // Iterate through each node and reverse the edges
    for (auto& node : graph) {
        char u = node.first;
        for (auto& neighbor : node.second) {
            char v = neighbor.first;
            int weight = neighbor.second;
            reversed_graph[v].push_back({u, weight});  // Reverse the edge direction
        }
    }

    return reversed_graph;
}

// Function to implement Dijkstra's Algorithm
unordered_map<char, int> dijkstra(unordered_map<char, vector<pair<char, int>>>& graph, c
    unordered_map<char, int> distance;
    priority_queue<pair<int, char>, vector<pair<int, char>>, greater<pair<int, char>>> p

    // Initialize distances
    for (auto& node : graph) {
        distance[node.first] = INT_MAX;  // Set initial distances to infinity
    }
    distance[source] = 0;

    // Add the source to the priority queue
    pq.push({0, source});

    // Dijkstra's Algorithm
    while (!pq.empty()) {
        char current_node = pq.top().second;
        int current_distance = pq.top().first;
        pq.pop();

        // Skip if the distance is not optimal
        if (current_distance > distance[current_node]) {
            continue;
        }

        // Explore neighbors
```

```
            for (auto& neighbor : graph[current_node]) {
                char next_node = neighbor.first;
                int weight = neighbor.second;
                int new_distance = current_distance + weight;

                if (new_distance < distance[next_node]) {
                    distance[next_node] = new_distance;
                    pq.push({new_distance, next_node});  // Push updated distance to priorit
                }
            }
        }

        return distance;
    }

    // Main function to compute shortest paths from all nodes to a destination node
    unordered_map<char, int> shortest_paths_to_destination(unordered_map<char, vector<pair<c
        // Step 1: Reverse the graph
        unordered_map<char, vector<pair<char, int>>> reversed_graph = reverse_graph(graph);

        // Step 2: Run Dijkstra's algorithm from the destination node in the reversed graph
        unordered_map<char, int> distances = dijkstra(reversed_graph, destination);

        return distances;
    }

    // Example usage
    int main() {
        // Graph represented as an adjacency list
        unordered_map<char, vector<pair<char, int>>> graph = {
            {'A', {{'B', 1}, {'C', 4}}},
            {'B', {{'C', 2}, {'D', 5}}},
            {'C', {{'D', 1}}},
            {'D', {}}
        };

        char destination = 'D';
        unordered_map<char, int> shortest_paths = shortest_paths_to_destination(graph, desti

        // Output the shortest paths from all nodes to the destination node
        for (auto& node : shortest_paths) {
            cout << "Shortest path from " << node.first << " to " << destination << ": "
                 << (node.second == INT_MAX ? "No path" : to_string(node.second)) << endl;
        }
```

```
    return 0;
}
```

Output:
Shortest path from A to D: 0
Shortest path from D to D: 0
Shortest path from C to D: 1
Shortest path from B to D: 3

6.6 Time Complexity

The time complexity of Dijkstra's algorithm with a priority queue is O((V+E)logV), where V is the number of vertices and E is the number of edges in the graph. Reversing the graph takes O(E) time, as each edge in the graph needs to be processed once to reverse its direction. Therefore, the overall time complexity of the approach is O((V+E)logV). This is the same as the time complexity of Dijkstra's algorithm for the single-source shortest path problem, which is quite efficient for graphs with a large number of vertices and edges.

6.7 Advantages of this approach

This method computes the shortest paths from all nodes to the destination using a single run of Dijkstra's algorithm. By reversing the graph and treating the destination as the source, we avoid running Dijkstra multiple times, reducing computational overhead, especially for large graphs. The approach is simple to implement, involving just two steps: reversing the graph and running Dijkstra from the destination node. It's both efficient and intuitive.

6.8 Edge Cases and Limitations

1. While this approach works well in most scenarios, there are some limitations to consider. One key factor is that Dijkstra's algorithm is not suitable for graphs with negative edge weights. This method assumes all edge weights are non-negative. If negative weights are present, alternative algorithms like Bellman-Ford would need to be applied.
2. Another potential issue arises in graphs with disconnected components. If the graph is not fully connected, some nodes may not have a path to the destination node. In such cases, the shortest path distance for these nodes will remain infinite, indicating that no valid path exists between those nodes and the destination.

References

1. University of Minnesota, Minneapolis. Oral history interview with Edsger W. Dijkstra, Charles Babbage Institute.

2. The Clean Code Blog. Implementation of Dijkstra's algorithm using TDD, Robert Cecil Martin.
3. Dijkstra, E.W. (1959). "A Note on Two Problems in Connexion with Graphs". Retrieved from https://link.springer.com/article/10.1007/BF01386390.
4. Cormen, T.H., et al. (2009). *Introduction to Algorithms*. Retrieved from https://drive.google.com/file/d/0B3RHrbxFb7PfYjk4ZG01Z3lrbnc/view?pli=1resourcekey=0-aHyhqxUeXCNvRK3$_Q fNurg$.
5. Sedgewick, R. & Wayne, K. (2011). *Algorithms*. Retrieved from https://drive.google.com/file/d/0B9$_P JNHU4_dcTJLOHJjQkZrbDg/view?resourcekey =$ $0 - G7JAF1EIgOcdoQR4hfOcEw$.
6. Barnett, J.H. (2009). Early Writings on Graph Theory: Euler Circuits and The Königsberg Bridge Problem. In: Hopkins, B. (Ed.), *Resources for Teaching Discrete Mathematics: Classroom Projects, History Modules, and Articles*. MAA Notes. Mathematical Association of America, pp. 197–208. Retrieved from https://www.cambridge.org/core/books/abs/resources-for-teaching-discrete-mathematics/early-writings-on-graph-theory-euler-circuits-and-the-konigsberg-bridge-problem/859A9DB1C6FDA9C581788160BB734446.
7. Graph (abstract data type). (2024, October 14). In *Wikipedia*. Retrieved from https://en.wikipedia.org/wiki/Graph$_(abstract_d ata_t ype)$.
8. Dijkstra's algorithm. (2024, November 9). In *Wikipedia*. Retrieved from
9. Early Writings on Graph Theory, Euler Circuits, and the Königsberg Bridge Problem. (n.d.). In *Resources for Teaching Discrete Mathematics*. Cambridge University Press. Retrieved from https://www.cambridge.org/core/books/abs/resources-for-teaching-discrete-mathematics/early-writings-on-graph-theory-euler-circuits-and-the-konigsberg-bridge-problem/859A9DB1C6FDA9C581788160BB734446.
10. Chapter abstract. (n.d.). In *Resources for Teaching Discrete Mathematics*. Oxford University Press. Retrieved from https://academic.oup.com/book/45622/chapter-abstract/394863558?redirectedFrom=fulltextlogin=true.