Technical Report

Algorithm Introduction

Description: Dijkstra's Algorithm is designed to find the shortest path from given source node to all other nodes in a weighted graph. It uses a greedy approach to continuously select the closest unvisited node and update the shortest path estimates. The algorithm is useful in networks where the requirement is to find minimize distance, cost, or any other metric, where all edge weights are non-negative.

How It Works:

1. Initialization: Set the distance to the source node as 0, while all other nodes are initialized to infinity (float('inf')) because they are not reachable initially.
2. Priority Queue: A min-heap (priority queue) is used to always select the node with the minimum distance that hasn't been processed yet.
3. Relaxation: For each node, the algorithm checks each of its neighbors. If a shorter path to a neighbor is found, the distance is updated.
4. Termination: The algorithm completes when all reachable nodes are processed, and the shortest paths to each of these nodes are recorded.

Pseudocode:

```
Function Dijkstra_Algorithm(Graph, Source):

    Print "Input Graph: ", Graph

    If Graph is empty:
        Print "The graph is empty."
        Return

    If Source is not in Graph:
        Print "Source node does not exist in the graph."
        Return

    For each Node, Edges in Graph:
        For each Neighbor, Weight in Edges:
            If Weight is negative:
                Print "Negative weight edges are not supported."
                Return

    Distance[Node] = infinity for each Node in Graph
    Distance[Source] = 0

    PriorityQueue = [(0, Source)]

    While PriorityQueue is not empty:
        CurrentDistance, CurrentNode = Extract node with minimum
distance from PriorityQueue

        For each Neighbor, Weight in Graph[CurrentNode]:
            UpdatedDistance = CurrentDistance + Weight
```

```
                If UpdatedDistance < Distance[Neighbor]:
                    Distance[Neighbor] = UpdatedDistance
                    Insert (UpdatedDistance, Neighbor) into PriorityQueue

        Print "Distances from source node ", Source, ": ", Distance

    End Function
```

Background: Dijkstra's Algorithm is grounded in the principles of graph theory and the greedy algorithm paradigm. It is specifically designed to operate on weighted graphs, where each edge weight quantifies a metric such as cost, time, or distance required to traverse between two nodes. For the algorithm to function correctly, all edge weights must be non-negative. A graph is composed of vertices (nodes) and edges (connections), and the primary objective of Dijkstra's Algorithm is to determine the shortest distance from a given source node to all other nodes in the graph. The algorithm leverages efficient data structures, notably priority queues, which are often implemented using heaps. These structures enable the algorithm to quickly and efficiently select the next node with the minimum tentative distance, ensuring optimal performance during the computation of shortest paths.

Complexity: The time complexity of Dijkstra's Algorithm is influenced by the choice of data structure for the priority queue. When implemented using a priority queue, the algorithm achieves a time complexity of $O((V + E) \log V)$, where V represents the number of vertices and E denotes the number of edges. This complexity makes the algorithm highly suitable for graphs requiring efficient computation of shortest paths. In terms of space complexity, the algorithm requires $O(V + E)$, accounting for the storage needed for the graph representation, the priority queue, and the distance mappings.

Comparison: Dijkstra's Algorithm is more efficient than the Bellman-Ford algorithm in terms of time complexity, provided that all edge weights are non-negative. However, this efficiency comes with a limitation, as Dijkstra's Algorithm cannot handle graphs containing negative edge weights, a scenario where the Bellman-Ford algorithm proves to be more versatile. Another alternative to Dijkstra's Algorithm is the A* algorithm, which enhances Dijkstra's approach by incorporating heuristics to guide the search process. This heuristic-based guidance often makes A* more efficient for specific pathfinding problems, particularly when a target node is known.

Implementation Details

Design Choices: I chose a min-heap (implemented using Python's heapq module) because it efficiently extracts the node with the smallest tentative distance. This data structure assigns and stores priority values based on distance, allowing quick access to the node with the smallest distance at each step, which is crucial for Dijkstra's algorithm.

Challenge Faced: One significant challenge during the implementation was ensuring that the shortest distance calculations were consistently accurate and efficient. Managing these updates required processing nodes in the correct order to avoid unnecessary revisits

or overwriting a shorter path with a longer one. Using a priority queue with a min-heap helped overcome this challenge by always selecting the node with the current shortest known distance, ensuring optimal path updates.

Architecture: This code implements Dijkstra's algorithm to find the shortest paths from a source node to all other nodes in a given graph. All node distances are initialized to infinity except for the source, which is set to zero. A priority queue (heapq) is used to extract nodes with the shortest distance efficiently. The main loop processes each node and updates the distances to its neighbors if a shorter path is found, prioritizing nodes with the least cost.

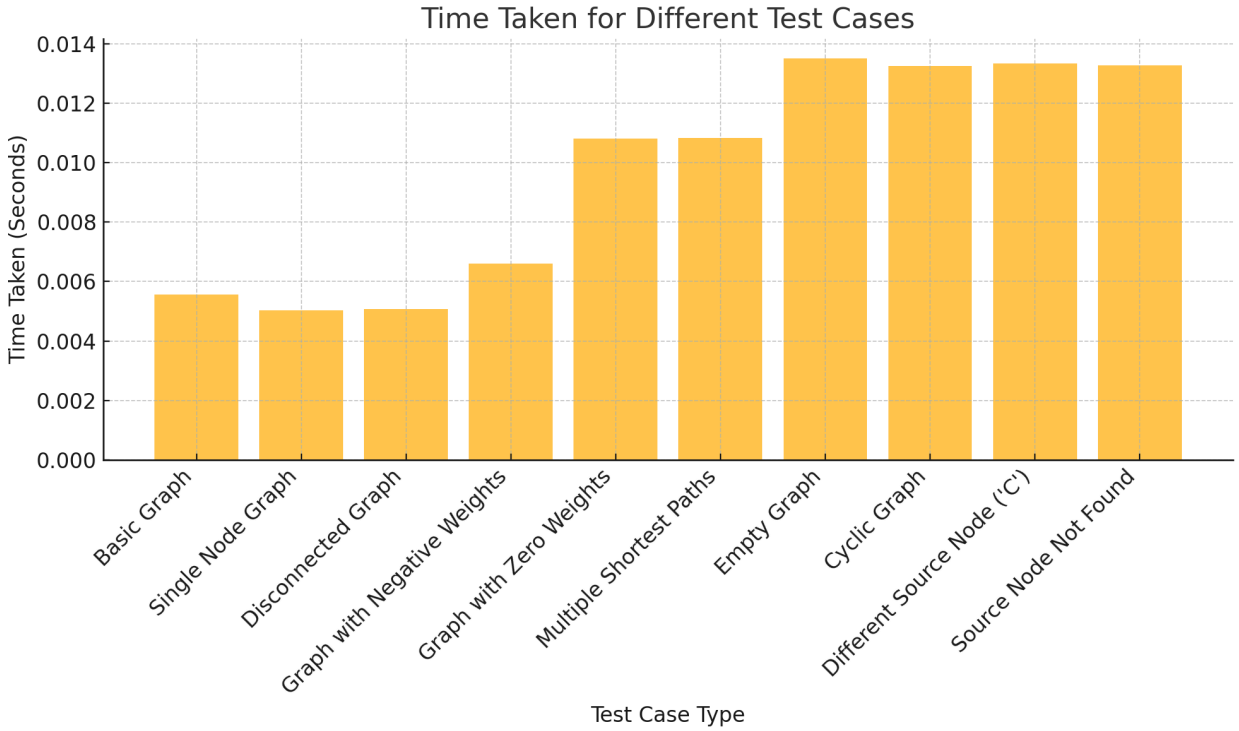
Benchmarking Methodology

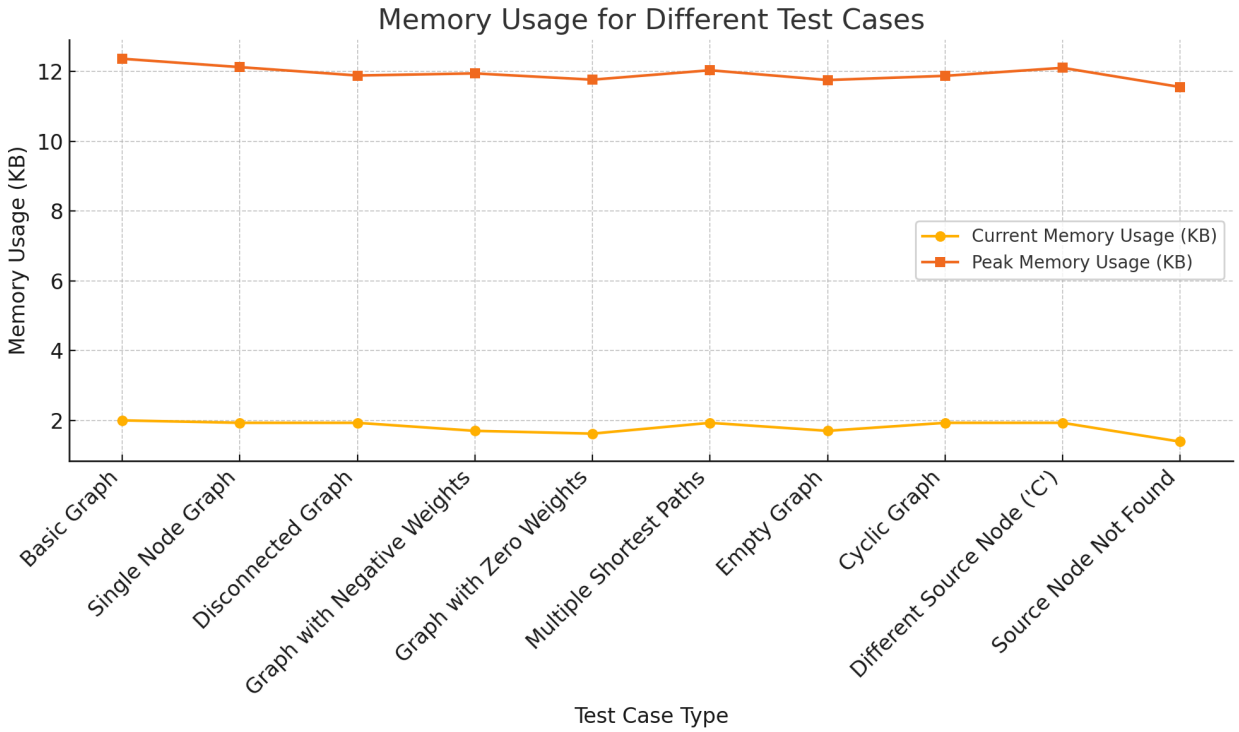
Workload: The workload consists of a collection of graph inputs used to test the Dijkstra algorithm implementation. These inputs include different types of graph structures, such as basic connected graphs, single-node graphs, disconnected graphs, graphs with zero-weight edges, and cyclic graphs. These diverse input cases are meant to represent a range of real-world scenarios and edge conditions, allowing for a comprehensive evaluation of the algorithm.

Dataset: The dataset consists of graph examples that were manually created using Python. Each graph is represented as an adjacency list in the form of a dictionary, where nodes are keys and edges are represented by lists of tuples containing neighboring nodes and weights. The graphs vary in size and complexity, including examples with five nodes, graphs containing cycles, graphs with zero-weight edges, disconnected parts, and graphs with negative weights to test the limitations of the algorithm. These graphs were specifically constructed to thoroughly test different situations the algorithm might encounter.

Setup: The implementation was benchmarked on a system running Windows 11 Home with an AMD Ryzen 7 7845HS processor (8 cores, 16 logical processors) and 32GB of RAM. The programming language used for the implementation was Python 3, specifically Python 3.13. The Python language was chosen for its built-in data structures, including the heapq module, which efficiently supports min-heap operations required for Dijkstra's algorithm, thus ensuring optimal performance during shortest path calculations.

Result:


Time Taken for Different Test Cases
Time Taken (Seconds)
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
Basic Graph
Single Node Graph
Disconnected Graph
Graph with Negative Weights
Graph with Zero Weights
Multiple Shortest Paths
Empty Graph
Cyclic Graph
Different Source Node ('C')
Source Node Not Found
Test Case Type


Memory Usage for Different Test Cases
Memory Usage (KB)
12
10
8
6
4
2
Current Memory Usage (KB)
Peak Memory Usage (KB)
Basic Graph
Single Node Graph
Disconnected Graph
Graph with Negative Weights
Graph with Zero Weights
Multiple Shortest Paths
Empty Graph
Cyclic Graph
Different Source Node ('C')
Source Node Not Found
Test Case Type

Summary and References

Conclusion: Dijkstra's Algorithm, a fundamental tool for solving shortest path problems in weighted graphs, was analyzed, implemented, and evaluated in this project. Using a greedy approach and a priority queue (min-heap), the algorithm efficiently calculates shortest paths, as demonstrated through diverse test cases, including disconnected graphs, cyclic structures, and zero-weight edges. The implementation's performance, with a time complexity of O((V + E) log V), confirmed its suitability for large graphs, though its limitation with negative edge weights was noted. Benchmarking highlighted its accuracy and resource efficiency, making it a reliable choice for real-world applications in networking and logistics. This project underscores the importance of algorithmic design and paves the way for exploring advanced heuristic-based approaches like the A* algorithm for specialized use cases.

Citations:

1. GeeksforGeeks, “Applications of Dijkstra’s Shortest Path Algorithm.” [Online]. Available: https://www.geeksforgeeks.org/applications-of-dijkstras-shortest-path-algorithm/. [Accessed: Nov. 29, 2024].
2. Javatpoint, "Dijkstra's Algorithm," [Online]. Available: https://www.javatpoint.com/dijkstras-algorithm. [Accessed: Nov. 29, 2024].
3. Medium, “Pathfinding: Dijkstra’s Algorithm.” [Online]. Available: https://medium.com/swlh/pathfinding-dijkstras-algorithm-65e71c346629. [Accessed: Nov. 29, 2024].
4. OpenGenus IQ, “Time and Space Complexity of Dijkstra’s Algorithm.” [Online]. Available: https://iq.opengenus.org/time-and-space-complexity-of-dijkstra-algorithm/. [Accessed: Nov. 29, 2024].
5. Python Software Foundation, "time — Time access and conversions," Python 3.13 documentation. [Online]. Available: https://docs.python.org/3/library/time.html. [Accessed: Nov. 29, 2024].
6. Python Software Foundation, "tracemalloc — Trace memory allocations," Python 3.13 documentation. [Online]. Available: https://docs.python.org/3/library/tracemalloc.html. [Accessed: Nov. 29, 2024].
7. ScienceDirect, "Dijkstra's Algorithms," [Online]. Available: https://www.sciencedirect.com/topics/computer-science/dijkstra-algorithms. [Accessed: Nov. 29, 2024].
8. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 4th ed. Cambridge, MA, USA: MIT Press, 2022.
9. WsCube Tech, “Dijkstra Algorithm.” [Online]. Available: https://www.wscubetech.com/resources/dsa/dijkstra-algorithm. [Accessed: Nov. 29, 2024].