Algorithms Report  
Jason gaynor, C23409212  
  
collaboration note: I worked with David Byrne C23308943 & Utibe Akpan C23783095

1. Overview

This report covers the implementation and execution of four graph algorithms: Depth-First Traversal (DFS), Breadth-First Traversal (BFS), Prim’s Minimum Spanning Tree (MST) algorithm, and Dijkstra’s Shortest Path Tree (SPT) algorithm. The graph is represented using adjacency lists.   
The program reads a graph from a file and executes all four algorithms from a specified starting vertex, providing step-by-step console output for clarity and debugging

1. Adjacency list diagram

A paper with numbers and a pen

AI-generated content may be incorrect.  
  
Adjacency Lists:

A → B(1), F(2), G(6)

B → A(1), C(1), D(2), E(4)

C → B(1), E(4)

D → B(2), E(2), F(1)

E → B(4), C(4), D(2), F(2), L(4)

F → A(2), D(1), E(2), L(2)

G → A(6), H(3), J(1), L(5)

H → G(3), I(2)

I → H(2), K(1)

J → G(1), K(1), L(3), M(2)

K → I(1), J(1)

L → E(4), F(2), G(5), J(3), M(1)

M → J(2), L(1).

1. MST

Step by step process for Prim’s Minimum spanning tree:

Step 1 Extract L

Process edges

L→E(4), L→F(2), L→G(5), L→J(3), L→M(1)

dist[]: [∞,∞,∞, ∞,4,2,5,∞,∞,3,∞,0,1]

parent[]: [-1,-1,-1,-1,12,12,12,-1,-1,12,-1,-1,12]

heap: [(M,1),(F,2),(J,3),(E,4),(G,5)]

Step 2 Extract M

Process edges

M→J(2) [dist[J]=3 vs. 1+2=3 → no update]

dist[]: No change

parent[]: No change

heap: [(F,2),(J,3),(E,4),(G,5)]

Step 3 Extract F

Process edges

F→A(2), F→D(1), F→E(2)

Updates

A: dist=2, parent=F

D: dist=1, parent=F

E: dist improved from 4→2, parent=F

dist[]: [2,∞,∞,1,2,2,5,∞,∞,3,∞,0,1]

parent[]: [6,-1,-1,6,6,12,12,-1,-1,12,-1,-1,12]

heap: [(D,1),(A,2),(E,2),(J,3),(G,5)]

Step 4 Extract D

Process edges

D→B(2), D→E(2) [no improvement]

dist[]: [2,2,∞,1,2,2,5,∞,∞,3,∞,0,1]

parent[]: [6,4,-1,6,6,12,12,-1,-1,12,-1,-1,12]

heap: [(A,2),(B,2),(E,2),(J,3),(G,5)]

Step 5 Extract A

Process edges

A→B(1) [improves B's dist from 2→1]

dist[]: [2,1,∞,1,2,2,5,∞,∞,3,∞,0,1]

parent[]: [6,1,-1,6,6,12,12,-1,-1,12,-1,-1,12]

heap: [(B,1),(E,2),(J,3),(G,5)]

Step 6: Extract B

Process edges:

B→C(1)

dist[]: [2,1,1,1,2,2,5,∞,∞,3,∞,0,1]

parent[]: [6,1,2,6,6,12,12,-1,-1,12,-1,-1,12]

heap: [(C,1),(E,2),(J,3),(G,5)]

Step 7: Extract C

dist[] & parent[]: No updates

heap: [(E,2),(J,3),(G,5)]

Step 8: Extract E

Process edges:

E→G(1) [improves G's dist from 5→1]

dist[]: [2,1,1,1,2,2,1,∞,∞,3,∞,0,1]

parent[]: [6,1,2,6,6,12,5,-1,-1,12,-1,-1,12]

heap: [(G,1),(J,3)]

Step 9: Extract G

Process edges:

* G→H(3), G→J(1) [improves J's dist from 3→2]

dist[]: [2,1,1,1,2,2,1,3,∞,2,∞,0,1]

parent[]: [6,1,2,6,6,12,5,7,-1,7,-1,-1,12]

heap: [(J,2),(H,3)]

Step 10: Extract J

Process edges

* J→K(1), J→M(2)

Updates:

* K: dist=1, parent=J
* M: already in MST

dist[]: [2,1,1,1,2,2,1,3,∞,2,1,0,1]

parent[]: [6,1,2,6,6,12,5,7,-1,7,10,-1,12]

heap: [(K,1),(H,3)]

Step 11: Extract K

Process edges

* K→I(1)

dist[]: [2,1,1,1,2,2,1,3,1,2,1,0,1]

parent[]: [6,1,2,6,6,12,5,7,11,7,10,-1,12]

heap: [(I,1),(H,3)]

Step 12: Extract I

Process edges

* I→H(2) [improves H's dist from 3→2]

dist[]: [2,1,1,1,2,2,1,2,1,2,1,0,1]

parent[]: [6,1,2,6,6,12,5,9,11,7,10,-1,12]

heap: [(H,2)]

Step 13: Extract H

No updates

dist[]: [2,1,1,1,2,2,1,2,1,2,1,0,1]

parent[]: [6,1,2,6,6,12,5,9,11,7,10,-1,12]

heap: []

MSP Complete

1. SPT Dijkstra

Initial Setup

dist[]: [∞, ∞, ∞, ∞, ∞, ∞, ∞, ∞, ∞, ∞, ∞, 0, ∞]

parent[]: [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]

Priority queue (min-heap): [(L, 0)]

Step 1: Extract vertex L (index 12)

Relax neighbors:

L→E (4), L→F (2), L→G (5), L→J (3), L→M (1)

Updated state:

dist[]: [∞, ∞, ∞, ∞, 4, 2, 5, ∞, ∞, 3, ∞, 0, 1]

parent[]: [-1, -1, -1, -1, 12, 12, 12, -1, -1, 12, -1, -1, 12]

Heap: [(M, 1), (F, 2), (J, 3), (E, 4), (G, 5)]

(Note: B not yet discovered)

Step 2: Extract vertex M (index 13)

Relax neighbor:

M→J (2) → dist[J] = 3, potential update is 1 + 2 = 3 → no change

State remains the same.

Heap: [(F, 2), (J, 3), (E, 4), (G, 5)]

Step 3: Extract vertex F (index 6)

Relax neighbors:

F→A (2), F→D (1), F→E (2)

Updates:

A: dist = 2 + 2 = 4, parent = F

D: dist = 2 + 1 = 3, parent = F

dist[]: [4, ∞, ∞, 3, 4, 2, 5, ∞, ∞, 3, ∞, 0, 1]

parent[]: [6, -1, -1, 6, 12, 12, 12, -1, -1, 12, -1, -1, 12]

Heap: [(D, 3), (A, 4), (J, 3), (E, 4), (G, 5)]

(Note: First connection to D — possible path to B forming)

Step 4: Extract vertex D (index 4)

Relax neighbors:

D→B (2), D→E (2)

Updates:

B: dist = 3 + 2 = 5, parent = D

E: current = 4, potential = 5 → no update

dist[]: [4, 5, ∞, 3, 4, 2, 5, ∞, ∞, 3, ∞, 0, 1]

parent[]: [6, 4, -1, 6, 12, 12, 12, -1, -1, 12, -1, -1, 12]

Heap: [(J, 3), (A, 4), (B, 5), (E, 4), (G, 5)]

(Note: B now reachable via L → F → D → B with distance 5)

Step 5: Extract vertex J (index 10)

Relax neighbors:

J→G (1), J→K (1)

Updates:

G: new dist = 3 + 1 = 4 (improved from 5), parent = J

K: dist = 3 + 1 = 4, parent = J

dist[]: [4, 5, ∞, 3, 4, 2, 4, ∞, ∞, 3, 4, 0, 1]

parent[]: [6, 4, -1, 6, 12, 12, 10, -1, -1, 12, 10, -1, 12]

Heap: [(A, 4), (B, 5), (E, 4), (G, 4), (K, 4)]

Step 6: Extract vertex A (index 1)

Relax neighbors:

A→B (1), A→F (2), A→G (6)

B: potential new dist = 2 + 2 + 1 = 5 (same as current) → no update

F already processed

G has a better distance via J → no update

No changes.

Heap: [(B, 5), (E, 4), (G, 4), (K, 4)]

Step 7: Extract vertex B (index 2)

Target node reached.

Confirmed shortest path: L → F → D → B

Total distance: 2 (L→F) + 1 (F→D) + 2 (D→B) = 5

Final details for B:

Distance from L: 5

Parent: D (index 4)

Path reconstruction:

B ← D ← F ← L

Edges: L→F (2), F→D (1), D→B (2)

1. Diagram of mst superimposed

A diagram of a diagram

AI-generated content may be incorrect.

1. Diagram of SPT superimposed

A diagram of a diagram

AI-generated content may be incorrect.

1. Program execution

A screenshot of a computer program

AI-generated content may be incorrect. A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

1. Description of challenging world graph

Description of the Challenging World Graph

The graph I used was crumlin\_roads.txt.   
It models a simplified map of a road network in Crumlin, Dublin consisting of 13 vertices and 20 edges.  
Each edge has an associated weight representing the travel cost.  
 The graph includes a combination of:

Low-cost direct paths (e.g., 1–2 with weight 1, 2–3 with weight 1),

High-cost alternative routes (e.g., 1–7 with weight 6),

Cycles (e.g., 4–5–6–4), adding to complexity.

This setup challenges Dijkstra’s algorithm by including several nodes with multiple incoming and outgoing connections, and various competing paths with differemt weights.

Dijkstra’s Algorithm Performance:

Dijkstra’s algorithm performed well on this graph

Correctness: The algorithm correctly built the Shortest Path Tree (SPT) from the given starting vertex. It accurately gave the shortest path to all other nodes, confirmed by the parent array output.

Efficiency:

Time: The total running time was low (usually under 10ms)

Memory Usage: Minimal memory usage was reported 0mb due to the efficient heap operations.

Heap Usage:  
The custom min-heap implementation enabled efficient extraction of the next closest vertex (remove()) and fast updates via siftUp() when shorter paths were found.  
Challenges I Encountered:

Multiple paths between nodes meant that Dijkstra had to perform frequent comparisons and heap updates.

Nodes like 5 and 6 had three or more paths feeding into them, so their distance updates occurred multiple times before finishing.

Overall, the algorithm demonstrated good handling of a realistic road network making it suitable for pathfinding in road navigation systems.

Discussion, Analysis, and Reflection

This assignment provided me an insight into the implementation and application of graph algorithms using adjacency lists. I gained experience with 4 foundational graph algorithms, Depth-First Search ,Breadth-First Search, Prims Algorithm for Minimum Spanning Tree and Dijkstras Algorithm for Shortest Path Tree (SPT)

What I Learned

1. Understanding Graph Representations  
   I learned how sparse graphs can be efficiently represented using adjacency linked *lists*. This memory-efficient structure avoids unnecessary storage of empty elements as would occur in an adjacency matrix and is especially useful when the number of edges is significantly less than the maximum possible.
2. Heap Usage in Graph Algorithms  
   The implementation of a custom min-heap class played an important role in optimizing both Prim’s and Dijkstra’s algorithms. I understood how the heap helps in retrieving the next minimum-distance vertex quickly, and how the siftUp and siftDown methods maintain the heap properties after insertion or updates.
3. Prim’s Algorithm   
   While implementing Prim’s algorithm, I saw how it continuously grows the MST by selecting the edge with the minimum weight that connects a vertex in the tree to one outside. Maintaining dist[], parent[], and hPos[] arrays helped in tracking the best available connections and updating the heap efficiently.
4. Dijkstra’s Algorithm   
   This algorithm is similar in structure to Prim’s but differs in how it updates the distance. It considers the path weight from the source to each vertex. I learned how Dijkstra's efficiently finds the shortest paths to all other vertices using a priority queue (heap) and dynamic updates.
5. Traversal Techniques   
   Implementing both DFS and BFS gave me a stronger grasp of traversal strategies. DFS helped in understanding exploration of paths, while BFS reinforced the use of queues to explore levels of a graph

What I Found Useful

* The use of a sentinel node z in the adjacency list simplified edge-case handling during traversal.
* Tracing the state of arrays like dist[], parent[], and the contents of the heap during execution gave me a much clearer picture of how these algorithms progress step by step.
* Being able to visually confirm algorithm progress and correctness through printed outputs such as the MST structure or SPT parent arrays was very useful for debugging and understanding.

Overall Reflection

This assignment allowed me to feel more confident in reading, building, and analyzing graph-based systems, which are fundamental in various domains like network design, route planning, and data mining. Working through these algorithms also strengthened my understanding of Java data structures, object-oriented programming, and algorithmic thinking.