OOPS Java project

Report on: Prim's Algorithm

Aggimalla Abhishek [23BDS004]

Nenavath Likhith Naik [23BDS037]

I. Introduction

Prim's algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a weighted, undirected graph. The MST is a subset of the edges of the graph that connects all the vertices together with the minimum total weight.

The key idea behind Prim's algorithm is to start with a single vertex and gradually add the closest (lowest-weight) unvisited vertex to the growing tree. This process is repeated until all vertices are included in the MST.

II. Key steps

Here's how Prim's algorithm works:

1. Initialize an empty MST.
2. Select a starting vertex and add it to the MST.
3. For each vertex in the MST, find the closest unvisited vertex and add the connecting edge to the MST, as long as adding the edge doesn't create a cycle.
4. Repeat step 3 until all vertices are included in the MST.

The algorithm maintains two main data structures:

1. A set of vertices that are already in the MST.
2. A priority queue (or min-heap) that stores the edges connecting the vertices in the MST to the unvisited vertices, sorted by weight.

At each step, the algorithm selects the edge with the minimum weight from the priority queue and adds the corresponding vertex to the MST. The algorithm continues this process until all vertices are included in the MST.

III. Comparision with other algorithm

Prim's algorithm is often compared to Kruskal's algorithm, another popular algorithm for finding the MST. The main difference is that Kruskal's algorithm builds the MST by considering the edges in order of increasing weight, while Prim's algorithm builds the MST by considering the vertices in order of their proximity to the growing tree.

IV. Time complexity

The time complexity of Prim's algorithm is primarily determined by the operations performed on the priority queue, which is used to efficiently select the next edge to add to the minimum spanning tree (MST).
In the provided code, the key operations performed on the priority queue are:

1. **Insertion**: When adding a new vertex to the priority queue (line 65 in the primMST method).
2. **Extraction of the minimum element**: When removing the vertex with the minimum key from the priority queue (line 58 in the primMST method).
3. **Decreasing the key value**: When updating the key (weight) of a vertex in the priority queue (line 72 in the primMST method).

The time complexity of these operations on a priority queue (implemented using a binary heap) is as follows:

1. **Insertion**: O(log V), where V is the number of vertices in the graph.
2. **Extraction of the minimum element**: O(log V).
3. **Decreasing the key value**: O(log V).

In the primMST method, the algorithm iterates through the adjacency list of each vertex in the graph (lines 61-77). The time complexity of this operation is O(E), where E is the number of edges in the graph.
Therefore, the overall time complexity of the Prim's algorithm implementation in the provided code is:
O(E log V)
This is because the algorithm performs the following steps:

1. Initializing the priority queue and other data structures: O(V)

2. Iterating through the adjacency list of each vertex: O(E)
3. Performing priority queue operations (insertion, extraction, and key decrease): O(log V) per operation, with a total of O(E log V)

The space complexity of the algorithm is O(V + E), as the code maintains an adjacency list representation of the graph, which requires O(V + E) space, and additional data structures such as the parent, key, and inMST arrays, which require O(V) space.

In summary, the time complexity of the provided Prim's algorithm implementation is O(E log V), which makes it efficient for large, sparse graphs, as the logarithmic factor in the time complexity is based on the number of vertices, rather than the number of edges.

V. Potential Applications

This graph could represent various real-world scenarios, such as:

- **Transportation Network**: The vertices could represent cities, and the edges could represent roads or flight routes between them, with the weights corresponding to the travel distances or costs.
- **Telecommunication Network**: The vertices could represent network nodes, and the edges could represent communication links between them, with the weights representing the cost or bandwidth of the connections.
- **Utility Network**: The vertices could represent consumers or distribution points, and the edges could represent pipelines or power lines, with the weights representing the cost or capacity of the connections.

In such scenarios, the Minimum Spanning Tree found by Prim's algorithm would represent the most cost-effective way to connect all the vertices while ensuring that the entire network remains functional.

VI. ALGORITHM STEPS

Initialization:

- **Create arrays to store the MST**:
  - **parent**[]: To store the parent of each vertex in the MST.
  - **key**[]: To store the minimum weight (key) of each vertex.
  - **inMST**[]: To track which vertices are already included in the MST.
- Initialize all **key** values to **Integer.MAX_VALUE** and all **inMST** values to false.
- Create a **PriorityQueue** to efficiently retrieve the vertex with the minimum key value.
- Start with vertex 0 by setting its **key** to 0, its **parent** to -1, and adding it to the priority queue.

2. Main Algorithm Loop:

- While the priority queue is not empty:
  - Poll the vertex **u** with the minimum key value from the priority queue.
  - Mark **u** as included in the MST by setting **inMST[u]** to **true**.
  - For each adjacent vertex **v** of **u**:
    - If v is not yet in the MST **(!inMST[v])** and the weight of the edge **(u, v)** is less than the current key value of **v (weight < key[v])**:
      - Update the **parent[v]** to **u**.
      - Update the **key[v]** to the weight of the edge **(u, v)**.
      - Add the updated vertex **v** to the priority queue.

3. Output:

- The **parent[]** array now contains the parent of each vertex in the MST, allowing you to reconstruct the MST.
- You can print the edges and weights in the MST using the **parent[]** array.

VII. Input Dataset

(0, 1) with weight 4
(0, 7) with weight 8
(1, 2) with weight 8
(1, 7) with weight 11
(2, 3) with weight 7
(2, 8) with weight 2
(2, 5) with weight 4
(3, 4) with weight 9
(3, 5) with weight 14
(4, 5) with weight 10
(5, 6) with weight 2
(6, 7) with weight 1
(6, 8) with weight 6
(7, 8) with weight 7

VIII. Output

- (0, 1) with weight 4
- (1, 2) with weight 8
- (2, 3) with weight 7
- (2, 8) with weight 2
- (2, 5) with weight 4
- (3, 4) with weight 9
- (5, 6) with weight 2
- (6, 7) with weight 1
- The total weight of this **MST** is **37.**

Report on: DFS Algorithm

I. INTRODUCTION

Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It is one of the fundamental algorithms in computer science, used in a variety of applications such as pathfinding, web crawling, and topological sorting.

The main idea behind DFS is to start at the root node (or any arbitrary node of a graph) and explore as far as possible along each branch before backtracking. This is achieved by maintaining a stack (or recursion) to keep track of the nodes that need to be visited.

II. KEY STEPS

Here's how the DFS algorithm works:

1. Start at the root node (or any arbitrary node of the graph).
2. Mark the current node as visited.
3. Push the current node onto the stack (or call a recursive function).
4. If the current node has unvisited neighbors, select one of them and repeat steps 2-3.
5. If the current node has no unvisited neighbors, pop the top node from the stack (or return from the recursive call).
6. Repeat steps 2-5 until the stack is empty (or the recursive calls have completed).

The algorithm can be implemented using both iterative (using a stack) and recursive approaches. The recursive approach is often more concise and easier to understand, while the iterative approach can be more memory-efficient for large graphs.

III. PROPERTIES AND APPLICATION

DFS has several important properties and applications:

1. Depth-first property: DFS explores the deepest unvisited node first before backtracking and exploring other branches.
2. Cycle detection: DFS can be used to detect cycles in a graph by checking if a node is visited more than once during the traversal.
3. Topological sorting: DFS can be used to perform topological sorting, which is useful in tasks like scheduling and dependency management.
4. Connected components: DFS can be used to identify the connected components of an undirected graph.
5. Maze solving: DFS can be used to solve mazes by exploring all possible paths until the exit is found.

IV. TIME COMPLEXITY

The time complexity of DFS can be broken down as:

Time Complexity: O(V + E)

- V = number of vertices
- E = number of edges

Here's why:

1. Each vertex is visited exactly once: O(V)
2. Each edge is traversed exactly once: O(E)

Space Complexity: O(V)

- This comes from the recursion stack in worst case (for a skewed/linear graph)
- For an iterative implementation, the stack/queue would also take O(V) space

V. POTENTIAL APPLICATIONS

The practical applications of DFS (Depth-First Search) across different domains:

1. Graph Problems
   - Cycle detection
   - Connected components analysis
   - Topological sorting (for DAGs)
   - Path finding between nodes
2. Game Development
   - Solving puzzles (e.g., maze generation/solving)
   - Game state exploration
   - AI decision trees
   - Move generation in games like chess
3. Web Crawling

    - Following links deeply into websites
    - Site mapping
    - Content indexing
4. File System Operations
    - Directory traversal
    - File searching
    - Calculating directory sizes
5. Compiler Design and Programming Languages
    - Expression parsing
    - Syntax tree traversal
    - Dead code elimination
    - Garbage collection (mark phase)
6. Social Networks
    - Finding connections between users
    - Analyzing social graphs
    - Recommendation systems
7. Networking
    - Network topology analysis
    - Routing algorithms
    - Broadcasting in networks
8. Artificial Intelligence
    - Constraint satisfaction problems
    - State space searching
    - Problem-solving algorithms
9. Dependency Resolution
    - Building dependency trees
    - Package management
    - Build systems
10. Data Analysis
    - Decision tree traversal
    - Hierarchical clustering
    - XML/JSON parsing

Most Common Use Cases:

1. Finding paths between nodes
2. Tree/graph traversal problems
3. Cycle detection
4. Topological sorting
5. Connected component analysis

VI. ALGORITHM STEPS

1. Graph Representation Fundamentals

- Choice between adjacency matrix or adjacency list
- Adjacency list preferred for sparse graphs (fewer edges)
- Need to track visited nodes
- Consider direction of edges (directed vs undirected)

2. Implementation Choices

- Recursive vs Iterative approaches
    - Recursive: Cleaner code, uses system stack
    - Iterative: Better space control, uses explicit stack
- Stack usage is essential (either way)
- Need to handle backtracking

3. Key Tracking Mechanisms

- Visited nodes tracking
- Parent nodes (for path reconstruction)
- Discovery times (when node is first visited)
- Finish times (when node processing completes)
- Component numbers (for disconnected graphs)

4. Critical Edge Cases to Handle

- Cycles in the graph
- Disconnected components
- Back edges
- Self-loops
- Empty graphs
- Single-node graphs

5. Performance Considerations

- Space complexity depends on graph structure
- Stack depth in worst case
- Memory usage in large graphs
- Choice of data structures impacts performance

VII. INPUT DATASET

INPUTS:

1. Graph Construction:
    - Number of vertices: 10
    - Edges added:

$0 \rightarrow 1$

$0 \rightarrow 2$

$1 \rightarrow 3$

$1 \rightarrow 4$

$2 \rightarrow 5$

$2 \rightarrow 6$

$3 \rightarrow 7$

$4 \rightarrow 8$

$5 \rightarrow 9$

- Starting vertex: 0

Visual representation of the input graph:

```
       0

     /   \

    1     2

   / \    / \

  3   4 5   6

  |   |   |

  7   8 9
```

VIII. OUTPUT

OUTPUT:

- DFS traversal sequence: 0 1 3 7 4 8 2 5 9 6

The output shows the vertices in the order they were visited during the depth-first traversal, starting from vertex 0 and exploring as far as possible along each branch before backtracking.

JAVA CODE USING PRIM'S ALGORITHM

```java
import java.util.ArrayList;
import java.util.PriorityQueue;

class PrimsAlgorithmWithAdjList {
    static class Edge {
        int destination;
        int weight;

        Edge(int destination, int weight) {
            this.destination = destination;
            this.weight = weight;
        }
    }

    static class Graph {
        int V;
        ArrayList<ArrayList<Edge>> adj;

        Graph(int v) {
            V = v;
            adj = new ArrayList<>();
            for (int i = 0; i < v; i++) {
                adj.add(new ArrayList<>());
            }
        }

        void addEdge(int src, int dest, int weight) {
            adj.get(src).add(new Edge(dest, weight));
            adj.get(dest).add(new Edge(src, weight));
        }
    }

    static class Node implements Comparable<Node>
{
        int vertex;
        int key;

        Node(int vertex, int key) {
            this.vertex = vertex;
            this.key = key;
        }

        @Override
        public int compareTo(Node other) {
            return this.key - other.key;
        }
    }

    public void primMST(Graph graph) {
        int V = graph.V;
        int[] parent = new int[V];
        int[] key = new int[V];
        boolean[] inMST = new boolean[V];

        for (int i = 0; i < V; i++) {
            key[i] = Integer.MAX_VALUE;
            inMST[i] = false;
        }

        PriorityQueue<Node> pq = new
PriorityQueue<>(V);

        key[0] = 0;
        parent[0] = -1;
        pq.offer(new Node(0, key[0]));

        while (!pq.isEmpty()) {
            int u = pq.poll().vertex;
            inMST[u] = true;

            for (Edge edge : graph.adj.get(u)) {
```

```
            int v = edge.destination;
            int weight = edge.weight;

            if (!inMST[v] && weight < key[v]) {
                parent[v] = u;
                key[v] = weight;
                pq.offer(new Node(v, key[v]));
            }
        }
    }

    printMST(parent, key, V);
  }

  private void printMST(int[] parent, int[] key, int V)
{
    int totalWeight = 0;
    System.out.println("Edge \tWeight");

    for (int i = 1; i < V; i++) {
        System.out.println(parent[i] + " - " + i + "\t" +
key[i]);
        totalWeight += key[i]; // Add the weight of
each edge to the total weight
    }

    System.out.println("Total Weight of MST: " +
totalWeight);
  }

  public static void main(String[] args) {
    int V = 9;
    Graph graph = new Graph(V);

    graph.addEdge(0, 1, 4);
    graph.addEdge(0, 7, 8);
    graph.addEdge(1, 2, 8);
    graph.addEdge(1, 7, 11);
    graph.addEdge(2, 3, 7);
    graph.addEdge(2, 8, 2);
    graph.addEdge(2, 5, 4);
    graph.addEdge(3, 4, 9);
    graph.addEdge(3, 5, 14);
    graph.addEdge(4, 5, 10);
    graph.addEdge(5, 6, 2);
    graph.addEdge(6, 7, 1);
    graph.addEdge(6, 8, 6);
    graph.addEdge(7, 8, 7);

    PrimsAlgorithmWithAdjList prim = new
PrimsAlgorithmWithAdjList();
    System.out.println("Edges in the constructed
Minimum Spanning Tree:");
      prim.primMST(graph);
  }
}
```

JAVA CODE USING DFS ALGORITHM

```
import java.util.LinkedList;
public class DFS {
  private LinkedList<Integer> adjlist[];
  private boolean visited[];
  public DFS(int vertices) {
    adjlist = new LinkedList[vertices];
    visited = new boolean[vertices];
    for (int i = 0; i < vertices; i++) {
      adjlist[i] = new LinkedList<>();
    }
  }

  public void addEdge(int src, int dest) {
    adjlist[src].add(dest);
  }

  public void dfs(int vertex) {
    visited[vertex] = true;
    System.out.print(vertex + " ");
    for (int neighbor : adjlist[vertex]) {
      if (!visited[neighbor]) {
        dfs(neighbor);
      }
    }
  }

  public static void main(String[] args) {
    int vertices = 10;
    DFS graph = new DFS(vertices);
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(1, 4);
    graph.addEdge(2, 5);
    graph.addEdge(2, 6);
    graph.addEdge(3, 7);
    graph.addEdge(4, 8);
    graph.addEdge(5, 9);
    System.out.println("Depth First Search traversal
starting from vertex 0:");
    graph.dfs(0);
  }
}
```