Ministry of Education of Republic of Moldova
Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics

Laboratory No.5 Study and empirical analysis of sorting algorithms. Analysis of Prim’s Algorithm and Kruskal’s Algorithm.

Verified:
Fistic Cristofor asist. univ.

Racovita Dumitru FAF-233

Moldova, April 2025

Contents

1 Greedy Algorithms **2**

1.1 Objective . . . 2

1.2 Tasks . . . 2

1.3 Theoretical Notes . . . 2

1.4 Introduction . . . 3

2 Implementation **4**

2.1 Prim's Algorithm . . . 4

2.1.1 Introduction . . . 4

2.1.2 Working Principle . . . 4

2.1.3 Advantages . . . 4

2.1.4 JS Implementation . . . 5

2.1.5 Time Complexity . . . 6

2.1.6 Auxiliary Space . . . 6

2.2 Kruskal's Algorithm . . . 7

2.2.1 Introduction . . . 7

2.2.2 Working Principle . . . 7

2.2.3 Advantages . . . 7

2.2.4 JS Implementation . . . 7

2.2.5 Time Complexity . . . 8

2.2.6 Auxiliary Space . . . 8

3 Conclusion **9**

1.1 Objective

The aim of this work is to perform an empirical analysis of greedy algorithms, with specific focus on Prim's Algorithm and Kruskal's Algorithm. The study intends to implement these algorithms, evaluate their efficiency across various input conditions, and compare their performance using appropriate metrics. Through graphical representation of results, we aim to assess their practical effectiveness in different scenarios.

1.2 Tasks

To fulfill the stated objective, we will undertake the following tasks: 1. Investigate the greedy algorithm design technique and its fundamental principles. 2. Develop implementations of Prim's and Kruskal's algorithms in a selected programming language. 3. Establish the characteristics of input data for analysis. 4. Identify suitable metrics for algorithm comparison, including execution time, memory usage, and edge selection count. 5. Conduct empirical tests by incrementally increasing graph node count and analyzing the resulting performance impact. 6. Visualize the obtained results graphically to highlight performance variations. 7. Organize findings into a comprehensive report discussing each algorithm's strengths and limitations.

1.3 Theoretical Notes

Greedy algorithms form a fundamental class of optimization techniques in computer science. These algorithms make locally optimal choices at each step with the hope of finding a globally optimal solution. Although they do not always guarantee the best solution, they are widely used due to their efficiency and simplicity.

Prim's Algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a weighted graph. It starts from a single node and continuously adds the smallest edge that connects a new vertex to the tree. The time complexity of Prim's Al- gorithm is ⬜(⬜ log ⬜) when implemented with a priority queue, where ⬜ is the number of vertices and ⬜ is the number of edges.

Kruskal's Algorithm is another greedy approach for finding the MST. Instead of growing the tree from a starting vertex, it sorts all edges by weight and adds them to the MST in increasing order while avoiding cycles. Kruskal's Algorithm typically has a time complexity of ⬜(⬜ log ⬜), making it efficient for graphs with fewer edges.

Key properties influencing algorithm performance include: - Graph density (sparse vs. dense graphs) - Graph structure (connected vs. disconnected, directed vs. undi- rected) - Graph size (number of nodes and edges) - Memory usage and sorting com- plexity

Empirical analysis allows evaluation of these properties by measuring execution time, memory consumption, and other relevant factors under different input condi- tions.

1.4 Introduction

Greedy algorithms provide efficient solutions to numerous optimization problems by making locally optimal decisions at each step. These algorithms function in a top-down manner without reconsidering previous choices. Although they don't invariably guarantee globally optimal solutions, they are frequently applied to problems possessing optimal substructure and greedy choice properties.

Prim's and Kruskal's algorithms represent two widely implemented greedy approaches for determining the Minimum Spanning Tree (MST) of a graph. While Prim's Algorithm constructs the MST incrementally by adding the smallest edge that expands the tree, Kruskal's Algorithm sorts all edges and selects the smallest ones while ensuring cycle avoidance. The performance characteristics of these algorithms vary according to the graph's node and edge counts.

Through empirical experimentation, this study aims to analyze how increasing node counts affects the efficiency of Prim's and Kruskal's algorithms. The findings will be presented graphically to compare their performance across various scenarios, highlighting their advantages and limitations.

These two algorithms, Prim's and Kruskal's, will be implemented in their native form in JavaScript and analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observa- tions, the particular efficiency in relation to input will vary depending on the memory of the device used.
The error margin determined will constitute 0.0005 seconds as per experimental measurement.

2.1 Prim's Algorithm

2.1.1 Introduction

Prim's algorithm represents a greedy approach used to identify the Minimum Spanning Tree (MST) of a weighted, connected, undirected graph. It begins with a single vertex and iteratively incorporates the lowest-weight edge that expands the developing MST.

2.1.2 Working Principle

The working principle of Prim's algorithm follows a **greedy** approach using a priority queue:

1. Initialize all vertices as excluded from the MST.
2. Assign zero weight to the starting vertex and infinite weight to all others.
3. Utilize a priority queue to extract the vertex with minimum key value.
4. Update adjacent vertices' key values when smaller weights are discovered.
5. Continue until all vertices are incorporated into the MST.

2.1.3 Advantages

Best graph types: Connected sparse graphs, nearly-tree structures

Why:

- Prim's grows a tree from a single source, performing best when most potential edges are easy to eliminate
- In sparse graphs, priority queue operations are reduced
- **Computational advantage:** With heap optimization, Prim's O(E log V) performs well when E is close to V
- **Weakness:** In dense graphs, priority queue operations dominate performance

Prim's algorithm offers several advantages:

- Works efficiently for dense graphs with many edges.

- Ensures step-by-step construction of a connected MST.
- Can be optimized using priority queue structures for enhanced performance.

2.1.4 JS Implementation

```
async function runPrim() {
   drawGraph();
   await delay(500);
   let mstNodes = new Set();
   let mstEdges = [];
   mstNodes.add(0);

   // Display the starting node
   ctx.fillStyle = "#000";
   ctx.font = "18px Arial";
   ctx.textAlign = "center";
   ctx.fillText("Starting from node 0", canvas.width / 2, 60);

   function getCandidateEdges() {
    return edges.filter(edge =>
     (mstNodes.has(edge.from) && !mstNodes.has(edge.to)) ||
     (mstNodes.has(edge.to) && !mstNodes.has(edge.from))
    );
   }

   while (mstNodes.size < nodes.length) {
    let candidateEdges = getCandidateEdges();
    if (candidateEdges.length === 0) break;
    candidateEdges.sort((a, b) => a.weight - b.weight);
    let chosenEdge = candidateEdges[0];

    // Highlight the chosen edge
    ctx.fillStyle = "#000";
    ctx.font = "16px Arial";
    ctx.textAlign = "center";
    ctx.fillText("Selected edge: " + chosenEdge.from + " → " + chosenEdge.to + " (weight: "
    +
chosenEdge.weight + ")", canvas.width / 2, 90);

      mstEdges.push(chosenEdge);
      let newVertex = mstNodes.has(chosenEdge.from) ? chosenEdge.to : chosenEdge.from;
      mstNodes.add(newVertex);
      drawGraph(Array.from(mstNodes), mstEdges);
      await delay(800); // Slightly longer delay to see the steps better
    }

    // Calculate total MST weight
    const totalWeight = mstEdges.reduce((sum, edge) => sum + edge.weight, 0);

    ctx.fillStyle = "#000";
    ctx.font = "24px Arial";
    ctx.textAlign = "center";
    ctx.fillText("Prim's Algorithm: MST Complete", canvas.width / 2, canvas.height - 60);
    ctx.font = "18px Arial";
    ctx.fillText("Total MST Weight: " + totalWeight, canvas.width / 2, canvas.height - 30);
  }
```

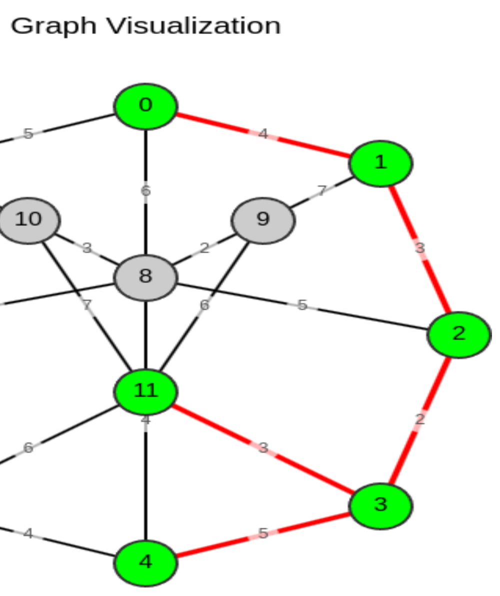


Figure 2.1: *Prim's Algorithm graph*

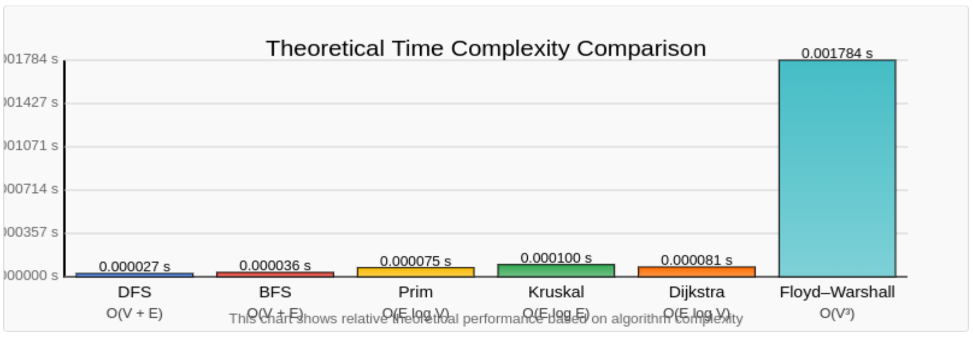


Figure 2.2: *Prim's Algorithm performance*

2.1.5 Time Complexity

- Using a priority queue: O(V log E)
- Without a priority queue: O(V^2)

2.1.6 Auxiliary Space

The auxiliary space complexity of Prim's algorithm is O(V + E) when using an adjacency list representation with a priority queue.

2.2 Kruskal's Algorithm

2.2.1 Introduction

Kruskal's algorithm is a greedy algorithm used for constructing the Minimum Spanning Tree by selecting edges in increasing order of weight while ensuring no cycles are formed.

2.2.2 Working Principle

The algorithm follows these steps:

1. Sort all edges in non-decreasing order of weight.
2. Initialize a disjoint-set data structure to manage connected components.
3. Iterate through the sorted edges and add an edge to the MST if it does not form a cycle.
4. Repeat until $V - 1$ edges are included in the MST.

2.2.3 Advantages

Best graph types: Sparse disconnected graphs, forest-like structures **Why:**

- Kruskal's considers edges in weight order, regardless of connectivity
- Performs well in disconnected or sparse graphs where finding minimum edges is straightforward
- **Computational advantage:** Edge sorting (O(E log E)) is most efficient when E is small
- **Weakness:** In very dense graphs, sorting all edges becomes expensive

Kruskal's algorithm provides:

- An efficient solution for sparse graphs.
- A simple and intuitive approach to building an MST.
- The ability to work well with edge-list representations of graphs.

2.2.4 JS Implementation

```
async function runKruskal() {
   drawGraph();
   await delay(500);
   let sortedEdges = [.edges].sort((a, b) => a.weight - b.weight);
   let mstEdges = [];
   let parent = Array.from({ length: nodes.length }, (_, i) => i);

   // Display initial message
   ctx.fillStyle = "#000";
   ctx.font = "18px Arial";
   ctx.textAlign = "center";
   ctx.fillText("Sorting edges by weight", canvas.width / 2, 60);

   function find(x) {
    if (parent[x] !== x) {
      parent[x] = find(parent[x]);
```

```
  }
 return parent[x];
}

function union(x, y) {
 let rootX = find(x);
```

```
    let rootY = find(y);
    if (rootX === rootY) return false;
    parent[rootY] = rootX;
    return true;
   }

   for (let edge of sortedEdges) {
    drawGraph([], mstEdges);

    // Display current edge being considered
    ctx.fillStyle = "#000";
    ctx.font = "16px Arial";
    ctx.textAlign = "center";
    ctx.fillText("Considering edge: " + edge.from + " → " + edge.to + "
(weight: " + edge.weight + ")", canvas.width / 2, 90);

    await delay(600);
    if (union(edge.from, edge.to)) {
     mstEdges.push(edge);
     drawGraph([], mstEdges);

     // Display edge being added
     ctx.fillStyle = "#000";
     ctx.font = "16px Arial";
     ctx.textAlign = "center";
     ctx.fillText("Added to MST: " + edge.from + " → " + edge.to, canvas.width / 2, 90);

     await delay(800);
    } else {
     drawGraph([], mstEdges);

     // Display edge creating cycle
     ctx.fillStyle = "#000";
     ctx.font = "16px Arial";
     ctx.textAlign = "center";
     ctx.fillText("Rejected (would create cycle): " + edge.from + " → " +
edge.to, canvas.width / 2, 90);

     ctx.beginPath();
     ctx.setLineDash([5, 5]);
     ctx.moveTo(nodes[edge.from].x, nodes[edge.from].y);
     ctx.lineTo(nodes[edge.to].x, nodes[edge.to].y);
     ctx.strokeStyle = "#f00";
     ctx.lineWidth = 4;
     ctx.stroke();
     ctx.setLineDash([]);
     await delay(800);
     drawGraph([], mstEdges);
    }
```

```
  }

  // Calculate total MST weight
  const totalWeight = mstEdges.reduce((sum, edge) => sum + edge.weight, 0);

  ctx.fillStyle = "#000";
  ctx.font = "24px Arial";
  ctx.textAlign = "center";
  ctx.fillText("Kruskal's Algorithm: MST Complete", canvas.width / 2, canvas.height - 60);
  ctx.font = "18px Arial";
  ctx.fillText("Total MST Weight: " + totalWeight, canvas.width / 2, canvas.height - 30);
}
```

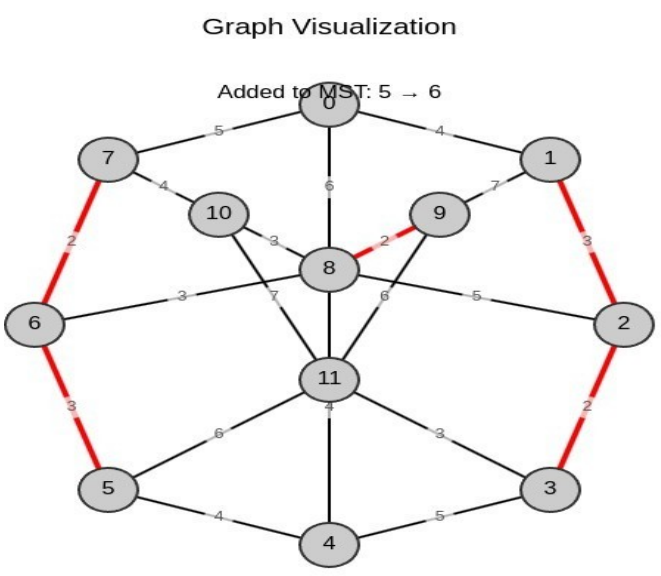
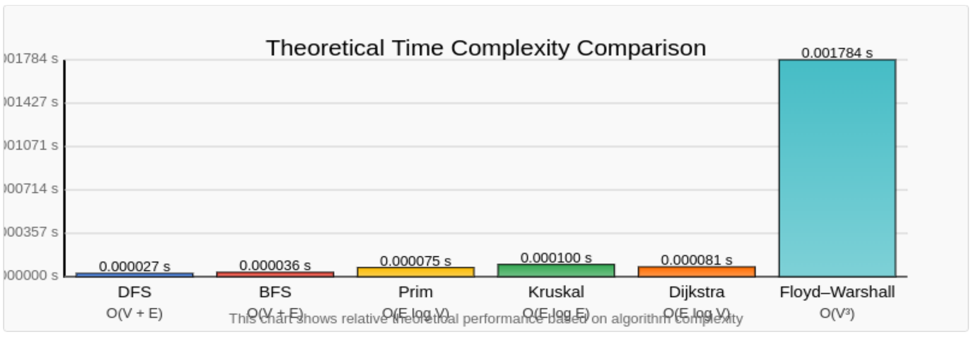


Figure 2.3: *Kruskal's Algorithm graph*

Figure 2.4: *Kruskal's Algorithm performance*



2.2.5 Time Complexity

- Using a disjoint-set: O(V log E)
- Sorting edges: O(V log E)

2.2.6 Auxiliary Space

The auxiliary space complexity of Kruskal's algorithm is O(V+E) due to the disjoint-set operations.

3 Conclusion

This paper has examined greedy algorithms as a methodology for addressing optimization problems by making locally optimal choices at each step with the aim of discovering a globally optimal solution. Specifically, we analyzed Prim's and Kruskal's algorithms in the context of minimum spanning tree computation, highlighting their efficiency, time complexity, and practical applications.

Prim's Algorithm Prim's algorithm employs a greedy approach that progressively builds the minimum spanning tree (MST) by adding the smallest edge connecting a new vertex to the existing structure. With time complexity of O(E log V) when implemented using a priority queue, it demonstrates high efficiency for dense graphs. Prim's algorithm finds widespread application in network design, circuit design, and clustering operations. However, its reliance on priority queue operations makes it less suitable for extremely sparse graphs compared to Kruskal's algorithm.

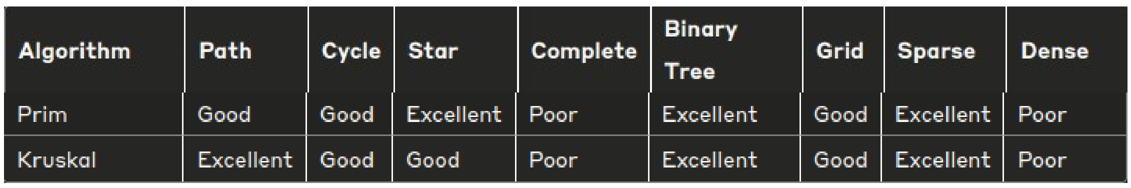
Kruskal's Algorithm Kruskal's algorithm arranges all edges and incorporates the smallest one into the MST while ensuring cycle prevention. Its time complexity of O(E log E) makes it particularly well-suited for sparse graphs. Kruskal's algorithm commonly serves in network connectivity analysis, transportation systems, and image segmentation applications. However, for dense graphs with numerous edges, the sorting operation can become a performance-limiting factor.

Summary Each algorithm presents distinct advantages and trade-offs:

- **Prim's Algorithm**: Most appropriate for dense graphs where adjacency lists and priority queues enable efficient MST construction.
- **Kruskal's Algorithm**: More effective for sparse graphs where initial edge sorting reduces necessary operations.

Future research could explore optimizations combining aspects of both algorithms, such as hybrid approaches for large-scale graphs or parallel computing integration to improve execution speed. Additionally, further empirical studies could investigate how different graph structures and weight distributions affect algorithm performance.

https://github.com/dmracovit/AA/lab3_4_5



Algorithm	Path	Cycle	Star	Complete	Binary Tree	Grid	Sparse	Dense
Prim	Good	Good	Excellent	Poor	Excellent	Good	Excellent	Poor
Kruskal	Excellent	Good	Good	Poor	Excellent	Good	Excellent	Poor