Exploring the Minimum Spanning Tree Problem: Kruskal's vs. Prim's Algorithms

Andrea A. Venti Fuentes

University of Miami

CSC 401 – Computer Science Practicum I

Dr. Dilip Sarkar

December 01, 2024

kruskal_algorithm/Kruskal.py

```
"""
Kruskal's Algorithm Implementation for Minimum Spanning Tree (MST)

This program reads a weighted undirected graph and computes its MST using Kruskal's algorithm.
It uses a Union-Find (Disjoint Set Union) data structure to efficiently detect cycles.

Usage:
    python Kruskal.py [input_file]

Input:
    The program reads the graph from a file or standard input.
    The graph is represented as a list of edges with their weights.

    The input format is:
        n m
        u1 v1 w1
        u2 v2 w2
        ...
        um vm wm

    where:
        - n: number of vertices (vertices are labeled from 0 to n-1)
        - m: number of edges
        - ui vi wi: edge between vertex ui and vi with weight wi

Output:
    The edges in the MST and the total weight of the MST.

Example:
    Input (graph.txt):
        4 5
        0 1 10
        0 2 6
        0 3 5
        1 3 15
        2 3 4

    Command:
        python Kruskal.py graph.txt

    Output:
        Edges in the MST:
        2 - 3: 4.0
        0 - 3: 5.0
        0 - 1: 10.0
        Total weight of MST: 19.0
"""
```

```python

import sys


class UnionFind:
    """
    Disjoint Set Union (Union-Find) data structure implementation.

    Attributes:
        parent (list): Parent of each element in the set.
        rank (list): Rank of each element to keep the tree flat.
    """

    def __init__(self, n):
        """Initialize Union-Find data structure with n elements.

        Args:
            n (int): Number of elements.
        """
        self.parent = [i for i in range(n)]
        self.rank = [0] * n

    def find(self, u):
        """Find the representative of the set that u is a member of.

        Args:
            u (int): Element to find.

        Returns:
            int: Representative of the set.
        """
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])  # Path compression
        return self.parent[u]

    def union(self, u, v):
        """Union the sets that contain u and v.

        Args:
            u (int): First element.
            v (int): Second element.

        Returns:
            bool: True if union was successful, False if u and v are already in the same set.
        """
        u_root = self.find(u)
        v_root = self.find(v)

        if u_root == v_root:
```

```
            return False  # u and v are already in the same set

        # Union by rank to keep tree shallow
        if self.rank[u_root] < self.rank[v_root]:
            self.parent[u_root] = v_root
        elif self.rank[u_root] > self.rank[v_root]:
            self.parent[v_root] = u_root
        else:
            self.parent[v_root] = u_root
            self.rank[u_root] += 1
        return True


def kruskal(n, edges):
    """Compute the MST of a graph using Kruskal's algorithm.

    Args:
        n (int): Number of vertices.
        edges (list): List of tuples (u, v, w) representing edges.

    Returns:
        tuple: A tuple containing:
            - mst_edges (list): List of edges in the MST.
            - total_weight (float): Total weight of the MST.
    """
    # Sort edges in non-decreasing order of weight
    edges.sort(key=lambda x: x[2])

    uf = UnionFind(n)
    mst_edges = []
    total_weight = 0.0

    for u, v, w in edges:
        if uf.union(u, v):
            mst_edges.append((u, v, w))
            total_weight += w

    return mst_edges, total_weight


def main():
    """Main function to read input and compute MST."""
    # Read input from file or standard input
    if len(sys.argv) > 1:
        # Read from file
        with open(sys.argv[1], "r") as f:
            lines = f.readlines()
    else:
        # Read from standard input
        lines = sys.stdin.readlines()
```

```

    n, m = map(int, lines[0].split())
    edges = []
    for line in lines[1:]:
        u, v, w = map(float, line.strip().split())
        edges.append((int(u), int(v), w))

    # Compute MST using Kruskal's algorithm
    mst_edges, total_weight = kruskal(n, edges)

    # Output the result
    print("Edges in the MST:")
    for u, v, w in mst_edges:
        print(f"{u} - {v}: {w}")
    print(f"Total weight of MST: {total_weight}")


if __name__ == "__main__":
    main()

```

kruskal_algorithm/graph.txt

```
4 5
0 1 10
0 2 6
0 3 5
1 3 15
2 3 4
```

prim_algorithm/Prim.py

```
"""
Prim's Algorithm Implementation for Minimum Spanning Tree (MST)

This program reads a weighted undirected graph and computes its MST using Prim's algorithm.
It uses a priority queue to efficiently select the next edge with the minimum weight.

Usage:
    python Prim.py [input_file]

Input:
    The program reads the graph from a file or standard input.
    The graph is represented as an adjacency list with weights.

    The input format is:
        n m
        u1 v1 w1
        u2 v2 w2
        ...
        um vm wm

    where:
        - n: number of vertices (vertices are labeled from 0 to n-1)
        - m: number of edges
        - ui vi wi: edge between vertex ui and vi with weight wi

Output:
    The edges in the MST and the total weight of the MST.

Example:
    Input (graph.txt):
        4 5
        0 1 10
        0 2 6
        0 3 5
        1 3 15
        2 3 4

    Command:
        python Prim.py graph.txt

    Output:
        Edges in the MST:
        0 - 3: 5.0
        3 - 2: 4.0
        0 - 1: 10.0
        Total weight of MST: 19.0
"""
```

```

import heapq
import sys


def prim(n, adj):
    """Compute the MST of a graph using Prim's algorithm.

    Args:
        n (int): Number of vertices.
        adj (list): Adjacency list where adj[u] is a list of tuples (v, w).

    Returns:
        tuple: A tuple containing:
            - mst_edges (list): List of edges in the MST.
            - total_weight (float): Total weight of the MST.
    """
    visited = [False] * n
    min_heap = []

    # Start from vertex 0
    visited[0] = True
    for v, w in adj[0]:
        heapq.heappush(min_heap, (w, 0, v))

    mst_edges = []
    total_weight = 0.0

    while min_heap and len(mst_edges) < n - 1:
        w, u, v = heapq.heappop(min_heap)
        if not visited[v]:
            visited[v] = True
            mst_edges.append((u, v, w))
            total_weight += w
            for to, weight in adj[v]:
                if not visited[to]:
                    heapq.heappush(min_heap, (weight, v, to))

    return mst_edges, total_weight


def main():
    """Main function to read input and compute MST."""
    # Read input from file or standard input
    if len(sys.argv) > 1:
        # Read from file
        with open(sys.argv[1], "r") as f:
            lines = f.readlines()
    else:
        # Read from standard input
```

```python
        lines = sys.stdin.readlines()

    n, m = map(int, lines[0].split())
    adj = [[] for _ in range(n)]
    for line in lines[1:]:
        u, v, w = map(float, line.strip().split())
        u = int(u)
        v = int(v)
        adj[u].append((v, w))
        adj[v].append((u, w))  # Since the graph is undirected

    # Compute MST using Prim's algorithm
    mst_edges, total_weight = prim(n, adj)

    # Output the result
    print("Edges in the MST:")
    for u, v, w in mst_edges:
        print(f"{u} - {v}: {w}")
    print(f"Total weight of MST: {total_weight}")


if __name__ == "__main__":
    main()
```

prim_algorithm/graph.txt

```
4 5
0 1 10
0 2 6
0 3 5
1 3 15
2 3 4
```

Output/Output.txt

```
❯ python Kruskal.py graph.txt
Edges in the MST:
2 - 3: 4.0
0 - 3: 5.0
0 - 1: 10.0
Total weight of MST: 19.0

❯ python Prim.py graph.txt
Edges in the MST:
0 - 3: 5.0
3 - 2: 4.0
0 - 1: 10.0
Total weight of MST: 19.0

❯ python performance_evaluation.py
n=50, density=0.1, Kruskal Time=0.0000s, Prim Time=0.0000s
n=50, density=0.3, Kruskal Time=0.0001s, Prim Time=0.0001s
n=50, density=0.5, Kruskal Time=0.0001s, Prim Time=0.0001s
n=50, density=0.7, Kruskal Time=0.0002s, Prim Time=0.0001s
n=50, density=0.9, Kruskal Time=0.0002s, Prim Time=0.0001s
n=100, density=0.1, Kruskal Time=0.0001s, Prim Time=0.0001s
n=100, density=0.3, Kruskal Time=0.0003s, Prim Time=0.0002s
n=100, density=0.5, Kruskal Time=0.0006s, Prim Time=0.0003s
n=100, density=0.7, Kruskal Time=0.0008s, Prim Time=0.0004s
n=100, density=0.9, Kruskal Time=0.0011s, Prim Time=0.0005s
n=200, density=0.1, Kruskal Time=0.0005s, Prim Time=0.0003s
n=200, density=0.3, Kruskal Time=0.0015s, Prim Time=0.0009s
n=200, density=0.5, Kruskal Time=0.0024s, Prim Time=0.0013s
n=200, density=0.7, Kruskal Time=0.0035s, Prim Time=0.0018s
n=200, density=0.9, Kruskal Time=0.0044s, Prim Time=0.0023s
n=300, density=0.1, Kruskal Time=0.0011s, Prim Time=0.0008s
n=300, density=0.3, Kruskal Time=0.0035s, Prim Time=0.0020s
n=300, density=0.5, Kruskal Time=0.0058s, Prim Time=0.0031s
n=300, density=0.7, Kruskal Time=0.0096s, Prim Time=0.0048s
n=300, density=0.9, Kruskal Time=0.0106s, Prim Time=0.0053s
n=400, density=0.1, Kruskal Time=0.0021s, Prim Time=0.0015s
n=400, density=0.3, Kruskal Time=0.0063s, Prim Time=0.0037s
n=400, density=0.5, Kruskal Time=0.0109s, Prim Time=0.0059s
n=400, density=0.7, Kruskal Time=0.0157s, Prim Time=0.0074s
n=400, density=0.9, Kruskal Time=0.0221s, Prim Time=0.0108s
n=500, density=0.1, Kruskal Time=0.0033s, Prim Time=0.0025s
n=500, density=0.3, Kruskal Time=0.0101s, Prim Time=0.0057s
n=500, density=0.5, Kruskal Time=0.0176s, Prim Time=0.0092s
n=500, density=0.7, Kruskal Time=0.0246s, Prim Time=0.0124s
n=500, density=0.9, Kruskal Time=0.0326s, Prim Time=0.0162s
Results saved to results/performance_results.csv
```

performance_evaluation.py

```python
"""
Performance Evaluation of Kruskal's and Prim's Algorithms

This script generates random graphs of varying sizes and densities to empirically
evaluate and compare the performance (execution time) of Kruskal's and Prim's algorithms.

Usage:
    python performance_evaluation.py

Requirements:
    - Python 3.x
    - matplotlib (for plotting results)
    - NetworkX (for generating random graphs)
    - pandas (for exporting data to CSV)
"""

import random
import time

import matplotlib.pyplot as plt
import networkx as nx
import pandas as pd

from kruskal_algorithm.Kruskal import kruskal
from prim_algorithm.Prim import prim


def generate_random_graph(n, m):
    """Generate a random undirected weighted graph.

    Args:
        n (int): Number of vertices.
        m (int): Number of edges.

    Returns:
        list: List of edges in the format (u, v, w).
        list: Adjacency list for Prim's algorithm.
    """
    G = nx.gnm_random_graph(n, m)
    edges = []
    adj = [[] for _ in range(n)]
    for u, v in G.edges():
        w = random.uniform(1, 100)
        edges.append((u, v, w))
        adj[u].append((v, w))
        adj[v].append((u, w))
    return edges, adj
```

```python


def evaluate_performance():
    """Evaluate and compare the performance of Kruskal's and Prim's algorithms."""
    num_vertices = [50, 100, 200, 300, 400, 500]
    densities = [0.1, 0.3, 0.5, 0.7, 0.9]
    data = []

    for n in num_vertices:
        for density in densities:
            m = int(
                density * n * (n - 1) / 2
            )  # Maximum number of edges in an undirected graph
            edges, adj = generate_random_graph(n, m)

            # Time Kruskal's algorithm
            start_time = time.time()
            kruskal(n, edges.copy())
            kruskal_time = time.time() - start_time

            # Time Prim's algorithm
            start_time = time.time()
            prim(n, adj)
            prim_time = time.time() - start_time

            print(
                f"n={n}, density={density:.1f}, Kruskal Time={kruskal_time:.4f}s, Prim Time={prim_time:.4f}s"
            )

            # Append results to the data list
            data.append(
                {
                    "Vertices": n,
                    "Density": density,
                    "Kruskal_Time": kruskal_time,
                    "Prim_Time": prim_time,
                }
            )

    # Save data to a CSV file
    df = pd.DataFrame(data)
    df.to_csv("results/performance_results.csv", index=False)
    print("Results saved to results/performance_results.csv")

    # Plotting the results
    for n in num_vertices:
        subset = df[df["Vertices"] == n]
        plt.figure(figsize=(10, 6))
        plt.plot(
```

```
            subset["Density"],
            subset["Kruskal_Time"],
            label="Kruskal's Algorithm",
            marker="o",
        )
        plt.plot(
            subset["Density"], subset["Prim_Time"], label="Prim's Algorithm", marker="s"
        )
        plt.title(f"Performance Comparison for n={n}")
        plt.xlabel("Density")
        plt.ylabel("Execution Time (seconds)")
        plt.legend()
        plt.grid(True)
        plt.savefig(f"results/performance_n_{n}.png")
        plt.close()


if __name__ == "__main__":
    evaluate_performance()

```

README.md

User Manual for Minimum Spanning Tree Algorithms

- User Manual for Minimum Spanning Tree Algorithms
  - Kruskal's Algorithm
    - Kruskal's Algorithm Usage
    - Kruskal's Algorithm Input Format
      - Kruskal's Algorithm Example:
      - Kruskal's Algorithm Sample Run:
  - Prim's Algorithm
    - Prim's Algorithm Usage
    - Prim's Algorithm Input Format
      - Prim's Algorithm Example
      - Prim's Algorithm Sample Run:
  - Performance Evaluation Script
    - Functionality Overview
    - How to Run the Script

This user manual provides instructions on how to use the implementations of Kruskal's and Prim's algorithms for computing the Minimum Spanning Tree (MST) of an undirected, weighted graph.

Kruskal's Algorithm

This implementation computes the MST of a given undirected, weighted graph using Kruskal's algorithm.

Kruskal's Algorithm Usage

To run the program, use the following command:

```
python Kruskal.py [input_file]
```

- If `input_file` is provided, the program reads the graph from the specified file.
- If `input_file` is not provided, the program reads the graph from standard input.

`Kruskal's Algorithm Input Format`

The input graph should be provided in the following format:

```
n m
u1 v1 w1
u2 v2 w2
...
um vm wm
```

- `n`: Number of vertices (vertices are labeled from `0` to `n-1`)
- `m`: Number of edges
- `ui vi wi`: Edge between vertex `ui` and `vi` with weight `wi`

Output:

- A list of edges included in the MST.
- The total weight of the MST.

Kruskal's Algorithm Example:

Input file `graph.txt`:

```
4 5
0 1 10
0 2 6
0 3 5
1 3 15
2 3 4
```

Kruskal's Algorithm Sample Run:

```
python Kruskal.py graph.txt
```

Output:

```
Edges in the MST:
2 - 3: 4.0
0 - 3: 5.0
0 - 1: 10.0
Total weight of MST: 19.0
```

Prim's Algorithm

This implementation computes the MST of a given undirected, weighted graph using Prim's algorithm.

`Prim's Algorithm Usage`

To run the program, use the following command:

```
python Prim.py [input_file]
```

- If `input_file` is provided, the program reads the graph from the specified file.
- If `input_file` is not provided, the program reads the graph from standard input.

`Prim's Algorithm Input Format`

The input graph should be provided in the following format:

```
n m
u1 v1 w1
u2 v2 w2
...
um vm wm
```

- `n`: Number of vertices (vertices are labeled from `0` to `n-1`)
- `m`: Number of edges
- `ui vi wi`: Edge between vertex `ui` and `vi` with weight `wi`

Output:

- A list of edges included in the MST.
- The total weight of the MST.

Prim's Algorithm Example

Input file `graph.txt`:

```
4 5
0 1 10
0 2 6
0 3 5
1 3 15
2 3 4
```

Prim's Algorithm Sample Run:

```
python Prim.py graph.txt
```

Output:

```
Edges in the MST:
0 - 3: 5.0
3 - 2: 4.0
0 - 1: 10.0
Total weight of MST: 19.0
```

Note: The order of edges may differ from Kruskal's algorithm, but the total weight of the MST should be the same.

Feel free to run the programs with your own input files or modify the existing ones to test different graphs. The code is thoroughly documented to help you understand each step of the algorithms.

Performance Evaluation Script

The performance_evaluation.py script is designed to assess and compare the execution time of Kruskal's and Prim's algorithms for constructing a Minimum Spanning Tree (MST) on random graphs of varying sizes and densities. This empirical evaluation helps understand the computational efficiency of both algorithms under different graph configurations.

Functionality Overview

1. **Graph Generation**:
   The script generates random undirected, weighted graphs with varying numbers of vertices and edge densities using the `NetworkX` library.

2. **Performance Timing**:
   The execution times of Kruskal's and Prim's algorithms are measured for each generated graph using Python's `time` module.

3. **Result Storage**:
   Execution times are stored in a CSV file (`results/performance_results.csv`) for further analysis.

4. **Visualization**:
   The script plots the performance results using `matplotlib`, creating separate graphs for different graph sizes. These plots show how execution times vary with graph density for each algorithm.

5. **Output Files**:

   - CSV file with performance data.
   - PNG files with performance graphs (one for each graph size).

How to Run the Script

1. **Dependencies**:

   - Ensure the following Python libraries are installed:

     ```
     pip install matplotlib networkx pandas
     ```

2. **Execution**:

   - Run the script from the command line:

     ```
     python performance_evaluation.py
     ```

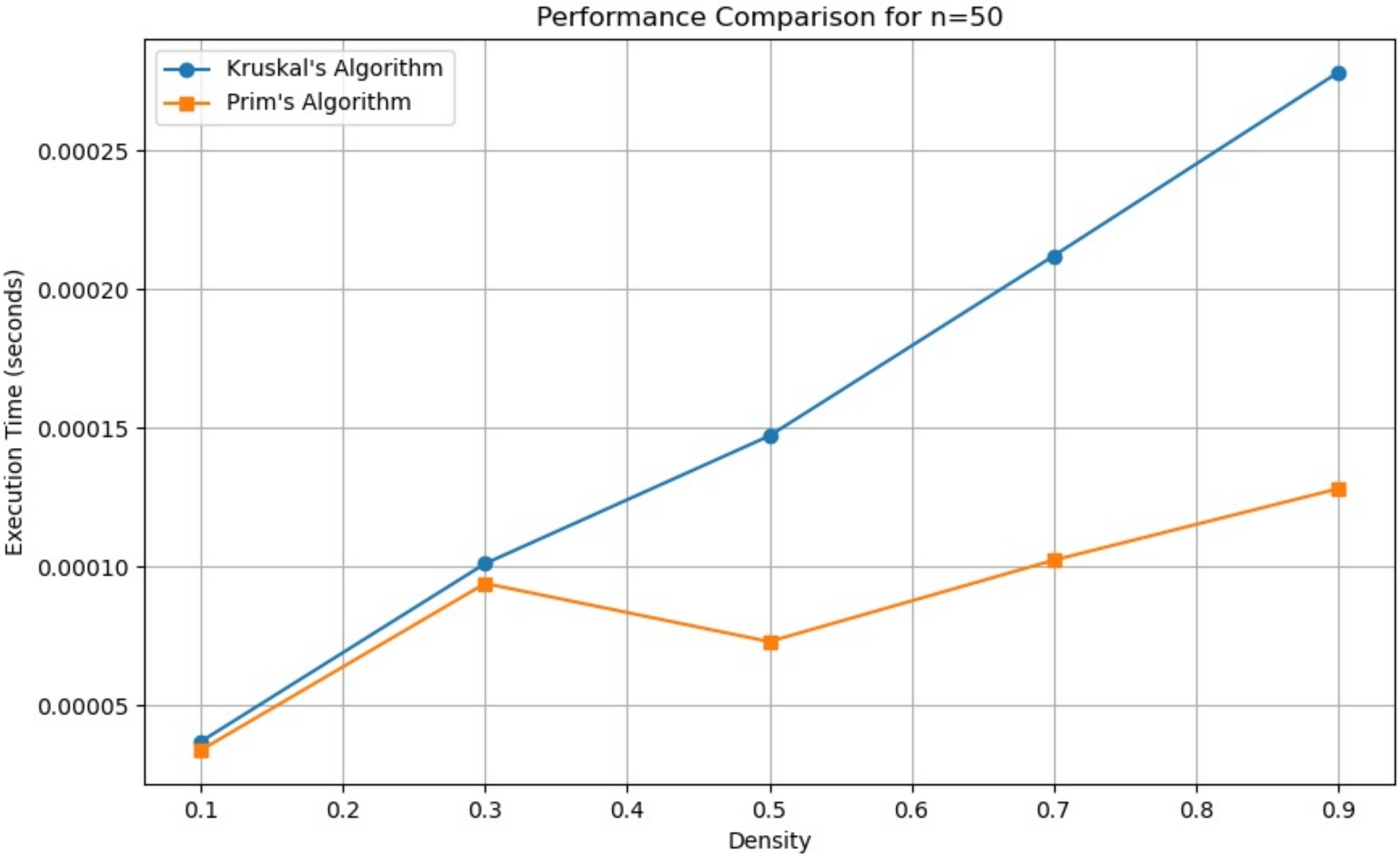
3. **Output Files**:

- Performance results are saved in the results directory as:
  - CSV file: `performance_results.csv`
  - Graph images: `performance_n_50.png`, `performance_n_100.png`, etc.

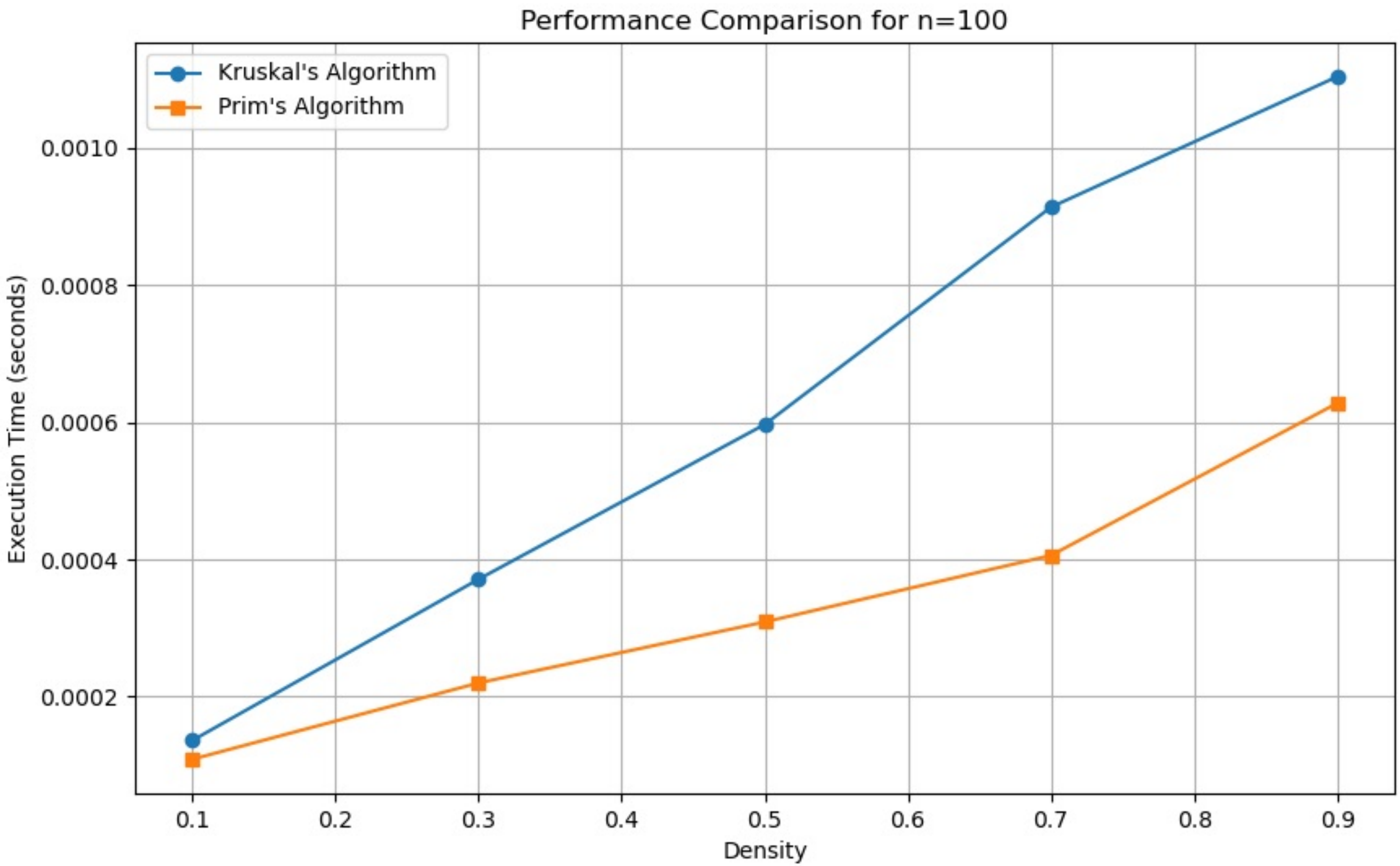
3. **Interpretation of Results**:
   - Open the CSV file to view the recorded execution times for different graph sizes and densities.
   - View the PNG files to analyze the execution time trends for Kruskal's and Prim's algorithms.

results/performance_results.csv

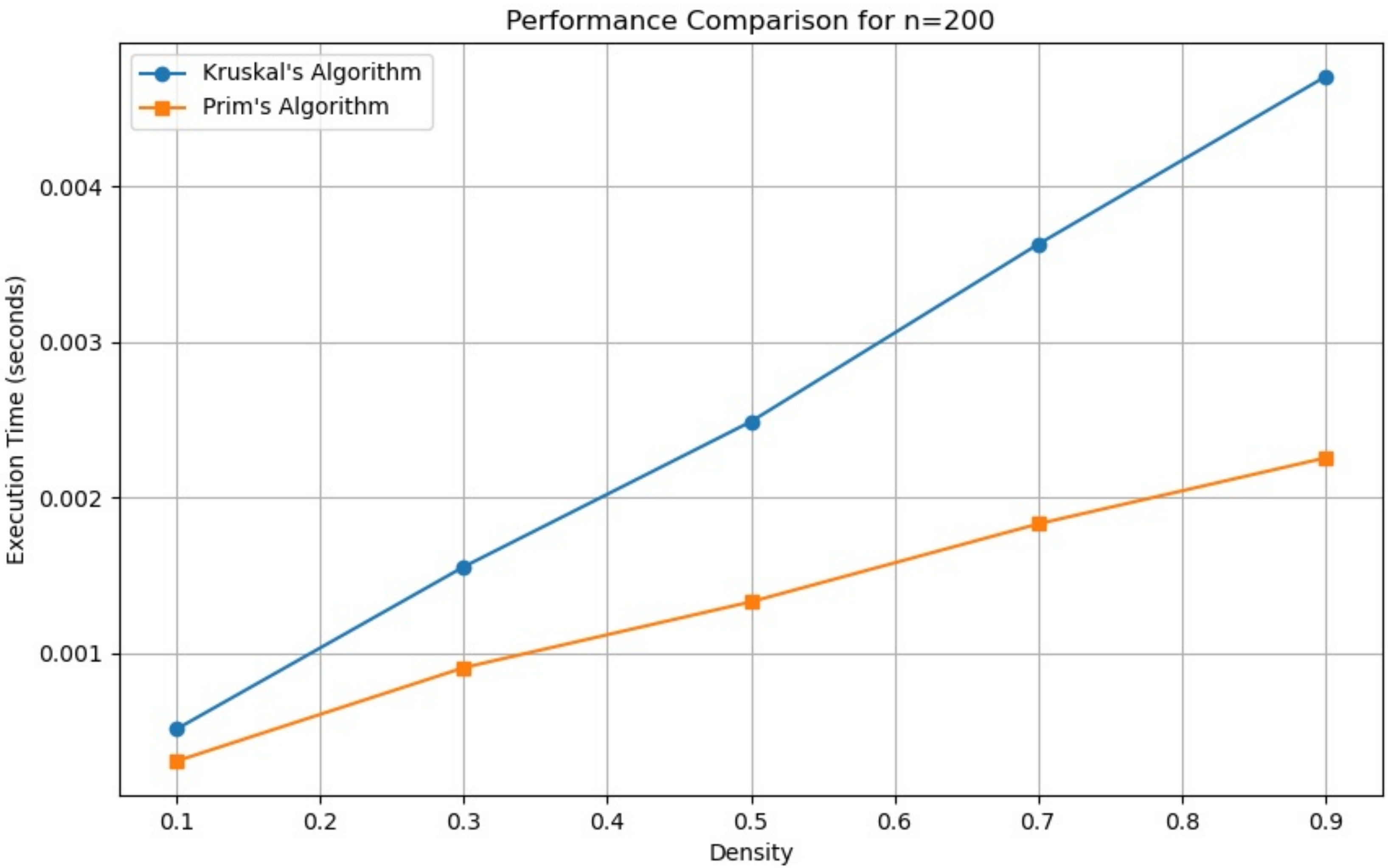
```
Vertices,Density,Kruskal_Time,Prim_Time
50,0.1,3.695487976074219e-05,3.3855438232421875e-05
50,0.3,0.0001010894775390625,9.393692016601562e-05
50,0.5,0.00014710426330566406,7.295608520507812e-05
50,0.7,0.00021195411682128906,0.00010228157043457031
50,0.9,0.0002779960632324219,0.00012803077697753906
100,0.1,0.0001361370086669922,0.0001087188720703125
100,0.3,0.0003712177276611328,0.0002200603485107422
100,0.5,0.0005970001220703125,0.000308990478515625
100,0.7,0.0009138584136962891,0.00040602684020996094
100,0.9,0.001104116439819336,0.0006279945373535156
200,0.1,0.0005140304565429688,0.00030803680419921875
200,0.3,0.001555919647216797,0.00090789794921875
200,0.5,0.0024881362915039062,0.0013308525085449219
200,0.7,0.0036270618438720703,0.0018310546875
200,0.9,0.004703044891357422,0.002254962921142578
300,0.1,0.0011968612670898438,0.0007979869842529297
300,0.3,0.003573894500732422,0.002081155776977539
300,0.5,0.005794048309326172,0.0029900074005126953
300,0.7,0.008552312850952148,0.0045588016510009766
300,0.9,0.01057291030883789,0.005097150802612305
400,0.1,0.002135038375854492,0.0015420913696289062
400,0.3,0.006395101547241211,0.0037398338317871094
400,0.5,0.010862112045288086,0.006065845489501953
400,0.7,0.015707731246948242,0.007939815521240234
400,0.9,0.021106958389282227,0.010153055191040039
500,0.1,0.003493070602416992,0.002363920211791992
500,0.3,0.012423992156982422,0.006505012512207031
500,0.5,0.018088817596435547,0.009444952011108398
500,0.7,0.0261538028717041,0.012809991836547852
500,0.9,0.033499717712402344,0.015898942947387695
```



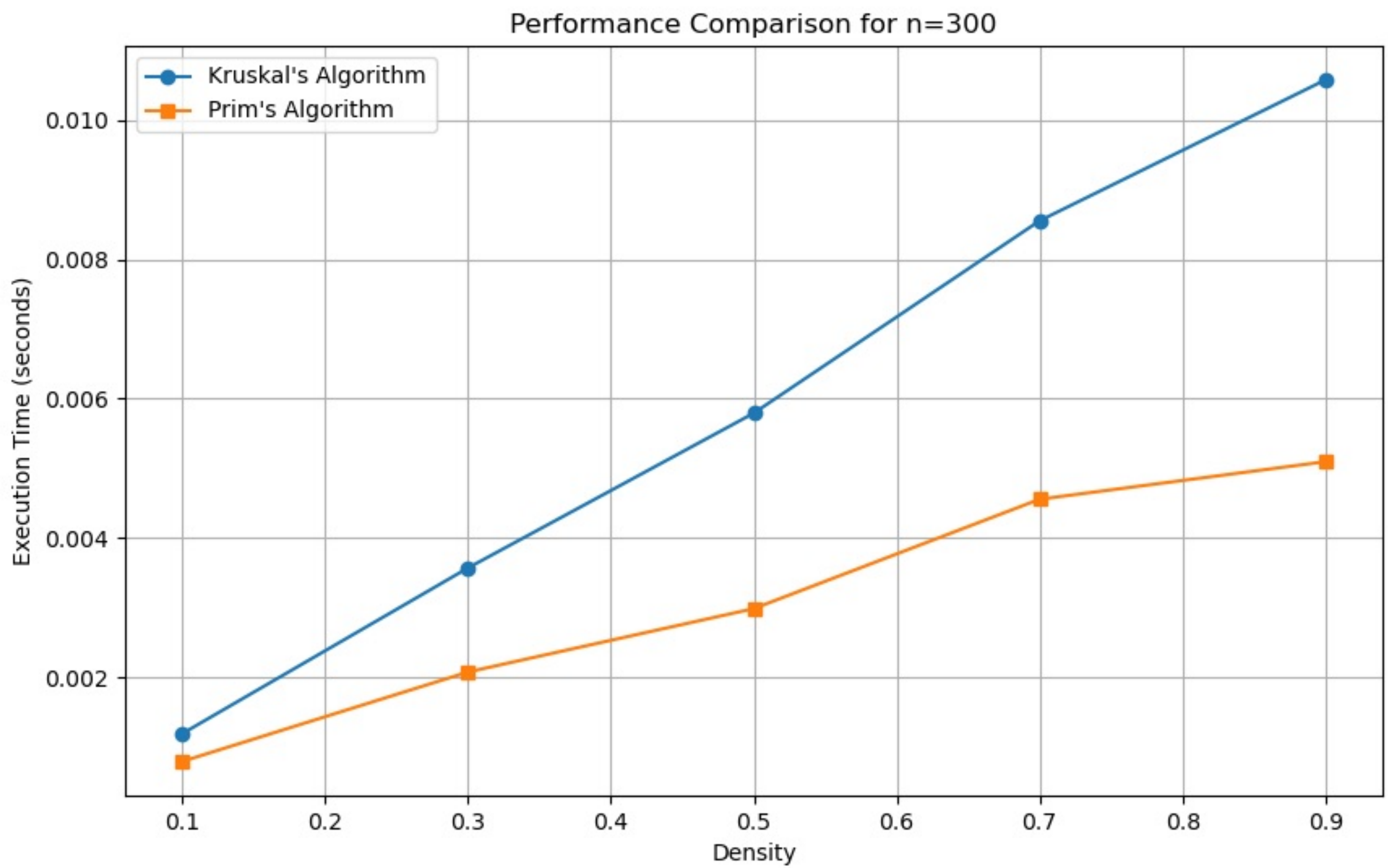
Performance Comparison for n=50
Kruskal's Algorithm
Prim's Algorithm
0.00025
0.00020
0.00015
0.00010
0.00005
Execution Time (seconds)
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Density



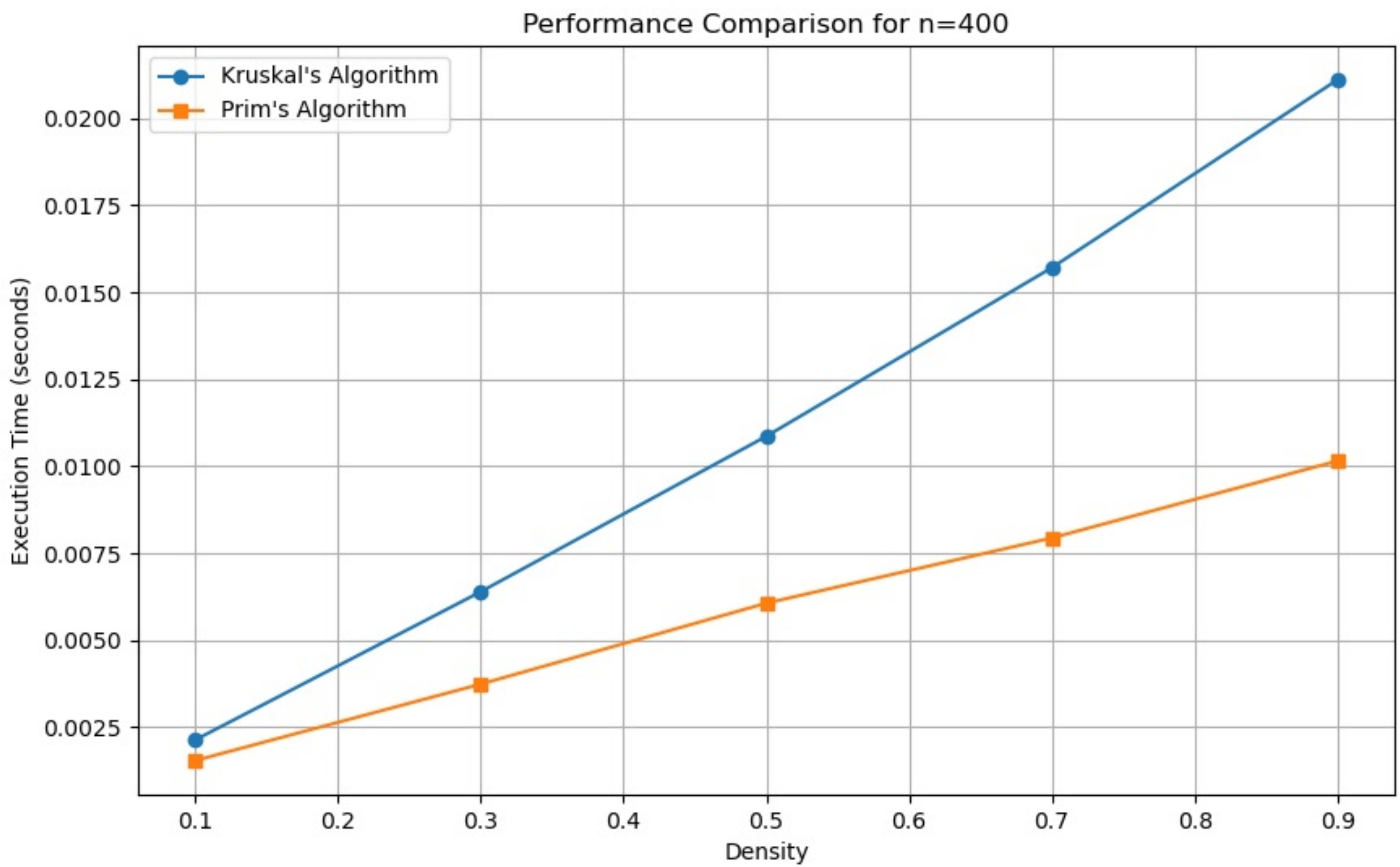
Performance Comparison for n=100
Kruskal's Algorithm
Prim's Algorithm
Execution Time (seconds)
0.0002
0.0004
0.0006
0.0008
0.0010
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Density

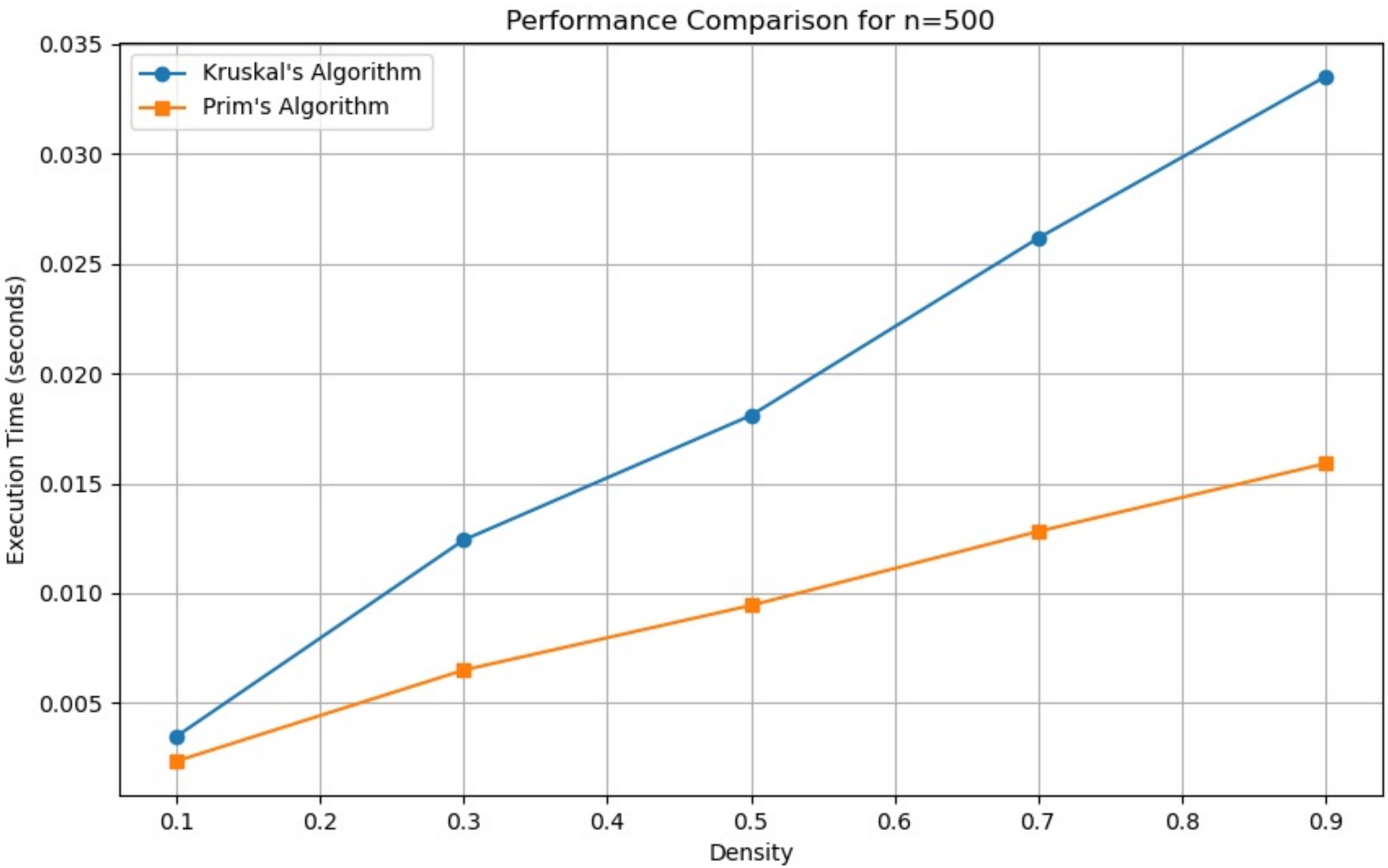


Performance Comparison for n=200
Kruskal's Algorithm
Prim's Algorithm
Execution Time (seconds)
0.001
0.002
0.003
0.004
Density
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9



Performance Comparison for n=300
Kruskal's Algorithm
Prim's Algorithm
Execution Time (seconds)
0.002
0.004
0.006
0.008
0.010
Density
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9


Performance Comparison for n=400
Kruskal's Algorithm
Prim's Algorithm
Execution Time (seconds)
Density
0.0025
0.0050
0.0075
0.0100
0.0125
0.0150
0.0175
0.0200
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9


Performance Comparison for n=500
Kruskal's Algorithm
Prim's Algorithm
Execution Time (seconds)
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Density