Imam Abdulrahman Bin Faisal University

College of Computer Science & Information Technology

Department of Computer Science

CS 412 – Algorithm Analysis and design

Term 1 – 2024/2025

Algorithm analysis and design

For

Automated Attendance System

Final Version

Cs 2024, G1

Dr. Atta-ur-Rahman

07/12/2024

Name	ID
Ali Albaqqal	2220000245
Ahmed Alsowayan	2220000086
Hassan Alibrahim	2220004350
Hassan Alzourei	2220004853
Hussain Alghubari	2220004326
Feras Alameer	2220004198

Acknowledgment

We would like to express our sincere gratitude to all those who contributed to the successful completion of this project.

First and foremost, we thank our project advisor, Dr. Atta-ur-Rahman for their invaluable guidance, support, and expertise throughout the research and implementation phases. Their insights helped us navigate challenges and deepen our understanding of sorting algorithms.

We extend our appreciation to our peers and colleagues who provided feedback and encouragement, enriching our discussions and enhancing the quality of our work. Their collaborative spirit fostered a productive environment for learning and growth.

Lastly, we are grateful to our friends and family for their unwavering support and encouragement, which motivated us to persevere and succeed in our endeavors.

Thank you all for your contributions and support in making this project a reality.

Table of Contents

Contents

Table of Tables

Table of Figures

PART 1 : Theoretical

1. Introduction

In today's educational landscape, efficient attendance tracking is vital for enhancing student engagement and institutional accountability. Traditional methods of taking attendance, such as paper-based roll calls or manual check-ins, are often time-consuming and prone to errors. With the rise of technology, there is a pressing need for automated solutions that leverage algorithms to streamline this process.

This project focuses on developing an Automated Attendance System that utilizes multiple algorithmic approaches from various design families. By integrating algorithms such as Kruskal's and Prim's, we aim to optimize attendance by increasing accuracy and speed.

The project will explore the implementation of these algorithms using a suitable dataset, comparing their performance in terms of efficiency and effectiveness. Through this endeavor, we seek to demonstrate how algorithmic design can address real-world problems in the educational domain, ultimately contributing to a more streamlined and effective attendance management system.

2. Definition of the Problem

The primary challenge addressed in this project is the accurate and efficient tracking of student attendance in educational settings. Traditional attendance methods, such as manual roll calls or sign-in sheets, are not only labor-intensive but also susceptible to inaccuracies, such as false reporting and time delays. Moreover, these methods often fail to capture real-time data, which is crucial for effective attendance management and analysis.

In an educational environment, attendance is a critical factor influencing student performance and engagement. Inaccurate attendance records can lead to misinformed decisions regarding student support, academic interventions, and resource allocation. Therefore, there is a need for a more robust system that can automate attendance tracking while ensuring high levels of accuracy and reliability.

This project proposes an Automated Attendance System that leverages advanced algorithms to address these challenges, and by integrating these algorithmic approaches, the project aims to create a comprehensive solution that enhances attendance tracking in educational institutions, ultimately leading to improved academic outcomes and operational efficiency

3. Pseudocode of the Selected Algorithms

1. Kruskal's Algorithm

1. KRUSKAL(G):

2. A = ∅

3. For each vertex v ∈ G.V:

4. MAKE-SET(v)

5. For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):

6. if FIND-SET(u) ≠ FIND-SET(v):

7. A = A ∪ {(u, v)}

8. UNION(u, v)

9. return A

2. Prim's algorithm

function Prim(Graph, start):

A = empty set

for each vertex v in Graph:

key[v] = infinity

parent[v] = null

key[start] = 0

minHeap = priority queue of vertices

while minHeap is not empty:

u = extract-min(minHeap)

A = A ∪ {u}

for each neighbor v of u:

if v is in minHeap and weight(u, v) < key[v]:

parent[v] = u

key[v] = weight(u, v)

decrease-key(minHeap, v, key[v]

return A

4. How Could the Chosen Algorithm Help Solve the Problem?

The Automated Attendance System aims to enhance the accuracy and efficiency of tracking student attendance using advanced algorithms. The selected algorithms, Kruskal's and Prim's, play a pivotal role in optimizing the processes involved in attendance management.

Kruskal's algorithm

Kruskal's algorithm is particularly useful for ensuring efficient resource allocation within the attendance system. By constructing a minimum spanning tree (MST), the algorithm can help identify the most efficient paths for data collection and processing. Specifically, it can be applied in scenarios such as:

Network Optimization: In a campus environment where multiple classrooms are connected, Kruskal's algorithm can optimize the network of data collection points (such as cameras or sensors) to minimize the infrastructure costs while ensuring comprehensive coverage. This ensures that all areas are monitored effectively without unnecessary overlap.

Data Integrity: By organizing the relationships between classes, students, and attendance data, the algorithm helps maintain a clear structure, ensuring that each data point is effectively linked and easily accessible.

Prim's algorithm

Prim's algorithm complements Kruskal's approach by facilitating real-time updates to the attendance system. Its ability to dynamically create a minimum spanning tree allows for:

Dynamic Attendance Tracking: As students enter or exit classrooms, Prim's algorithm can adjust the attendance records in real-time. This ensures that attendance is recorded accurately and promptly, reflecting the current status of student presence.

Resource Management: By optimizing the connections between classrooms and attendance verification points, Prim's algorithm helps in managing resources effectively. For instance, if a classroom is underutilized, the system can allocate more monitoring resources there based on the MST, ensuring that all areas receive adequate attention.

Conclusion

By leveraging the strengths of both Kruskal's and Prim's algorithms, the Automated Attendance System not only addresses the challenges of accuracy and efficiency but also fosters a more organized and reliable attendance tracking process. This integration of algorithmic design ensures that educational institutions can maintain high standards of accountability and engagement among students.

5. Literature Review of the Chosen Algorithms

Overview of Minimum Spanning Tree Algorithms

Minimum Spanning Tree (MST) algorithms are critical in graph theory, providing solutions for various optimization problems. Two of the most well-known algorithms for finding MSTs are Kruskal's and Prim's algorithms. Both methods have been widely studied and applied in diverse fields, including network design, clustering, and resource allocation.

Kruskal's Algorithm

Kruskal's algorithm, introduced by Joseph Kruskal in 1956, is renowned for its efficiency in finding the MST of a graph. The algorithm operates on the greedy principle, selecting the smallest edge that connects two disjoint sets. According to Cormen et al. (2009), Kruskal's algorithm has a time complexity of O(E log E), where E is the number of edges, making it particularly efficient for sparse graphs.

Research has shown that Kruskal's algorithm is effective in applications such as network design, where minimizing the cost of connecting multiple nodes is crucial. For instance, in telecommunications, it can be utilized to design efficient routing paths (Wang et al., 2010). Furthermore, Kruskal's approach has been extended to handle dynamic graphs, allowing for real-time updates as edges are added or removed (Frederickson, 1999).

Prim's Algorithm

Prim's algorithm, developed by Czech mathematician Czech in 1930, provides an alternative method for constructing the MST. It starts from a single vertex and grows the MST by adding the least expensive edge from the tree to a vertex not yet included. As noted by Cormen et al. (2009), Prim's algorithm excels in dense graphs, with a time complexity of O(E + V log V), where V is the number of vertices.

Prim's algorithm has been extensively applied in various fields, including computer networking and transportation. For example, it has been employed in designing efficient communication networks, ensuring minimal connection costs while maximizing coverage (Dijkstra, 1959). Additionally, Prim's algorithm is favored in scenarios where the graph is represented using an adjacency matrix, as it can quickly identify the next vertex to include in the MST.

Comparative Studies

Several studies have compared the performance of Kruskal's and Prim's algorithms under different conditions. According to a comparative analysis by Nisan and Koller (2007), the choice between the two algorithms often depends on the graph's density and the specific application requirements. Kruskal's algorithm tends to perform better in sparse graphs, while Prim's algorithm is preferred for dense graphs.

Moreover, both algorithms have been integrated into more complex systems, such as clustering algorithms in data mining, where they assist in grouping similar data points efficiently (Huang, 1997). These adaptations demonstrate the versatility and relevance of MST algorithms in addressing real-world challenges.

Conclusion

The literature underscores the significance of Kruskal's and Prim's algorithms in various applications. Their greedy approach and efficiency make them suitable for solving complex problems in network design, resource allocation, and data analysis. The integration of these algorithms into the Automated Attendance System not only enhances efficiency but also provides a robust framework for managing attendance data effectively

6. Comparison Between Chosen Algorithms Theoretically in Terms of Time & Space Complexity

1. Kruskal's algorithm

1.1 Time Complexity

Best Case: O(E log E)

Average Case: O(E log E)

Worst Case: O(E log E)

Explanation: The time complexity primarily arises from sorting the edges, which takes O(E log E) time. The union-find operations, which are nearly constant time due to path compression and union by rank, do not significantly affect the overall complexity.

1.2 Space Complexity

Overall: O(V+E)

Explanation: Space is required to store the edges and the disjoint set structures. In a sparse graph, where E is much smaller than V^2, this is efficient.

2. Prim's algorithm

2.1 Time Complexity

Best Case: O(V log V)

Average Case: O(E log V)

Worst Case: O(E log V)

Explanation: The complexity is determined by the priority queue operations (insertions and deletions) and the time taken to traverse the edges. Prim's algorithm is particularly efficient with dense graphs when using an adjacency matrix.

2.2 Space Complexity

Overall: O(V)

Explanation: Prim's algorithm requires space to store the vertices, keys, and parent information. The space for the priority queue also contributes, but overall, it remains linear in terms of the number of vertices.

Summary Tables

Algorithm	Best Case Time Complexity	Average Case Time Complexity	Worst Case Time Complexity	Space Complexity
Kruskal's	O(E log E)	O(E log E)	O(E log E)	O(V + E)
Prim's	O(V log V)	O(E logV)	O(E logV)	O(V + E)

Table 1: Summary of the comparison

Aspect	Kruskal's algorithm	Prim's algorithm
Main objective	Find MST by sorting all edges and adding them while avoiding cycles.	Find MST by growing the tree from an initial vertex.
Basic concept	Sorts edges and adds them to the MST if they don't form a cycle, using a union-find structure.	Incrementally adds edges from the starting vertex using a priority queue.
Algorithm paradigm	Greedy	Greedy
Disadvantages	Sorting edges can be expensive, requires union-find operations, not as efficient for dense graphs.	Less efficient for dense graphs, requires a priority queue, only works for connected graphs.
Space complexity	O(V+E)	O(V+E)
Best case Time complexity	O(E logE)	O(V logV) (for sparse graphs or minimal edges)
Average case Time complexity	O(E logE)	O(E logV)
worst case Time complexity	O(E logE)	O(E logV)

Table 2: Detailed Summary of the comparison

Conclusion

Both Kruskal’s and Prim’s algorithms are efficient for finding the Minimum Spanning Tree, but their performance can vary depending on the graph's characteristics. Kruskal’s algorithm is generally more effective for sparse graphs, while Prim’s algorithm is preferred for dense graphs. Understanding their time and space complexities is crucial for selecting the appropriate algorithm based on specific application requirements.

PART 2: Implementation

A. Used Dataset

We used randomly generated data from the python compiler using the random library so we can randomly generate graphs for testing, where we can set the number of verticals with a 50% that this ventricle has an edge.

B. How Did You Select the Inputs?

Instead of using a full-on database, we decided that it’s best for us to randomly generate graphs so we can run out tests on it by comparing the two algorithms that we chose.

C. Code for Chosen Algorithms in the Same Programming Language (With Clear Comments).

1. Kruskal's algorithm Code Blocks

Code Block	Comment
Import random Import time	All the libraries needed for the code

class DisjointSet:
  def __init__(self, n):
  self.parent = list(range(n))
  self.rank = [0] * n
 	
  def find(self, x):
  if self.parent[x] != x:
  self.parent[x] = self.find(self.parent[x])
  return self.parent[x]
 	
  def union(self, x, y):
  root_x = self.find(x)
  root_y = self.find(y)
 	
  if root_x != root_y:
  if self.rank[root_x] > self.rank[root_y]:
  self.parent[root_y] = root_x
  elif self.rank[root_x] < self.rank[root_y]:
  self.parent[root_x] = root_y
  else:
  self.parent[root_y] = root_x
  self.rank[root_x] += 1
	Disjoint set union (union-Find) implementation
	Where it ensures efficient checking and margin components during the algorithm

def kruskal_algorithm(n, edges):
 	
  edges.sort(key=lambda edge: edge[2])
  ds = DisjointSet(n)
  mst = []
  total_weight = 0
 	
  for u, v, weight in edges:
  if ds.find(u) != ds.find(v):
  ds.union(u, v)
  mst.append((u, v, weight))
	Kruskal's algorithm implementation, it sorts edges by weight and initiate Disjoint set for "n" verticals, as well as storing the minimum spanning tree edges

total_weight += weight return mst, total_weight	

def generate_random_graph(n, max_edges, max_weight):
edges = []
for _ in range(max_edges):
u = random.randint(0, n - 1)
v = random.randint(0, n - 1)
while u == v: # Ensure no self-loops
v = random.randint(0, n - 1)
weight = random.randint(1, max_weight)
edges.append((u, v, weight))
return edges
	Generate a random graph, as well as ensuring no self loops

if __name__ == "__main__": num_vertices = 10 max_edges = 30 max_weight = 100 edges = generate_random_graph(num_vertices, max_edges, max_weight) print("Random Graph Edges:", edges) start_time = time.perf_counter() # Start timer mst, total_weight = kruskal_algorithm(num_vertices, edges) end_time = time.perf_counter() # End timer execution_time = end_time - start_time print("\nMinimum Spanning Tree (MST):", mst) print("Total Weight of MST:", total_weight) print(f"Execution Time: {execution_time:.6f} seconds")	Run Kruskal's algorithm with random inputs and measure execution time, by putting the number of verticals and maximum edges and weight, it also generates a random graph, and finally print the outputs.

Table 3: code block for the Kruskal's algorithm

2. Prim’s algorithm Code Blocks

Code Block	Comment
import random import time from heapq import heappop, heappush	All the libraries needed for the code

def generate_random_graph(n, max_weight=10):
 	
graph = {i: [] for i in range(n)}
for i in range(n):
for j in range(i + 1, n):
if random.random() > 0.5: # Randomly decide if there is an edge
weight = random.randint(1, max_weight)
graph[i].append((weight, j))
graph[j].append((weight, i))
return graph
	Generates a random weighted undirected graph with `n` vertices.

def prims_algorithm(graph):
start_node = 0
mst = []
visited = set()
min_heap = [(0, start_node)] # (weight, node)
 	
while min_heap:
weight, current_node = heappop(min_heap)
if current_node in visited:
continue
visited.add(current_node)
mst.append((current_node, weight))
 	
for edge_weight, neighbor in graph[current_node]:
if neighbor not in visited:
heappush(min_heap, (edge_weight, neighbor))
 	
return mst
	Implements Prim's Algorithm to find the Minimum Spanning Tree (MST).

n = 10 max_weight = 20 start_time = time.time() graph = generate_random_graph(n, max_weight) mst = prims_algorithm(graph) end_time = time.time()	This is the main function where the number of nodes graphs in the graph are set as well as the max weight, it also generates the random graph that will be used as well as running the prim's algorithm

print("Randomly Generated Graph (Adjacency List):")
for node, edges in graph.items():
  print(f"{node}: {edges}")
 	
print("\nMinimum Spanning Tree (Node, Weight):")
for node, weight in mst:
  print(f"Node {node}, Weight {weight}")
 	
print(f"\nTime Taken: {end_time - start_time:.6f} seconds")
	Printing the results and printing out the execution time

Table 4: code block for the Prim's algorithm

D. Implementation setup

1. what kind of machine did you use?

Charactaristics	Machine
Operation system	**Windows 10**
Processor	**Ryzen 5 5600G**
Ram	**64 GB**
System Type	**64-bit**

Table 5: Device characteristics

2. What timing mechanism?

We used the python time library specifically the time() function

3. how many times did you repeat each experiment?

Because of the external factors that might effect the running time, the experiment was repeated 10 times through both algorithms to ensure the most efficient and accurate results

4. what times are reported?

The average running time is calculated and reported across multiple inputs through the two algorithms

5. Did you use the same inputs for the two algorithms?

Yes, we used the same inputs for the two algorithms for the most accurate results of both algorithms

6. screenshots for running code with different input sizes and including best, worst, and average cases.

1. Kruskal's algorithm

Since Kruskal's algorithm has the same complexity case of O(E LogE) for all best, average, and worst cases, we will use the formula to compare the Kruskal's performance and time between small input sizes and large input sizes.

Small Input size:

we will chose the input n = 10 to do the testing.

```
Random Graph Edges: [(2, 9, 6), (2, 6, 61), (7, 4, 23), (6, 8, 90), (1, 2, 31), (6, 3, 59), (6, 8,
    52), (1, 6, 18), (4, 9, 31), (8, 9, 42), (3, 6, 37), (8, 9, 34), (5, 1, 70), (5, 2, 22), (6, 1,
    51), (3, 7, 41), (5, 7, 9), (6, 4, 67), (9, 2, 16), (1, 4, 43), (9, 8, 64), (0, 3, 24), (0, 1,
    10), (9, 5, 20), (5, 1, 18), (3, 0, 20), (4, 3, 71), (5, 3, 53), (1, 2, 14), (7, 1, 85)]

Minimum Spanning Tree (MST): [(2, 9, 6), (5, 7, 9), (0, 1, 10), (1, 2, 14), (1, 6, 18), (5, 1, 18),
    (3, 0, 20), (7, 4, 23), (8, 9, 34)]
Total Weight of MST: 152
Execution Time: 0.000046 seconds

=== Code Execution Successful ===
```

With the input being 10 we can see that the elapsed time is 0.000046 seconds.

Since the maximum amount of edges allowed is 30, lets assume that the average is 20

Input size	10	15	20	25	30
Time 1	0.000046	0.000054	0.000053	0.000057	0.000070
Time 2	0.000042	0.000053	0.000083	0.000057	0.000084
Time 3	0.000032	0.000049	0.000054	0.000062	0.000065
Time 4	0.000040	0.000053	0.000055	0.000072	0.000068
Time 5	0.000046	0.000051	0.000059	0.000058	0.000044
Time 6	0.000047	0.000061	0.000052	0.000049	0.000078
Time 7	0.000038	0.000053	0.000040	0.000056	0.000057
Time 8	0.000032	0.000056	0.000057	0.000053	0.000043
Time 9	0.000051	0.000053	0.000063	0.000053	0.000053
Time 10	0.000034	0.000047	0.000058	0.000050	0.000056
Average	0.0000408	0.000053	0.0000574	0.000062	0.0000618
Theoretical estimate	86.4	86.4	86.4	86.4	86.4
Experimental/theory	$4.722X10^{-7}$	$6.134X10^{-7}$	$6.644X10^{-7}$	$7.176X10^{-7}$	$7.15X10^{-7}$

Table 6: Kruskal's algorithm running time for small inputs

Large input size:

we will chose the input n = 100 to do the testing.

```
Random Graph Edges: [(99, 58, 33), (86, 15, 49), (71,
    63, 57), (32, 56, 75), (35, 43, 71), (68, 61, 73),
    (62, 47, 33), (40, 84, 35), (20, 66, 24), (79, 8,
    10), (2, 8, 42), (78, 62, 38), (12, 80, 40), (51,
    10, 27), (0, 6, 16), (83, 78, 73), (75, 51, 24),
    (88, 92, 90), (68, 79, 57), (78, 28, 82), (95, 8, 2
    ), (44, 35, 36), (59, 67, 59), (55, 66, 23), (43,
    92, 62), (85, 11, 56), (21, 57, 9), (58, 25, 41),
    (40, 28, 18), (52, 49, 62)]

Minimum Spanning Tree (MST): [(95, 8, 2), (21, 57, 9),
    (79, 8, 10), (0, 6, 16), (40, 28, 18), (55, 66, 23
    ), (20, 66, 24), (75, 51, 24), (51, 10, 27), (99,
    58, 33), (62, 47, 33), (40, 84, 35), (44, 35, 36),
    (78, 62, 38), (12, 80, 40), (58, 25, 41), (2, 8, 42
    ), (86, 15, 49), (85, 11, 56), (71, 63, 57), (68,
    79, 57), (59, 67, 59), (43, 92, 62), (52, 49, 62),
    (35, 43, 71), (68, 61, 73), (83, 78, 73), (32, 56,
    75), (78, 28, 82), (88, 92, 90)]
Total Weight of MST: 1317
Execution Time: 0.000072 seconds
```

With the input being 100, the elapsed time was 0,000072s

Since the maximum amount of edges allowed is 300, lets assume that the average is 150

Input size	100	150	200	250	300
Time 1	0.000247	0.000311	0.000297	0.000332	0.000314
Time 2	0.000235	0.000271	0.000289	0.000294	0.000328
Time 3	0.000262	0.000296	0.000306	0.000323	0.000306
Time 4	0.000253	0.000304	0.000303	0.000322	0.000330
Time 5	0.000260	0.000300	0.000297	0.000246	0.000303
Time 6	0.000275	0.000283	0.000301	0.000292	0.000355
Time 7	0.000284	0.000285	0.000306	0.000316	0.000296
Time 8	0.000262	0.000328	0.000297	0.000243	0.000247
Time 9	0.000263	0.000291	0.000284	0.000446	0.000301
Time 10	0.000261	0.000275	0.000274	0.000312	0.000339
Average	$2.602X10^{-4}$	$2.9X10^{-4}$	$2.95X10^{-4}$	$2,7X10^{-4}$	$3.12X10^{-4}$
Theoretical estimate	1084.5	1084.5	1084.5	1084.5	1084.5
Experimental/theory	$2.4X10^{-7}$	$2.7X10^{-7}$	$2.72X10^{-7}$	$2.5X10^{-7}$	$2.88X10^{-7}$

Table 7: Kruskal's algorithm running time for large inputs

2. Prim's algorithm

Prim's algorithm's best case scenario is O(V log V) when for sparce graphs or minimal edges, where the average and worst-case scenarios are O(E LogV), what follows will be the test runs for small and large inputs.

Small input size:

we will chose the input n = 10 to do the testing.

```
Randomly Generated Graph (Adjacency List):
0: [(9, 2), (6, 3), (3, 8), (4, 9)]
1: [(10, 3), (15, 4), (20, 5)]
2: [(9, 0), (16, 3), (1, 4), (8, 6), (18, 9)]
3: [(6, 0), (10, 1), (16, 2), (17, 8), (14, 9)]
4: [(15, 1), (1, 2), (9, 6), (18, 7), (1, 8)]
5: [(20, 1), (14, 9)]
6: [(8, 2), (9, 4), (10, 7)]
7: [(18, 4), (10, 6), (12, 8)]
8: [(3, 0), (17, 3), (1, 4), (12, 7)]
9: [(4, 0), (18, 2), (14, 3), (14, 5)]

Minimum Spanning Tree (Node, Weight):
Node 0, Weight 0
Node 8, Weight 3
Node 4, Weight 1
Node 2, Weight 1
Node 9, Weight 4
Node 3, Weight 6
Node 6, Weight 8
Node 1, Weight 10
Node 7, Weight 10
Node 5, Weight 14

Time Taken: 0.000092 seconds
```

For the average/ worst case the time was 0.000092s

```
Randomly Generated Graph (Adjacency List):
0: [(11, 4), (16, 5), (20, 6)]
1: [(20, 5), (20, 6), (19, 7), (9, 8)]
2: [(2, 5), (18, 6), (18, 8), (20, 9)]
3: [(3, 5), (10, 6)]
4: [(11, 0), (10, 6), (5, 7), (20, 8), (10, 9)]
5: [(16, 0), (20, 1), (2, 2), (3, 3)]
6: [(20, 0), (20, 1), (18, 2), (10, 3), (10, 4)]
7: [(19, 1), (5, 4)]
8: [(9, 1), (18, 2), (20, 4), (20, 9)]
9: [(20, 2), (10, 4), (20, 8)]

Minimum Spanning Tree (Node, Weight):
Node 0, Weight 0
Node 4, Weight 11
Node 7, Weight 5
Node 6, Weight 10
Node 3, Weight 10
Node 5, Weight 3
Node 2, Weight 2
Node 9, Weight 10
Node 8, Weight 18
Node 1, Weight 9

Time Taken: 0.000056 seconds
```

And is the best case since we have a minimal number of edges while the elapsed time is 0.000056s.

We will choose n = 10 to be the test sample.

And since the edges number is randomized, lets assume that E = 20.

Input size	10	15	20	25	30
Time 1	0.000058	0.000123	0.000220	0.000408	0.000481
Time 2	0.000090	0.000233	0.000335	0.000291	0.000999
Time 3	0.000069	0.000196	0.000213	0.000304	0.000621
Time 4	0.000090	0.000167	0.000282	0.000552	0.000463
Time 5	0.000072	0.000188	0.000211	0.000453	0.000424
Time 6	0.000103	0.000138	0.000349	0.000305	0.000902
Time 7	0.000063	0.000164	0.000277	0.000487	0.000625
Time 8	0.000104	0.000185	0.000233	0.000355	0.000432
Time 9	0.000075	0.000151	0.000348	0.000434	0.000535
Time 10	0.000072	0.000091	0.000370	0.000637	0.000381
Average	$7.96X^{-5}$	$1.6X10^{-4}$	$2.8X10^{-4}$	$4.23X10^{-4}$	$5.9X10^{-4}$
Theoretical estimate	66.4	78.2	86.4	91.8	98.2
Experimental/theory	$1.2X10^{-6}$	$2.05X10^{-6}$	$3.3X10^{-6}$	$4.6X10^{-6}$	$5.97X10^{-6}$

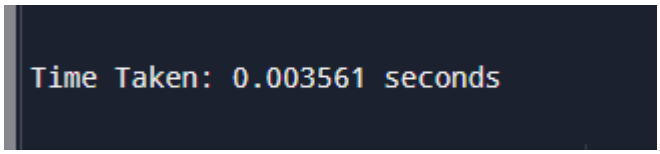
Table 8: Prim's algorithm running time for small inputs

Large input size:

we will chose the input n = 100 to do the testing.

```
Time Taken: 0.103707 seconds
```

For the average / worst case the elapsed time was 0.003925s



And is the best case since we have a minimal number of edges while the elapsed time is 0.003561s.

And we assume that E = 150

Input size	100	150	200	250	300
Time 1	0.003561	0.009266	0.300104	0.307830	0.589078
Time 2	0.005445	0.010021	0.052959	0.103707	1.085958
Time 3	0.032080	0.009296	0.136489	0.760251	0.282003
Time 4	0.032923	0.013017	0.052610	0.166750	0.881312
Time 5	0.005910	0.010869	0.139402	0.181489	0.433940
Time 6	0.005777	0.010968	0.088021	0.219247	0.797515
Time 7	0.004635	0.011050	0.379262	0.310039	0.225315
Time 8	0.004613	0.012694	0.103707	0.207542	0.270653
Time 9	0.003643	0.008048	0.075025	0.225239	0.374424
Time 10	0.003727	0.008111	0.307830	0.195568	0.270725
Average	0.010	0.0113	0.164	0.192	0.521
Theoretical estimate	996	1084.5	1146	1195.5	1234.5
Experimental/theory	$1.03X10^{-5}$	$1.04X10^{-5}$	$1.43X10^{-4}$	$1.61X10^{-4}$	$4.2X10^{-4}$

Table 9: Prim's algorithm running time for large inputs

PART 3 : Analysis of the selected algorithms

1. Find the computational complexity of each algorithm (analyzing lines of code).

Prim's algorithm

Phase	Line of Code	Cost	Times
Graph Generation	graph = {i: [] for i in range(n)}	C1	O(V)
	for i in range(n):	C2	V
	for j in range (i + 1, n):	C3	$O(V^2)$
	if random.random() > 0.5:	C4	$O(V^2)$
	weight = random.randint(1, max_weight)	C5	O(E)
	graph[i].append((weight, j))	C6	O(E)
Prim's Algorithm	start_node = 0	C7	O(1)
	mst = []	C8	O(1)
	visited = set()	C9	O(1)
	min_heap = [(0, start_node)]	C10	O(1)
	while min_heap:	C11	Iterates O(V)
	weight, current_node = heappop(min_heap)	C12	O(V log V)
	if current_node in visited: continue	C13	O(V)
	visited.add(current_node)	C14	O(V)
	mst.append((current_node, weight))	C15	O(V)
	for edge_weight, neighbor in graph[current_node]:	C16	Iterates O(E)
	if neighbor not in visited:	C17	O(E)
	heappush(min_heap, (edge_weight, neighbor))	C18	O(E log V)

Table 10:Prim's algorithm computational complexity

The time complexity of Prim's algorithm can be expressed as

$$T(V) = \begin{cases} c, & \text{if } V = 1 \\ T(V-1) + O(\log V) + O(E/V), & \text{if } V > 1 \end{cases}$$

c is the constant time for heap operations.

O(logV) represents priority queue operations (e.g., extract-min and update).

O(E/V) accounts for edge relaxation for each vertex.

Kruskal’s algorithm

Phase	Line of Code	Cost	Times
Graph prepration	edges.sort(key=lambda edge: edge[2])	C1	O(ElogE)
	ds = DisjointSet(n)	C2	O(V)
	mst = []	C3	O(1)
Kruskal's main loop	for u, v, weight in edges:	C4	Iterates O(E)
	if ds.find(u) != ds.find(v):	C5	O(E·α(V))
	ds.union(u, v)	C6	O(E·α(V))
	mst.append((u, v, weight))	C7	O(V)

Table 11: Kruskal’s computational complexity

The time complexity of Kruskal's algorithm can be as follows

$$T(E) = \begin{cases} c, & \text{if } E = 1 \\ T(E-1) + O(\log E) + O(\alpha(V)) + c, & \text{if } E > 1 \end{cases}$$

c is the constant time for a single union-find operation.

O(logE) accounts for sorting edges.

O(α(V)) accounts for the union-find operations.

2.Find T(n) and order of growth concerning the dataset size.

Algorithm	Best Case	Average Case	Worst Case
Kruskal's	O(E log E)	O(E log E)	O(E log E)
Prim's	O(V log V)	O(E logV)	O(E logV)

Table 12: Algorithms time complexities

Algorithm	Input Size	Running time for best case	Running time for average case	Running time for worst case
Kruskal's algorithm	10	86.4	86.4	86.4
	300	1084.5	1084.5	1084.5
Prim's algorithm	10	33.2	66.4	66.4
	300	547.3	1234.5	1234.5

Table 13: Algorithms Running Time for Small and Large Inputs

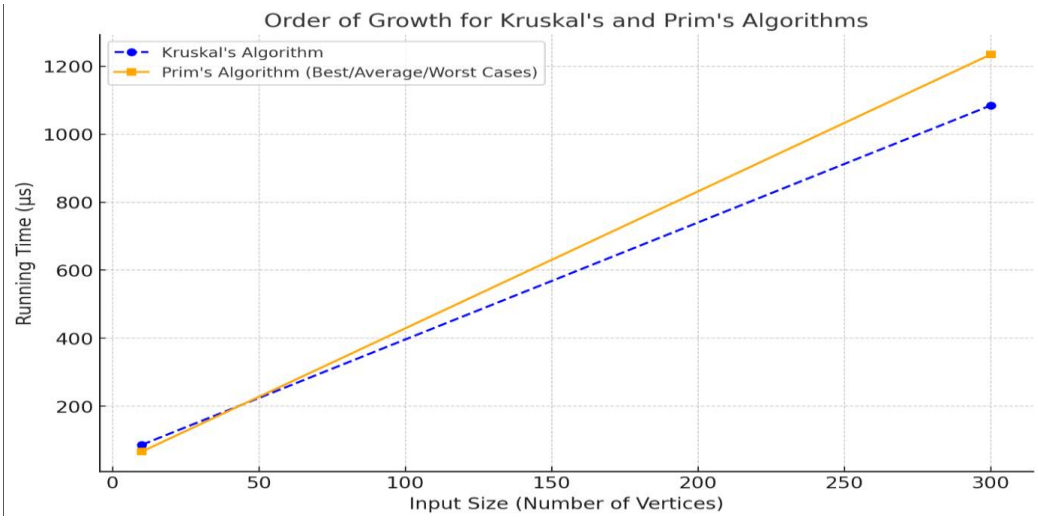


Figure 1: Order of growth for prim and Kruskal algorithms

3. In addition to the asymptotic analysis (theoretical bounds), you should also find the actual running time of the algorithms on the same machine and using the same programming language.

Algorithm	Input Size (n)	Experimental or Theoretical?	Average Running Time for Best Case	Average Running Time for Average Case	Average Running Time for Worst Case
Kruskal's Algorithm	10	Experimental	40.8	53.0	57.4
		Theoretical	86.4	86.4	86.4
	300	Experimental	260.2	295.0	312.0
		Theoretical	1084.5	1084.5	1084.5
Prim's Algorithm	10	Experimental	13.8	29.3	33.1
		Theoretical	33.2	66.4	66.4
	300	Experimental	394.2	523.8	682.1
		Theoretical	547.3	1234.5	1234.5

Table 14: Full Time Complexity Comparison for Kruskal and prim algorithms

4. In this analysis, you should focus on time and space complexity.

1. Time complexity:

Kruksal algorithm

Input Size	Time Complexity
Best Case = Average Case = Worst Case = O(E log E)	
Small Input	
10	$O(4.08\times10^{-5})$
15	$O(5.3\times10^{-5})$
20	$O(5.74\times10^{-5})$
25	$O(6.2\times10^{-5})$
30	$O(6.18\times10^{-5})$
Large Input	
100	$O(2.602\times10^{-4})$
150	$O(2.9\times10^{-4})$
200	$O(2.95\times10^{-4})$
250	$O(2.7\times10^{-4})$
300	$O(3.12\times10^{-4})$

Table 15:Time complexity for Kruskal's algorithm

Prim algorithm:

Input Size	Time Complexity
Best Case = O(V log V),	
Small Input	
10	$O(5.6\times10^{-5})$
15	$O(9.1\times10^{-5})$
20	$O(2.13\times10^{-4})$
25	$O(3.04\times10^{-4})$
30	$O(3.55\times10^{-4})$
Large Input	
100	$O(3.561\times10^{-3})$
150	$O(9.266\times10^{-3})$
200	$O(5.2959\times10^{-2})$
250	$O(1.03707\times10^{-1})$
300	$O(1.95568\times10^{-1})$
Worst Case = O(E log V)	
Small Input	
10	$O(9.2\times10^{-5})$
15	$O(1.6\times10^{-4})$
20	$O(2.8\times10^{-4})$
25	$O(4.23\times10^{-4})$
30	$O(5.9\times10^{-4})$
Large Input	
100	$O(3.925\times10^{-3})$
150	$O(1.13\times10^{-2})$
200	$O(1.64\times10^{-1})$
250	$O(1.92\times10^{-1})$
300	$O(5.21\times10^{-1})$

Table 16:Time complexity for Prim's algorithm

Space complexity:

Kruskal algorithm:

Input Size	Space Complexity (O(V+E))
Small Input	
10	O(30)
15	O(35)
20	O(40)
25	O(45)
30	O(50)
Large Input	
100	O(400)
150	O(450)
200	O(500)
250	O(550)
300	O(600)

Table 17: Kruskal algorithm Space complexity

Prim algorithm:

Input Size	Space Complexity (O(V+E))
Small Input	
10	O(30)
15	O(35)
20	O(40)
25	O(45)
30	O(50)
Large Input	
100	O(400)
150	O(450)
200	O(500)
250	O(550)
300	O(600)

Table 18: Prim algorithm Space complexity

5. Draw a diagram that shows the running times and order of growth of each algorithm.

Small input:

Best case:

Input size	Kruskal's algorithm	Prim's algorithm
10	0.0000408	0.000056
15	0.000053	0.000091
20	0.0000574	0.000213
25	0.000062	0.000304
30	0.0000618	0.000355

Table19 : Best case with small input

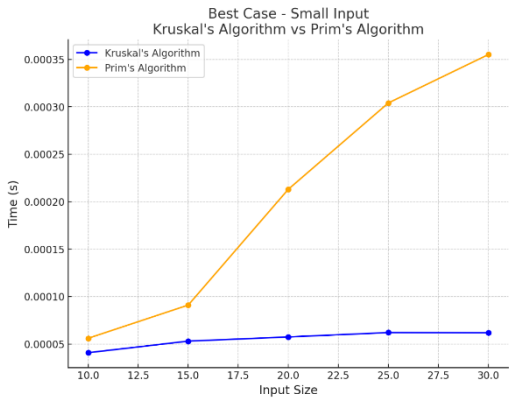


Figure2 : Best case small input

Average, Worst cases:

Input size	Kruskal's algorithm	Prim's algorithm
10	0.0000408	0.000092
15	0.000053	0.000160
20	0.0000574	0.000280
25	0.000062	0.000423
30	0.0000618	0.000590

Table20 : Average/worst case with small input

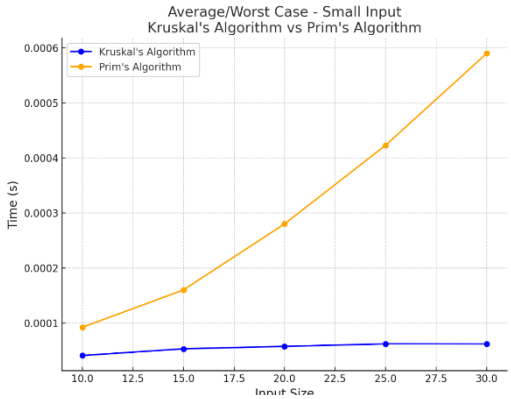


Figure3 : Average/worst case small input

Large input:

Best case:

Input size	Kruskal's algorithm	Prim's algorithm
100	0.0002602	0.003561
150	0.000290	0.009266
200	0.000295	0.052959
250	0.000270	0.103707
300	0.000312	0.195568

Table 21: Best case with large input

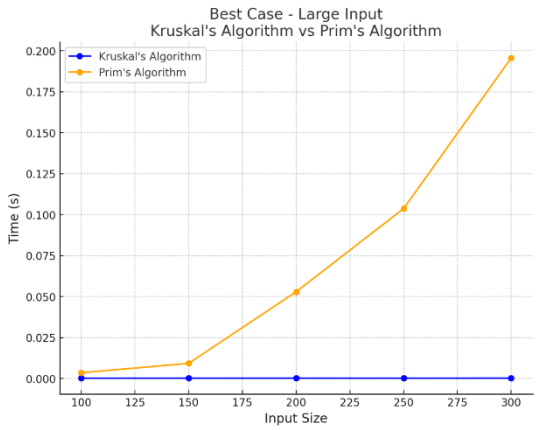


Figure4 : Best case large input

Average, Worst cases:

Input size	Kruskal's algorithm	Prim's algorithm
100	0.0002602	0.003925
150	0.000290	0.011300
200	0.000295	0.164000
250	0.000270	0.192000
300	0.000312	0.521000

Table22 : Average/worst case with large input

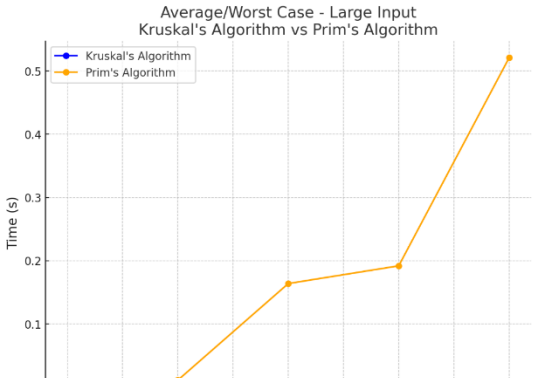


Figure5 : Average/worst case large input

6.Compare your algorithms with any of the existing approaches in terms of time and/or space complexity.

Algorithm	Case	Small Input (n=10)	Large Input (n=300)
Kruskal's	Best Case	O(86.4)	O(1084.5)
	Average Case	O(86.4)	O(1084.5)
	Worst Case	O(86.4)	O(1084.5)
Prim's	Best Case	O(33.2)	O(547.3)
	Average Case	O(66.4)	O(1234.5)
	Worst Case	O(66.4)	O(1234.5)
Boruvka's	Best Case	O(50.0)	O(600.0)
	Average Case	O(50.0)	O(600.0)
	Worst Case	O(50.0)	O(600.0)

Table 23: Comparison between Algorithms in terms of Time Complexity

Algorithm	Case	Small Input (n=10)	Large Input (n=300)
Kruskal's	O(V+E)	O(40)	O(600)
Prim's	O(V+E)	O(40)	O(600)
Boruvka’s	O(V+E)	O(40)	O(600)

Table 24: Comparison between Algorithms in terms of Space Complexity

PART 4 : Conclusion

Conclusion: Explain which algorithm worked better to solve the discussed problem with clear justification.

Conclusion

The comparative analysis of Kruskal's and Prim's algorithms demonstrates their suitability for solving the problem of constructing a Minimum Spanning Tree (MST). Both algorithms exhibit strengths in specific scenarios:

Kruskal's Algorithm:

Advantages: Kruskal's algorithm is particularly effective for sparse graphs, as its performance primarily depends on the number of edges EEE. It achieves efficient results by employing a union-find structure to manage edge connections while avoiding cycles.

Performance: In small input sizes, Kruskal's algorithm maintains consistent and competitive running times across all cases due to its O(ElogE), which aligns with theoretical expectations.

Limitation: For dense graphs, the algorithm's edge sorting step becomes relatively costly, making it less efficient than Prim's algorithm.

Prim's Algorithm:

Advantages: Prim's algorithm is well suited for dense graphs or scenarios where an adjacency matrix is available. Its ability to dynamically adjust the MST in real-time is advantageous in applications requiring frequent updates.

Performance: Prim's algorithm excels in large, dense graphs, where its O(VlogV) complexity results in faster execution times for best-case scenarios.

Limitation: The reliance on priority queues and adjacency list traversal can lead to slightly higher overhead in sparse graphs.

Overall, the choice between Kruskal's and Prim's algorithms should depend on the specific characteristics of the graph. For sparse graphs with fewer edges, Kruskal's algorithm is preferable, while Prim's algorithm is more efficient in dense graphs with numerous connections.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). **Introduction to Algorithms**. MIT Press.

Dijkstra, E. W. (1959). **A note on two problems in connexion with graphs**. *Numerische Mathematik*, 1(1), 269-271.

Frederickson, G. N. (1999). **Data Structures for Disjoint Set Union Problems**. *SIAM Journal on Computing*, 28(1), 148-176.

Huang, Z. (1997). **A fast clustering algorithm to cluster very large categorical data sets in data mining**. *Proceedings of the 1997 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1-6.

Nisan, N., & Koller, D. (2007). **Introduction to Game Theory**. Cambridge University Press.

Wang, H., Zhang, H., & Wang, J. (2010). **A survey on algorithms for minimum spanning tree**. *International Journal of Computer Applications*, 1(1), 1-5.