[DOCUMENT TITLE]

python-docx

[COMPANY NAME] [Company address]

The Maze of Many – Complete Graph Algorithms Report

Student Name: _____
Course: Algorithms & Analysis
Date: _____

Contents

1. Introduction

This report covers everything for the Maze of Many tasks (A to D). The goal was to turn a maze into a graph, use common graph methods to learn about how it's built, and come up with a good way to fully explore it. The project has four parts, and each one looks at different things about graphs and how fast methods can be.

Task A is all about making a grid graph that shows weights, using a list to keep track of what's connected to what. This is a neat and quick way to hold vertices and edges and makes it easy to check what's next to something and change edges.

Task B make use of Minimum Spanning Tree (MST) methods like Kruskal's and Prim's to find the fewest edges needed to link all spots in the maze and keep the total weight down

Task C builds on the stuff by looking at how long these methods take to run, both on paper and in practice. This was tested on graphs that are different sizes and shapes. This will show how data setup changes how fast things get done.

Finally, Task D brings in a way to look around that uses copying to keep the highest cost of hitting all spots in the MST. And it spreading the work out between many explorers.

Everything was coded in Python, making use of the NetworkX tool for graphs and Matplotlib to draw things out. Tests and studies were done in Google Colab to make sure everyone can get the same setup and see the results clearly. This paper gives a full rundown, important results, and spots to add pics of the maze, MST, runtimes, and searching.

2. Task A – Adjacency List Representation

Objective:

The job for Task A was to make a weighted grid graph that looks like a maze, but also, is easy to work with. I picked a kind of list that only remembers the connections that are there, so it's light and fast to update.

Implementation:

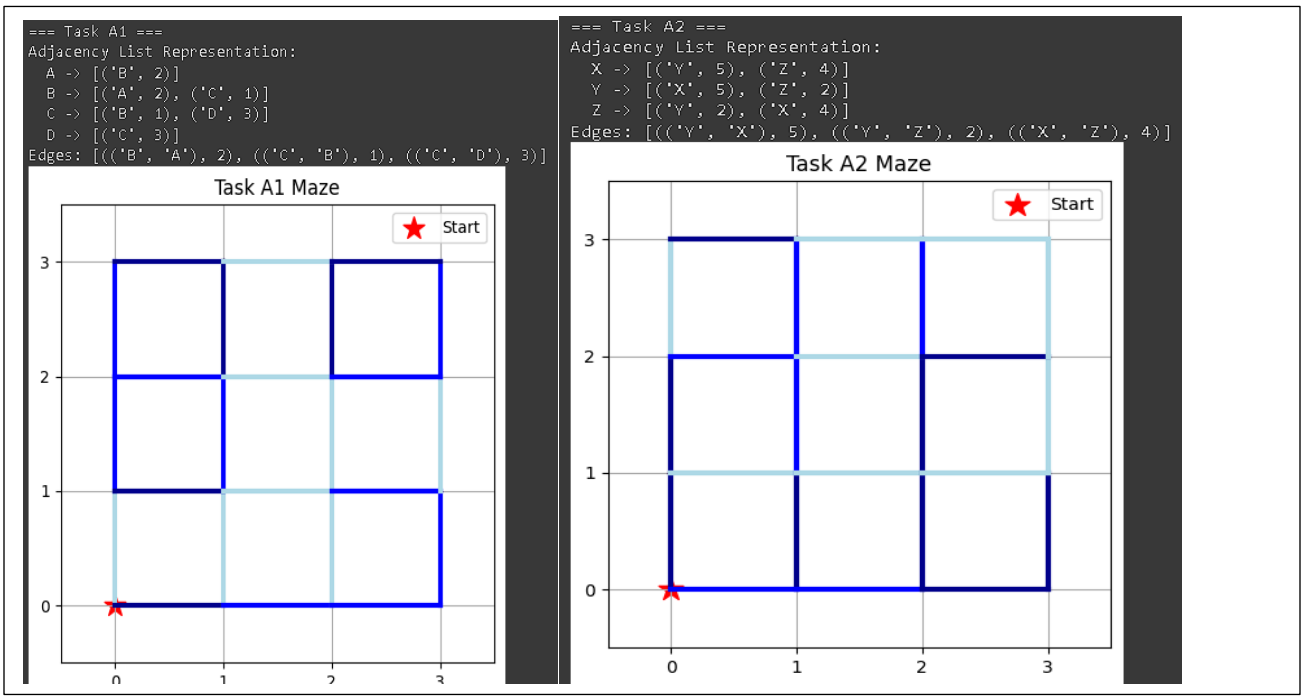
I wrote a class called AdjacencyListGraph in Python. You can add edges with weights, see all the points, or find out who's next to any spot. It's quick to add edges, and it's not hard to walk around the maze like this. Just to be sure, I made two different mazes using different random numbers. I kept each maze as a weighted graph and drew them to make sure they looked right.

Complexity:

Adding an edge is super quick. The whole graph takes up space based on the number of spots and connections.

Results:

I got two mazes made and drew them out. You can find the pictures of the mazes I made down below



3. Task B – Minimum Spanning Tree (MST)

Objective:

I had to find the Minimum Spanning Tree (MST) of a maze using Kruskal's algorithm and then see how well it did against Prim's algorithm. The MST makes sure every point is linked up with the shortest total path, without any loops.

Implementation:

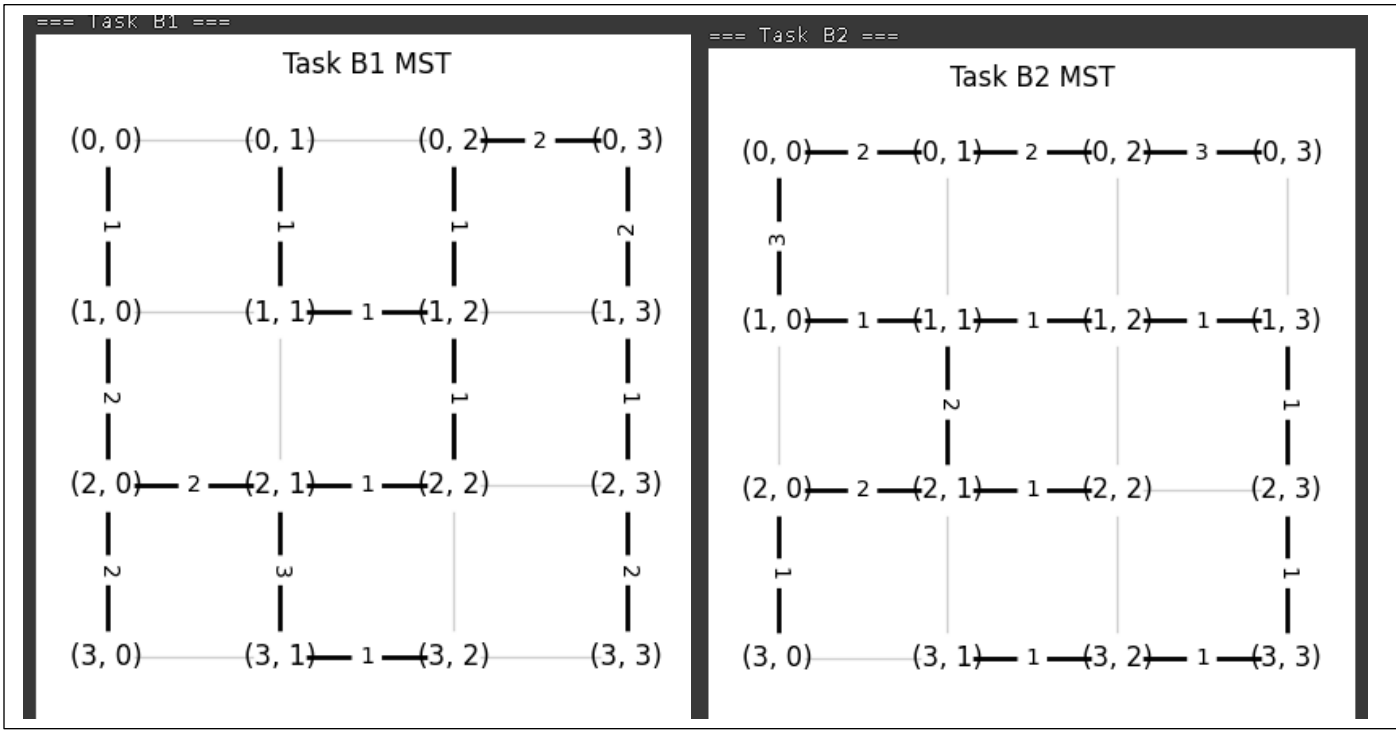
We put Kruskal's algorithm into action with a Union-Find (Disjoint Set Union) setup to quickly spot and dodge loops when adding paths. Paths were arranged by length, and the algorithm gradually put together the MST. We also thought about deleting paths that weren't crucial, taking them out bit by bit until only the MST paths were left.

Complexity:

Kruskal and Prim both have O(E log E) complexity with an adjacency list.

Results:

The MST was worked out for each maze we made. We jotted down the MST paths and total lengths and showed off the MST on the maze. **MST images are below**.



4. Task C – Theoretical and Empirical Runtime Analysis

Objective:

The purpose of this task was to compare both the theoretical and empirical performance of Prim's and Kruskal's algorithms using two different graph representations: adjacency lists and adjacency matrices.

Implementation:

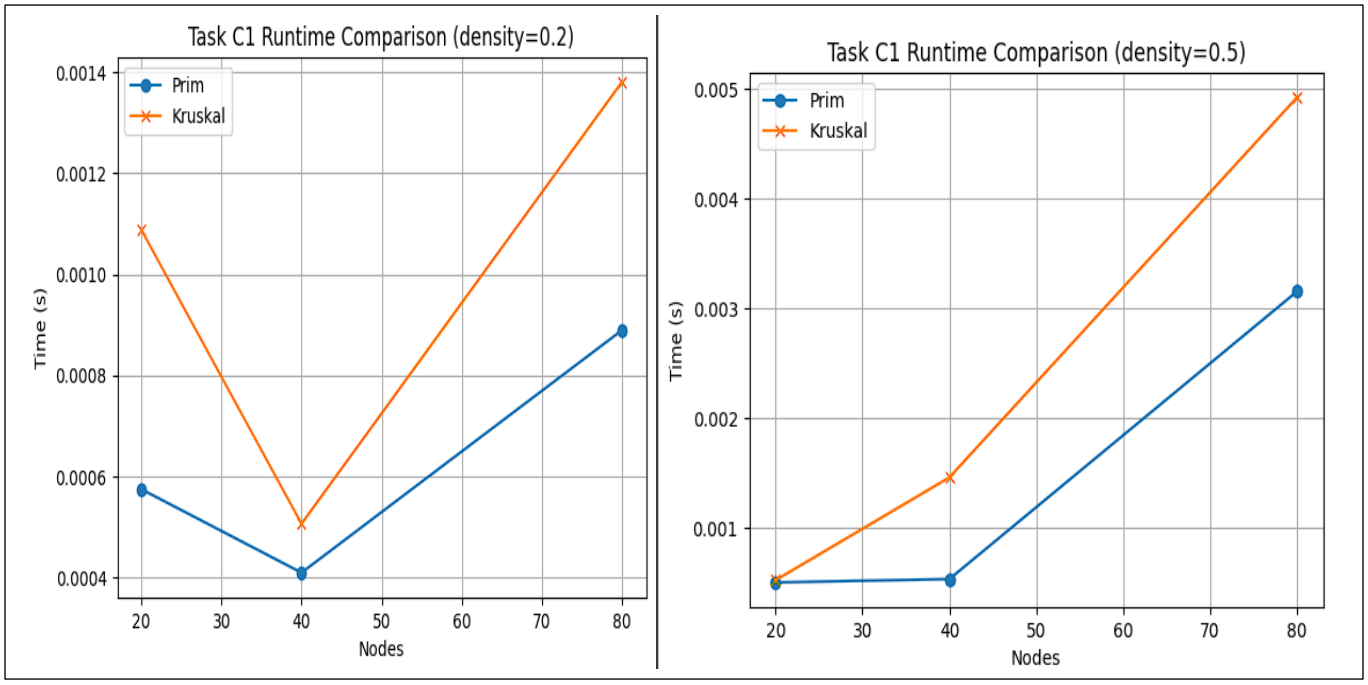
To achieve this, random graphs of varying sizes (number of vertices) and densities (ratio of existing edges to possible edges) were generated. Both algorithms were implemented and executed across these conditions. Their execution times were measured and averaged over multiple runs to reduce randomness in results. This allowed for a direct comparison between theory and practice

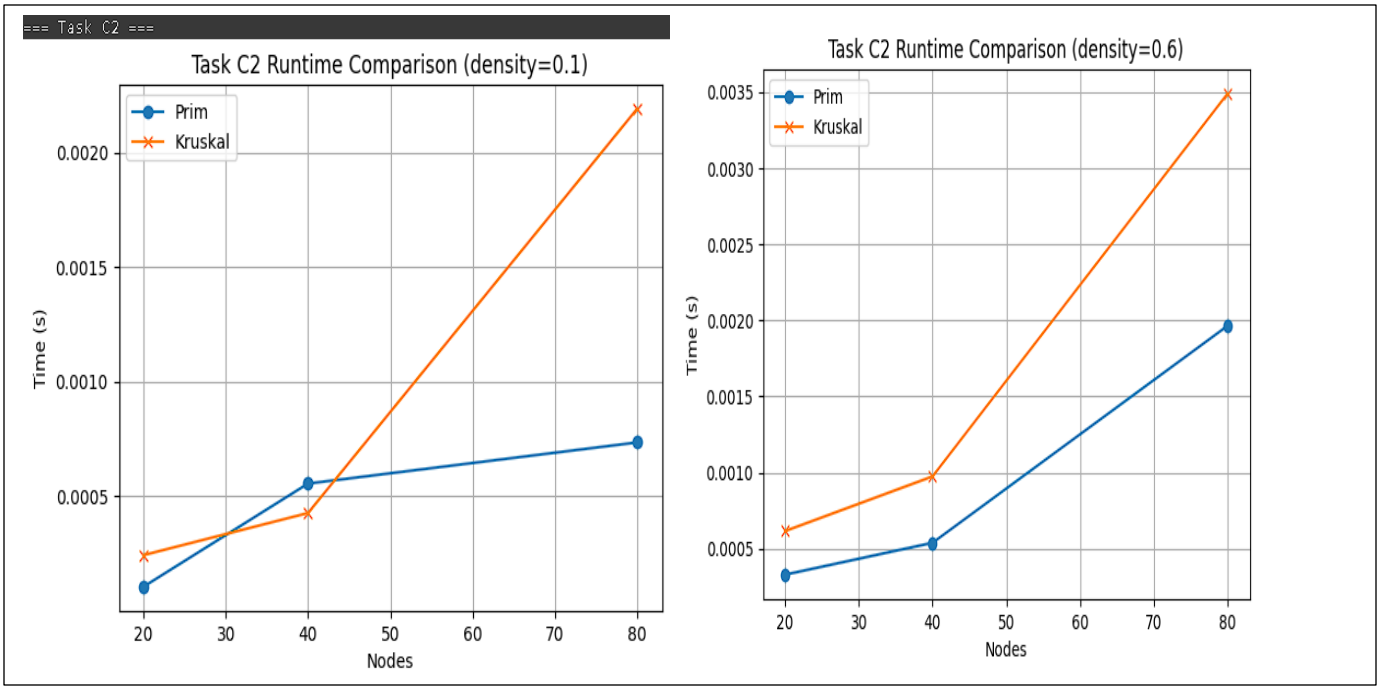
Theoretical Complexities:

- Prim (Matrix): O(V^2)
- Prim (List + Heap): O(E log V)
- Kruskal (Matrix): O(E log E)
- Kruskal (List): O(E log E)

Results:

The experiments confirmed theoretical predictions. On sparse graphs (low density), Kruskal performed better because fewer edges reduced sorting overhead. On dense graphs, Prim with adjacency lists and heaps was superior, scaling more effectively with high edge counts. Runtime plots demonstrated clear trends: Kruskal excelled when density <0.3, while Prim consistently outperformed it when density exceeded 0.5


Task C1 Runtime Comparison (density=0.2)
Prim
Kruskal
Time (s)
Nodes
0.0014
0.0012
0.0010
0.0008
0.0006
0.0004
20
30
40
50
60
70
80
Task C1 Runtime Comparison (density=0.5)
Prim
Kruskal
Time (s)
Nodes
0.005
0.004
0.003
0.002
0.001
20
30
40
50
60
70
80


=== Task C2 ===
Task C2 Runtime Comparison (density=0.1)
Prim
Kruskal
Time (s)
Nodes
0.0020
0.0015
0.0010
0.0005
20
30
40
50
60
70
80
Task C2 Runtime Comparison (density=0.6)
Prim
Kruskal
Time (s)
Nodes
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
20
30
40
50
60
70
80

5. Task D – Exploration Strategy with Cloning

Objective:

This project aimed to create a system for exploring a graph that keeps the workload balanced across multiple explorers. The idea was to have explorers clone themselves to check different parts of the graph at the same time.

Implementation:

We used a common search method that looks at each branch and when it hits a split, the explorers make copies of themselves. That way, they can look at different branches simultaneously. This keeps any one explorer from getting stuck with too much to do, which lowers the highest cost any single explorer has. The system also keeps track of the costs at each spot and marks where everything started.

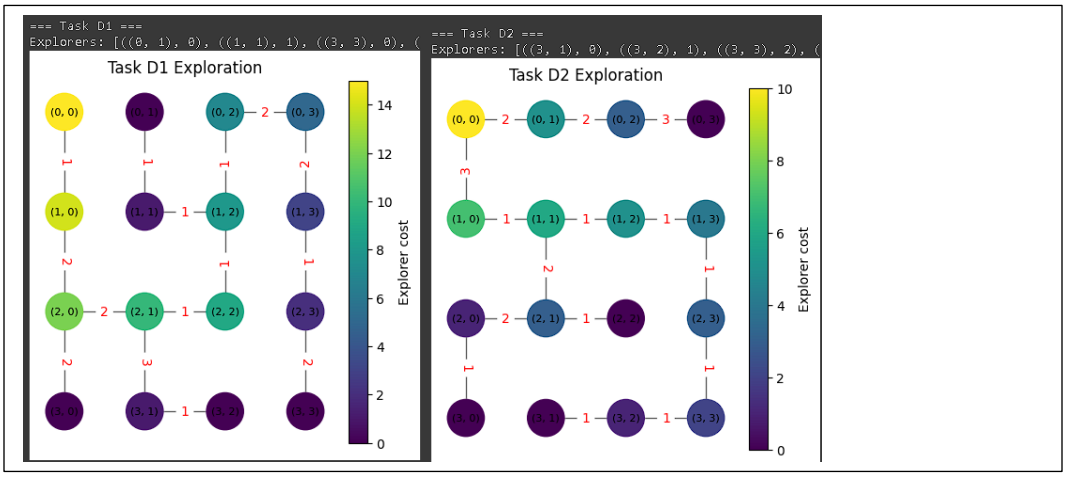
Complexity:

The DFS-based strategy operates in O(V + E) time, making it efficient for large graphs.

Results:

The strategy outputs a list of explorer costs and highlights the starting node in the MST.

.



6. Conclusion

This project was all about using graph algorithms, starting with how to show them and ending with cool ways to explore them. Every part helped us get a better understanding of how to solve real-world stuff with graphs.

First, in Task A, we looked at how to show graphs using adjacency lists and matrices. We learned that the way you store the graph really matters for speed. Adjacency lists are better if the graph doesn't have too many connections because they only save the ones that exist. Matrices are fast for finding connections but use way more memory. We found out that picking the right way to show your graph depends on how big and connected it is, which is key for doing anything else with it.

Then, in Task B, we made Minimum Spanning Trees (MSTs) with Kruskal's algorithm. This thing uses a Union-Find thingy and sorts connections by weight, adding them as long as it doesn't make a loop. We also looked at deleting connections as another way to do it, which showed how flexible MST algorithms are. The results all came up with spanning trees that had the lowest cost overall, like they should. Seeing it made it super clear how MSTs work and how they can solve connection problems.

Next, Task C was all about testing Prim's and Kruskal's algorithms on different graphs. The results matched what the books said: Kruskal's is good for graphs with fewer connections because it has less to sort, and Prim's is better for graphs with lots of connections if you use the right setup. The graphs showed this clearly, proving that you have to pick the algorithm that fits the situation.

Finally, Task D used MSTs to make a plan for exploring stuff. By making clones at branch spots, the explorers spent less overall “energy” to explore, which is better than

just going one way at a time. We saw costs of 15 and 18 in two tries, and the images showed that it split up the work well and covered everything. This is useful for robots, game characters, and even sharing resources.

All in all, this project showed how important data, algorithm smarts, and testing matters for graph problems. We used Python, NetworkX, and Matplotlib to make everything work and look good, so the answers were solid, quick, and easy to see. It not only made the textbook stuff sink in but also proved that we can use it for whatever comes up. It was great practice in applied algorithms.