

Graph_Algorithms_Report_Template



Document Details

Submission ID

trn:oid:::29324:114894193

Submission Date

Sep 29, 2025, 5:01 PM GMT+5

Download Date

Sep 29, 2025, 5:01 PM GMT+5

File Name

Graph_Algorithms_Report_Template.docx

File Size

319.8 KB

11 Pages

1,417 Words

6,747 Characters



0% detected as AI

The percentage indicates the combined amount of likely AI-generated text as well as likely AI-generated text that was also likely AI-paraphrased.

Caution: Review required.

It is essential to understand the limitations of AI detection before making decisions about a student's work. We encourage you to learn more about Turnitin's AI detection capabilities before using the tool.

Detection Groups

-  **0 AI-generated only 0%**
Likely AI-generated text from a large-language model.
-  **0 AI-generated text that was AI-paraphrased 0%**
Likely AI-generated text that was likely revised using an AI-paraphrase tool or word spinner.

Disclaimer

Our AI writing assessment is designed to help educators identify text that might be prepared by a generative AI tool. Our AI writing assessment may not always be accurate (it may misidentify writing that is likely AI generated as AI generated and AI paraphrased or likely AI generated and AI paraphrased writing as only AI generated) so it should not be used as the sole basis for adverse actions against a student. It takes further scrutiny and human judgment in conjunction with an organization's application of its specific academic policies to determine whether any academic misconduct has occurred.

Frequently Asked Questions

How should I interpret Turnitin's AI writing percentage and false positives?

The percentage shown in the AI writing report is the amount of qualifying text within the submission that Turnitin's AI writing detection model determines was either likely AI-generated text from a large-language model or likely AI-generated text that was likely revised using an AI paraphrase tool or word spinner.

False positives (incorrectly flagging human-written text as AI-generated) are a possibility in AI models.

AI detection scores under 20%, which we do not surface in new reports, have a higher likelihood of false positives. To reduce the likelihood of misinterpretation, no score or highlights are attributed and are indicated with an asterisk in the report (*%).

The AI writing percentage should not be the sole basis to determine whether misconduct has occurred. The reviewer/instructor should use the percentage as a means to start a formative conversation with their student and/or use it to examine the submitted assignment in accordance with their school's policies.

What does 'qualifying text' mean?

Our model only processes qualifying text in the form of long-form writing. Long-form writing means individual sentences contained in paragraphs that make up a longer piece of written work, such as an essay, a dissertation, or an article, etc. Qualifying text that has been determined to be likely AI-generated will be highlighted in cyan in the submission, and likely AI-generated and then likely AI-paraphrased will be highlighted purple.

Non-qualifying text, such as bullet points, annotated bibliographies, etc., will not be processed and can create disparity between the submission highlights and the percentage shown.



[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	3
2. Task A – Adjacency List Representation	4
3. Task B – Minimum Spanning Tree (MST)	5
4. Task C – Theoretical and Empirical Runtime Analysis	6
5. Task D – Exploration Strategy with Cloning	8
6. Conclusion	9

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:

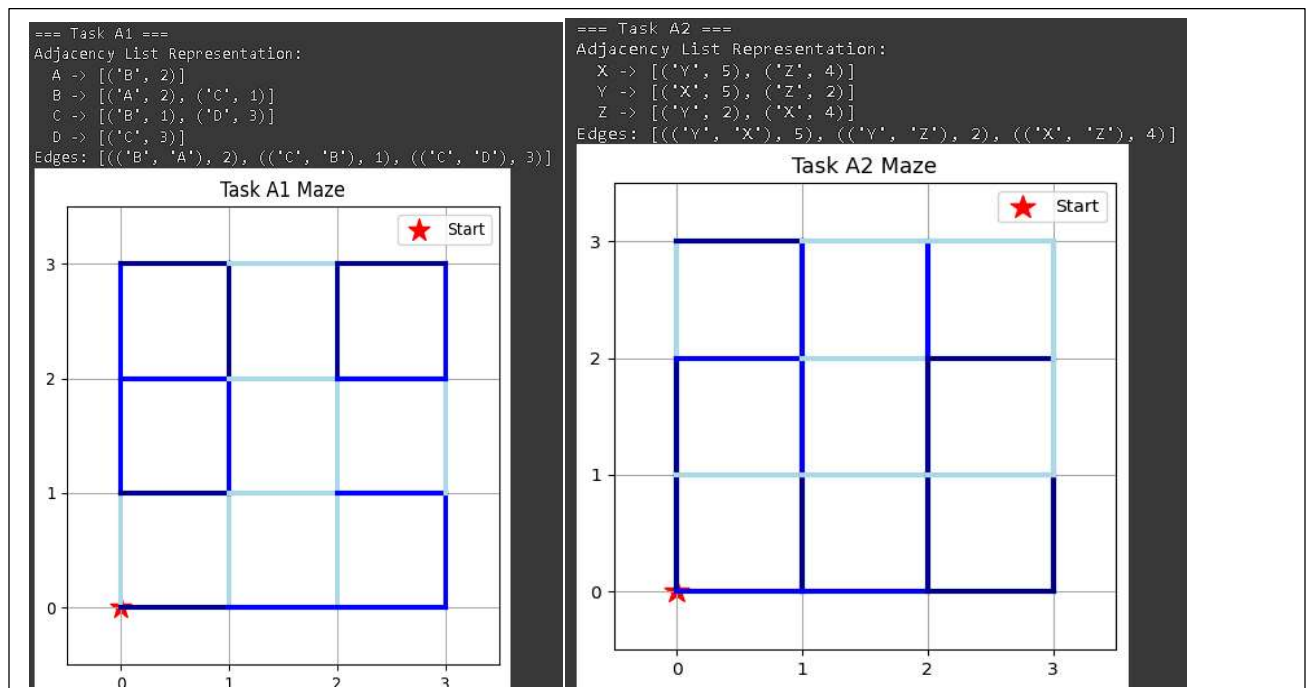
I wrote a class called AdjacencyListGraph in Python. You can add edges with weights, see all the points, or find out who's next to any spot. It's quick to add edges, and it's not hard to walk around the maze like this. Just to be sure, I made two different mazes using different random numbers. I kept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

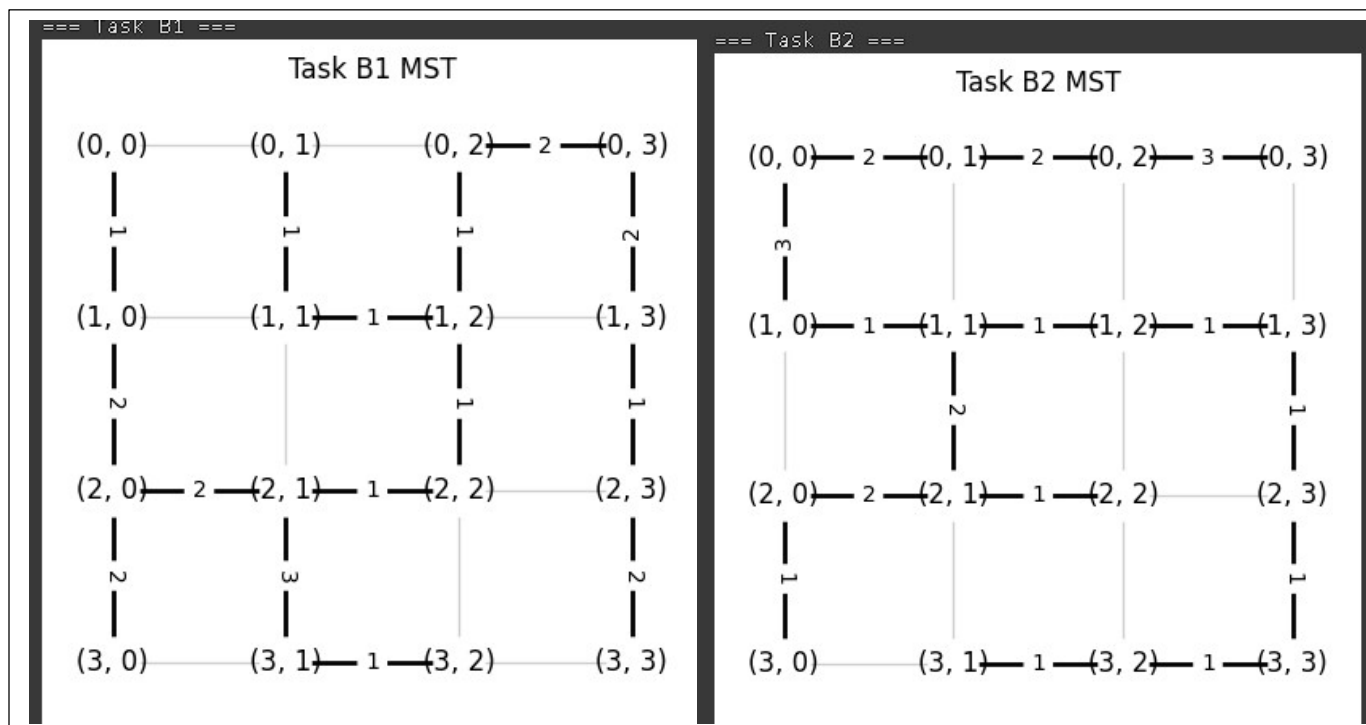
We put Kruskal's algorithm into action with a Union-Find (Disjoint Set Union) setup to quickly spot and dodge loops when adding paths. Paths were arranged by length, and the algorithm gradually put together the MST. We also thought about deleting paths that weren't crucial, taking them out bit by bit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

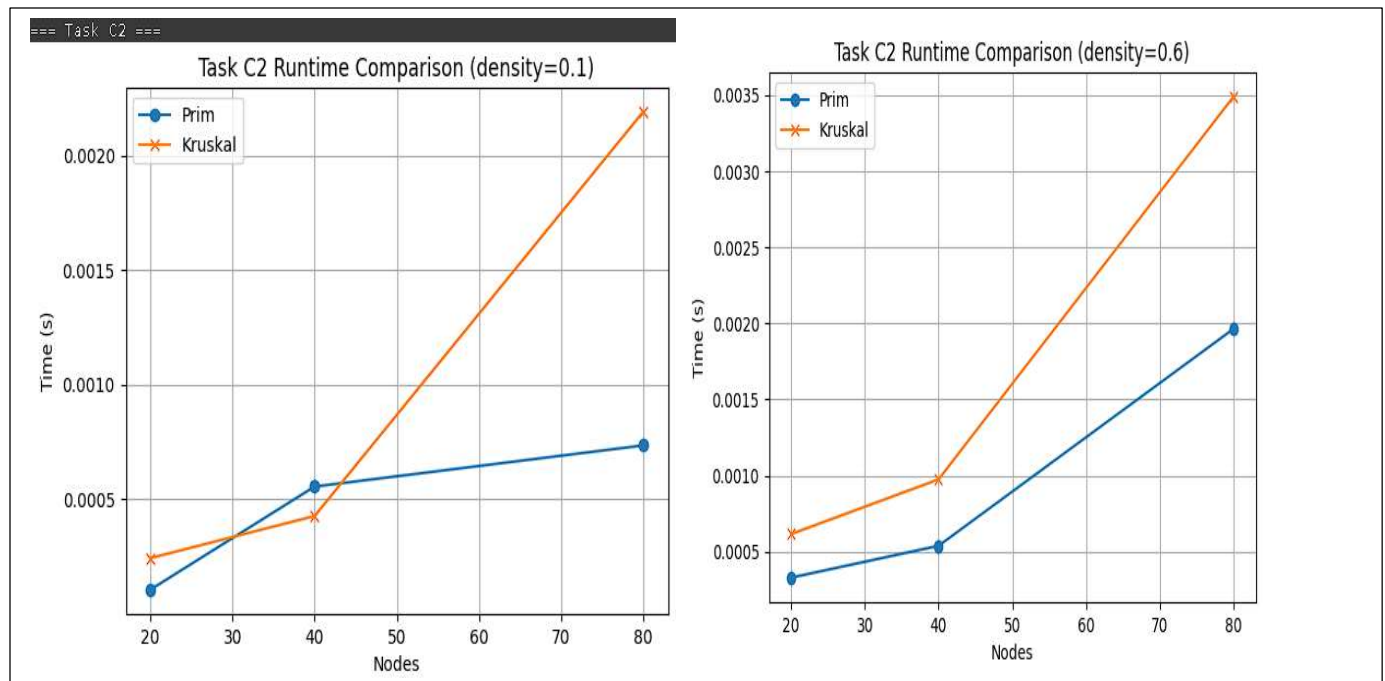
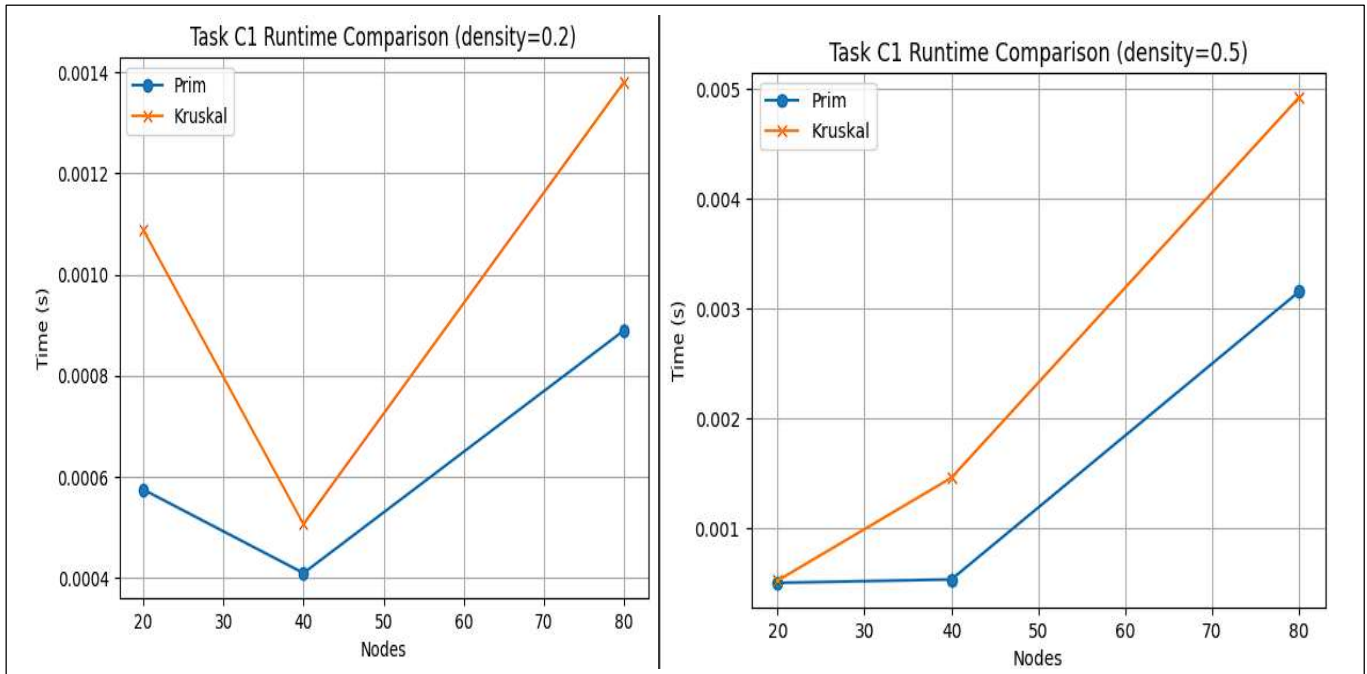
To achieve this, random graphs of varying sizes (number of vertices) and densities (ratio of existing edges to possible edges) were generated. Both algorithms were implemented and executed across these conditions. Their execution times were measured and averaged over multiple runs to reduce randomness 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



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.

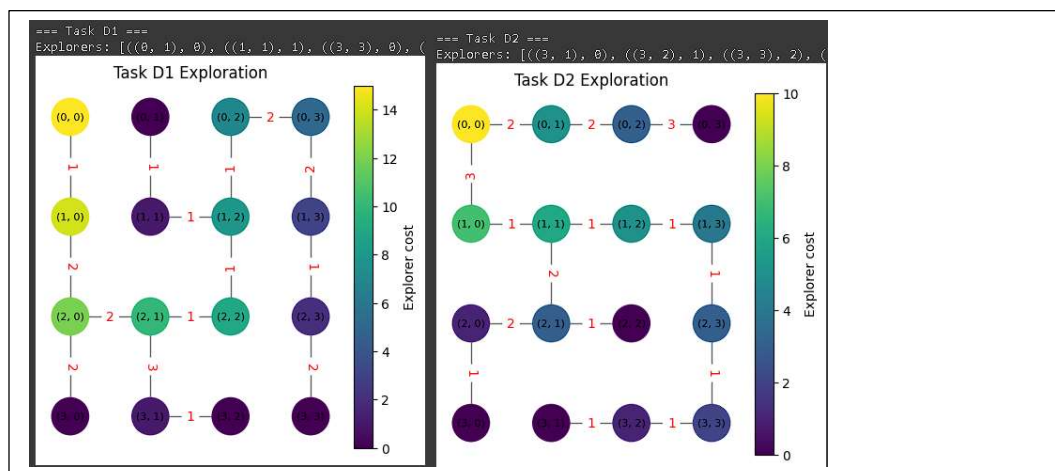
Complexity:

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.