RMIT UNIVERSITY

Algorithms and Analysis
COSC2123/3119

The Maze of Many:

An Exploration of Graph Traversal and Planning Problems

Assessment Type	Individual assignment. Submit online via GitHub.
Due Date	Week 11, Friday October 10, 19:59 (before 8pm). A late penalty will apply to assessments submitted after 7:59 pm.
Marks	30

1 Learning Outcomes

This assessment relates to four learning outcomes of the course which are:

- CLO 1: Compare, contrast, and apply the key algorithmic design paradigms: brute force, divide and conquer, decrease and conquer, transform and conquer, greedy, dynamic programming and iterative improvement;
- CLO 2: Compare, contrast, and apply key data structures: trees, lists, stacks, queues, hash tables and graph representations;
- CLO 3: Define, compare, analyse, and solve general algorithmic problem types: sorting, searching, graphs and geometric;
- CLO 4: Theoretically compare and analyse the time complexities of algorithms and data structures; and
- CLO 5: Implement, empirically compare, and apply fundamental algorithms and data structures to real-world problems.

2 Overview

Across multiple tasks in this assignment, you will design and implement data structures that represent mazes as graphs and algorithms to explore these mazes. You will address both fully connected, non-cyclical mazes and mazes with cycles. Some of the components ask you to critically assess your solutions through both theoretical analysis and controlled empirical experiments to encourage reflection on the relationship between algorithm design and real-world performance. The assignment emphasizes on strategic thinking and the ability to communicate solutions clearly and effectively.

2.1 Important Notes

Please read all the following information before attempting your assignment.

- This is an *individual* assignment. You may not collude with any other person (or people) and plagiarize their work. Everyone is expected to present the results of their own thinking and writing. Never copy another student's work (even if they "explain it to you first") and never give your written work to others. Keep any conversation high-level and never show your solution to others. Never copy from the Web or any other resource or use Generative AI like ChatGPT to *generate solutions or the report*. Suspected cases of collusion or plagiarism will be dealt with according to RMIT Academic integrity policy.

- This assignment requires submitting both a written report in PDF format and your implemented code via **GitHub** Classroom (Details in Section 5 - *Submission*).

- Before implementing any code, please carefully read the `README` section provided in the skeleton code repository and add or modify code **only** within the files explicitly marked with the comment `# EDIT THIS FILE TO IMPLEMENT X` at the top of the file. You may add helper functions to editable files, but do not add any additional code files. Any changes outside these marked sections may negatively affect the reliability of your solution and could cause your submission to fail our automated tests.

- You are expected to properly use git version control and **commit regularly, with clear and meaningful messages** to reflect the progress of your development. Submitting your entire solution in a single commit, or in just a few large commits, is considered poor practice. Please note that marks from automatic tests will be adjusted based on how well you follow these practices. Full marks will only be awarded when correct implementation is paired with clear evidence of good version control and development discipline.

- Respect the word and page limits specified for the report. Content beyond the specified limits will not be read and **will not be considered in marking**.

- Strictly adhere to the assignment submission deadline. The deadline will not be extended under any circumstances, except in the event of natural disasters or similar emergencies.

- Please check the Ed forum discussion for further clarifications about specifications. In addition, a video will be released explaining the assignment specification (see the Ed forum post for the assignment release).

- Please do use the Ed forum and consultation sessions to ask questions around the tasks. You are welcome to ask any question you like, but we may limit our answers slightly to ensure we do not take away learning opportunities.

- In the submission (a single PDF file for the report) you will be required to certify that the submitted solution *represents your own work only* by including the following statement:

> *I certify that this is all my own original work. If I took any parts from elsewhere, then they were non-essential parts of the assignment, and they are clearly attributed in my submission. I will show I agree to this honor code by typing "Yes":*

Contents

3 Assessment details

The assignment is broken up into a number of tasks (A, B, C, and D) to help you progressively complete the project. Please note that although completing one task can assist with subsequent tasks, each task can be attempted independently. Moreover, the tasks are designed in a way that even in cases where your implementation does not fully achieve the expected output, you can still discuss and present the theoretical aspects of your algorithmic design through your report.

Marks are broken up into two types:

1. implementation (15 marks); and
2. report (15 marks)

Implementation marks are given for your code. Your code will be examined using a series of tests. The more tests you pass, the more marks your implementation will be awarded. 3 of your implementation marks are awarded for good software engineering (clean code, good comments, descriptive and small commits, etc), and so 12 marks are strictly for tests.

Report marks are given for your written work. Tasks B, C, and D all have a written element that you will be expected to submit as **a single PDF file**. You will be awarded marks for correctness, rigour, clarity, and good mathematical communication. Tasks that require written work will give motivation for what to write. Please adhere to the page limit.

3.1 Motivation

You are a sorcerer's apprentice, and have been tasked with exploring the *Maze of Many* - an ancient, ever changing labyrinth full of arcane traps and dangerous obstacles. Legends say that the maze is full of magical items waiting to be reclaimed, but only those who possess deep knowledge of graph theory may enter the maze and survive...

Objective

The overall objective is deceptively simple:

The maze can be represented as a weighted, undirected graph $G = (V, E)$, where:

- V is the set of rooms/vertices ($|V| = n \times m$);
- E is the set of corridors/edges, each having a positive cost/weight $w : E \mapsto \mathbb{N}^+$
- $s \in V$ is a starting node

You are searching for a traversal path:

$$P = (s = v_0, v_1, v_2, ..., v_k)$$

such that:

- set$(P) = V$ (all rooms have been explored);
- Each sequential vertex in the path P should:

- be adjacent, so if $v_i = (a, b)$ then $v_{i\pm1} \in \{(a \pm 1, b), (a, b \pm 1)\}$; and
- share an edge, so $(v_i, v_{i\pm1}) \in E$

- Cost$(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$ is minimised, where $w(v, u) > 0$ is the weight between vertex $v \in V$ and $u \in V$ (cost of moving from room v to room u).

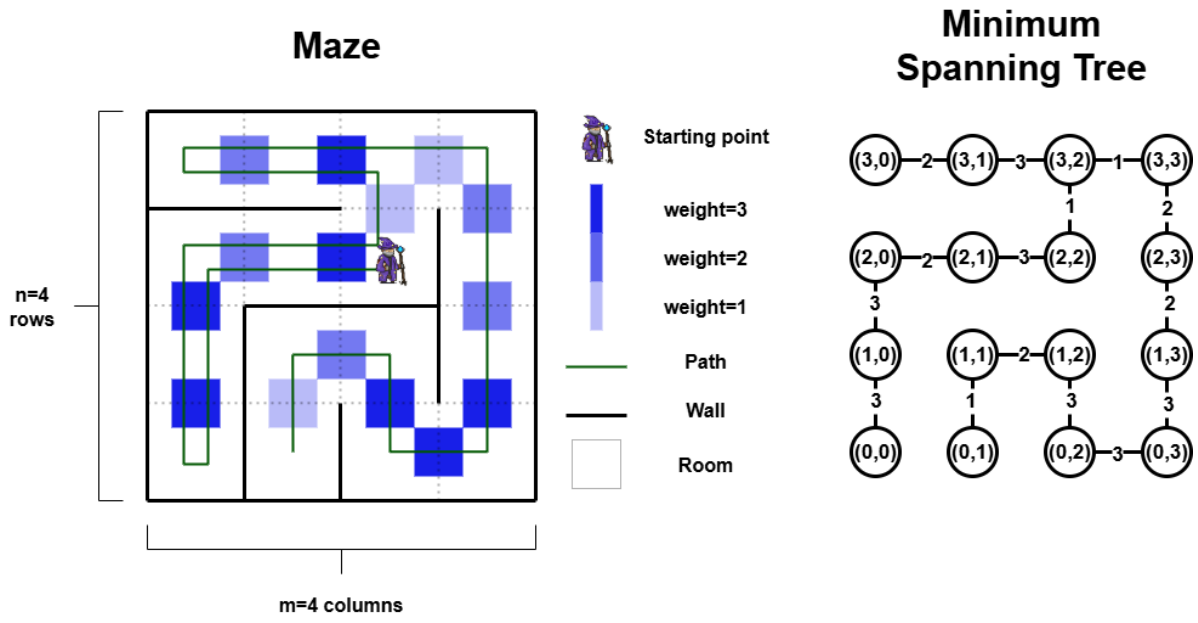

Figure 1: A breakdown of the problem. You are presented with a randomly generated 2 dimensional $n \times m$ maze structure. Once the maze is generated, you know its entire structure. All cells in the maze are square, and therefore have a maximum of 4 connections (we can only travel to adjacent rooms). Two adjacent rooms are disconnected if a wall separates them. If not, we may enter the new room at some cost, and this cost is also randomly generated (the costs given in the figure are just examples). The maze can be represented as a *weighted, undirected graph*.

3.2 Tasks in Brief

Completing Tasks A, B, C, and D will take you through the stages of completing the above objective for different scenarios, and allow you to assess which methods are the most suitable for different tasks.

Task A Implement an adjacency list (data structure) for a graph (abstract data type).

Task B Implement Kruskal's Algorithm for generating a minimum-spanning tree of a graph and discuss different minimum-spanning tree behaviour.

Task C Using theoretical and empirical analysis, prove which data structure (matrix vs list) and minimum-spanning tree algorithm (Prim's vs Kruskal's) combination is best for this problem (if any).

Task D Use the minimum-spanning tree to plan an optimal exploration strategy using clones.

Please read each task carefully, and implement only what is asked in each Task. Please read the README.md file found in the repository for instructions on how to run the code and the sample tests.

3.3 Suggested Four-Week Timeline

It is highly recommended that you set up your developer environment, read the specification, and run/experiment with the code and configuration file as soon as possible. **You should start by reading the README file in your repository** - this contains instructions on setting up and running the code. Doing so will allow you to gauge how challenging you believe this assignment will be for you personally and where you are likely to have the greatest difficulties. The tasks do, generally, build in difficulty, and so Task D is more difficult than Task C and so on.

Please do read the tasks early and try to understand what is being asked of you. When you know the motive and objective of each task, you will be able to plan how best to create a great solution to this assignment. On average, since the assignment is 30 marks we expect the average student to spend approximately 36 hours on it in order to achieve the average mark (according to the accredition process). Some will spend more, and some will spend less - that is up to the individual. This amounts to 9 hours per week or around an hour and fifteen minutes per day. Marks for each task are a good proxy for difficulty, therefore you should look to spend approximately:

1. 17% of your time on Task A;
2. 23% of your time on Task B;
3. 27% of your time on Task C; and
4. 33% of your time on Task D.

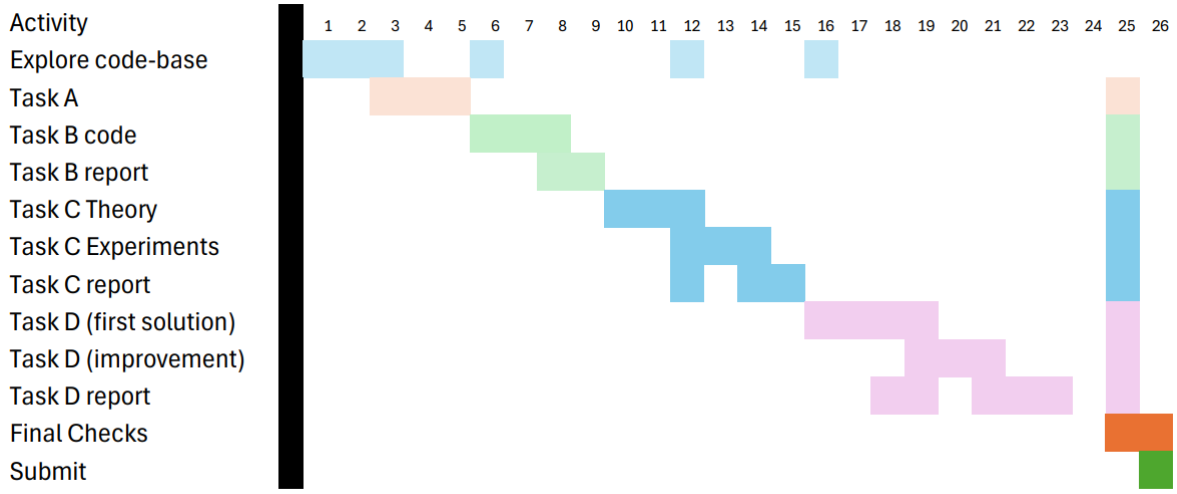

Figure 2: Gantt chart for a suggested timeline. Task D, which is more open ended, should (probably) have more time dedicated to it than other tasks to achieve top marks. Notice that coding tasks require re-exploring the code base. You may use any code in the code base to inspire your own solutions, so you need to have a look at how it works. It is not expected that you work on this assignment every single day - this is just a suggested timeline.

Do remember that we have access to your repository and commit history, and so can see when you start and work on the assignment. Do not panic - there is no punishment for starting the assignment late! But students who start early perform better, in general.

4 Tasks

4.1 Task A: Implementing an Adjacency List (5 marks)

Before leaving the Arcane University on your quest, you read all you can about mazes and graphs. You learn that there are two ways to represent graphs: (1) adjacency matrices; and (2) adjacency lists. You already know how to represent a maze as an adjacency matrix, but you must learn a spell for the adjacency list representation.

4.1.1 Implementation Task (5 marks)

The current implementation in `graph/adjacency_matrix.py` stores the maze/graph as an adjacency matrix. You must implement the class in `graph/adjacency_list.py` to represent the maze/graph as an adjacency list.

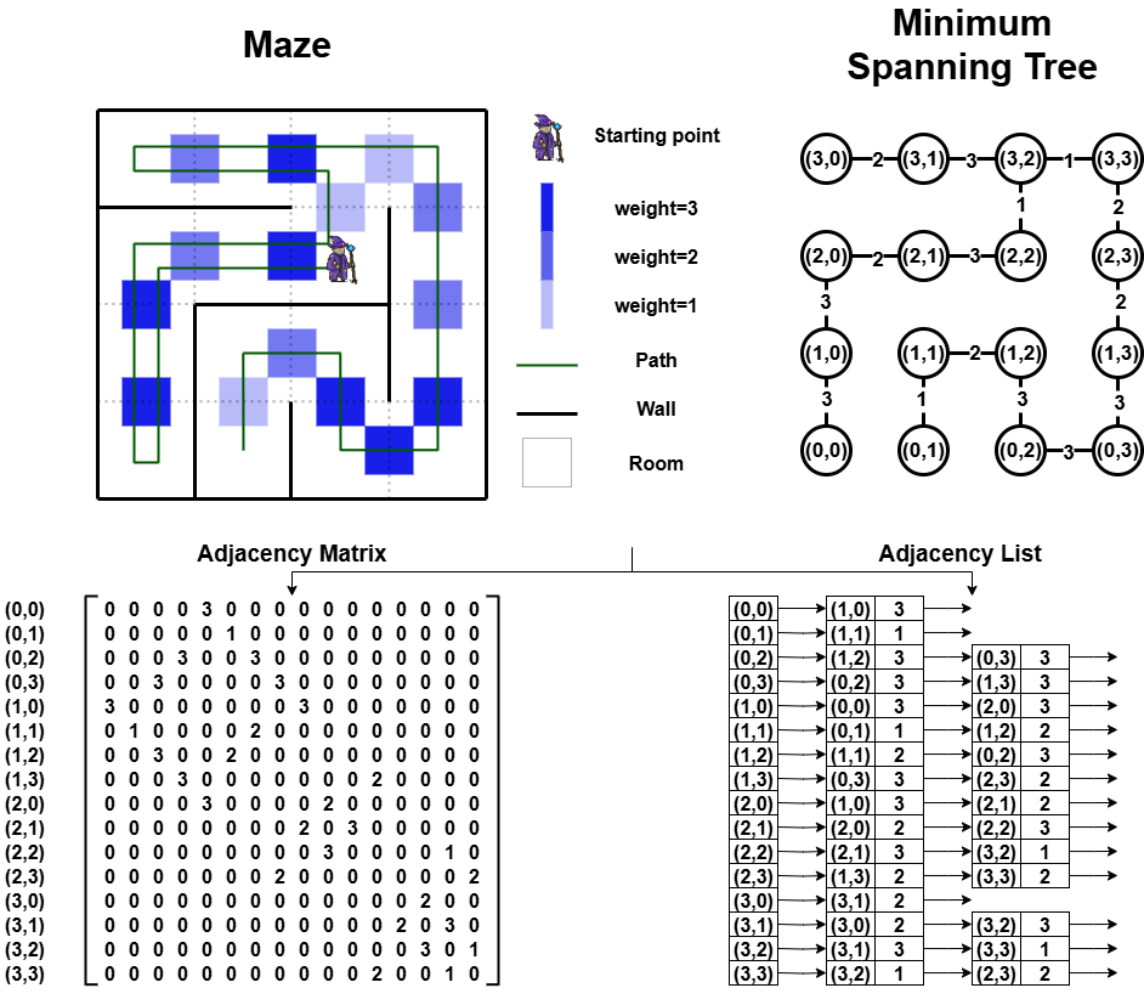

Figure 3: Flow of representation. Here we go from maze/graph→matrix/list representation.

4.1.2 Report Task (0 marks)

You are NOT required to write anything in your report for Task A.

Tips

Here are a few tips to assist you with Task A:

- Changing the configuration file variable *graph_type* from "matrix" to "list" will make the maze/graph use an adjacency list;
- The list representation should mimic the behaviour of the matrix implementation, but algorithms will be slightly modified as the data structures are different;
- Some basic, non-exhaustive tests are given to ensure your methods are on the right track. To run these tests, run `python -m tests.test_list`; and
- The initial visualisation shows a maze of all walls with a single node being the minimum-spanning tree. When the task is complete you should have a fully connected maze in the left panel (with path) and a minimum-spanning tree graph in the right panel (matching the output for the matrix) - see Figure 4 in the README.
- You can load the example maze in the above figure by setting *load_maze* to "true" and *maze_name* to "specMaze" in the configuration file.
- Changing *print_struct* to "true" in the configuration file will print the data structure to the console (this is useful for debugging).

4.2 Task B: Creating a Minimum Spanning Tree (7 marks)

After a long and tiring journey, you finally arrive at the entrance of the *Maze of Many*. Carved into a great, obsidian obelisk outside the stone door is a map of the labyrinth—its twisting passages drawn as a graph of nodes and edges only you can interpret. Beneath the map is an inscription, glowing faintly with ancient magic:

> *"Only those who can unveil the skeleton of the maze may enter.*
>
> *A path that connects all, yet weighs the least.*
>
> *Reveal in both ways, lest you be lost."*

You recall overhearing two of the greatest known magic wielders arguing over a minimum graph weight problem in the local tavern outside the Arcane University. The Grand Wizard Prim's method grew a tree step by step from a starting point. But Kruskal the Wise's way was different: He did not begin inside the maze at all. Instead, he laid out all possible paths in order of strength, and joins them one by one, always taking the lightest path that does not form a cycle. For both methods, what remains is the Minimum Spanning Tree—the simplest skeleton of the maze.

The riddle seems to ask for both spells...

4.2.1 Implementation Task (5 marks)

The *MST* folder in your code contains functions for both Prim's and Kruskal's algorithm. However, only Prim's has been implemented. Your task is to implement Kruskal's algorithm in *MST/kruskals*, written in pseudo-code below.

Algorithm 1 Kruskal's Minimum Spanning Tree (MST)

Require: Graph $G = (V, E)$
Ensure: MST T
if $V = \emptyset$ **then**
    return empty graph of same type ▷ If input graph is empty, return an empty graph
$T \leftarrow$ empty graph with vertices V ▷ This is will be the MST (currently no edges)
$E' \leftarrow$ empty list ▷ Collect all the edges into this list
for each vertex $u \in V$ **do**
    for each neighbor $v \in \text{neighbours}(u)$ **do**
        if (u, v) already considered (due to undirected graph) **then**
            continue ▷ We can skip this because $(u, v) \equiv (v, u)$
        $w \leftarrow \text{getWeight}(u, v)$
        Append (w, u, v) to E' ▷ Add edge to edges list
Sort E' by weight ascending ▷ We want to process lighter edges first
Initialize parent map: $\text{parent}[v] \leftarrow v$ for all $v \in V$ ▷ Stores vertex sets
for each edge $(w, u, v) \in E'$ **do**
    if $\text{union}(u, v, \text{parent})$ **then** ▷ Check adding this edge won't create a cycle
        Add edge (u, v, w) to T ▷ Adds this edge to the MST
return T

4.2.2 Report Task (2 marks, maximum 1 page)

Answer the following questions about minimum-spanning trees:

1. Design a greedy algorithm that operates by deleting edges from the original graph to create a minimum spanning tree, rather than adding them like Kruskal's does. In a similar style to the above example, write pseudo-code for your algorithm.

2. With reference to your pseudo-code, justify the complexity of your algorithm. Explain why it may it not be efficient and how density effects its runtime.

Total page limit for Task B (pseudo-code and answers to problems above) is **one page**.

Tips

Below are some tips for completing this task:

- To change from Prim's algorithm to Kruskal's, change the *mst_generator* method from "prims" to "kruskals" in the configuration file;
- Kruskal's algorithm works by iteratively selecting the lightest edge in the maze, provided selecting it (1) connects two previous unconnected parts of the graph; and (2) does not create a cycle. It terminates when the number of selected edges $|E_T| = |V| - 1$.
- Setting *wall_removal_perc* (which deletes X% of remaining walls) to a value larger than 0 will create a maze with cycles. You should check that that the minimum-spanning tree on the right panel of the visualisation has no cycles even if the maze does.
- You are given some basic, non-exhaustive tests to check your progress - to run them run `python -m tests.test_kruskals`;
- You have been provided with a union function in *MST/kruskals* - it may come in useful.

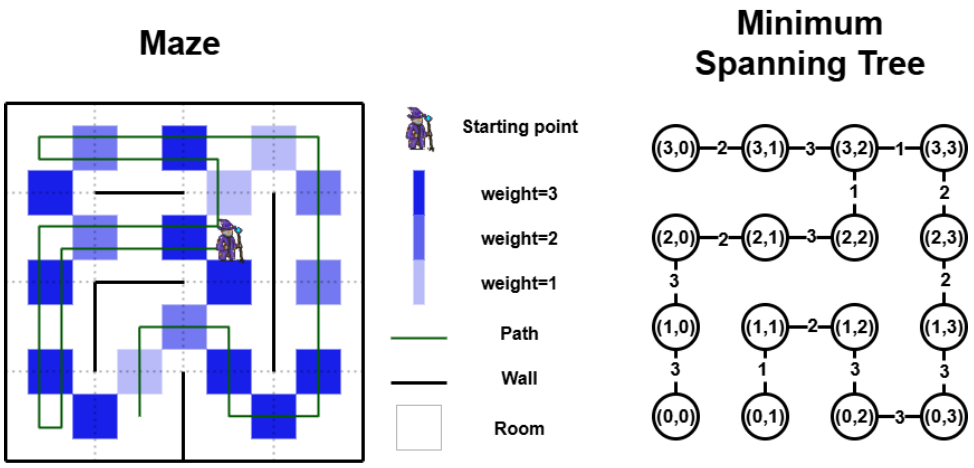

Figure 4: An example output of a maze, a valid exploration strategy and its minimum-spanning tree. Here we observe that, even though the maze has cycles, the minimum spanning tree does not. We use the minimum-spanning tree to guide exploration.

4.3 Task C: Analysis of the different Strategies (8 marks)

With your knowledge of arcane maps and spanning trees, you now face a deeper challenge. The *Maze of Many* constantly shifts its form, sometimes sprawling with countless passages, other times condensed into tight clusters. As a sorcerer's apprentice, you must learn not only the spells themselves, but also when each spell is most powerful. Two questions remain:

1. Should you represent the maze using the **adjacency matrix** or the **adjacency list**?
2. Should you rely on **Prim's algorithm** or call upon **Kruskal's algorithm**?

To survive the shifting labyrinth, you must compare these approaches and decide which is most effective under different conditions. Only then will you wield true mastery over the maze.

4.3.1 Implementation Task (0 marks)

Your code is not marked for this task.

4.3.2 Report Task (8 marks, maximum 2 pages)

Your report should include:

1. **Theoretical Analysis:** Strongly justify (with reference to equations and/or the algorithms) the theoretical time complexities of both Kruskal's Algorithm and Prim's algorithm with the two different graph representations (4 complexities in total).
2. **Empirical Design:** Identify which variables are most important when comparing the four combinations (matrix+Prim, matrix+Kruskal, list+Prim, list+Kruskal). Explain why these variables matter, and why you may ignore others.
3. **Empirical Analysis:** Provide a thorough analysis using one or more plots to compare the performance of these combinations on mazes of varying size and density.
4. **Reflection:** Discuss whether your empirical results align with your theoretical predictions. How do the results compare to your theoretical expectations? If there are discrepancies, explain possible causes — e.g., data structures, constant factors, or implementation details. Is there an objectively best choice algorithm/representation here? Why?

Total page limit for Task C is **two pages**.

Tips

Below are some tips for completing this task:

- Begin by reading and understanding the algorithms behind Prim's/Kruskal's. How do they use the matrix/list?
- You are given a start/stop timer function - one of your tasks is to workout what you should be timing.
- Make sure to provide plots for your experiments and include the plots in your report.
- If you need advice on how to begin your empirical analysis, have a look at the lab exercise at the end of tutorial 3.
- Be sure to try and link your theoretical and empirical analysis together.

4.4 Task D: The Final Problem (10 marks)

Now that you understand the full structure of the maze, it is time to begin exploring. Your job is to explore every room in the maze and report your findings to the Arcane University. This initially sounds simple, but in your possession you have *the mirror of reflections*, a powerful artifact that allows you to create an ethereal copy of yourself. These ghostly copies may, in turn, make their own copies, who may make their own copies, etc. Your copies may explore parts of the maze on your behalf whilst you explore elsewhere - however, every copy you make has a cost.

Goal

Your objective is to devise a strategy that explores *every* cell while minimising the exploration cost. Exploration cost is defined as the **maximum total traversal cost** incurred by any explorer (the sorcerer or one of their clones). When a clone is created, it **inherits the cost of reaching its creation point**, and then continues exploring its assigned sub-tree. Because clones explore in parallel, the overall exploration cost is not the sum of all traversal and cloning costs, but rather the cost of the longest individual exploration. Clones may also create further clones recursively if your strategy deems this optimal. **Once a clone has finished exploring its designated branch it disappears**. In addition, you **may assume that the maze has no cycles**. This task connects traversal planning with decrease/divide-and-conquer trade-offs.

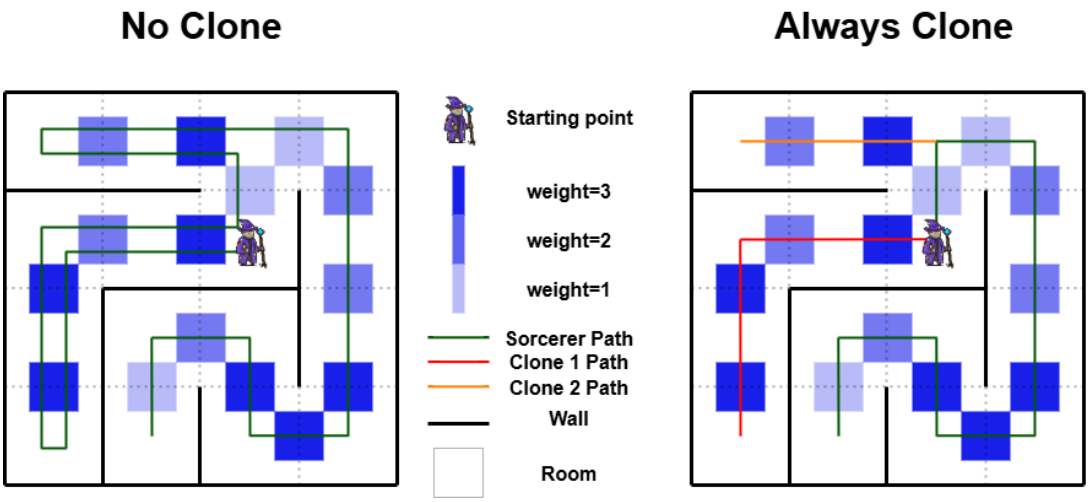

Figure 5: Example paths of two different exploration strategies. Left: We never make clones. Right: We make clones at every junction.

Take the above figure as an example. The no clone strategy would always traverse this maze (using this path) at a cost of 50 - the cost of traversing all the weights using the path above. The always clone strategy now has cloning in play at a cost of d. If $d = 1$ then:

- the original sorcerer has a total cost of 20 (makes 2 clones, traversal cost is 18);
- clone 1 has a total cost of 12 (the cost of the journey up to that point of 0, the cost of creating the clone 1 and the cost of exploring its sub-tree of 11); and
- clone 2 has a total cost of 8 (the cost of the journey up to that point of 2 (the sorcerer created a clone then moved), the cost of creating the clone 1 and the cost of exploring its sub-tree of 5).

Since exploration happens in parallel, the cost of this exploration is the maximum of all individual costs - in this case, 20. Clearly the cloning strategy was more optimal here. However, as the cost of cloning grows the always cloning strategy becomes less optimal.

4.4.1 Implementation (5 Marks)

The file *solvers/task_d_solver* contains a function called *task_d_explore*. Here you should implement a maze exploration strategy that may utilise recursive cloning. Your function takes in: (1) a graph (which is an minimum-spanning tree); (2) a current coordinate (initially the starting vertex v_0); (3) the set of visited vertices (initially empty); (4) a list of lists containing paths of any previous clones and the original sorcerer; and (5) the explorer id (0 is the sorcerer, 1 is clone 1, etc). Your function should edit the *all_paths* variable so that it contains a list of paths (each list being the path traversed by the sorcerer or a clone). See the function comment for *task_d_explore* for more information.

The total cost for a traversal C is defined as

$$C = \max_i \left(\underbrace{H_i}_{\text{inherited cost}} + \underbrace{\sum_{j=0}^{|P_i|-1} w(v_j, v_{j+1})}_{\text{path cost}} + \underbrace{k_i\, d}_{\text{clone cost}} \right)$$

Here:

- H_i is the inherited cost (the cost accumulated up to the point where explorer i is created);
- $P_i = (v_0, v_1, \ldots, v_{|P_i|})$ is the sequence of vertices in the path traversed by explorer i;
- $w(v_j, v_{j+1})$ is the weight of the edge connecting consecutive rooms v_j and v_{j+1};
- k_i is the number of clones created by explorer i; and
- d is the cost of creating a clone.

All paths must be physically traversable. Specifically, for two consecutive rooms v_j and v_{j+1} in any path:

- $w(v_j, v_{j+1}) > 0$, i.e. the two rooms are connected by an edge; and
- v_j must be adjacent to v_{j+1} in the graph, so if $v_j = (a, b)$ then:

$$v_{j+1} \in \{(a-1, b), (a+1, b), (a, b-1), (a, b+1)\}$$

Additionally, the starting vertex of any new clone must coincide with the current vertex of an already active explorer (either the original sorcerer or another clone). A separate function will then calculate the total cost of the completed traversal.

4.4.2 Report Task (5 marks, maximum 2 pages)

Your written report (maximum two pages) must address the following:

1. **Describe your solution:** Explain the general idea behind the exploration strategy you implemented in `task_d_explore`.

2. **Mathematical motivation:** Clearly justify why your approach is reasonable using the cost function

$$C = \max_i \Big(H_i \; + \; \sum_{j=0}^{|P_i|-1} w(v_j, v_{j+1}) \; + \; k_i \cdot d \Big).$$

   Explain how your chosen strategy minimises this maximum cost.

3. **Pseudo-code:** Provide concise pseudo-code that captures the essential steps of your solution. (Keep it short—this is not a code listing.)

4. **Complexity analysis:** Justify the time complexity of your strategy in terms of $|V|$ (vertices) and $|E|$ (edges).

In the problem above we assumed:

(i) clones disappear once they finish exploring their assigned sub-branch; and

(ii) the maze has no cycles.

Briefly explain why removing these assumptions makes the problem significantly more challenging. Specifically:

- **If clones are immortal:** Why does allowing clones to explore beyond their initial sub-branch increase the complexity of choosing where and when to create them? How might you adapt your strategy in this case?
- **If the maze has cycles:** Why does this complicate traversal planning, and how might your strategy need to adapt?

Note: You are not required to implement strategies for the above two scenarios—only to reason about them. The total page limit for Task D is **two pages**.

Tips

Below are some tips for completing this task:

- Remember, you have full access to a map of the maze. This means that you know the structure in advance.
- To enforce mazes with no cycles, set *wall_removal_perc* to 0 in the configuration file - setting it to 100 will delete all the internal walls!
- Changing the *maze_solver* variable in the configuration file to "task_d" will use your strategy to generate paths.
- When starting your implementation, begin by reading through the other two exploration strategies - use them as inspiration! You may switch between them by setting the *maze_solver* variable to "no_clone" and "always_clone".
- Sometimes cloning is good, sometimes it is bad - can you develop a criterion or rule that dictates when the right time to clone is?
- How do cycles impact the uniqueness of minimum-spanning trees?

5 Submission

This assignment is due on the October 10^{th} by 19:59 (before 8pm).

5.1 Submission Guidelines

Your assignment is to be submitted via the Github Classroom. When accepting the assignment via Github Classroom, the template code is forked creating a new, personal repository for you. To begin working on your code, you should clone the repository to your local machine. All development of your assignment's code should be documented via *Git commits* in this repository.

When you are happy with your code, you may begin the submission process.

Step 1: Github Classroom Begin by checking that you are connected properly to the github classroom. To do this, navigate to this page on EdStem and download the *classroom_roster.csv* file. Check that your student number matches the GitHub username you used to create your repository. If it does not, leave a comment on the post with:

- your name;
- your student number; and
- your GitHub username

and we will help you resolve it.

Step 2: Upload PDF The report for your assignment should be stored as a PDF document. Please ensure that it is uploaded into your repository with a reasonable name, such as *(student_number)_report.pdf*. The report makes up 15 marks of this assignment, and if it is not in the repository then we cannot mark it.

Step 3: Tag Submission When you are happy that your repository is in the correct state, you may make a submission. We do this by **tagging the commit you want marked as "submission"**. Doing this requires you to:

1. Find the commit you want to submit (red box below shows an example commit message);
2. Copy the commit's hash (green box below) by pressing the copy button (the two overlapping squares);

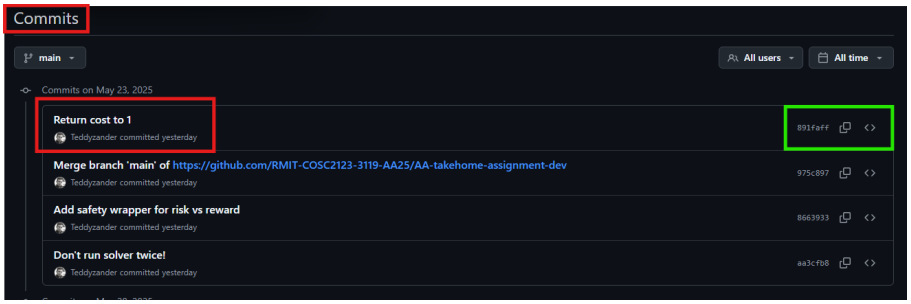

Figure 6: Example of what a commit history looks like. The red box is a commit message, and the green box is a commit's hash.

3. run the following commands in git bash:

   (a) `git tag -a submission <commit hash> -m "assignment submission"`

   (b) `git push origin submission`

   This will create a submission tag, tag the commit ¡commit hash¿ as this tag and then push it to your repository.

If you ever want to make a change to your submission, you may delete the tag using the commands:

1. `git tag -d submission`
2. `git push origin :refs/tags/submission`

and then go through the tagging process again with a different commit hash.

Step 4: Validation At this point, you should do a quick sanity check to ensure everything above has been done correctly. You should:

1. Check your student number is assigned to the right github account;
2. Check that your repository:

   (a) has a tag and that the tag is called "submission" (not upper case or anything else):

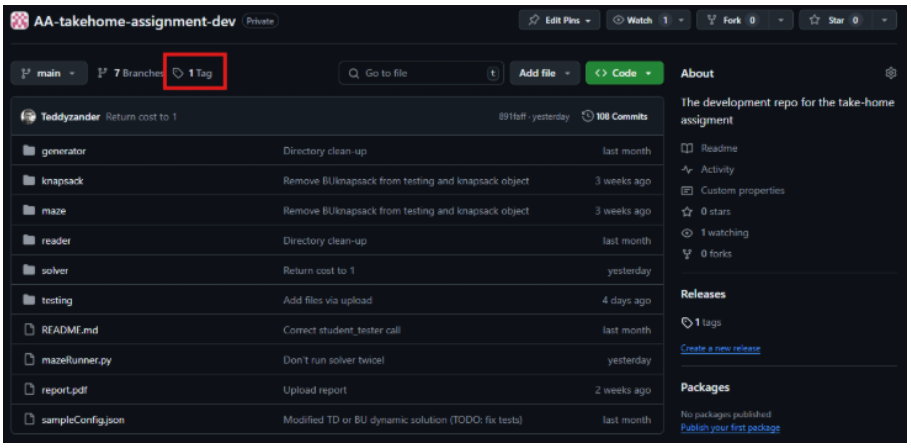

Figure 7: State of a repository when it has a tag. Observe that the red box shows us we have 1 tag. Clicking it will take us to the tags.

   (b) and going into tags and clicking on the submission tag takes you to your submission folder:

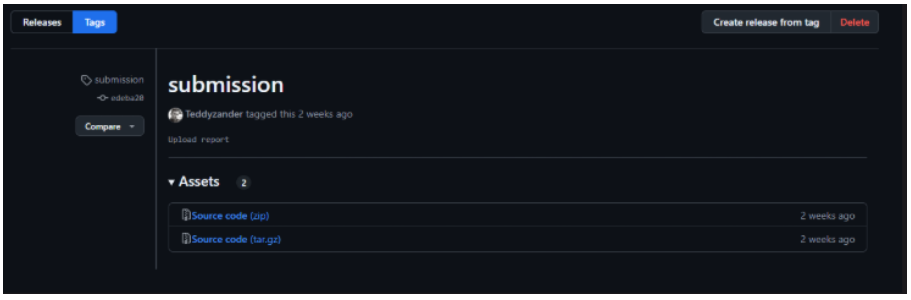

Figure 8: Your submission tag contains a source file (the state of your code at the tagged commit). This is what will be downloaded and marked.

3. Download the source file and check that:

   (a) the source file contains the report you want marked; and

   (b) the source file contains the code you want marked.

Step 5: Confirm Submission Your final step is to fill in this form that confirms you have completed the above tasks. During this process, you will sign-off that you have checked that all the above has been completed correctly.

LINK TO THE FORM

You are responsible for resolving any issues with your assignment, though please do talk to us so we can assist you with resolving them.

5.2 Late Submission Penalty

Late submissions will incur a 10% penalty on the total marks of the corresponding assessment task per day or part of day late, i.e, 3 marks per day. Submissions that are late by 5 days or more are not accepted and will be awarded zero, unless special consideration has been granted. Please ensure your submission is correct as re-submissions after the due date will be considered as late submissions. We strongly advice you submit **at least one hour before the deadline**. Late submissions due to slow Internet will not be looked upon favorably, even if it is a few minutes late. Any claim of late submission due to slow Internet will require documentation and evidence that submission attempts were made at least **one hour before the deadline**.

6 Academic integrity and plagiarism (standard warning)

Academic integrity is about honest presentation of your academic work. It means acknowledging the work of others while developing your own insights, knowledge and ideas. You should take extreme care that you have:

- Acknowledged words, data, diagrams, models, frameworks and/or ideas of others you have quoted (i.e. directly copied), summarised, paraphrased, discussed or mentioned in your assessment through the appropriate referencing methods.
- Provided a reference list of the publication details so your reader can locate the source if necessary. This includes material taken from Internet sites. If you do not acknowledge the sources of your material, you may be accused of plagiarism because you have passed off the work and ideas of another source without appropriate referencing, as if they were your own.

RMIT University treats plagiarism as a very serious offense constituting misconduct. Plagiarism covers a variety of inappropriate behaviours, including:

- Failure to properly document a source
- Copyright material from the internet or databases
- Collusion between students

For further information on our policies and procedures, please refer to this link.

7 Getting Help

There are multiple venues to get help. There are two weekly consultation hours (see Canvas or the Ed Forum for time and location details). In addition, you are encouraged to discuss any issues you have with your Tutor. We will also be posting common questions on the Ed Forum and we encourage you to check and participate in the discussion. However, please **refrain from posting solutions**, particularly as this assignment is focused on algorithmic design. Please carefully read the FAQ page on the Forum Etiquette or download it here to make sure you do not mistakenly post something that would be considered inappropriate.

8 Assessment

The project will be marked out of 30. The assessment in this project will be broken down into several parts for each Task. The criteria discussed in Table 1 will be considered when allocating marks.

Table 1 - Assessment Criteria

Criteria	Ratings	Pts
Task A Implementation Automatic Testing Results	**4 to 0 pts** Number of passed tests normalized by appropriate software engineering practices (subject to valid implementation)	4 pts
Task A Implementation Quality of Implementation	**1 to 0 pts** Valid, correct and clean implementation	1 pt
Task B Implementation Automatic Testing Results	**4 to 0 pts** Number of passed tests normalized by appropriate software engineering practices (subject to valid implementation)	4 pts
Task B Implementation Quality of Implementation	**1 to 0 pt** Valid, correct and clean implementation	1 pt
Task B Report	**Complete - 2 to 1.5 pts** The report is well-organized and clearly answers the questions. Description of the algorithm and the steps are clear and sound. The discussion about the complexity analysis of the proposed solution is clear, sound and well-justified against the pseudo-code. **Satisfactory - 1.5 to 1 pts** The report is generally understandable but may have some unclear sections. The algorithm is provided but it includes ambiguous steps. The complexity analysis is provided but it is not clear or sound at times.	2 pts

	Incomplete - 1 to 0 pts The report does not include a proper discussion regarding the algorithm. The complexity analysis is not provided or is not accurate.	
Task C Report	**Complete - 8 to 6 pts** The report is well-organized and all four aspects are comprehensively addressed: (i) complexity analysis is correct and the justification is clear and sound, (ii) the design of the test is sound and clearly communicated. The inclusion and exclusion of all parameters are well-justified, (iii) the results of the empirical analysis are presented and communicated clearly, and (iv) the comparison between theoretical and empirical analysis is clear and sound. **Satisfactory - 6 to 4 pts** The report is generally understandable but may have some unclear sections in regards to one or two of the listed aspects above. **Incomplete - 4 to 0 pts** The report does not include a proper discussion on the empirical design and/or the empirical analysis lacks depth and proper justification. The theoretical analysis is incorrect and/or not well-explained and the report lacks a sound comparison between the theoretical and empirical analyses.	8 pts
Task D Implementation Automatic Testing Results	The marking will be awarded following a comparison with our solution using the formula: $$\text{mark} = \frac{\text{tests passed}}{\text{number of tests}} \cdot \text{solution optimality} \cdot 4$$ where solution optimality, $\in [\frac{1}{4}, 1]$. All below bands are determined based on the number of passed tests normalized by appropriate software engineering practices (subject to valid implementation): **Efficient & Optimal solution - 4 pts** the solution achieves comparable net rewards compared to our solution **Efficient but sub-optimal solution - 3 pts** the solution is close to optimality, but occasionally gives sub-optimal solutions	4 pts

	Inefficient but not trivial solution - 2 pts the solution explores more number of cells mostly explore sub-optimally, but solution is better than base-line **Inefficient & sub-optimal solution - 1 pts** the solution is valid, but it is near base-line implementation **Solution not valid - 0 pts** the implementation gives back a path that does not satisfy as a solution to the problem	
Task D Implementation Quality of Implementation	**1 to 0 pts** Valid, correct and clean implementation. All cells are explored.	1 pt
Task D Report	**Complete - 5 to 3.5 pts** A well-articulated exploration and selection strategy is provided which includes detailed and correct pseudo-code. Clear explanation of how the algorithm balances clone cost and traversal as well as a thorough and correct complexity analysis with respect to input size is provided. The report clearly discusses how the changes in to clone life-span and cycles would impact the strategy and performance of the algorithm and the provided discussion is sound and relevant to the proposed design. **Satisfactory - 3.5 to 2 pts** Strategy is explained but lacks depth or precision. Pseudo-code and complexity analysis are present but may have minor errors or be incomplete. Some discussion regarding the impact of clone life-span and cycles on the algorithm is provided, but it lacks depth and/or is not clearly relevant to the proposed design. **Incomplete - 2 to 0 pts** The report provides no clear strategy presented, or lacks logical structure. Pseudo-code is missing or significantly flawed. Complexity analysis has incorrect reasoning and no meaningful discussion of how assumptions affect the algorithm is provided.	5 pts