RMIT UNIVERSITY

Algorithms and Analysis
COSC 1285/2123
Assignment 2: Exploring Maze Generation and Solving Algorithms

Assessment Type	Individual assignment. Submit online via Canvas → Assignments → Assignment 2. Marks awarded for meeting requirements as closely as possible. Clarifications/updates may be made via announcements/assignment FAQ/relevant discussion forums.
Due Date	Week 12, October 18, Friday, 11:59pm
Marks	30

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 plagiarise 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*. Remember you are meant to develop the solution by yourself - assessment is designed to encourage everyone to learn, and you are not doing yourself any favours by taking shortcuts. Suspected cases of collusion or plagiarism will be dealt with according to RMIT Academic integrity policy.

In the submission (your PDF file for the report of Task B) 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":*

Clarification to Specifications

Please periodically check the assignment FAQ for further clarifications about specifications. In addition, the lecturer will go through different aspects of the assignment each week, so even if you cannot make it to the lectorials, be sure to check the course material page on Canvas to see if there are additional notes posted.

1 Overview

In this assignment, you will implement a maze generator and solver that handle *weighted mazes*. You will implement specific tasks focused on generating and solving the maze using different approaches, including both optimal and heuristic-based solutions.

2 Learning Outcomes

This assessment relates to two 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 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.

3 Background

In Assignment 1, we explored 2D mazes and implemented two data structures to represent and store them. In this assignment, we will extend our work to weighted mazes. Instead of considering uniform cell weights, where every cell has the same value, we will introduce two distinct weighting strategies: random weighting and checkered weighting. These weighted mazes will influence how paths are generated and solved, as each path's cost will now depend on the sum of edge weights.
In our weighted maze, each cell is assigned a weight, representing the difficulty level to traverse that cell. You can think of it as travelling through different types of terrain in a game: moving between cells with the same weight attracts no additional cost, but switching between cells of different weights requires a higher cost, reflecting the difficulty of transitioning between different terrains.
Figure 1 shows two examples of weighted mazes corresponding to two different weighting approach. For simplicity, weighted mazes will have cell weights ranging from 1 to 4, depending on the strategy used:

- Random Weighting (Figure 1, left): Each cell is randomly assigned a weight between 1 and 4.
- Checkered Weighting (Figure 1, right): Cell weights follow a checkered pattern similar to a chessboard. However, instead of alternating between two weights (as in a traditional chessboard), we will use four distinct weights to create a more complex pattern.

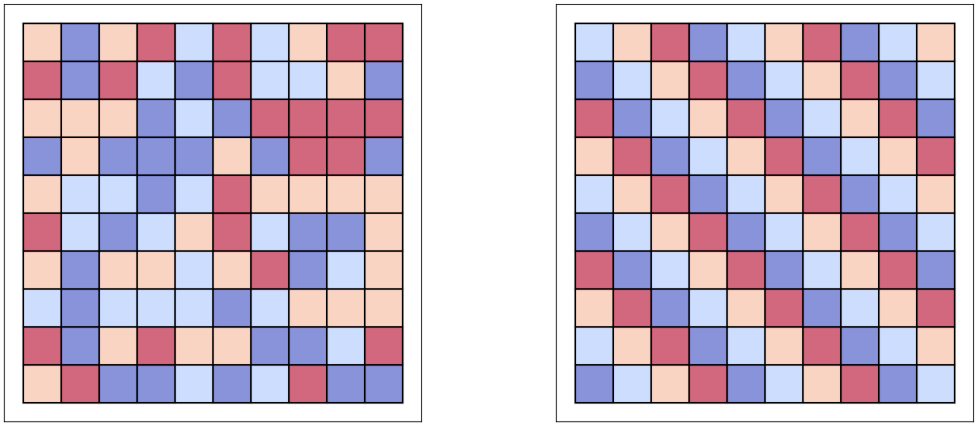

Figure 1: Two weighted mazes where the weights are generated by either the *random* approach (left) or the *checkered* approach. The cell colours correspond to the weight of the cell ranging from dark blue (1) to red (4).

In both cases, boundary cells (representing walls or buffer zones around the maze) will be assigned a weight of 0.
Figure 2 shows the edge weights for the first (bottom) row of the random weights presented in Figure 1. The objective remains to find a path from an entrance to an exit, ensuring that the paths adhere to the constraints imposed by the maze's weights. Weighted mazes offer additional challenges, as paths with fewer steps may not always be the optimal solution due to higher cell weights. The cost of a path in our set up is the sum of the *edge weights* between the cells on that path, and the goal is to minimize this total cost.
To facilitate navigation in the maze, entrances and exits are defined in the same way as in Assignment 1. Also, similar to Assignment 1, we will focus on *perfect mazes* where there is always a path from the entrance to the exit, ensuring full connectivity between all cells. Our aim is to develop and implement both *generation* and *solving* algorithms for these weighted mazes.

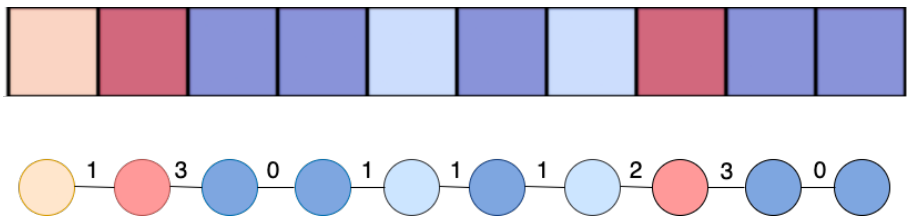

Figure 2: The vertices and edges representing the partial maze shown above. Edge weights capture the absolute difference between the weights of adjacent cells. The total cost of a path from the leftmost to the rightmost cell (or vice versa) is the sum of these edge weights, i.e., $1 + 3 + 0 + 1 + 1 + 1 + 2 + 3 + 0 = 12$.

4 Assessment details

The assignment is broken up into a number of tasks, 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. Except for the empirical analysis section of Task X, no task is dependent on the completion of another.

Task A: Maze Generation and Solver Implementation (8 marks)

To gain a better understanding of how weighted mazes are generated and solved, you will implement a maze generator and a maze solver in this Task. We also provide an implementation for you to study the problem, and to allow you to progress with subsequent steps if you cannot get your own maze generator and solver working. See the provided skeleton code for additional information on how we implemented the Recursive Backtracking maze generator and solver. You are required to understand and implement a weighted maze generator using Kruskal's algorithm and a weighted maze solver using Dijkstra's algorithm:

Maze Generators

- **Recursive BackTracking Maze Generator (provided)** The recursive backtracking maze generator is essentially a DFS generator. Starting with a maze with walls between all adjacent pairs of cells, it randomly selects a cell in the maze, and perform a DFS traversal of the whole maze, from that initial cell. During the DFS traversal, we generate all the unvisited neighbours of the current cell, and select the closest neighbour to the current cell (the neighbour with the least distance in cell weights). When we go to an unvisited cell, we remove/destroy the wall between the current cell to the selected unvisited neighbouring cell. If there is no unvisited neighbouring cell from our current one, we backtrack to the cell we were at previously. The DFS ends when we have visited all the cells in the maze.

- **Kruskal's Maze Generator (required)** This algorithm also starts with a maze with walls between all adjacent pairs of cells, and whilst for an unweighted maze the algorithm would treat the walls equally, in a weighted maze the walls need to be ordered based on their edge weights in an ascending manner. The algorithm then processes each wall in the order, checking if removing the wall would connect two previously disconnected sections of the maze (using a union-find data structure to manage the connected components). If removing the wall connects two distinct sections, the wall is removed, and the two sections are merged into a single connected component. If removing the wall does not connect two sections, it is left in place. The union-find data structure will be briefly discussed during the lectorials to help you with this task.

Maze Solvers

- **Recursive BackTracking Maze Solver (provided)** The recursive backtracking maze solver is the equivalent of the recursive backtracking maze generator. It starts at an entrance, and from the current cell, finds the closest unvisited neighbouring cell and go there (and mark it visited). Then it continues until either it reaches an

exit, or a dead end, by which then it backtracks to a cell where there is at least one unvisited neighbour.

- **Dijkstra's Maze Solver (required)** Dijkstra's solver starts by initialising the distance at the entrance to 0 and infinity to all other cells. Using a priority queue (remember our discussion on heaps?), the algorithm iteratively explores the neighboring cells with the smallest distance. For each cell, it updates the distances to its neighboring cells based on the sum of the current cell's distance to the neighbor, i.e., the difference between the current cell's weight and the weight of its neighbours. If a shorter path to a neighbor is found, the distance is updated, and the neighbor is added to the priority queue for further exploration. This process continues until the algorithm reaches an exit or all reachable cells (including other exits) have been processed.

Task B: Brute Force Solver for All Entrance-Exit Pairs (6 marks)

In this task, you are required to devise and implement a brute force algorithm to find non-overlapping paths in a weighted maze. The goal is to compute multiple paths from a set of entrances to a corresponding set of exits, ensuring that the paths do not overlap, i.e., no cell is used by more than one path. In this task You will need to check all possible combinations of paths between the entrance-exit pairs to find a valid set of non-overlapping paths and if there are multiple sets of such paths, report the option with the shortest total distance.
To help you get started, select one of the solvers, either provided or implemented by you, and solve the problem where we have one entrance and exit pairs. Once you solved this, then extend this to the case where there are multiple entrance-exit pairs. To help us understand your solution, write a report of up to one page as part of this task, in addition to your implementation.

Task C: Greedy Solver for All Entrance-Exit Pairs (8 marks)

In this task, you are required to design and implement a greedy algorithm to compute non-overlapping paths in a weighted maze. The goal is to find paths from a set of entrances to a corresponding set of exits, ensuring that the paths do not overlap (i.e., no cell is used by more than one path). The algorithm should incorporate heuristics to guide the selection of paths. You are free to choose and justify the heuristics that you think will yield the best results.
Along with the implementation, include a maximum 1-page report to discuss and justify the heuristics you used and why you believe they will lead to efficient path-finding when solving all entrance-exit pairs in a maze.

Assumptions and Considerations for Task B & C

- One-to-One Mapping of Entrances and Exits: The number of entrances is equal to the number of exits. For simplicity, assume that each entrance is paired with a corresponding exit in the same order they are provided in the configuration files.

- The set of paths that meet the non-overlapping criterion may not be the shortest path available between the corresponding entrance-exit pairs. We are interested in

minimising the total cost for all paths instead of finding the shortest path for each pair.

- Perfect Maze Assumption: Since we are working with perfect mazes, it is guaranteed that a path exists from each entrance to each exit. However, it is not guaranteed that a valid solution with non-overlapping paths will always exist.
- Handling No Solution: If it is impossible to find a set of non-overlapping paths between all the entrance-exit pairs, your program should return False, indicating that no valid solution exists (see the code skeleton for more details).
- Efficiency Considerations: Given the brute force nature of the task, the algorithm will be slow for larger mazes. In this assignment we will only consider up to 10 by 10 mazes with maximum 3 entrance-exit pairs.

Task D: Performance Evaluation and Comparison (6 marks)

In this task, you are required to write a report of up to two pages and evaluate the performance of your greedy algorithm in Task C compared to the brute force solution developed in Task B. The objective is to assess both algorithms in terms of time efficiency and correctness.

To complete this task, you should first analyze how each algorithm performs with respect to time. This involves measuring how long each algorithm takes to find solutions as the maze size and complexity (in terms of the number of entrance-exit pairs). You should test your mazes, varying the cell weights and generators if possible, and document the time differences observed. You will need to also compare the total path costs produced by the greedy algorithm and the brute force approach, and determine whether the greedy method results in solutions that are close to those found by brute force.

Whether or not you have implemented the solver in Task C and/or B, provide a detailed discussion outlining the expected time complexity of the devised algorithms for each task as well as how you think the two approaches (brute force and greedy) would scale with larger mazes.

Your solutions will be tested against our own set of mazes, and marks will be awarded based on a ranking system. This ranking will consider both the correctness and efficiency of your solution. The more accurately and quickly your algorithms perform compared to ours, the higher your score will be for this task. More details will be shared later in the lectorial sessions.

Task E: Recorded Interview (2 marks)

After your implementations and report are submitted, you will be asked to record your responses to a number of questions in Canvas. These questions will ask you about aspects of your implementation or report. You'll have a set time to consider the questions, make a recording then upload that recording.

Code Framework

We provide Python skeleton code (see Table 1) to help you get started and ensure we have consistent interfacing so we can have consist correctness testing.

We also listed the files that you really need to modify/implement. Unlike Assignment 1, there are more files you need to implement, hence see Table 1 to confirm those.

Notes

- If you focus on the parts with "TODO" and/or "Please implement" parts of the provided skeleton, you in fact do not need to do anything else to get the correct output formatting. mazeTester2.py will handle this.
- We will run your implementation on the university's core teaching servers, e.g., `titan.csit.rmit.edu.au`, `jupiter.csit.rmit.edu.au`, and `saturn.csit.rmit.edu.au`. If you develop on your own machines, please ensure your code compiles and runs on these machines. You don't want to find out last minute that your code doesn't compile on these machines. If your code doesn't run on these machines, we unfortunately do not have the resources to debug each one and cannot award marks for testing. *Please note this carefully, this is a firm requirement.*
- All submissions should be compiled with no warnings on **Python 3.6.8** when compiling the files specified in Table 1 - this is the default Python3 version on the Core teaching servers. Please ensure your code runs on the core teaching servers and with this version of Python. *Please note this carefully, this is a firm requirement.*

5 Report Structure

As a guide, the report could contain the following sections:

- Your solution for Task B, and the rationale or justification behind it. (up to 1 page in length)
- Your solution for Task C, and the rationale or justification behind it. (up to 1 page in length)
- A detailed reflection in Task B and Task C algorithms. (up to 2 pages in length)
- You can also have an appendix, which doesn't count towards the overall page count.

6 Submission

The final submission will consist of three parts:

1. Your **Python source code** of your implementations. This must include all files, including the provided skeleton as well as any extra files you have created. Your source code should be placed into the same structure as the supplied skeleton code, and the root directory/folder should be named as `Assign2-<your student number>`. Specifically, if your student number is s12345, when `unzip Assign2-s12345.zip` is executed then all the source code files should be in directory Assign2-s12345. We use automated testing and compilation, and the testing script will expect this structure, so if is different, the script may not be able to compile your code. So please make sure not to change the structure.

2. Your **written report for Tasks B, C and D** in a single PDF file, called "s12345-assign2.pdf" if your student number is s12345. Separately submit this to another submission location, for Turnitin checking.

3. The Python source file folder (and files within) should be zipped up and named as `Assign2-<your student number>.zip`. E.g., if your student numbers is s12345, then your code submission file should be called `Assign2-s12345.zip`, and when we unzip that zip file, then all the submission files should be in the folder Assign2-s12345.

Note: **submission of the report and code will be done via Canvas**. We will provide details closer to the submission deadline.

7 Assessment

The project will be marked out of 30. The assessment in this project will be broken down into three parts. The following criteria will be considered when allocating marks.

Tasks A (8/30):

- Your implementation will be assessed based on the number of tests it passes in our automated testing. The tests will assess things such as whether the generated graph is perfect, whether a provided maze is solved, whether it has similar traces, when using the same random number seeds as our implementations of the algorithms.
- Your implementation will also be assessed whether it implements the requested algorithms.
- While the emphasis of this project is not programming, we would like you to maintain decent coding design, readability and commenting, hence commenting and coding style will make up a portion of your marks.

Task B & C (14/30):

- The clarify and soundness of the description of your algorithmic solution, and the rationale/justification of the approach in the report. This is important for us to understand your reasoning. We highly recommend attempting this element even if you do not manage to implement your solution.
- Your implementation will be partially assessed based on a number of tests in our automated testing. The tests will assess things such as whether the correct set of paths with minimum total cost is returned.
- While the emphasis of this project is not programming, we would like you to maintain decent coding design, readability and commenting, hence commenting and coding style will make up a portion of your marks.

Task D (**8/30**):

- The clarity and soundness of the discussions on the theoretical analysis of the algorithms devised in Tasks B and C, as well as the clarity and completeness of the explanation on the experimental setup and evaluation process, including clear and well-reasoned reflections on the results.
- Your implementation will be partially assessed based on its performance in terms of correctness and efficiency when compared to our solution.

Task E **2/30**:

- This is a pass/fail assessment, and you'll be assessed based on your ability to answer some questions about your implement and report.

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. Granted Special Considerations with new due date set after the results have been released (typically 2 weeks after the deadline) will automatically result in an equivalent assessment in the form of a practical test, assessing the same knowledge and skills of the assignment (location and time to be arranged by the coordinator). Please ensure your submission is correct (all files are there, compiles etc), re-submissions after the due date and time will be considered as late submissions. The core teaching servers and Canvas can be slow, so please do double check ensure you have your assignments done and submitted a little before the submission deadline to avoid submitting late. We strongly advice you submit **at least one hour before the deadline**. *Late submissions due to slow processing of Canvas or slow Internet will not be looked upon favourly, even if it is a few minutes late. Slow processing of Canvas or slow Internet will require documentation and evidence submission attempts was made at least one hour before the deadline.*

Assessment declaration: By submitting this assessment, you agree to the assessment declaration - `https://www.rmit.edu.au/students/student-essentials/assessment-and-exams/assessment/assessment-declaration`

8 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 person without appropriate referencing, as if they were your own.

RMIT University treats plagiarism as a very serious offence 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 the following: `https://www.rmit.edu.au/students/student-essentials/rights-and-responsibilities/academic-integrity`.

9 Getting Help

There are multiple venues to get help. There are weekly consultation hours (see Canvas 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 Assignment 2 Q&A section on Canvas and we encourage you to check and participate in the EdForum discussion forum. However, please **refrain from posting solutions**, particularly as this assignment is focused on algorithmic and data structure design.

file	description
`mazeTester2.py`	Code that reads a configuration file and runs the appropriate generator and solver. *No need to modify this file.*
`maze/maze.py`	Implementation of the weighted maze. *No need to modify this file.*
`maze/util.py`	Coordinates class, and other utlity classes and methods. *No need to modify this file.*
`maze/graph.py`	Abstract class for graphs. All graph implementations should implement the Graph class (same as assignment 1). *No need to modify this file.*
`maze/edgeListGraph.py`	Code that implements an edge list (for maze) *No need to modify this file.*
`maze/maze_viz.py`	Modified code used to visualise generated mazes and paths using matplotlib. *No need to modify this file.*
`generator/mazeGenerator.py`	Abstract class for maze generators. *No need to modify this file.*
`generator/recurBackGenerator.py`	Implementation of the recursive backtracking maze generator. *No need to modify this file.*
`generator/kruskalGenerator.py`	Implementation of the Kruskal's generator algorithm. Used for Task A. *Please complete implementation.*
`solver/mazeSolver.py`	Abstract class for maze solvers. *No need to modify this file.*
`solver/recurBackSolver.py`	Implementation of the recursive backtracking maze solver. *No need to modify this file.*
`solver/dijkstraSolver.py`	Implementation of the Dijkstra's algorithm solver. Used for Task A. *Please complete implementation.*
`solver/allPairsSolvers.py`	Abstract class for maze solvers that find non-overlapping solutions for all entrance-exit pairs. *No need to modify this file.*
`solver/taskBSolver.py`	Implementation of the brute force solver for Task B. *Please complete implementation.*
`solver/taskCSolver.py`	Implementation of the greedy solver for Task C. *Please complete implementation.*
`README.txt`	Please read this first, it mentions how to run the code. *No need to modify this file.*

Table 1: Table of provided Python skeleton files.