Algorithms and Analysis

COSC 2123/1285

Assignment 2: Maze Generators and Solvers

	Assessment Type	Pair (Group of 2) Assignment.
	Due Date	Week 12, 11:59pm, 20th of Oct, 2019
	Marks	15

1 Overview

Mazes have a long history and many of us would remember seeing our first mazes in a puzzle book and trying to trace a path from entrance to exit. Maze solving seeks to find a path to the exit(s), either from a set of entrance(s), or from somewhere inside the maze. Maze generation, conversely, seeks to use algorithms to construct mazes.

In this assignment, you will explore and implement algorithm(s) to generate and solve mazes. We first provide some background about maze types, maze generation and maze solving algorithms.

1.1 Mazes

There are multiple types of mazes. In this assignment, we are interested in 2D, rectangular, perfect mazes. Mazes are made up of a number of cells and walls (see Figure 1 for an example cell with four walls). 2D mazes are layout on a 2d plane. Rectangular mazes have rectangular cells with 4 sides, where each side can have a wall, see Figure 2a for an illustration. Perfect mazes are ones where every cell in the maze can be potentially visited and have no loops, see Figure 2 for examples. In this assignment, we additionally introduce tunnelled mazes. These mazes have a number of tunnels, which link non-adjacent cells, see Figure 2b. If we enter one end of the tunnel, we will exit at the other end of the tunnel.

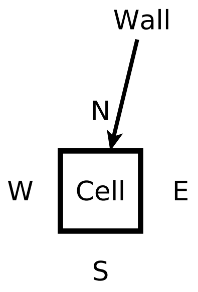
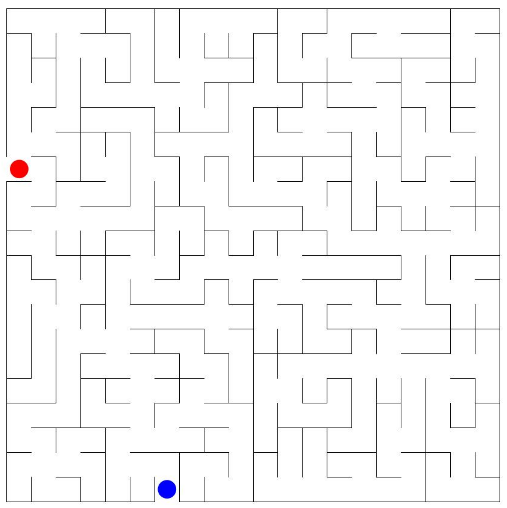


Figure 1: A rectangular cell, with four walls and the four directions of North (N), East (E), South (S), West (W).


(a) A 20 by 20 rectangular maze.

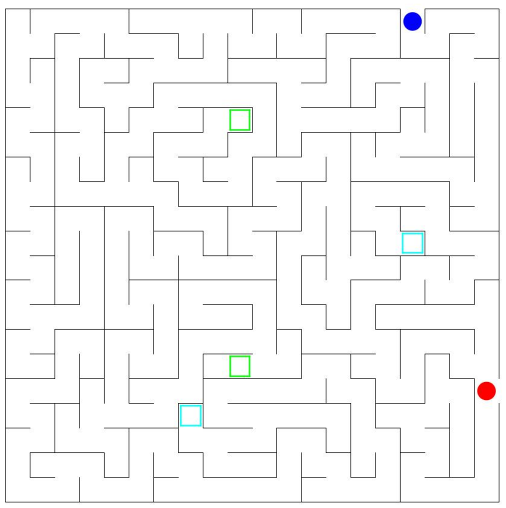

(b) A 20 by 20 rectangular maze with 2 tunnels, with tunnel entry points drawn in coloured squares (squares with same colour are entries of the same tunnel).

Figure 2: Perfect mazes with entrance specified with blue dot and exit with red dot.

1.2 Maze Generation Algorithms

There are many maze generation algorithms, each generating mazes of different characteristics. We focus on the following two algorithms.

1.2.1 Hunt and Kill

Starting with a maze where all walls are present, i.e., between every two adjacent cells is a wall, the algorithm uses the following procedure to generate a maze:

1. Pick a random starting cell.
2. Randomly select an unvisited neighbouring cell and carve a passage to the neighbour. Repeating this until the current cell has no unvisited neighbours.
3. Enter the "hunt" mode, where the algorithm scans the grid searching for an unvisited cell that is adjacent to a visited cell. If found, carve a passage between the two and let the formerly unvisited cell be the new starting location.
4. Repeat steps 2 and 3 until the hunt mode scans the entire grid and finds no unvisited cells.

Following the following link for more details http://weblog.jamisbuck.org/2011/1/24/maze-generation-hunt-and-kill-algorithm.

1.2.2 Kruskal's

This generator is based on Kruskal's algorithm for computing minimum spanning trees (hence the name of this generator). Starting with a maze where all walls are present, i.e., between every two adjacent cells is a wall, it uses the following procedure to generate a maze:

1. For each pair of adjacent cells, construct an edge that links these two cells. Put all possible edges into a set. Maintain a set of trees. Initially this set of trees contains a set of singleton tree, where each tree has one of the cells (its index) as it only vertex.
2. Select a random edge in the set. If the edge joins two disjoint trees in the tree set, join the trees. If not, disgard the edge. When a selected edge joins two trees, carve a path between the two corresponding cells.
3. Repeat step 2 until the set of edges is empty. When this occurs, then the maze will be a perfect one.

1.3 Maze Solving Algorithm

Similar to generation algorithms, there have been many solvers proposed. In this assignment we focus on only one.

1.3.1 Recursive Backtracker

This solver is basically a DFS. Starting at the entrance of the maze, the solver will initially randomly choose an adjacent unvisited cell. It moves to that cell, update its visit status, then selects another random unvisited neighbour. It continues this process until it hits a deadend (no unvisited neighbours), then it backtracks to a previous cell that has an unvisited neighbour. Randomly select one of the unvisited neighbour and repeat process until we reached the exit (this is always possible for a perfect maze). The path from entrance to exit is the solution.

2 Tasks

The project is broken up into a number of tasks.

Task A: Implement Maze Generators (5 marks)

To explore the different approaches to maze generation, in this task, you will implement twp different maze generation algorithms for the different maze types. The maze generators to implement are:

- Hunt and Kill
- Kruskal's

The types of mazes we want to generate with these algorithms are as follows:

- 2D rectangular perfect maze with no tunnels (type A)
- 2D rectangular perfect maze with tunnels (type B)

You need to implement the Hunt and Kill and Kruskal algorithms to generate all two types of mazes.

Generation Algorithm	Maze types
Hunt and Kill	A, B
Kruskal's	A, B

Task B: Implement Maze Solver (5 marks)

In this task, you are to implement one solver: the recursive backtracker solver. It should be able to solve the two types of mazes.

Details for both tasks

As explained above, your task is to implement the two generation and one solving approaches for different types of mazes. Which maze type, generator and solver will be specified in a parameter setting file.

To help you get started, you are provided with skeleton code that implements the data structures for the two types of mazes. In addition, we provide a main class (MazeTester) which parses parameters, specified in a file, and correctly call the specified generator, optionally visualise the generated maze and optionally call the specified solver on the maze. The list of files provided are listed in Table 1.

file	**description**
`MazeTester.java`	Code that reads in parameter file then uses the specified algorithms to generate, visualise and solve mazes. Has options to visualise the maze and the cell visited by a solver. *No need to modify this file*.
`Maze.java`	Interface class for maze. *No need to modify this file*.
`Cell.java`	Class implementing a maze cell. *No need to modify this file*.
`Wall.java`	Class implementing a maze wall. *No need to modify this file*.
`NormalMaze.java`	Class that extends the Maze interface to implement a 2D rectangular maze (with no tunnels). *No need to modify this file*.
`TunnelMaze.java`	Class that extends the Maze interface to implement a 2D rectangular maze with tunnels. *No need to modify this file*.
`StdDraw.java`	Helper class to visualise maze. *No need to modify this file*.
`MazeGenerator.java`	Interface class for maze generation algorithms. *No need to modify this file*
`HuntAndKillGenerator.java`	Class implements a Hunt and Kill maze generator. *Complete the implementation of this file.*
`KruskalsGenerator.java`	Class implements a kruskal's algorithm generator. *Complete the implementation of this file.*
`MazeSolver.java`	Interface class for maze solving algorithms. *No need to modify this file*
`RecursiveBacktrackerSolver.java`	Class implements a recursive backtracking maze solver. *Complete the implementation of this file.*

Table 1: Table of supplied Java files.

Note, please do not modify MazeTester class at all, as this contains some of the evaluation code, hence we will be using our copy to do the evaluation and any changes will not be included. We also strongly suggest not to modify any of the "No need to modify" files. You may add methods and java files, but it should be within the structure of the skeleton code, i.e., keep the same directory structure. Similar to assignment 1, this is to minimise compiling and running issues. Note that the onus is on you to ensure correct compilation on the core teaching servers.

As a friendly reminder, remember how packages work and IDE like Eclipse will automatically add the package qualifiers to files created in their environments.

Compiling and Executing

To compile the files, run the following command from the root directory (the directory that MazeTester.java is in):

```
javac -cp .:mazeSolver/SampleSolver.jar *.java
```

To run the maze visualisation and evaluation algorithm, run the following command:

```
java -cp .:mazeSolver/SampleSolver.jar MazeTester <parameter file> <visualise maze and solver>
```

where

- parameter file: name of the file that contains the parameter specifications of the run.
- visualise maze and solver path: [y/n], to indicate whether to visualise (y) or not (n).

The jar file contains bytecode for a solver we have already coded up. This is to help you see a solver in action on your maze (before you have implemented your own solvers).

We now describe the contents of the parameter file.

Parameters

The parameter file specifies all the settings for the maze generation, evaluation and visualisation. This file has the followng format:

```
mazeType
generatorName solverName
numRow numCol
entranceRow entranceCol
exitRow exitCol
tunnelList
```

Where parameters are as follows:

- mazeType: The type of maze to generate, where it should be one of {normal, tunnel}.
- generatorName: Name of maze generation algorithm, where it should be one of {huntandkill, kruskal}. "huntandkill" is Hunt and Kill algorithm and "kruskal" is Kruskal's algorithm.
- solverName: Name of the maze generation algorithm, where it should be one of {recurBack, sample, none}. "recurBack" is recursive backtracker solver, "sample" is a sample solver for you to visualise solving and "none" is to specify that no solving is wanted.

- numRow: Number of rows in the generated maze, it should be a positive integer (1 or greater).
- numCol: Number of columns in the generated maze, it should be a positive integer (1 or greater).
- entranceRow: the row of the entrance cell, should be in range [0, numRow-1].
- entranceCol: the column of the entrance cell, should be in range [0, numCol-1].
- exitRow: the row of the exit cell, should be in range [0, numRow-1].
- exitCol: the column of the exit cell, should be in range [0, numCol-1].

For tunnel mazes, we have additional parameters (the non-tunnelled mazes will ignore these parameter settings if they are specified):

- tunnelList: list of tunnels, one per line. Each line consists of the (row, column) of the cells on each end of the tunnel. For example, "a b c d" means one end of the tunnel is at (a, b) while the other end is (c, d). Each row must be in range [0, numRow-1], each column in range [0, numCol-1].

An example parameter file is as follows:

```
normal
kruskal recurBack
30 30
2 0
0 0
```

This specifies the following parameters:

- Generating rectangular mazes
- Using the Kruskal's algorithm for generation
- Using recursive backtracker for solving
- The maze has 30 rows and 30 columns (30 by 30 maze)
- Entrance is located at cell (2,0).
- Exit is located at cell (0,0).

An example parameter file for tunnel mazes.

```
tunnel
huntandkill sample
50 50
0 5
49 12
5 9 15 9
3 7 14 8
0 0 22 12
```

This specifies the following parameters:

- Generating (rectangular) tunnel mazes
- Using Hunt and Kill algorithm for generation
- Sample solver provided by us
- The maze has 50 rows and 50 columns (30 by 30 maze)
- Entrance is located at cell (0,5).
- Exit is located at cell (49,12).
- There are three tunnels.
- First tunnel has one end at (5,9) and the other end at (15,9)
- Second tunnel has one end at (3,7) and the other end at (14,8)
- Third tunnel has one end at (0,0) and the other end at (22,12)

Task C: Theoretical Analysis (5 marks)

Question 1. Solving Recursive Relations [3 mark]. A naive multiplication of two matrices of order n requires $O(n^3)$ ***additions***. By using a divide and conquer approach, Strassen devised another algorithm that requires $T(n)$ additions where

$$T(n) = 7T(n/2) + cn^2,$$

where c is a constant independent of n and $T(1) = 0$ (as multiplying two numbers requires no additions). Use the method of *backward substitution* (introduced in Week 2's lecture) to show that Strassen's algorithm requires $O(n^{\log_2 7}) \approx O(n^{2.81})$ additions, which is better than $O(n^3)$. *Provide all the steps of your proof* to get your mark (make sure you use $T(1) = 0$ in your proof). Note that there is the so-called *Master Theorem* to deal with a general recurrence relation of the type $T(n) = aT(n/b) + O(n^c)$, where a, b, c are constants (see Levitin's textbook Chapter 5). However, here it is required that you present the method of backward substitution to understand how a proof of the Master Theorem should work.

Question 2 [2 mark]. You are given the task of sorting a large array of size n. Moreover, the array only contains integers lying between 0 and n^2 (inclusive) and has many repeated values.

(a) Which algorithm would you recommend - *distribution sort* or *mergesort*? Justify your answer taking into account both *space* and *time complexities*.

(b) What if you know in addition that the array only contains *multiples* of n, that is, every element in the array belongs to the set $\{0, n, 2n, 3n, \dots, n^2\}$, which of the two aforementioned algorithms would you recommend? Justify your answer taking into account both *space* and *time complexities*.

3 Assessment

The project will be marked out of 15.

The assessment in this project will be broken down into three components. The following criteria will be considered when allocating marks.

Task A (5/15):

We will test your generator implementation by specifying different parameter settings and then evaluating their correctness.

Your maze generators will be evaluated on whether:

1. there is a connected path from the entrance to the exit of the maze, i.e., the maze is *solvable*.
2. the generated mazes are *perfect*, i.e., contains *no loops* and a solver can potentially visit *every* cells.
3. your code implements the particular algorithm in question. The emphasis is on you, via commenting and code design, to show this.
4. the entrance and exit are correctly generated.
5. for tunnel maze, whether your maze has the specified tunnels.

Task B (5/15):

Similar to task A, we will test your solver implementation by specifying different parameter settings and then evaluating their correctness.

Your maze solvers will be evaluated on whether:

1. the solver is able to find a path from entrance to exit for the input mazes that are of one of the three maze types.
2. whether your code implements the particular algorithm in question. The emphasis is on you, via commenting and code design, to show this.

Task C (5/15):

To get the full mark for this task, you should make sure to provide all the important steps in your solution to Question 1 and all necessary explanation in your solution to Question 2.

3.1 Late Submissions

Late submissions will incur a *deduction of 1.5 marks per day or part of day late*. Note, there will be **no leeway** given for late submissions, incorrect uploading etc. Please ensure your submission is correct, resubmissions after the due date and time will be considered as late submissions. Even if you are one minute late, it will be counted late. The core teaching servers and canvas can be slow, so please ensure you have your submission done a little before the submission deadline to avoid submitting late.

4 Submission

The final submission will consist of:

- Your Java source code of your implementations. We will provide details closer to submission date.
- Your solutions to Task C Question 1 and Question 2 in a PDF file of font size 12pt and at most *two* pages.
- Contribution sheet

Note: submission of the code and solutions will be done via Canvas.

5 Team Structure

This project should be done in **pairs** (group of two). If you have difficulty in finding a partner, post on the discussion forum or contact your lecturer. If there are issues with work division and workload in your group, please contact your lecture as soon as possible.

In addition, please submit what percentage each partner made to the assignment (a contribution sheet will be made available for you to fill in), and submit this sheet in your submission. The contributions of your group should add up to 100%. If the contribution percentages are not 50-50, the partner with less than 50% will have their marks reduced. Let student A has contribution X%, and student B has contribution Y%, and $X > Y$. The group is given a group mark of M. Student A will get M for assignment 1, but student B will get $\frac{M}{\frac{X}{Y}}$.

6 Plagiarism Policy

University Policy on Academic Honesty and Plagiarism: You are reminded that all submitted project work in this subject is to be the work of you and your partner. It should not be shared with other groups. Multiple automated similarity checking software will be used to compare submissions. It is University policy that cheating by students in any form is not permitted, and that work submitted for assessment purposes must be the independent work of the student(s) concerned. Plagiarism of any form will result in zero marks being given for this assessment, and can result in disciplinary action.

For more details, please see the policy at `http://rmit.info/browse;ID=sg4yfqzod48g1`.

7 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 or Lab Demonstrator. We will also be posting common questions on Canvas and we encourage you to check and participate in the discussion forum on Canvas. Although we encourage participation in the forums, please refrain from posting solutions.