COSC 2123/1285 Algorithms and Analysis
Semester 1, 2016

Assignment 2 Description

Due date: 11:59pm Sunday, May 29th 2016 **Weight:** 15%

Pairs Assignment

1 Objectives

There are three key objectives for this project:

- Design and implement a number of algorithms to generate 2D mazes.
- Design and implement a number of algorithms to solve these mazes.
- Have fun!

This assignment is designed to be completed in *groups of 2 or pairs*. We suggest that you work in pairs.

2 Background

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.

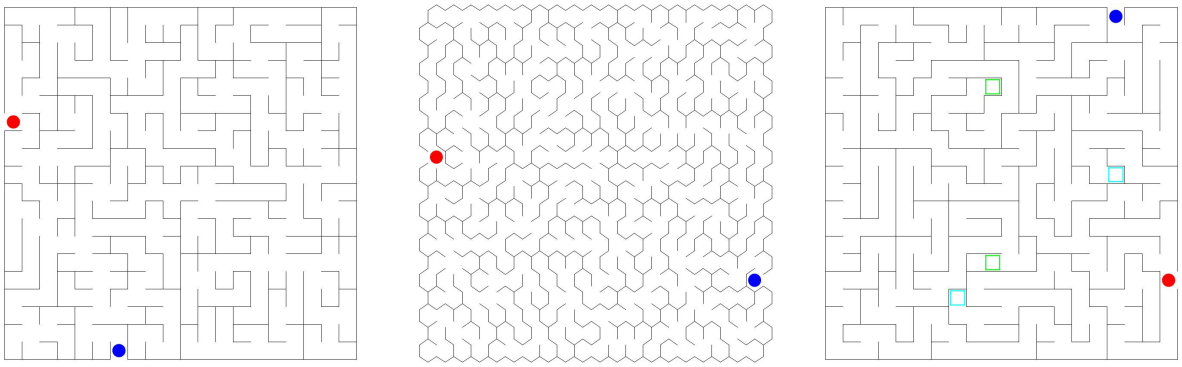
We first provide some background about maze types, maze generation and maze solving algorithms.

2.1 Mazes

There are multiple types of mazes. In this assignment, we are interested in 2D, rectangular and hexagon, perfect mazes. Mazes are made up of a number of cells and walls (see Figure 2 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 1a for an illustration. Hexagon mazes have hexagon cells with 6 sides, that can have 6 walls which are at 120 degree angle to their adjacent walls, see Figure 1b for an illustration. Perfect mazes are ones where every cell in the maze can be potentially visited and have no loops, see Figure 1 for examples. In this assignment, we additionally introduce tunnelled mazes. These mazes have a number of tunnels, which link non-adjacent cells, see Figure 1c. If we enter one end of the tunnel, we will exit at the other end of the tunnel.

2.2 Maze Generation Algorithms

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



(a) A 20 by 20 rectangular maze.

(b) A 20 by 20 hexagon maze.

(c) 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 1: Perfect mazes with entrance specified with blue dot and exit with red dot.

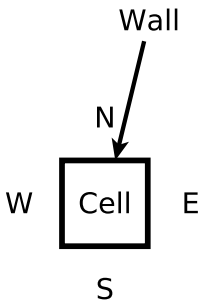


Figure 2: A rectangular cell, with four walls and the four directions of North (N), East (E), South (S), West (W).

2.2.1 Recursive Backtracker

This generator uses the DFS principle to generate mazes. Starting with a maze where all walls are present, i.e., between every cell is a wall, it uses the following procedure to generate a maze:

1. Randomly pick a starting cell.
2. Pick a random unvisited neighbouring cell and move to that neighbour. In the process, carve a path (i.e, remove the wall) between the cells.
3. Continue this process until we reach a cell that has no unvisited neighbours. In that case, backtrack one cell at a time, until we backtracked to a cell that has unvisited neighbours. Repeat step 2.
4. When there are no more unvisited neighbours for all cells, then every cell would have been visited and we have generated a perfect maze.

2.2.2 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 cell 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.

2.2.3 Prim's

This generator is based on Prim's algorithm for computing minimum spanning tree. We used the modified version of it. Starting with a maze where all walls are present, i.e., between every cell is a wall, it uses the following procedure to generate a maze:

1. Pick a random starting cell and add it to set Z (initially Z is empty, after addition it contains just the starting cell). Put all neighbouring cells of starting cell into the frontier set F.

2. Randomly select a cell c from the frontier set and remove it from F. Randomly select a cell b that is in Z and adjecent to the cell c. Carve a path between c and b.

3. Add cell c to the set Z. Add the neighbours of cell c to the frontier set Z.

4. Repeat step 2 until Z includes every cell in the maze. At the end of the process, we would have generated a perfect maze.

2.3 Maze Solving Algorithm

Similar to generation algorithms, there have been many solvers proposed. In this assignment we focus on the following two.

2.3.1 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.

2.3.2 Bidirectional BFS

This solver performs BFS searches starting at both the entrance and exits. When the two BFS fronts first meet, the path from the entrance to the point they meet, and the path from the exit to the meeting point forms the two halves of a shortest path (in terms of cell visited) from entrance to exit. Combine these paths to get the final path solution.

3 Tasks

The project is broken up into a number of tasks.

Task A: Implement Maze Generators (8 marks)

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

- Recursive backtracker
- Kruskal's
- Prim'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 hexagon perfect maze with no tunnels (type B)
- 2D rectangular perfect maze with tunnels (type C)

However, not every algorithms can generate all three types of mazes. Hence, you only need to implement the recursive backtracker and kruskal algorithms to generate all three types of mazes; for prim's algorithm, you only need to generate the mazes without tunnels, i.e., the first two. The following table states which type of maze each of your generator implementation should be able to construct:

Generation Algorithm	**Maze types**
Recursive Backtracker	A, B, C
Kruskal's	A, B, C
Prim's	A, B

Task B: Implement Maze Solver (5 marks)

In this task, you are to implement two solvers to see how different mazes affect the different solvers. The solvers to implement are the recursive backtracker and bidirectional BFS solvers. They should be able to solve all three type of mazes.

Details for both tasks

As explained above, your task is to implement the three generation and two 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 three 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.

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, hex}.
- generatorName: Name of maze generation algorithm, where it should be one of {recurBack, kruskal, modiPrim}. recurBack is recursive backtracker, kruskal is kruskal's algorithm, and modiPrim is (modified) prim's algorithm.
- solverName: Name of the maze generation algorithm, where it should be one of {bidir, recurBack, sample, none}. bidir is bidirectional BFS, 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:

```
hex
modiPrim biDir
30 30
2 0
0 0
```

This specifies the following parameters:

- Generating hexagonal mazes
- Using modified Prim's algorithm for generation
- Using bidirectional BFS 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
recurBack none
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 recursive backtracker algorithm for generation
- No solver, just maze generation
- 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)

4 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 (8/15):

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.

Coding style and Commenting (2/15):

You will be evaluated on your level of commenting, readability and modularity. This should be at least at the level expected of a final undergraduate student.

4.1 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 blackboard can be slow, so please ensure you have your submission done a little before the submission deadline to avoid submitting late.

5 Submission

The final submission will consist of:

- Your Java source code of your implementations. We will provide details closer to submission date.

Note: submission of the code will be done via Blackboard.

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 Blackboard 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 Blackboard and we encourage you to check and participate in the discussion forum on Blackboard. Although we encourage participation in the forums, please refrain from posting solutions.

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.*
`HexMaze.java`	Class that extends the Maze interface to implement a 2D hexagonal maze. *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*
`RecursiveBacktrackerGenerator.java`	Class implements a recursive backtracking maze generator. *Complete the implementation of this file..*
`KruskalsGenerator.java`	Class implements a kruskal's algorithm generator. *Complete the implementation of this file..*
`ModifiedPrimsGenerator.java`	Class implements a prim's algorithm generator. *Complete the implementation of this file..*
`MazeSolver.java`	Interface class for maze solving algorithms. *No need to modify this file*
`BiDirectionalBFSSolver.java`	Class implements a bi-directional BFS maze solver. *Complete the implementation of this file..*
`RecursiveBacktrackerSolver.java`	Class implements a recursive backtracking maze solver. *Complete the implementation of this file..*

Table 1: Table of supplied Java files.