

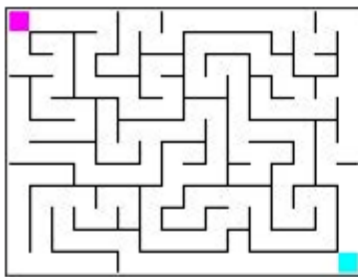
Homework 3 -- Mazes

Due Monday by 11:59pm**Points** 100**Available** after Oct 8 at 11:59pm

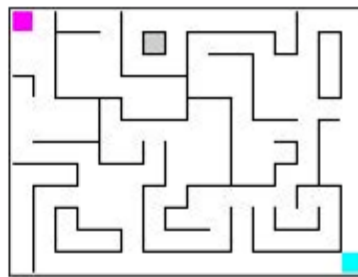
The goal of this assignment is to design and implement a program that can build a variety of mazes.

Build a Maze

A *perfect maze* is the simplest type of maze for a computer to generate and solve. It is defined as a maze which has one and only one path from any point in the maze to any other point in the maze. This means that the maze has no inaccessible sections, no circular paths, no open areas:



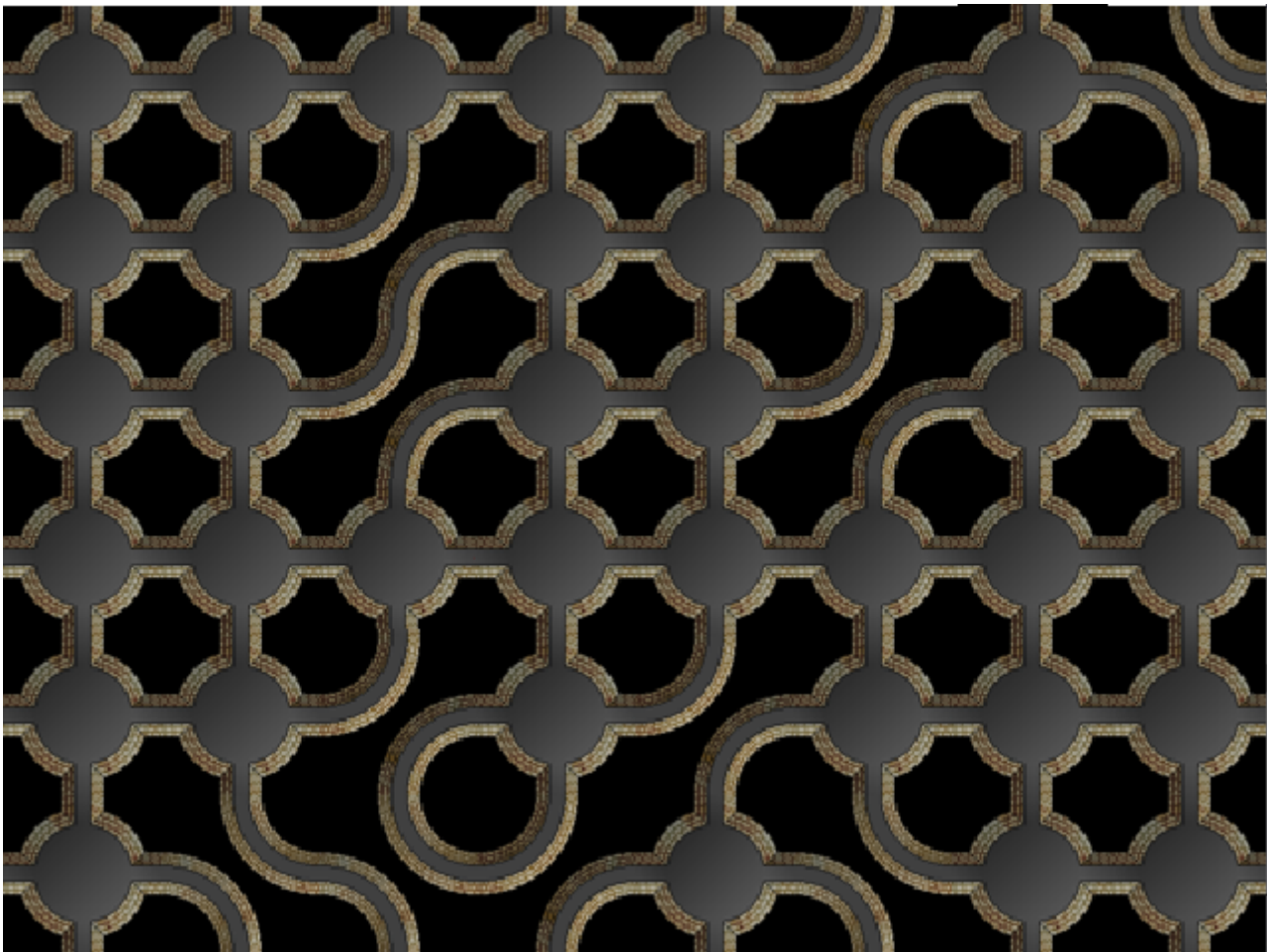
(a) A perfect maze



(b) A non-perfect maze

Each cell in the grid represents a location in the maze that has a potential exit to the north, south, east, and west. One way to look at the perfect maze is that each location is a hallway that twists and turns its way from one place to another. The challenge for this type of maze is to find the direct path from one place (the pink square) to another (the blue one).

In the *non-perfect maze*, each cell in the grid also represent a location in the maze, but there can be multiple paths between any two cells. This form is useful in several applications. Computer games, for instance, use this kind of maze to create a map of the world by giving locations in the maze different characteristics. For instance, a *room maze* categorizes locations in the maze as either *rooms* or *hallways*, where a hallway has exactly two doors while a room has 1, 3 or 4 doors. In this example of a room maze, rooms are pictured as circles, hallways are channels. Since locations at the top, bottom, left and right can "wrap" to the other side, we call this a *wrapping room maze*:



The placement of rooms, doors, and hallways in the grid is usually randomly selected with some constraint that is specified at the time of creation. In general, we start with a perfect maze (since it the simplest to build) and then additional pathways are added until the constraint is met.

Algorithm for building mazes

There are a number of algorithms for building perfect mazes including [Kruskal's algorithm](https://en.wikipedia.org/wiki/Kruskal's_algorithm) (https://en.wikipedia.org/wiki/Kruskal's_algorithm) and [Prim's algorithm](https://en.wikipedia.org/wiki/Prim's_algorithm) (https://en.wikipedia.org/wiki/Prim's_algorithm). Each of these requires starting with an undirected graph whose nodes represents locations in the maze, and whose edges represent walls between locations. For this assignment, you will implement a modification of *Kruskal's* algorithm.

To build a maze, we will need to specify the number of rows, the number of columns, and the number of walls remaining in the maze (this is our constraint). Given the number of locations in the maze as $n = \text{numberOfRows} \times \text{numberOfColumns}$, the remaining number of walls must be between 0 (a maze where all locations are rooms with 4 doors) and $\text{numberOfEdges} - n + 1$ (a perfect maze). In this video, we demonstrate how to modify Kruskal's algorithm to build these mazes:

2020fall-cs5010-p3-building-mazes



Powered by Panopto



What to do

Design and implement the interfaces/classes to generate perfect mazes, room mazes, and wrapping room mazes in a way that captures their similarities and accurately represents the relevant data. For the purposes of this assignment, the maze also has the following requirements:

- There is one goal location cell
- 20% of locations (at random) have gold coins in them that the player can pick up
- 10% of locations (at random) have a thief that takes some of the player's gold coins
- The maze can produce the player's location and gold count
- The maze can produce the possible moves of the player (North, South, East or West) from their current location
- The player can make a move by specifying a direction
- The player should be able to collect gold such that
 - a player "collects" gold by entering a room that has gold which is then removed from the room
 - a player "loses" 10% of their gold by entering a room with a thief

Your maze should **not** provide a way to dump the maze to the screen for anything other debugging purposes.

In addition, develop a driver program that uses command line arguments to:

- specify the size of the maze

- specify whether the maze is perfect or a room maze, and whether it is wrapping or non-wrapping
 - if it is non-perfect, specify the number of walls remaining
- specify the starting point of the player
- specify the goal location

Your program should then be able to


- demonstrate a player navigating within the maze, collecting gold, being robbed, and reaching the goal location (see details below for specific scenarios)

Hint: For testing purposes, use [Random \(Java Platform SE 11 \)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Random.html)

(<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Random.html>) to generate pseudo-random numbers to reproduce the same "unique" mazes each time.



Extra credit: Provide a way to specify the route from the player's current location to the goal location that retrieves the maximum amount of gold coins and loses the the minimum amount of gold coins

Create a JAR file of your program

In order to make your application easier to run, you are required to create and submit a  file.

What to submit

Log on to the [Handins submission server](https://handins.ccs.neu.edu/) (<https://handins.ccs.neu.edu/>) and upload a ZIP file of your assignment. Your ZIP file should contain three folders: src/, test/ and res/ (even if empty).

- All your code should be in src/.
- All your tests should be test/.
- Your original and/or revised design document should be in res/.
- Submit a correct  file in the res/ folder. We should be able to run your program from this jar file.
- Submit at least two example runs of your program the res/ folder that communicate to us how to specify commands to run your program to verify that it meets all of the above specifications.
 - One of the runs should show a perfect maze
 - One of the runs should show a wrapped room maze
 - One of the runs should show the player visiting every location in a room maze
 - One of the runs should show the player reaching the goal location
 - Each of the runs should state the player's location and gold count at each step
- Submit a  file that documents how to use your program, which parts of the program are complete, and design changes and justifications. You should distinguish the details of the two example so that we know what to look for in each run. The file should also include any necessary citations (see syllabus).

Criteria for grading

You will be graded on:

- Whether your code is well-structured and clean (i.e. not unnecessarily complicating things or using unwieldy logic).
- Correctness of your implementations including whether your example runs contain enough information for us to run your program from the provided `JAR` file and to see that all of the required features are working correctly.
- Whether your code is well documented using well-formed English sentences.
- Whether your code follows correct style (according to the style grader).
- Self-assessment to be completed on the Handins submission server.