CS 1XA3 Winter 2015 Project
Mazes

In this project we are going to build the data structures and the algorithms that we can use to build and solve perfect mazes. We will start with the simplest data structure called a stack:

- (10 points) A stack (also known as LIFO = Last In First Out) is a data structure that stores elements, and supports the following operations:
  1. push(item, S): adds the item to the top of stack S
  2. pop(S): removes the top item from S and returns it
  3. isEmpty(S): returns True if there are no elements in S, otherwise it returns False
  4. size(S): returns the number of elements in S

Build a class MyStack that implements a stack together with its operations.

We are going to use stacks later. First we need to define what is a perfect maze. The following pictures can help. Also, the site http://www.mazeworks.com/mazegen/mazetut/index.htm contains nice explanations of the algorithms we’ll implement and use.

We will start building a maze starting with an N-by-N grid. Note that a maze can be built by from such a grid by knocking down walls from the grid following certain rules (the first maze above comes from the 14-by-14 grid, and the second from the 22-by-22 grid). A maze is perfect if it has no open spaces, there is one and only one path between every pair of points in the maze, i.e., there are no inaccessible locations, and there are no cycles. Here’s a classic algorithm to generate perfect N-by-N mazes: Consider an N-by-N grid of cells, each of which initially has a wall between it and its four neighbouring cells. For each cell (x, y), maintain a variable north[x][y] that is true if there is wall separating (x, y) and (x, y + 1). We have analogous variables east[x][y], south[x][y], and west[x][y] for the corresponding walls. Note that if there is a wall to the north of (x, y) then north[x][y] = south[x][y+1] = True. Construct the maze by knocking down some of the walls as follows:
  1. Start at the lower leftmost cell (1,1).
  2. Find a neighbour at random that you haven't yet been to.
  3. If you find one, move there, knocking down the wall. If you don't find one, go back to the previous cell.
  4. Repeat steps (2) and (3) until you've been to every cell in the grid.

Note that the outer walls are never knocked down, except for the exit one. Also, pick your starting point and the exit randomly.

- (10 points) Build an object class Maze that takes a command line parameter N, and constructs a random N-by-N perfect maze. Hint: maintain an (N+2)-by-(N+2) grid of cells to avoid tedious special cases.
- (10 points) Use the graphics package graphics.py to add a Draw method to Maze that draws the maze like the figures above (together with the start and the exit).
The next natural problem is: Given a **Maze** object, can we compute a route from the starting point to the exit (we know that such a route always exists, because the maze is a perfect one)? There is a very simple and natural algorithm for exploring mazes (or rooms in a castle, or caves, or any structure that has interconnected spaces which can be modelled as the nodes of a graph, with the interconnections modelled as edges). It is called Depth-First-Search (DFS) and you most probably have already used it or heard about it (maybe in the form of the Hansel & Gretel fairy tale, which you can enjoy again here: http://en.wikipedia.org/wiki/Hansel_and_Gretel). The basic idea is to use exhaustive search with backtracking; in order to implement it, you don’t have pebbles to keep track of your progress, but you have a stack. The algorithm for solving a maze, starting from (1, 1) and stopping if we reach cell (N, N), is as follows:

$$\text{Explore}(x, y)$$

-------------

- Mark the current cell (x, y) as "visited"
- If no wall to north and unvisited, then $$\text{Explore}(x, y+1)$$
- If no wall to east and unvisited, then $$\text{Explore}(x+1, y)$$
- If no wall to south and unvisited, then $$\text{Explore}(x, y-1)$$
- If no wall to west and unvisited, then $$\text{Explore}(x-1, y)$$

- (10 points) Add a method Explore(x,y) to your Maze class that implements the Explore algorithm. This method must output the exit path, i.e., a sequence of cells that starts with the start cell, ends with the exit cell, doesn't go through walls (of course!), and it must be a simple path, i.e., it doesn't wander in useless cycles. For example, the first path wanders in useless cycles, while the second one goes straight from the start (3,4) to the exit (3,7); it’s the latter that you should output:

```
(3,4) (3,5) (4,5) (5,5) (5,5) (4,5) (4,6) (4,7) (3,7)
```

```
(3,4) (3,5) (4,5) (4,6) (4,7) (3,7)
```

*Hint:* Trace the content of your stack during the execution of Explore(). What do you notice?

- (10 points) So far, you place only two special points in the maze: the start and the exit. Extend your code to include a third special point, which is a key to open the exit door. Now, your Explore(x,y) method must first discover the key, before discovering the exit (i.e., even if you discover the exit first, you have to continue searching for the key, before coming back to it). Again, the method has to output a straight path from the start to the key and then to the exit. *Hint:* Is this complicated variation two simpler exploration problems you have already implemented, in disguise?

- (10 points) Extend your Draw method to be able to draw not only the maze, but also the path you calculate in the previous question (e.g., by painting the path cells with a different colour).

- (fun points) Build a game that allows the user to navigate in the maze until he or she finds the key and then the exit. Set a limit on the number of steps the user can take (you already know what is the minimum number of steps, because you have already calculated the simplest path that solves the maze – just give the user some slack; the smaller the slack, the higher the level of difficulty). You may want to add a time limit as well, or any other features you want. Your algorithms are actually quite general; for example, you don’t have to start with a grid, but you can start from any initial shape.