CSCI 204 – Introduction to Computer Science II

Project 2 – Maze

Assigned: Wednesday 09/27/2017
First Phase (Recursion) Due Friday, 10/06/2017
Second Phase (Stack) Due Monday, 10/16/2017

1 Objective

The purpose of this assignment is to give you an opportunity to practice stacks and recursion and understand better the relation between the two.

2 Introduction

Do you know how to find your way out through a maze? After you write this program, you will never be lost again in a maze!

Assume that a maze is a rectangular array of squares, some of which are blocked, others are empty (we call them *a passage*). All squares at the boundary of the maze that are not blocked are possible exits, if there is a way leading to it from the entrance point. (Note that the entrance point may be any passage square of the maze, not necessarily one on the boundary). A path from the entrance to an exit can consist of only passage squares that are next to each other. There may be many exits for a given maze and the entrance point. A mouse (or any other creature with some intelligence) chooses an entrance among the squares that are passages and it will try to find all possible ways leading to an exit from the given entrance.

For example, if a maze is given in Figure 1, the '*'s represent the squares that are blocked and the spaces represent the possible passage. Starting at location (5, 6), one will find the three possible paths to the exits of (0, 3), (11, 1), and (11, 1) through a different path. The squares on the paths are represented by the symbol '!'. There are four copies of the maze in Figure 1. The upper left one is the original maze; the rest three show three different paths from the starting to the three exits that can be reached from the starting location.

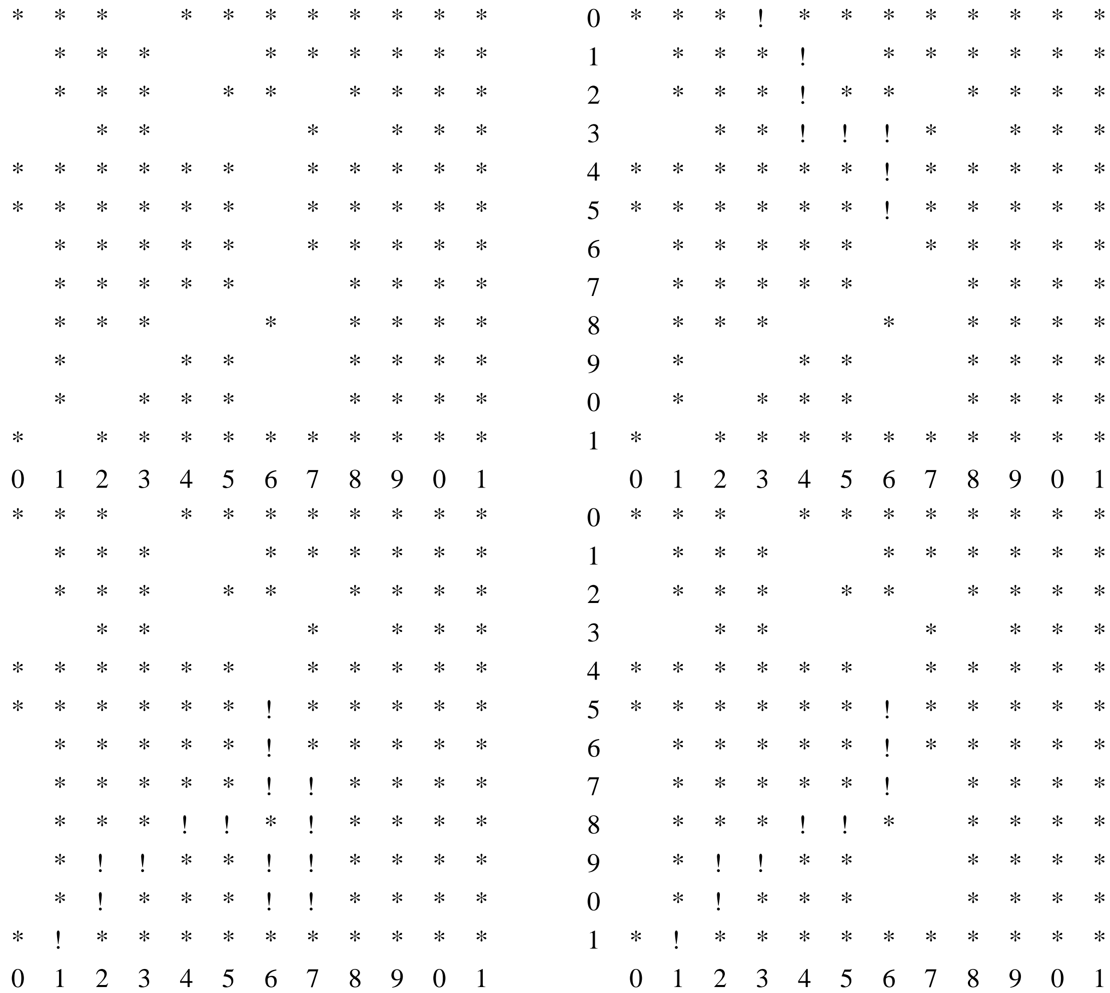

Figure 1. Four paths out of (5, 6)

The basic rules of moving around a maze are very simple.

1. The mouse starts at the location within the maze specified as the entrance point.
2. The mouse can move through any open space (possible passage) toward the exits, which are located at the four edges of the maze (or as specified by a pair of coordinates) and the squares at the exits have to be possible passage themselves.
3. The mouse can only move up, down, left and right. The mouse cannot move diagonally, although permitting it to do so will not increase the difficult level of the algorithm too much. (The figure shown above permits the mouse to move in all eight directions.)
4. In general, the mouse should be able to find all possible paths, not just one. But to simplify our problem, especially with the stack solution, our mouse is specified just to find one path in this project.

How does it work? Starting with the current mouse location (e.g., the entrance point), the mouse will look for a possible move in all four directions if the current square is not an exit. If one of the neighboring squares is a possible passage and it has not been visited before, the mouse will move to that square by marking the square as *visited* (as a part of a path). The algorithm continues from there. As one can see, this can be formed as a recursive solution very nicely. One can also solve the problem using stacks without recursion (though there is a very close relation between stack and recursion.)

3 Your Tasks

The project you are to complete is to design and implement a set of classes that work together to solve the maze problem. You will solve this problem in two different ways, one is using recursion, and the other is using stacks.

1. Your program generates (or reads from a text file) a maze, display it on the screen. This should depend on the user choice. The user can either indicate to use a randomly generated maze or the user prefers the use of an existing maze (read the maze from a text file).
2. The user is able to specify a particular entrance point and a particular exit point after seeing the maze.
3. Your program then takes this pair of input and exit coordinates and invokes the necessary methods in various objects to find a path between the entrance point and the exit point.
4. Your program then displays all these paths to the user on the screen.
   a. When using stack, print the single path (you do not need to erase other traces that are left on the maze as a part of the previous exploration.)
   b. When using recursion, print all possible paths between the entrance and exit point. Because of the nature of recursion, your program will print one maze after another if there is more than one path between the entrance and exit point; each of the mazes will display one path.

4 Detailed instructions

Copy all the files from the following Linux directory.

~csci204/2017-fall/student/project/p2/

In that folder, you should see a collection of six files, three test maze files, maze1.dat, maze2.dat, and maze3.dat, and three Python program files, maze.py, pyliststack.py, and testmouse.py. The file maze.py contains an incomplete implementation of a maze class and the file pyliststack.py contains an implementation of stack. You may implement your own stack (e.g., use the work you do in the stack lab.) The key requirement is that you have to solve the problem using two different approaches (though highly related), stack and recursion. The file testmouse.py contains the drive program where you can use to run the tests.

Your first task is to complete the *maze* class. Take a look at the maze class file. We have implemented a method to generate a random maze. You need to complete all other methods that are specified in the file. If your solution requires more methods from the maze class, please feel free to add them. The basic logic (take a look at testmouse.py) is that when the program starts from testmouse.py, the user is asked to enter

a file name representing the maze to be solved. If the user enters something that is not a valid file name, your program (the maze class) generates a random maze. After displaying the maze on the screen, the user can then specify an entrance and an exit point so your program can try to solve the problem.

4.1 Algorithm to find a maze path using recursion

The algorithm to find a maze path using recursion is very similar to that of using a stack. The only difference is that in recursion, we don't need an explicit while loop any more. We simply explore the four neighbors *recursively*! The exact algorithm is left for students to figure out. One key element is that the algorithm has to make a local copy of the maze before exploring the next level because the mouse needs a current version of the maze in order to decide which direction(s) to move. To save a local copy of the maze, you need to do two things.

1. Pass the maze as a parameter to the recursive call (among other parameters such as the current position coordinates and the exit coordinates)
2. At the beginning of the solving method, make a local copy of the maze that is passed in as a parameter by using Python's deepcopy() method in the copy package. The copy package allows one to make various types of copy. The method deepcopy() creates a separate object with identical attributes as those in the original copy. The copy() method makes a *shallow* copy with reference to the original object only.
```
import copy
……
localCopy = copy.deepcopy( maze )
```

Note: The logic of recursion should be implemented in a file that you can import into the testmouse.py program. If you examine the current testmouse.py file, you will find that the testmouse.py file imports the Mouse class from a file named mouserecursion.py for a recursive solution. Later on when you implement the solution using stack, you should consider that the file name to be ousestack.py. (You can certainly use other file name to contain the implementation of the Mouse class, you just have to change the import statement accordingly.)

4.2 Algorithm to find a maze path using a stack

The basic idea of using a stack to find a path in a maze is described in the following algorithm. (See Dr. Havill's website: http://personal.denison.edu/~havill/algorithmics/algs/maze.pdf. Dr. Havill is an alumnus of Bucknell's Computer Science Department.)

As one can see, the stack solution marks the path while exploring. When the exit is found, in addition to the correct path that has been marked, other cells on the maze may also have been marked as a possible path with unsuccessful exploration. (See a demo late in this description.) This is fine. Completely removing the unsuccessful trails involves saving a copy of local maze on a stack each time a cell is explored. We will not attempt to resolve this problem here in this project.

Name the file containing your Mouse class using stack to be something like "mousestack.py"

```
Algorithm Maze (maze, start)
# Find the goal in a maze using a stack (depth first search)

create an empty stack named S
push start onto S
while S is not empty do
        current ← pop from S
        if current is the goal then
                output "Success!"
                print the maze with the path marked
                break the loop
        else if current is not a wall and current is not marked as visited
        # Explore four neighbors
                mark current as visited
                mark the current as a part of the path on the maze
                if the right neighbor is valid to visit
                    push onto S  the point to the right of current
                if the left neighbor is valid to visit
                    push onto S the point to the left of current
                if the neighbor above is valid to visit
                    push onto S the point above current
                if the neighbor below is valid to visit
                    push onto S the point below current
        end if
end while
```

Figure 2. Algorithm for a stack solution

4.3 **Revising the drive program** testmouse.py

You need to run the program through the drive program testmouse.py. Currently testmouse.py will ask the user to enter a file name. If the file name is not "none", the program will read a maze file. You are asked to revise this part of the program so that the program will simply try to open a file with the name that the user types in, if the opening of the file is failed, an exception is raised such that your program will use the random number generator to generate a maze file (this part of the code is already given in the maze class.)

4.4 **Sample output**

The following is a sample output running the maze program using maze1.dat as the input. You can compare your output with it.

This result is from the stack solution. Notice that it only finds and prints one path while two paths exist.

```
Enter the name of the maze file ("none" if using random
file): maze1.dat
-------- The Original Maze --------
 012345678901
0*** ********
1***    *****
2*** **  ****
3***   *  ***
4***** ** ***
5***** ** ***
6***** ******
7*****   ****
8***   * ****
9*   **  ****
0* *    *****
1* **********
Please enter the starting row : 2
Please enter the starting column : 3
Please enter the exiting row : 11
Please enter the exiting column : 1
Success!
We found path!
 012345678901
0*** ********
1***    *****
2***!**  ****
3***!!!*  ***
4*****!** ***
5*****!** ***
6*****!******
7*****!  ****
8***!!!* ****
9*!!!**  ****
0*!*    *****
1*!**********
```

Figure 3. Sample result from a stack solution

For the same maze, if you use a recursive solution, multiple paths will be printed, each of which occupies a maze. For this particular maze (maze1.dat), in addition to the path found in Figure 3, another path between (2, 3) and (11, 1) is as follows.

```
 012345678901
0*** ********
1***    *****
2***!**  ****
3***!!!*  ***
4*****!** ***
5*****!** ***
6*****!******
7*****!!!****
8***   *!****
9*!!!**!!****
0*!*!!!!*****
1*!**********
```

Figure 4. Additional result from a recursive solution

Note that in the above example of the stack solution, there is no extra mark left on the maze except the path. In other cases, it is possible that extra marks left on the maze which are not part of the solution.

5 Submission

Make sure you submit all your files in your folder to Moodle with two separate submissions, one for each phase. A “readme-xxx.txt” file is required that contains the output of your test runs, where “xxx” indicates either the result of a stack solution, or that of a recursive solution, (e.g., “readme-stack.txt” or “readme-recursion.txt.” In your readme file, put your name and the assignment number at the top, mark clearly the name of the input file (e.g., *maze1.dat*) and the output associated with it. You are asked to include all three test cases given in the assignment, each of which should be tested against both a stack solution and a recursive solution.

We ask you to submit the two phases separately with each submission containing the program and a proper readme file. Make sure you name your solution program separately as well, e.g., mousestack.py and mouserecursion.py.

6 Grading Criteria

When grading, we will look for the quality of the program from various aspects. Here are the grading criteria.

1. Functionality: 60%

   User-friendly and well-formatted input and output (10%)

   Correctness of computation (30%)

   Program satisfies the problem specifications (20%)

2. Organization and style: 30%
   a. Program follows object-oriented design approach and contains at least classes for Stack, Maze, Mouse, and a separate test program (e.g., *testmouse.py*) (10%)
   b. Variables and functions have descriptive names following Python convention (5%), program contains sufficient amount of docstrings explaining the functionality of its major parts and comments explaining specific actions that blocks of code are performing, each code file has comments on the top including name, date and file description (5%)
   c. Code is structured, clean, organized and efficient. Each Python code file that doesn’t include the “main” function contains description of only one class (10%). *To earn full grade in this category, avoid leaving commented-out legacy code in your submission.*

3. Extraordinary elements, creativity and innovation: 10%

   This includes anything beyond the provided specification that makes your program stand out.