IE 172 Laboratory 3: Amazing Mazes

Dr. T.K. Ralphs

Due February 25, 2016

1 Laboratory Description and Procedures

1.1 Learning Objectives

1. Understand each of the key terms listed below.
2. Understand how to implement and use the stack ADT.
3. Understand how to write a client application using the interface to a data type.

1.2 Key Words

1. abstract data type
2. interface
3. stack
4. data attribute
5. method attribute
6. implementation
7. client

1.3 Scenario

The application for this lab is a program that could help a robot find its way from its starting point through a series of corridors to a specified destination or conclude that there is no path to the destination. Although this application may seem rather trivial for the size of mazes we are dealing with here, path-finding algorithms are in fact very serious business in the gaming world. Simulating characters that can navigate through the complex virtual worlds of on-line games such as World of Warcraft in a realistic fashion requires sophisticated path-finding algorithms. Path-finding programs could also be used to guide a robot through a factory or even to find a path through a space too confined for humans, such as a collapsed building. The very simple algorithm that we will use for this lab makes use of trial and error to find a path. We will use a *stack* data structure to keep track of the path followed so far so that we can retrace our steps when we reach a dead end.

In this scenario, we want the robot to be able to make fast decisions and therefore our stack data structure should be as efficient as possible at performing the necessary operations. Your primary task is to design a stack data structure that can be used with the maze program as a *client*. You will also be asked to program one of the methods of maze program itself, which uses the stack class.

1.4 Program Specifications

1.4.1 Stack Data Structure

For this lab, we will not focus on analyzing the efficiency of an algorithm, but rather on the use of data structures by their clients through an API. We will illustrate the design and use of a new data type, as well as the concept of separating the interface from the implementation. Finally, we'll demonstrate the use of a data type by a client program. The data structure we will use is a very simple one called a *stack* that can be implemented on top of either the linked lists class or on top of the Python list class. Your first task will be to implement a stack data structure on top of the linked list class with the API designated in the file `stacks.py`.

1.4.2 Maze Class

After implementing the stack API, the stack data structure will be used to implement a `Maze` class. The class is implemented in the file provided to you called `maze.py`. The constructor for the class generates a random maze. Upon construction of the maze, it can be solved automatically through a systematic search algorithm. To move through the maze, the robot may go from the square it currently occupies (always starting from the top left corner) to any of the four squares that may be reached by going right, left, up, or down. Once the robot chooses to move to a square, it is immediately marked as having been visited and the root is only allowed to visit there again if backtracking. Once at the new square, the robot should try each new direction in turn and if no direction is found, then backtrack by using the stack.

The main method of the class is the `solve()` method, which determines a path through the maze by exhaustively searching all unexplored squares until forced to backtrack. The progress of the algorithm can be displayed graphically if the display mode is set to `graphical_full` or `graphical_limited` (see documentation). This is done using the `pygame` module. The `solve` method returns the path (if found) as a stack of tuples representing the locations visited on the path. After solution, an ASCII representation of the solution can be printed with the built-in print method.

The stack stores all moves made to arrive at the current square. After making each move, the move is put on the stack so that it can be undone. Once the robot begins exploring (always from the top left corner), it should continue to follow the above algorithm until either finding a path to the exit (always at the bottom right corner) or concluding that there is no such path (when the stack is empty and the robot has backtracked all the way back to the starting point). If a path to the exit is found, then the stack can be read to determine the path taken. If a path is found, it should be printed to a file in the same format as the input except that a "*" should appear in any square that is part of the path.

2 Laboratory Assignments

2.1 Programming (15 points)

1. (15 points) Change the code so that the maze is stored as a dictionary, as in the code for the Game of Life.

2. (Extra Credit (10 points)) Explain using pseudo-code (or real code) how you would go about changing the current iterative implementation into a recursive one.

3. (A Little Fun) Change the white square into a picture of a robot.

2.2 Analysis (30 points)

1. (10 points) What is the theoretical worst case running time of finding a path through the maze?

2. (10 points) Test the empirical performance of the `solve()` method by generating random boards of dimension 100 and counting the total number of squares the robot visits (the number of steps) before finding the solution (this will require some modifications to the code).

3. (10 points) Test different densities and search orders to see if the performance depends on either of these factors (this also requires some modification to the code).

2.3 Follow-up Questions (20 points)

1. (10 points) Consider a randomly generated list of size n.

   (a) If the list is stored in random order, what is the expected (average) number of comparisons we will make over a sequence of M searches if αn of these searches are successful, i.e., we are searching for an item that is on the list.

   (b) What is the expected average number of searches if the items are stored in order (so that we can use a more efficient search strategy, such as bisection search).

2. (10 points) Use an induction argument to show that the merge sort algorithm is correct, as we discussed in class. Recall that there are two parts to the argument: proving that the merge is correct and then proving that the overall algorithm is correct, given that the merge is correct.