CSCI 262 Data Structures

Spring 2019

Project 2: Mazes

(Back to Assignments)

Purpose

- Understand how to use stacks and queues
- Learn a basic algorithm for navigating a maze

Before You Begin

- Read the entire assignment before starting to program.
- Download the starter code.
- Download the example inputs.

Introduction

Problem Description

Suppose that you are given a maze as depicted below; you start at the position marked 'o' and try to reach the position marked with a '*', without going through any walls ('#'). You are allowed to move up, down, left, or right only. Is there a solution to the given maze (i.e., can you traverse it from start to finish on the given open paths?

```
Example maze:
  ########
  #.##..*#
  #.##.###
  #.#..###
  #.##...#
  #o...#.#
  ########
```

For this project, you are to write a program which will determine whether a path exists from the start point to the finish point of a given maze.

Finding Your Way

There are many strategies for navigating mazes. One of the key strategies is to keep track of where you've been, so you don't keep exploring the same paths over and over. This is easy enough if you've brought some

chalk - just write a mark on the floor at regular intervals to show where you've been.

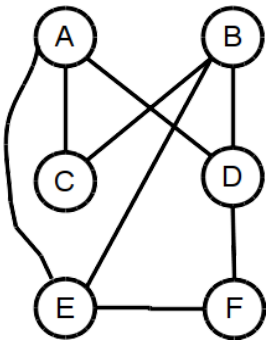
Another key is to have some way of figuring out where you haven't yet been and making sure you explore all of those places. If you've been marking your path as you go, then a simple strategy is to always go until you hit a dead-end; when you hit a dead-end, you then back up to the most recent intersection and check the passages you didn't yet go down (the ones that are not marked). If all of the passages are marked, you back up even further (make sure you remember which passage you came from originally, though - more chalk!) The computer equivalent of backing up and trying the other choices is called backtracking, and is basically an exercise in writing recursive functions, a topic we'll explore later in the semester.

Now suppose you have a GPS, paper, pen, and a magic teleporter (and the chalk!) Your new strategy is as follows; whenever you reach an intersection where you can go in more than one direction, after marking where you've been, you write down the GPS location of the start of each outgoing tunnel. Finally, you pick one of the tunnels to explore and cross it off your list. Now, whenever you hit a dead-end, instead of backing up, you simply choose an unexplored location from your list and use the magic teleporter to go straight there.

This latter approach is how we'll navigate for this assignment. Because our maze will be a regular two-dimensional grid, we don't really need the GPS - we can simply use the (x, y) coordinates of each point in the maze for our locations. The pen and paper in our solution will be implemented using a data structure; the classic data structures for this problem are the stack or the queue.

Graph Search

The problem of navigating a maze is actually a special case of a more general problem, that of graph search. A graph is a data structure that can be used to model all kinds of interesting problems, including mazes. Basically a graph is composed of locations (known as nodes or vertices) and connections between locations (known as edges). Here's an example of a simple graph structure:



When we follow the algorithm above using a stack to store locations we haven't yet explored, the algorithm is known as a depth-first search (DFS) because we go as far in the graph as we can before hitting a dead-end and trying another path. It turns out that DFS explores a graph's locations in basically the same order as the recursive backtracking approach described above; once again, we see that there is a connection between stacks and recursive function calls.

When we use a queue to store locations, we end up doing a breadth-first search (BFS). A BFS works outward from the start location like ripples on a pond. As the ripples move further outward, more locations are discovered.

Both of these important algorithms have applications to real-life problems; if you are moving on to a degree in computer science, you will learn more about DFS and BFS and their applications in your Algorithms course. For

now, your program will demonstrate how each algorithm explores a maze.

Example Program Interaction

The user interface will be pretty simple for this assignment (and it is mostly already written for you). The program will prompt the user to enter a file name and whether to use a stack or a queue. The program will then read in the file and begin stepping through the maze. At each step, the program will draw the maze, showing '@' symbols on squares that have already been visited. The program will prompt the user to continue. An example of the interaction can be seen below (using a stack):

```
Please enter a maze filename: 8-example.txt
Please enter (S) to use a stack, or (Q) to use a queue: S
Solving maze '8-example'.
Initial maze:

########
#.##..*#
#.##.###
#.#..###
#.##...#
#o...#.#
########

Hit Enter to continue...

########
#.##..*#
#.##.###
#.#..###
#.##...#
#o...#.#
########

Hit Enter to continue...

########
#.##..*#
#.##.###
#.#..###
#.##...#
#o@..#.#
########

Hit Enter to continue...

########
```

```
#.##..*#
#.##.###
#.#..###
#.##...#
#o@@.#.#
########

Hit Enter to continue...

########
#.##..*#
#.##.###
#.#..###
#.##...#
#o@@@#.#
########

Hit Enter to continue...

########
#.##..*#
#.##.###
#.#..###
#.##@..#
#o@@@#.#
########

Hit Enter to continue...

########
#.##..*#
#.##.###
#.#..###
#.##@@.#
#o@@@#.#
########

Hit Enter to continue...

########
#.##..*#
#.##.###
#.#..###
#.##@@@#
#o@@@#.#
########
```

```
...


########
#.####*#
#.###@##
#.#@@###
#.#####@#
#o@@@#@#
########

Hit Enter to continue...

########
#.####*#
#.###@##
#.#@@###
#.#####@#
#o@@@#@#
########

Hit Enter to continue...
Finished: goal reached!
```

Outline

Step 1 - Download and build the starter code

One of the skills a programmer must learn is how to read and modify code written by other programmers. For this assignment, you have been provided with starter code which implements the user interface and the main loop driving the algorithm. It also provides the start of a class named `maze_solver` which will do all of the important work of solving mazes. Your job is to fill in the methods which implement the algorithm as well as print the maze.

The provided code takes care of all the user prompts at the beginning. It will call a method you will write (`_read_maze()`) to read in a maze from an input stream (file opening/closing are handled for you). It will also call another method you will write (`_initialize`) to do any setup of the `maze_solver` object needed for the algorithm (basically, just find the start point and add it to your stack or queue). Then it begins looping. At each iteration of the loop, it checks to see if a boolean instance variable, `_no_more_steps`, has been set to true. If not, it calls `_step()` (another method you write) to advance one step in the algorithm. Once `_no_more_steps` is set to true, the loop exits and the code checks another boolean variable, `_goal_reached` to determine whether to print a success or a failure message. At each step along the way, the method `_print_maze()` is called to display the current state of the maze.

The starter code should build and run. Look for the **TODO** comments in the code to see where you should add or modify code.

Step 2 - Write the `_read_maze()` method

Before doing this step, take a look at the provided maze files - they are just text files containing an initial line providing the number of rows and columns, and then the maze as lines of text. Your job is to read in the number of rows and columns, and then read in the maze lines and store in the `_maze` object in your `_maze_solver` class. (As written, `_maze` is a vector of string, which is a convenient type for this problem - but you can change this if you prefer.)

Step 3 - Write the `_print_maze()` method

The `_write_maze` method will at least let the program print the initial maze for you. This is a very simple method - just write out the rows of the maze to cout - but it will let you at least verify that your `_read_maze()` method is working.

Step 4 - Write the `_initialize()` method

You must first initialize your maze-solving algorithm in the method named `_initialize`. Start by searching the maze for the start location. You will need to decide on some representation of locations in the maze - consider writing a simple `point` class to hold row, column locations in public variables.

The start point should also be put into the data structure. There is a boolean instance variable, `_use_stack`, which will tell you which data structure you should be using. You will need to add instance variables to `maze_solver` to store your stack and queue (just go ahead and make one for each - you will use only one of them each time your program runs). Then push or enqueue the starting location of the maze (if you are using a `point` class, for instance, you would push or enqueue a `point` object onto a stack or queue of `point`).

Step 5 - Write the `_step()` method

This will be the most complex method in your program. You will likely want to create additional helper methods to decompose the problem, as well to keep your code clean and reduce code duplication. Essentially, the `_step` method implements the graph search algorithms described above. Each "step" should result in the exploration of one previously unexplored space on the maze.

The basic algorithm for `_step()` is below, assuming you are using a stack (the queue version is identical except for the names of things). Note that `_step()` is called in a loop that is implemented in the method `run()` - you don't write a loop yourself! (The method `run()` also takes care of calling `_initialize()`, `_write_maze()`, and other stuff.)

- If no more steps are possible (your stack is empty), set _no_more_steps to true and return.
- Pop a point from the stack. This is your current point.
- If the current point is the goal ('*'), note this (set the `_goal_reached` and `_no_more_steps` variables to true) and return.
- If the point is other than the start point, mark it as visited (set it to '@').

- Examine the points above, to the right, below, and to the left of the current point. If they are valid (i.e., within the maze boundaries), reachable (i.e., not walls), and not already visited (not marked with '@' or the 'o'), then push them onto the stack.
- Make sure the top item on your stack is not visited (this can happen when, for instance, you have visited multiple neighbors of a point previously). Pop any already visited items.
- If the stack is empty, set `_no_more_steps` to true.

Step 5 - Test!

Make sure your program will run correctly on different inputs. We have provided you with a variety of inputs which may help catch some common problems. For instance, there are inputs that have loops or open corridors which can trap you in an infinite loop if you are not correctly marking and avoiding previously visited locations.

Beyond just testing, run your code on various inputs and observe how the solution steps vary depending on the data structure used. Do you see how, when using a stack, the algorithm probes as far as it can down one path before backing up a bit to try another? When using a queue, do you see how the visited points expand out from the start point such that it visits every location one step away before visiting any locations two steps away and so forth?

Grading:

For this project, we will be doing something a bit different for grading; we will be using something called interview grading, which means that in addition to grading the code you produce, we will conduct short interviews with each of you to ask questions about your understanding of what went into this project. Expect to answer questions about the algorithm used in this project and your implementation of it. Stay tuned for more information on how to sign up for an interview time.

This project is worth 80 points in total, evenly divided between evaluation of your code product and your interview.

Code (40 points)

Program compiles and executes	5 points
README	5 points
Code quality and style	5 points
Reading and printing maze	5 points
Correctly solves simple mazes with walls	5 points
Correctly solves mazes without walls	5 points
Correctly reports when mazes have no solutions	5 points
Correctly solves mazes with loops	5 points

Documentation:

README:

1. Include names of all people who helped/collaborated as per the syllabus
2. Describe the challenges you encountered and how you surmounted them
3. What did you like/dislike about the assignment?
4. How long did you spend working on this assignment?

Submit a zip file on Canvas containing:

- README
- all of the source files (header and cpp) for your program (including any unmodified starter code)