CS 120 (Spring 22): Introduction to Computer Programming II

Long Project #11 - Maze Solver

due at 5pm, Tue 12 Apr 2022

REMEMBER: The `itertools` and `copy` libraries in Python are **banned.**

```
####S....#######   ####
#  #    .      #####  #
#  ###  .....         #
#    #  #   .         #
#  ###  ### .         #
#           .         #
#         ...#   ######
#    E.....
```

1 Overview

In this program, you will solve a maze. You will read the map up from a file, build a tree to represent the maze, and then search through the tree, to find a path from the start to the end.

Name your program `maze_solver.py`

1.1 Dump Points

Solving the maze takes several steps, so I've broken it into smaller pieces. For each piece, I've added a "dump" option - if I ask for that particular dump, then you will stop the algorithm at that point, print out what you have so far, and then terminate the program. I've done this for two reasons. First, I hope that it will help guide you through the solution - since each step will be relatively small. Second, if you complete only part of the algorithm but not all of it, you can still get partial credit, because we'll have testcases that will stop you, and check your code, at various points in the process.

(Yes, this reduces your flexibility, and locks you into an algorithm I've devised. Sorry!)

1.2 User Input

When the program runs, the user will type two lines of input. The first will be a filename, which gives the map. The second will be the "command;" if it is one of the "dump" commands, then you will run the program to that point and dump out your current state. If the command is blank, then you will run the algorithm all the way to the end, and print out the solution only.

2 Map Format

The maze will be encoded in a text file, which will look something like this:

```
####S###########   ####
#  #    #      #####  #
#  ###  #####         #
#    #  #   #         #
#  ###  ### #         #
#           #         #
#         ####   ######
#    E#####
```

In this map the hash symbols represent the paths that you can follow (**not** walls!). The `S` represents the start point for the maze, and the `E` represents the end. (Don't assume that the start and end are at any special locations; they might be **anywhere** in the maze.)

Your program will find the path through the maze, and report it by printing out the maze again, but with the path marked with periods, like this:

```
####S....#######   ####
#  #    .      #####  #
#  ###  .....         #
#    #  #   .         #
#  ###  ### .         #
#           .         #
#         ...#   ######
#    E.....
```

When you read the file, don't assume that the lines in the file are a perfect grid; different lines might be different widths. Instead, simply focus on the locations where hash symbols exist (plus the start and end symbols).

2.1 Coordinates

When you read the map file, you need to record every "cell" that contains a path. Assign them (x, y) coordinates, where $(0, 0)$ is the upper-left corner of the map. In the example map above, we have lots of cells along the top edge; their coordinates are

$$(0,0), (1,0), (2,0), (3,0), (4,0), (5,0), (6,0), ...$$

There are also quite a few along the left edge:

$$(0,1), (0,2), (0,3), (0,4), (0,5), ...$$

The start point is at $(4, 0)$ and the end point is at $(5, 7)$.

2.2 Map Guarantees - and Things You Must Check

You may assume (without checking) that the map we give you is **acyclic** (meaning that it doesn't have any loops). Real mazes sometimes have loops, of course - but that makes the solver harder to write!

You can also assume that all of the paths in the map are connected to each other. Together, these two assumptions mean that you know for sure that there is **exactly one** path from the start to the end.

However, you must **check and confirm** that each of the following things is true about the map. If any of these are not true, you must immediately report an error and terminate the program. (See the testcases for the exact error messages.)

- You must be able to read the map file.
- The map must contain exactly one start state - no more, no less.
- The map must contain exactly one end state - no more, no less.
- The only characters allowed in a map file are space, newline, hash symbol, S, and E.

3 Trees and Mazes

In this program, you will be using a tree - but it won't be a binary tree! Build your own class, to hold the nodes of this new tree - but design it such that each node can have an unlimited number of children. While I won't tell you exactly how this tree must work - and so you can define its methods and data fields - note that your TA will be checking your design, and good design of this class will be part of your grade.

It may seem odd to use a tree to solve a maze, but - so long as the maze doesn't contain any loops - it works remarkably well. The root of the tree will represent the start position in the maze; the parent/child relationships in the tree represent moving further away from the start. That is, the start location itself will be the root, and its children will be the cells (one, or perhaps many) which are adjacent to the start. The next level of children will be locations that are two steps away from the start, and so on. The end location, then, will be one of the nodes, somewhere in the tree - it might be a leaf node, but don't count on it!

The cool thing about representing a maze with a tree is that it makes the maze very easy to solve: to find a path from the start to the end, really all we're doing is doing a search through an un-ordered, non-binary tree!

(spec continues on next page)

4 Required Dump Points

As noted above, the "command" that the user types will tell you how far to run the algorithm, before you dump out a result. You must support the following commands; if the user types anything other than one of these (a blank line is OK), then report an error and terminate the program. For each of these commands, check the testcases to find the proper output.

> **WARNING:** For this program, we're going to need to be more picky about **exactly** matching the required output than we normally do. (This is because we need to verify exactly how many spaces you print out in your map.) So make sure to **test your code on GradeScope early** - and be prepared that the autograder will be quite picky about blank spaces, and blank lines.
>
> Sorry!

4.1 Command: `dumpCells`

After reading the map in from the file, print out all of the cells in the file, as (x, y) pairs. Sort the pairs. Mark the START and END cells.

4.2 Command: `dumpTree`

After converting the set of cells to a tree, print out the tree, using a pre-order traversal. Use indentation (with spaces and vertical bar characters) to make it clear which nodes are parents and children.

(If a node has multiple children, the order in which you print the children matters. Make sure, when you build the tree, that you always build the children in the following order: up,down,left,right. Then you will match the expected output.)

4.3 Command: `dumpSolution`

After finding the solution (by seaching the tree), print out all of the steps in the path (including the start and end points)

4.4 Command: `dumpSize`

The last little step, before actually printing the output from your maze, is to figure out the height and width of the input maze you were given. This must be calculated **only based on the cells you've been checking -** don't try to keep track of how many lines the file had, or how wide they were.

If I ask you to dump this out, then scan through the set of cells, and find the maximum x and y values.

4.5 No Command

If there is no command, then simply print out the map, back to the user - but with the path (except for the start and end cells) replaced by periods.

5 Hints: Tree Building

Probably the most challenging single part of this program is buildling the tree. You won't be able to build it like we build BSTs (adding one node at a time); instead, you will need to **recurse through the maze,** building the entire tree in one recursive algorithm.

The set of cells that you have read from the map will be central to this. Any time that you want to build a node of the tree, create a new node (its value should be the (x, y) coordinates of this cell). Then, check, in all 4 directions, to see which ones have adjacent cells. Where you find adjacent cells, build child nodes for the tree.

Of course, we're overlooking some details. We must never go back into our own parent! So, we'll need some way to prevent any node from treating its own parent as a child. There are several possible strategies for this: feel free to choose any one of these (or come up with your own):

- Pass, as a parameter, information about your parent node - so you can know which neighbor you can't treat as a child.
- Have a set of all of the cells - pass it as a parameter - but **modify this set** as you go, removing cells from it as soon as you build them.

  Haven't I told you, repeatedly, not to modify an array (or set) that you've been given as a parameter? Not exactly! I said, don't modify it unless **the function spec says that you can.** Since you get to write the function spec, you are allowed to make it legal, if you want!
- Build a **new** set of nodes (passed around as a parameter), that gets **larger** every time you add a node. These are the **already-built** cells - and so you should never build a new node for any of them, a second time.

5.1 Child Order

Which child should you explore first? If it weren't for the autograder, you actually would have **complete freedom** - any order of the children would work as well as any other. But, since the autograder expects you to print out the nodes in a **very specific order** in the `dumpTree` command, you have to create your children in the same order as I did. (Sorry!)

Always, from any given node, look for children in the following order: up,down,left,right.

6 Hints: Tree Searching

We already know how to search through an unordered tree; we have to check each of the children, to see if they have the value we want. But this program requires a more advanced search than you've done before: instead of returning a boolean value (`True` or `False`), you need to return an **entire path** through the maze.

I'm not going to give you the solution here, but here are some things to be thinking about:

- What sort of data structure would you use to store the path, if you had your choice? Try returning exactly that from the search function.
- Many searches through the tree will fail, because we made a wrong turn in the maze. What should your search function return in that case?
- If a node recurses into a child, and the child finds the thing we're looking for, it will return some information about how to get to the node. But that information will (probably) only get us from the **current node,** to the destination. It doesn't show the whole path from the root. What new information can the current node add, before it returns an answer to its own parent?

7 Hints: Printing Out the Solved Maze

While the input file is not required to be a perfect grid of characters, your output must be. It should be a grid of characters, of exactly the required width and height. This will often include some trailing spaces, at the end of a line, to fill up space on the line - and perhaps leading spaces, too - if the current line doesn't have cells all the way at the edges.

There are a couple ways to do this (both pretty good). First, we could declare a 2D grid, and fill the grid up with individual characters. This is sometimes a cool trick, since you can fill the grid up with some starting default (say, spaces?) and then "draw" into the grid by changing individual cells.

Second, we could iterate through the grid, one line and character, at a time, and ask ourselves each time: "what character should be at location (x, y)?" There are only a few characters that you have to print: open space, paths not in the solution, paths that are part of the solution, the start location, and the end location.

8 Turning in Your Solution

You must turn in your code using GradeScope.