Lab 14: Backtracking

Mar 22, 2018

Main Event

1. Setup and instructions

Today we'll use recursion to solve a maze. Before you begin, there are several files you will need to download:

- helpers.py Contains convenience methods for
  - determining if a coordinate is out of the board bounds;
  - building a maze from a text file; and
  - printing a maze.

  The functions that perform these operations are described in more detail below.
- maze.py A skeleton Python file you will develop.
- board.txt A maze, described as a text file. Again, you do not have to parse this file, the `helpers.get_board()` will do this for you (and can help you print it):

```
>>> board = helpers.get_board('board.txt')
>>> helpers.print_board(board)
o @ @ @ @ @
. . . . . @
@ . @ @ @ @
@ . @ @ @ @
. . . @ . x
@ @ . . . @
```

Once you have completed the assignment you can alter this file to try new mazes. The board file has the following format:

- o (`helpers.markers.start`) The starting position. This is (generally) the coordinate you should specify to start your program. In the provided board, the start position is at row-column 0-0 for convenience.
- . (`helpers.markers.open`) Denotes an “open” space. If you were working a maze using pencil and paper, this would be a blank position (to which you could move in a next step).
- @ (`helpers.markers.boundary`) A boundary. Again, in a pencil-paper based maze, this would be a wall that you could not cross.
- x (`helpers.markers.goal`) Gooooaaaaaallllll!

2. Main

You can think of `find_path` as determining what to do at a given position on the board. Put yourself in its place: the maze is described as a grid—what do you do?

Base case

You should first determine whether the coordinates specified are within the bounds of the board. A negative coordinate, or a coordinate that is larger than the board size, denotes an out-of-bounds request. The function `helpers.inbounds` can help in making this decision: given a board (a list-of-lists) and a row and column (as integers), it returns True of the coordinates exist within the board and False otherwise:

```
>>> l = [[0, 0, 0], [0, 0, 0]]
>>> helpers.inbounds(l, 0, 0)
True
>>> helpers.inbounds(l, 10, 5)
False
```

Unlike traditional Python indexing, this function considers negative values to be out-of-bounds. In the even that this function returns

Whether coordinates are in-bounds in an indication of whether you can make progress from this position. If you cannot, you should return the previous setup. If so, however, you should next consider the other two base cases: is this position legal or is this position the goal? If so—either are True—you should return the appropriate answer.

Recursive step

If none of the base case tests passed, it is time to recurse! Remember, that the recursive step should get us closer to the goal—in this case the `helpers.markers.goal`. Thus, we should call ourself with an updated coordinate. From this location, there are four possible positions to which we can move: up, down, left, and right. You are given the current coordinates as parameters to the function—how to move in these directions should be clear.

Given that we have four separate moves to make, we'll need to call ourself four separate times (with the new coordinates). If any of the recursive calls returns True, we should immediately return True; otherwise, we should move on to trying the next coordinate.

If none of the attempts gets us to our goal, we should end the function by returning False.

Backtracking works best if we leave some breadcrumbs to remember where we've been. Without breadcrumbs, we might recurse forever. To add breadcrumbs, after checking the base case, but before recursing, update the positions value to a character of your choosing. The only requirement is that this character cannot be a character otherwise used in the board.

Once we have finished with the recursive step, but before returning the final False, you should "reset" this breadcrumb to be the value that it was prior to changing it.

3. Tracing the path

To build an understanding of what is actually happening, add new print statements to `find_path` to log its activity. For this, use the helper function `indent_print`:

```
>>> helpers.indent_print(0, 'Hello')
Hello
>>> helpers.indent_print(5, 'Hello')
     Hello
```

The function prints your string, with the specified number of spaces in front. To get this to work, do the following:

1. Add a parameter spaces to `find_path`. By default, make it zero.
2. For each recursive call you make, increment the value of this parameter by one.

3. Prior to each return statement (in the line immediately prior to returning), add a call to `helpers.indent_print`. You should pass a string of your choice, and the `spaces` variable. The string should be unique; that is, don't use the same two strings within the `find_path` function.

4. If you haven't already, separate your four recursive call into four distinct if-statements (the alternative is to have a single if-statement, with the disjunction of all the calls). Prior to each if-statement, add a call to `helpers.indent_print` with a message that is *slightly* different than the message that gets printed before the potential return statement.

Run your code again—you should get an interesting trace displaying the winding and unwinding of the recursion.

The Python file developed to complete this question should be what is turned in for this lab.

Solution:

```
import helpers

def find_path(board, row, col, spaces=0):
    #
    # Base cases
    #

    # 1. Are we in bounds?
    if not helpers.inbounds(board, row, col):
        helpers.indent_print(spaces, 'nolegal')
        return False

    spot = board[row][col]

    # 2. Are we on the goal?
    if spot == helpers.markers.goal:
        helpers.indent_print(spaces, 'goal!')
        return True

    # 3. Is it okay to be here?
    if spot != helpers.markers.start and spot != helpers.markers.open:
        helpers.indent_print(spaces, 'bad spot:' + spot)
        return False
```

```
    #
    # Recurse!
    #
    forward_indicator = '>'
    backward_indicator = '<'

    previous = board[row][col]
    board[row][col] = '+' # breadcrumb

    helpers.indent_print(spaces, forward_indicator + 'left')
    if find_path(board, row, col - 1, spaces + 1):
        helpers.indent_print(spaces, backward_indicator + 'left')
        return True

    helpers.indent_print(spaces, forward_indicator + 'right')
    if find_path(board, row, col + 1, spaces + 1):
        helpers.indent_print(spaces, backward_indicator + 'right')
        return True

    helpers.indent_print(spaces, forward_indicator + 'up')
    if find_path(board, row - 1, col, spaces + 1):
        helpers.indent_print(spaces, backward_indicator + 'up')
        return True

    helpers.indent_print(spaces, forward_indicator + 'down')
    if find_path(board, row + 1, col, spaces + 1):
        helpers.indent_print(spaces, backward_indicator + 'down')
        return True

    board[row][col] = previous

    helpers.indent_print(spaces, 'nope: (' + str(row) + ',' + str(col)

    return False

board = helpers.get_board('board.txt')
helpers.print_board(board)
print('-' * 50)

find_path(board, 0, 0)

print('-' * 50)
helpers.print_board(board)
```

4. Deeper understanding

- *Within* the `find_path` function, print the board. If you print the board just after adding the breadcrumb, you will get an interesting trace of the steps your function took to reach the goal.
- What happens if you change the order in which your move? For example, instead of going up-down-left-right, try left-right-up-down. Do you notice a difference in your board trail?
- What happens if you don't leave a breadcrumb?
- Alter the maze file and try new boards! Remember, the power of our (relatively simple) algorithm is that it will work on any board size. What happens if you design a board with no solution?

For this question, what should be turned in are two text files. The first containing answers to the questions posed; the second being an altered maze.

Introduction to Computer Science

Introduction to Computer Science
jerome.white@nyu.edu

jerome-white

Learning computer science concepts through practice.