2023-04-03

Unit 10 Project

Maze Solver

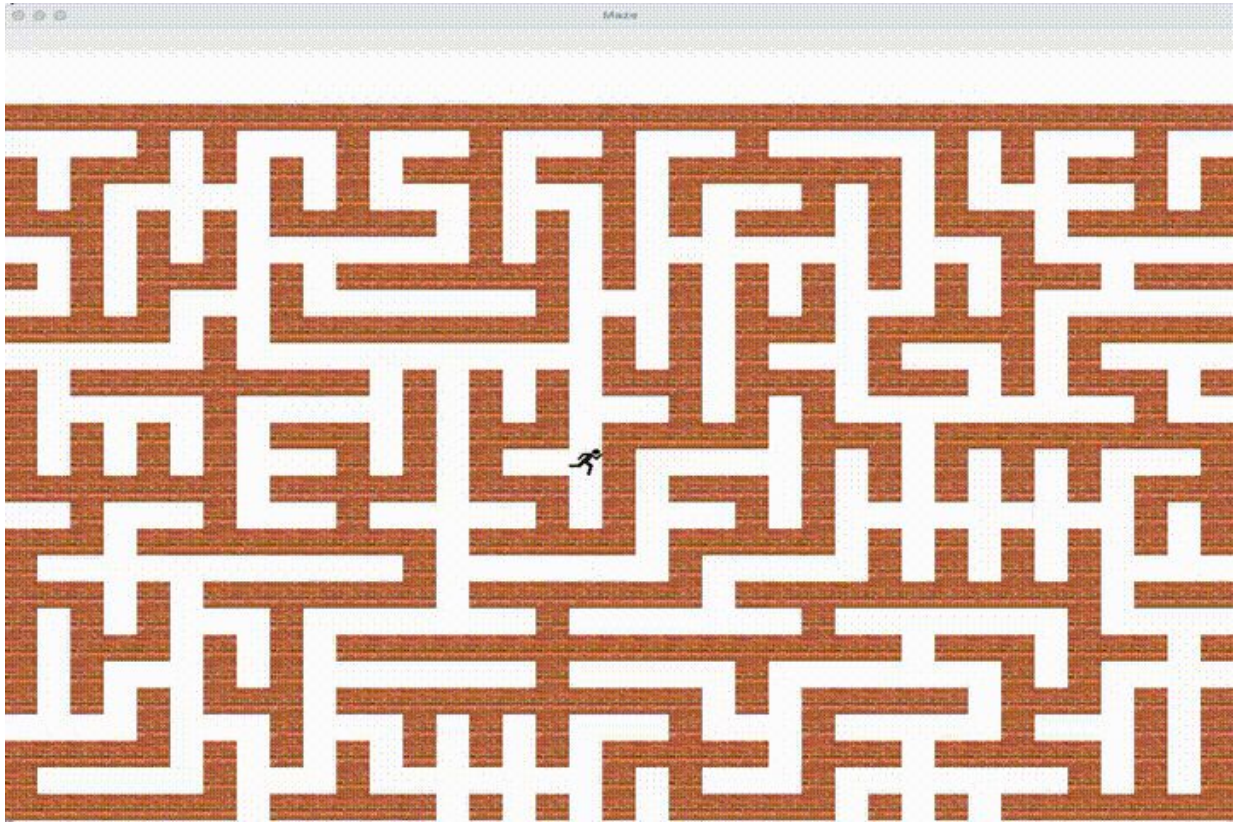
Maze Solver

Remember testing your Unit 9 Project to make sure your winning logic worked correctly?

Wouldn't it have been nice if the Maze Game had a cheat code that showed you how to escape the maze?

We can do it... with recursive algorithms!

The Unit 10 Project will be the same starting code as the Unit 9 Project... but making the computer solve the maze!



Right-Hand Rule

If you ever personally find yourself stuck in a maze, you can use the **right-hand rule** or **wall follower** algorithm to get out. Put your right hand on the wall, and consistently follow the right wall. This works for all *simply connected* mazes, that is, without island walls. (So not the one below.)

Since our computer doesn't have to physically walk the paths, we'll implement an algorithm that may do more walking but can always find the exit.

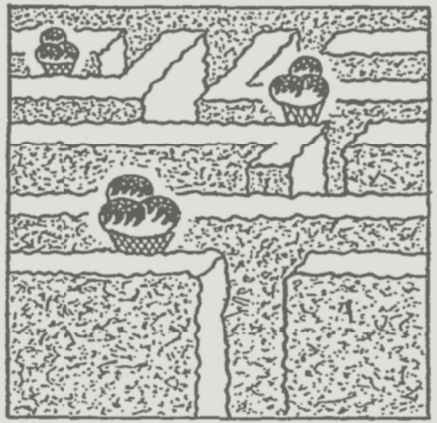
Why Right-Hand Rule for Mazes Works

To the Editor:

Guy Maxtone-Graham's criticism of D. Q. O'Brien's maze extrication procedure ("Bearing Right Can Make You Go in Circles," letter, Aug. 15) may lead many not to use the right-hand rule when it does work. If upon entering a maze, one immediately puts out one's right hand, touches the entryway wall and then faithfully follows the right wall, the exit will be found without fail.

As Mr. Maxtone-Graham points out, many mazes have unconnected, or island walls. If one were to stroll into a maze, become disoriented and then try to use the right-hand rule, one might unwittingly follow an island section of wall. This would indeed mean going "around the same circuit forever." If however, one starts with the entryway right-hand wall and never breaks contact with it, one will never become attached to an island wall. Thus, those who are consistent in applying Mr. O'Brien's right-hand-wall method, will find it to work without fail. ALEX JOYCE

New York, Aug. 15, 1989



Campbell Laird

Ice Cream Solution

To the Editor:

Several summers ago, while vacationing in England, my wife and I became completely lost in the maze at Hampton Court among schoolchildren who were also on holiday. Suddenly, we realized that most, if not all, of the children we met in the maze were eating ice cream cones. Those with small cones were in the same predicament we were in. Those with larger cones had obviously come into the maze later. Our problem was all but solved. To get to the entrance of the maze, we merely walked in the direction opposite from that being walked by the big ice cream cones.

For our solution to work, the following conditions are necessary: (1) You must take your holiday during summer when school is out; (2) there must be an ice cream vendor at the entrance of the maze, and (3) the weather must not be so hot as to melt your clues quickly. FRANK OWENS

Newark, Del., Aug. 17, 1989

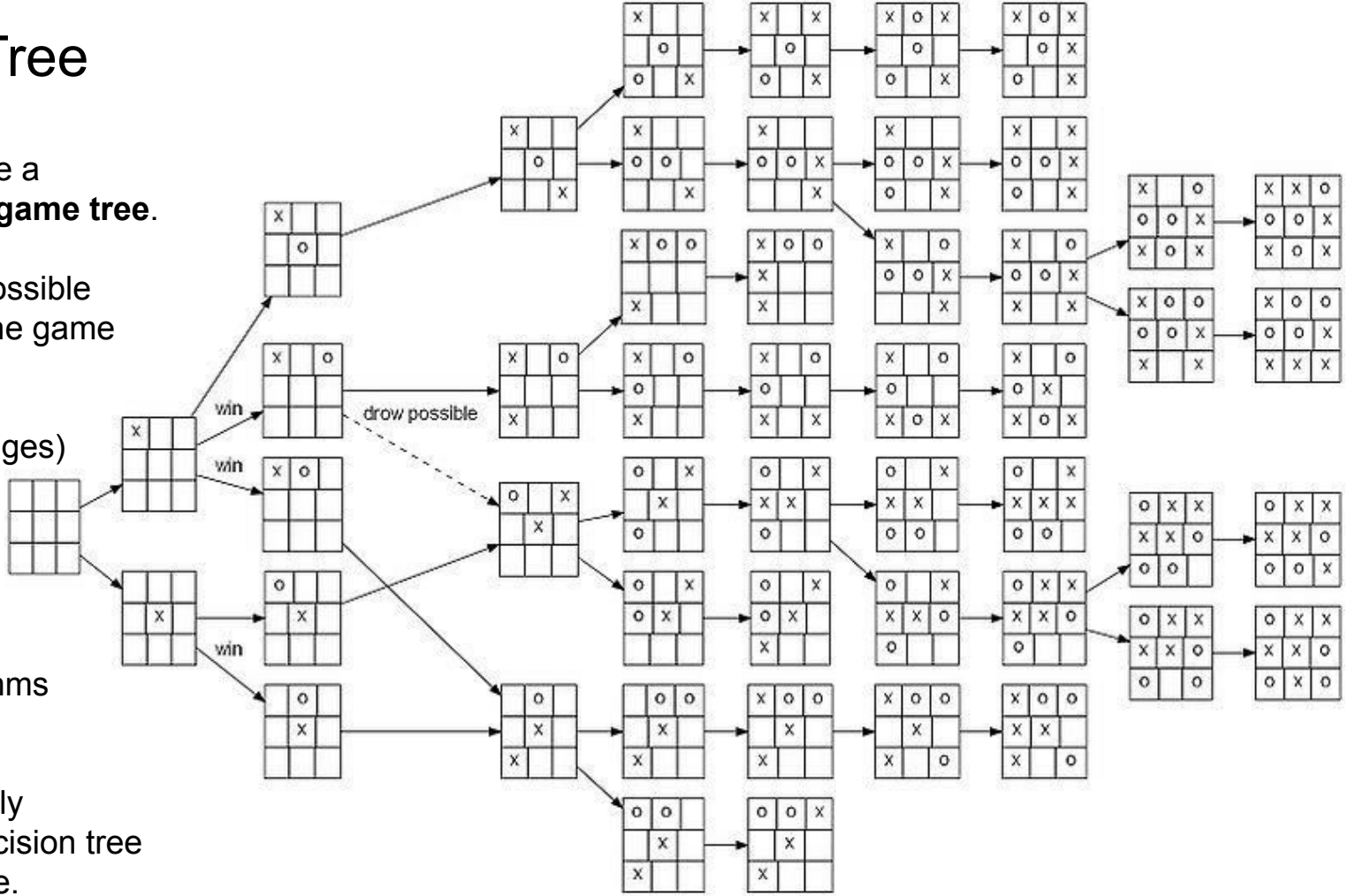
Decision Tree

Many games have a **decision tree** or **game tree**.

Each node is a possible configuration of the game board/state.

The branches (edges) are the possible moves that can be made from each state.

Computer algorithms that play games, including maze solving, are usually searching this decision tree for a winning state.



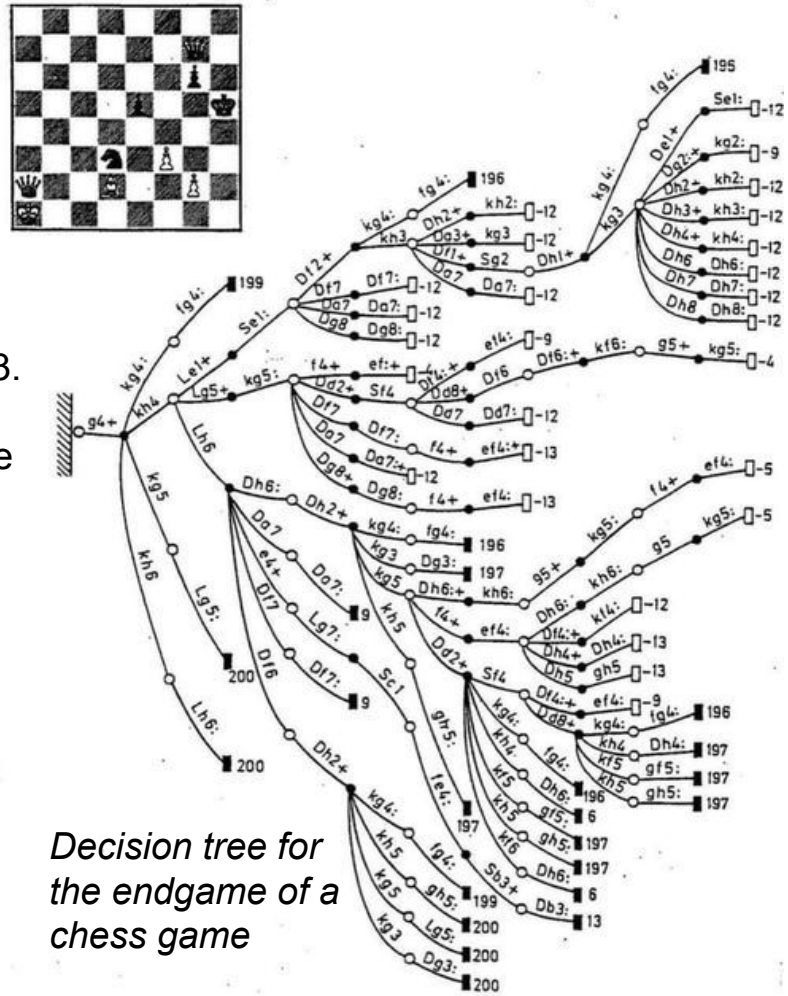
Decision Tree

In our maze, a decision tree node has at most four children: north, south, east, and west.

In Tic Tac Toe, there's a child node for each of the remaining empty spaces where a move can be made, so up to 9 for 3x3.

Chess and Go have enormous decision trees, because of the wide variety of moves that can be made and multitude of possible game states. This is why it took until we had advanced supercomputers in the '90s for the first chess-playing computer to beat a world chess champion. (Kasparov vs. Deep Blue, 1996)

A computer didn't beat a human champion at Go until 2016 (Google DeepMind)... ending 2,500 years of human Go supremacy.



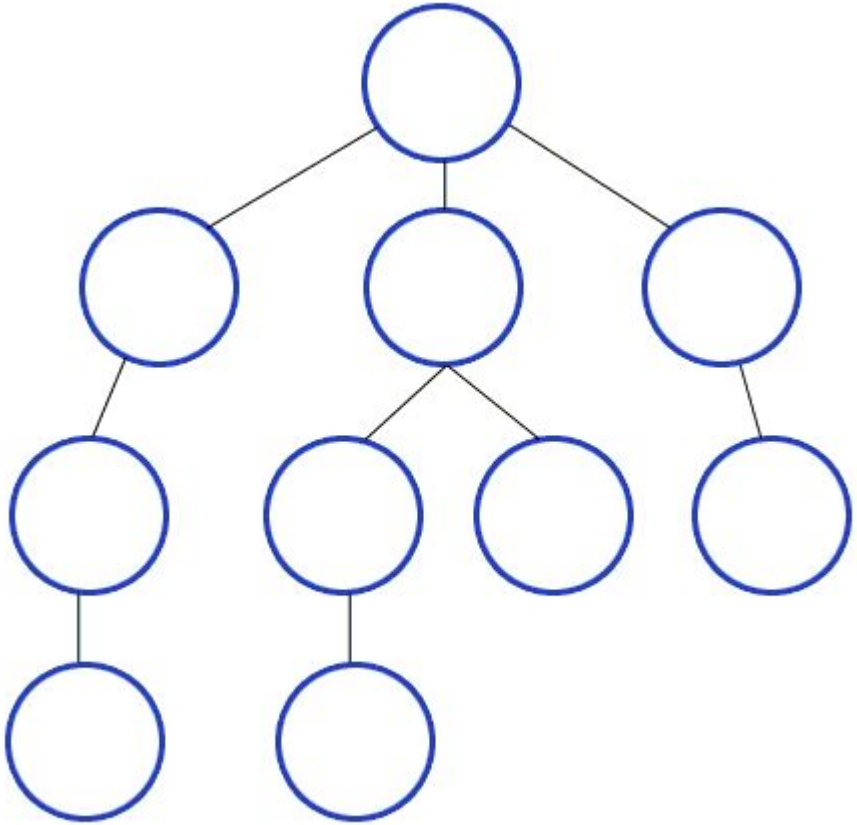
Decision tree for the endgame of a chess game

Depth-First Search

Depth-first search is an algorithm for traversing a recursive data structure such as a tree.

The overall algorithm is quite simple:

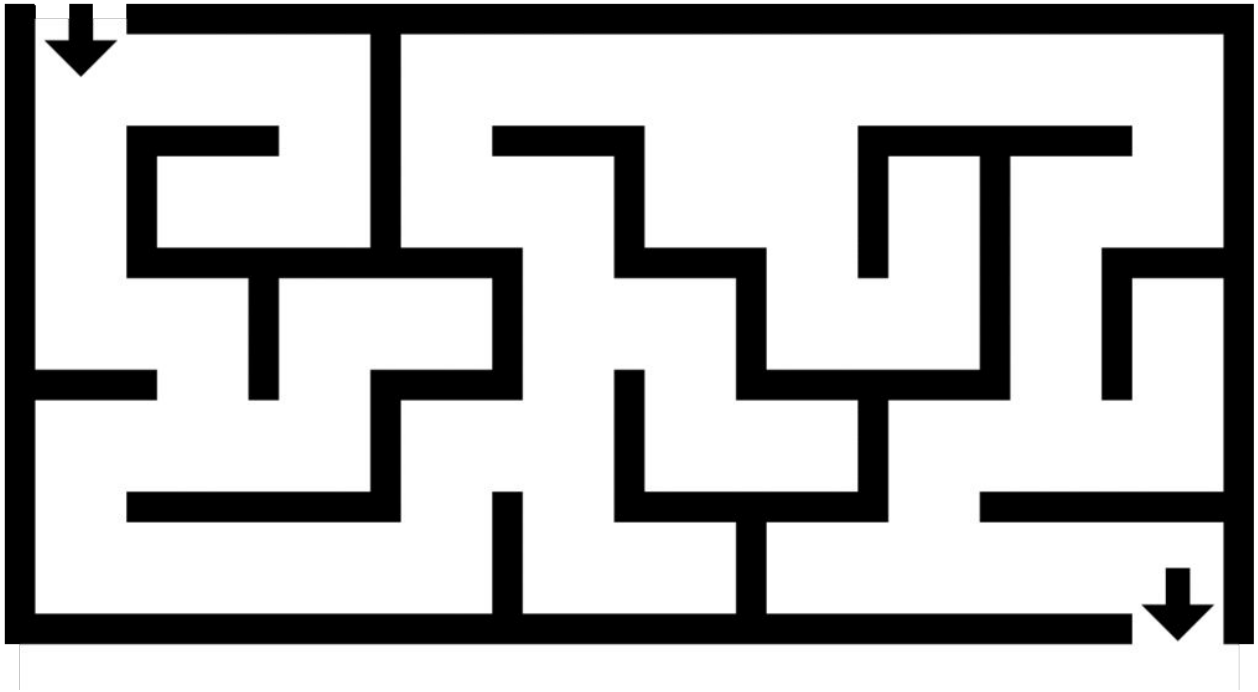
```
def visit(node):
        do something with node
        (this might return if node satisfies some
         criteria being searched for)

        for each child of node:
                visit(chld)
```



Depth-First Search

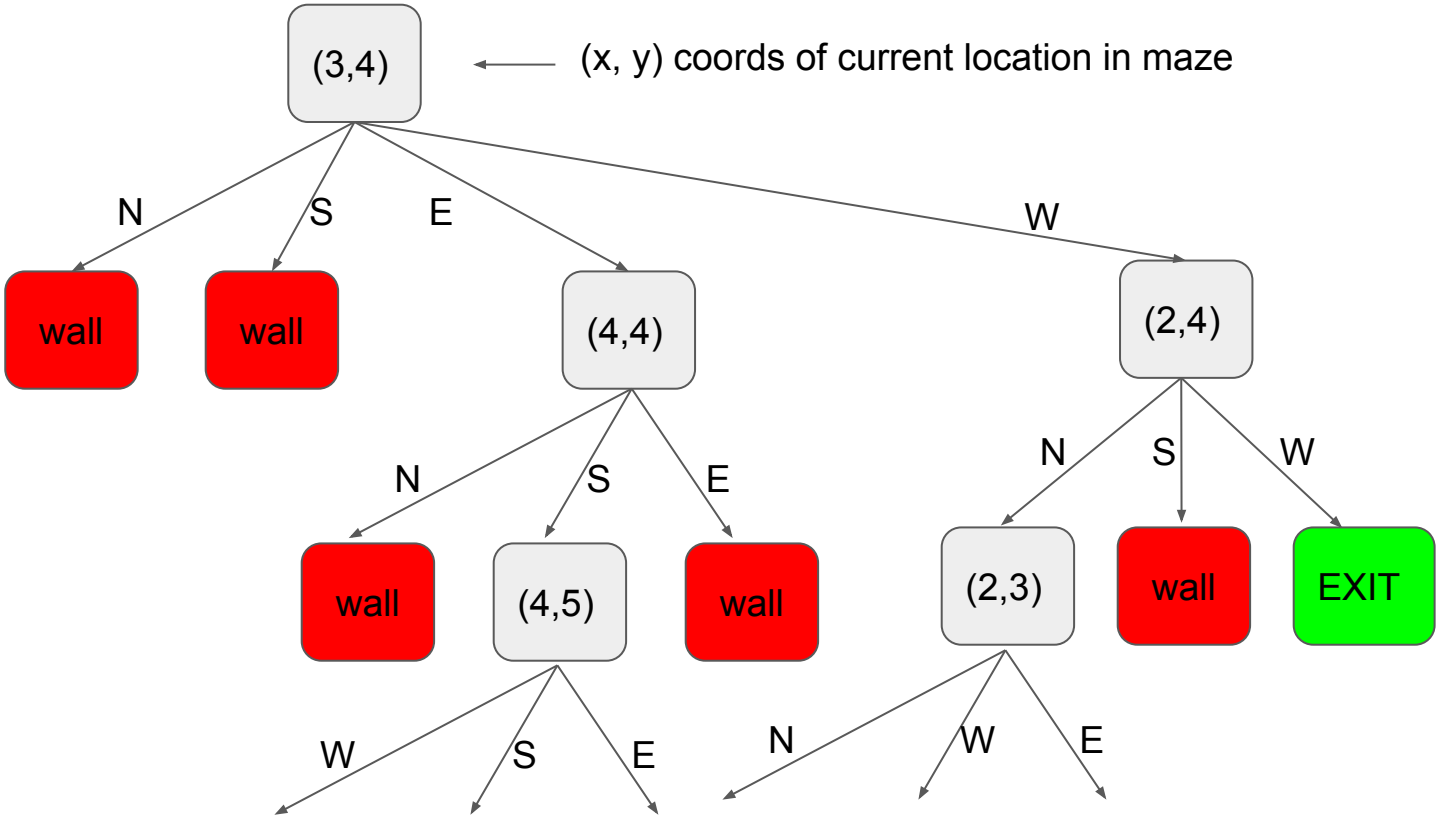
Whenever the depth-first search encounters a fork in the maze where multiple directions are possible, it recursively explores all of the directions.

Whenever all possible ways to an exit in one direction are exhausted, the algorithm "backtracks" by returning out of levels of recursion back to an earlier fork in the maze, and explores any other available directions.



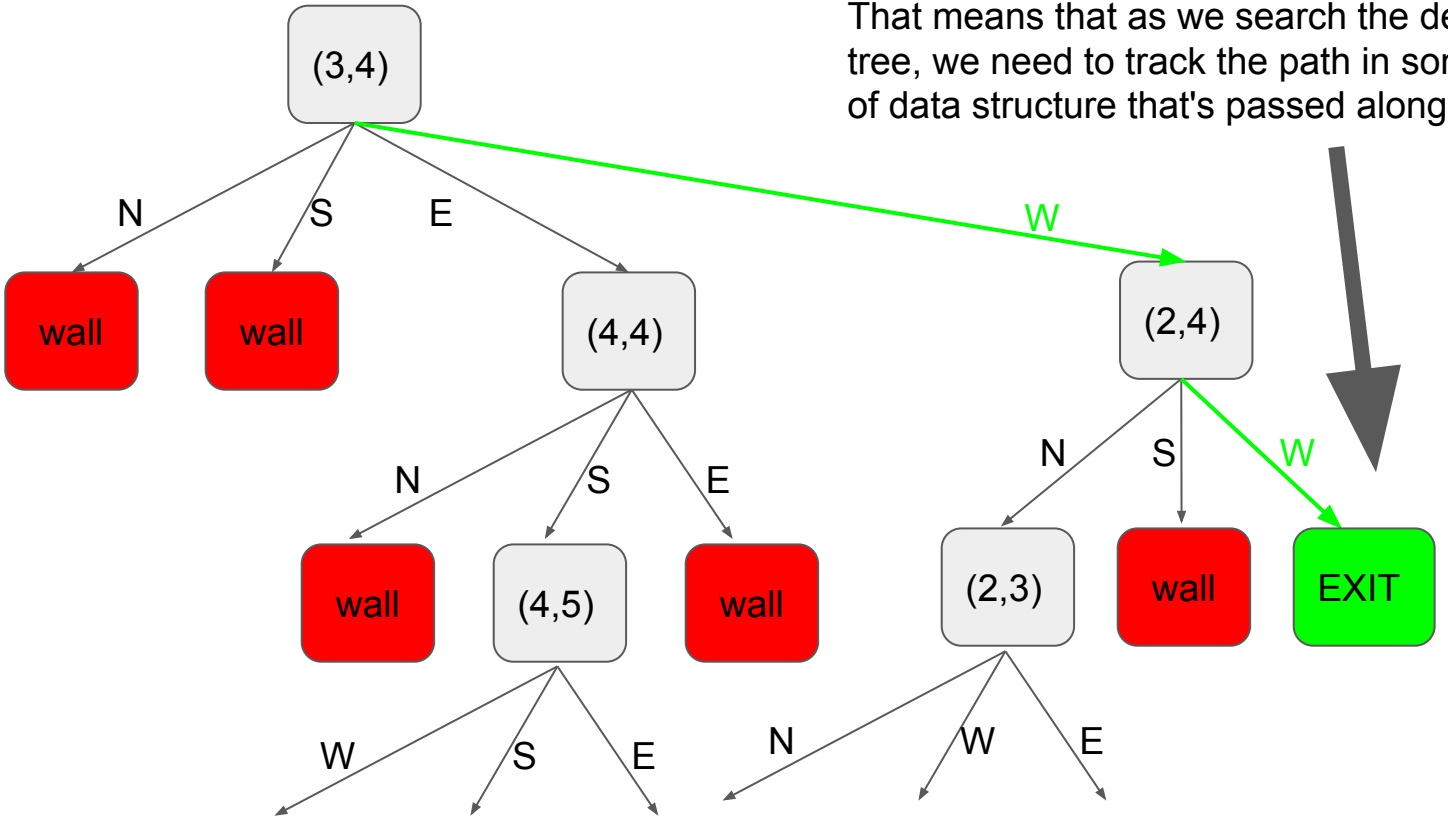
One way to visualize this is with the 2D maze itself. Another is to think of it as searching a decision tree.

Maze Decision Tree



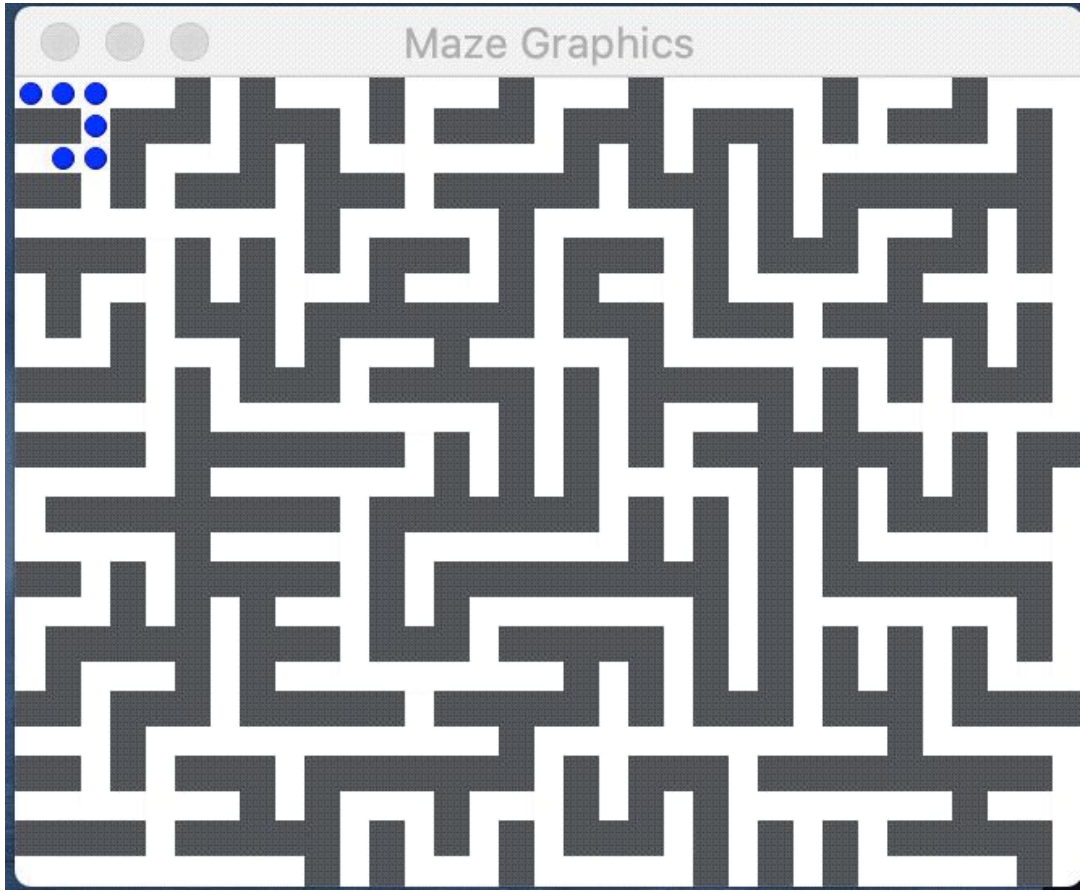
Maze Decision Tree

When we find an EXIT node in the decision tree, we return the path that got us there. That means that as we search the decision tree, we need to track the path in some kind of data structure that's passed along.



Here's another visualization of how the algorithm progresses.

You can see it trying each possible direction, and then exhausting all possibilities to find an exit in that particular direction and moving onto other directions.



Path FindEscapePath(int x, int y, Path previousSteps)

Base case:

If we're at the exit, then the winning path is the path that got us here! Return it!

If we're at a wall, then there is no path this way, return nothing.

Recursive case:

The recursive case should be smaller subproblems of this problem.

In this case, the smaller subproblems are going each available direction from this point, and **excluding** the current cell (along with any other cells in previousSteps) from any further consideration, since we don't want to go in circles.

Algorithm Pseudocode

```
Path FindEscapePath(int x, int y, Path previousSteps)
```

Base cases:

If (x, y) is an exit, return previousSteps + (x, y) as the winning path.

If (x, y) is a wall, this is a dead end, return null.

Recursive case:

For each adjacent cell (x', y'):

Call FindEscapePath(x’, y’, previousSteps + (x,y))

Compare any paths that are returned... return the shortest one.

Rubric

1. Customize the graphics for the maze again. You can simply copy over your graphics from the Unit 9 Project, or you can pick new graphics.	10 points
2. Your maze-solving algorithm should be demonstrated on more than one maze. Create at least five mazes of varying sizes (maze1.txt, maze2.txt, etc.)	10 points
3. Make it possible to switch between the mazes. The game should start on maze #1. Handle the key presses 1, 2, 3, 4, 5. Pressing one of these keys should switch to a different maze.	10 points
4. Implement the maze solving algorithm. Partial credit will be given for partially working algorithms, but **take care that your code compiles, and that the game doesn't crash or hang.**	40 points
5. In the method plotEscapePath(), plot the path out of the maze that was found by your algorithm. There is already code for highlighting cells with translucent yellow: call Maze.highlight, Maze.clearHighlighting.	20 points
6. Your algorithm should be optimal, that is, it should find not only a path out of the maze, but the minimum length path out of the maze.	10 points

Repl.it: Unit 10 Project

https://replit.com/@MsMolinaECHS/Unit-10-Project

Good luck!

Ask questions if you're stuck... this is a challenging project!