Analysis of solveMaze within MazeSolver.java

The method solveMaze requires a Maze and a MazeGUI object, which it uses to determine whether a maze is solvable (in which case it returns a path from the start of the maze, to the end in the form of an ArrayList), or whether it isn’t (in which case it returns an empty ArrayList).

Let n be the number of locations in the maze (width * height). Lines 40-93 contain four conditional statements, each of which are relative clones of each other, so I will explain how one of them has a time complexity of O(1), rather than all four, to be succinct. For instance, the if statement in lines 44-51 takes O(1) time since all commands within it are constant-time assignments. The if statement in lines 43-52 takes O(1) time since the it’s checking for a boolean value. Line 42 indexes a position within a double array, which is an O(1) operation. Finally, the if statement in lines 41-53 makes a comparison between two values, which is an O(1) operation. Since the if statements within lines 40-93 are all similar, the whole block (lines 40-93) have a total time complexity of O(1). Line 95 takes O(1) time because it is a simple return. Lines 22-23, 25, 27 are assignments/indexing operations and therefore also take O(1) time. The for-loop in lines 36-38 takes O(n) time, since it iterates through the list of maze tiles that precede the goal location to the start location and colors those tiles gold. Line 39 is a simple return, and therefore takes O(1) time. The if statement in lines 35-40 takes O (1) time since it is only checking whether one object equals another, and so it’s a constant-time comparison. Lines 29-34 are all O(1) operations, since they do not depend on the number of locations in the maze. Lastly, the while-loop overall (lines 28-94) only takes O(n) time (despite having an O(n) for-loop [lines 36-38]) due to amortized efficiency. That is, the for-loop (lines 36-38) is not called each time the while loop runs--in fact, significantly less times--and so, distributed over an arbitrarily large n, the time complexity for the while-loop (lines 28-94) becomes O(n), and since that is the most expensive operation of the method, the time complexity for the entire method is O(n).

Stack vs. Queue Solvers

The QueueAgenda seems to be more efficient when solving mazes that have open spaces, and a goal adjacent to the start. The StackAgenda, on the other hand, seems to be more efficient when solving mazes that have narrower paths, or those that have more walls than open space.

Neither solver appears to return a faster path to the goal consistently. The mazes contained within fastqueue.txt and faststack.txt demonstrate this. For instance, using a stack to solve the maze in fastqueue.txt provides a huge path, while solving with a queue does not. However, using a queue to solve the maze in faststack.txt also provides a huge path to the goal, whereas using a stack does not. It appears that mazes with goals significantly close to the start, or those unobstructed by many walls, are solved faster using a queue. However, if a goal is a winding path away from the start, it seems as though stacks do a better job, since they handle crossroads better. This can be seen in mazefile2; where the QueueAgenda adds all of the squares as part of its solution path, the StackAgenda adds only relevant squares (in that scenario). In any case, there doesn’t appear to be any evidence supporting the claim that one agenda type always finds shorter solutions than the other in all situations.