Daniel Nedrow

Homework 1

CEG 4110: Introduction to Software Engineering

The problem is to extract the contours of a solid square from within an n x n 2D array. The cells of the array representing the square have value 1, while the other cells have value 0. We don’t know the size or location of the square within the array, and we are to check all possible paths through the array to find the square. I’ve made the following assumptions: we begin the search in the top-left corner of the array, and the goal of checking all possible paths is to find the shortest path to uncovering the contours of the square.

Figure 1 is a flow diagram for the program execution to accomplish the above goal. Note that we immediately set the maximum allowable path length as n + 3, where n is the length of one dimension of the array to be searched. This bears some discussion.

Define the **path length** to be the number of array cells visited in order to uncover the contours of the square. We are only tasked with finding the contours of the square (we know it is solid inside). Since the shape in this case is guaranteed to be square, it is only necessary to find 3 of the 4 borders of the square: we can then use the side length of the square to deduce its overall contours. We will not wastefully increase the path length by tracing all the way around the square (and certainly not by uncovering its insides). Given that this allows for a shorter path length, I have found by observation that any given square’s contours can be deduced with a path length of less than n + 3. I do not have a mathematical proof of this, but neither was I able to find a square location/size which disproves it.

Why belabor this point? The problem tasked us with checking “all possible paths.” Since this is a wide-open array and not a maze (which would at least restrict our movement to a manageable set of cells), the combinatorial explosion of checking all paths is staggering. I found that to even be able to reasonably check a 10 x 10 array with my laptop, a restriction needed to be placed on the algorithm to stop following a path as soon as it was shown to be at least as long as the current minimum path found (which we initialize to be n + 3 for further time savings).

As shown by Figure 1, we use a depth-first recursive search to check all possible paths through the array. If we have uncovered the contours of the square, or if we are about to exceed the current minimum path length, we stop searching along these lines and return control to the calling function for further exploration. If we still have an active search for this path, we recursively call the search function for each of the 8 possible directions (including the diagonals). Eventually, the function returns control to the calling program, and the program displays the minimum path length found, the original secret array, and the shortest path taken to uncover the contours of the square inside. Note that there will be many ties for shortest path through a wide-open array, so only the first path of this shortest length will be chosen.

Figure 2 represents a function to check whether the contours of the square have been uncovered. The diagram is straightforward, but again note my goal is to uncover only 3 distinct borders of the square, since the 4th border can be deduced. To uncover the location of a border we merely need to find a single cell on that border. Clearly, such a cell would be identified as either having an empty cell beside it (to the north, east, south, or west), or by similarly bordering the outer edge of the array. (This, however, is an implementation detail and is not covered by the flow diagram).

Finally, it should be noted that the recursive depth-first search function requires a lot of information to be passed forward each time it is called. In order to check all possible paths, we must regard calls to the function as representing a new path. The algorithm I’ve described needs to keep track of a lot of information for each path, so it is useful to encapsulate that information in a single Path object (otherwise the search function’s parameter list would be extensive). With this encapsulation, the search function can be called with a Path object and a couple of integers to indicate where we are about to move in the x and y direction in the array. Figure 3, below, is a possible UML representation of the Path class.

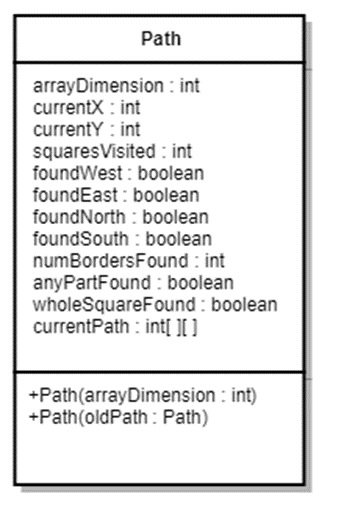


Figure 3: UML Diagram for Path Class

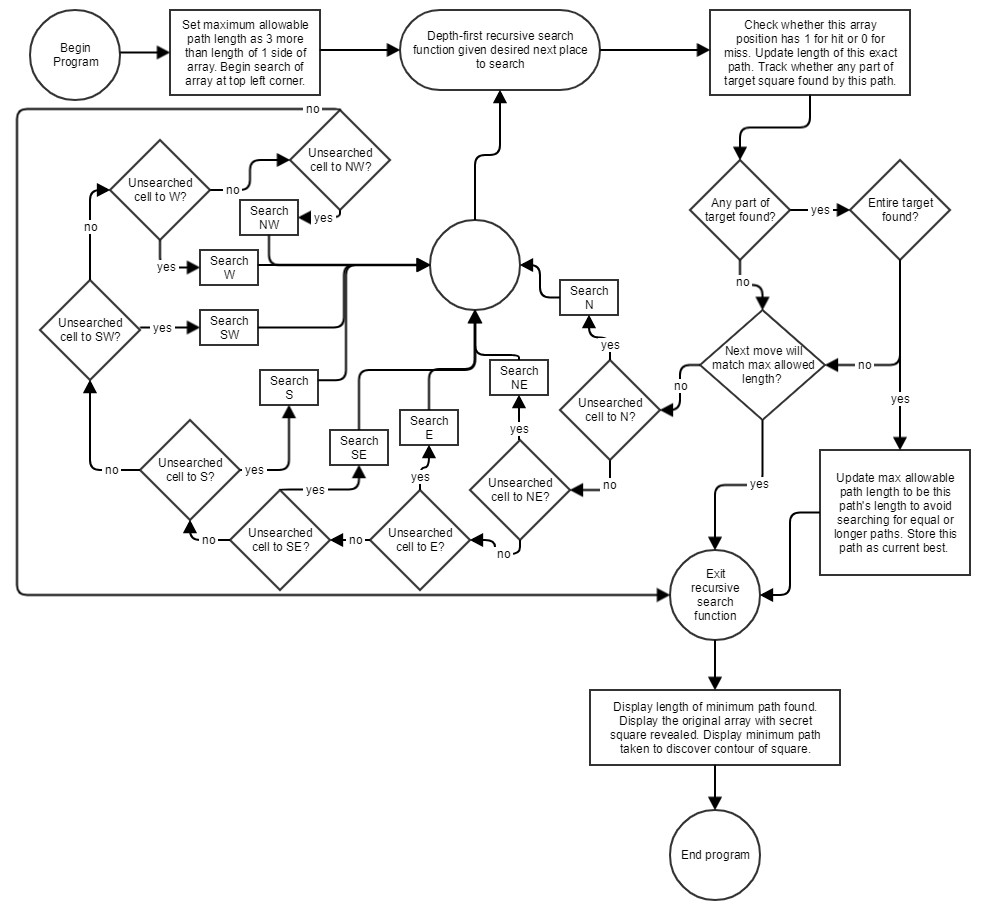


Figure 1: Flow Diagram for Program Execution with Depth-First Recursive Search Function

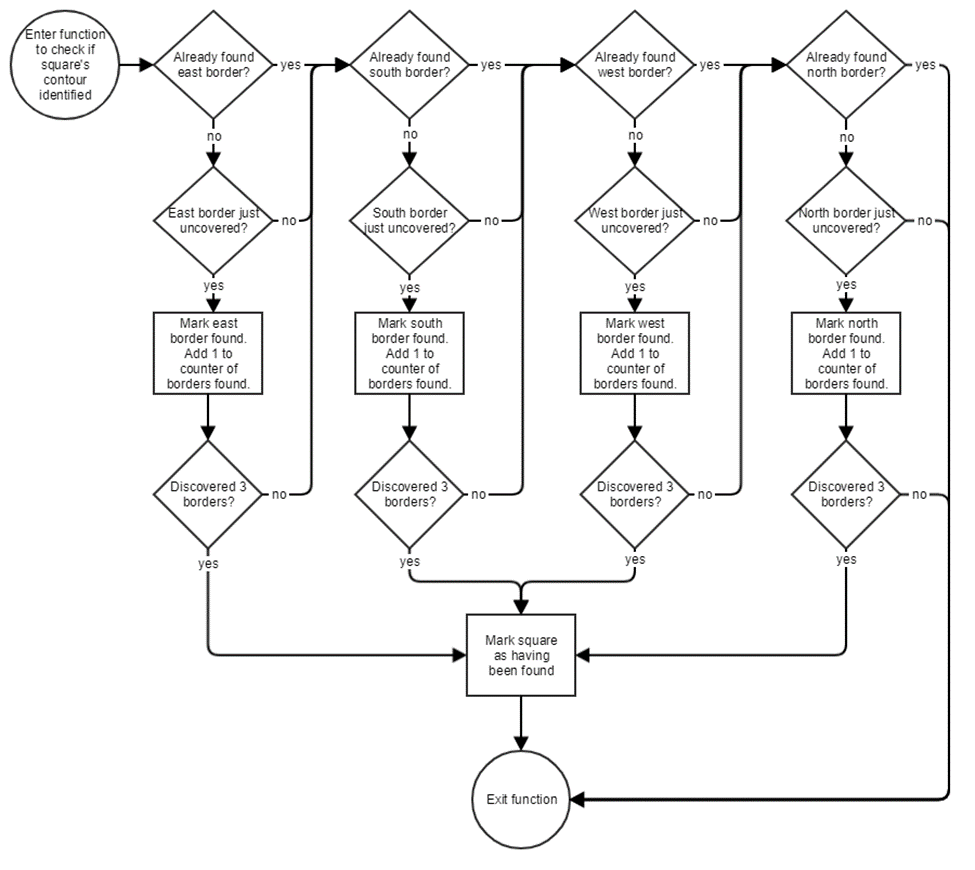


Figure 2: Flow Diagram for Function to Check if Square's Contours Have Been Found