1 Readings

DSO Ch. 5, 7 (8-Queens project 11), 9, 15.1. For Part 2 add: DSO Ch. 10, 15.3. For Part 3 add: DSO Ch. 8.

2 Assignment

Your assignment is to write a program that uses a recursive functions, non-recursive functions and choices of data structures to find solutions to a puzzle. The puzzle is to find paths within a maze. The algorithmic approach is finding all paths from a start square to a goal square is similar to the approach that solves the 8 queens problem. One choice for the chosen data structure is to use a list that will represent the current partial solution so far.

Another choice for the data structure is to maintain information about the progress of the search right in an array that represents the maze. This second choice is an alternative for part 1 below; but it is necessary for parts 2 and 3.

The project will have 3 parts:

1. Find, print within the maze, or count ALL paths that begin at a given square and end at another given square (60 points).

   The executable file produced by your build script will be be named proj5.

   The relevent input data (see below) will be read and the part 1 operation will be performed when the command

   proj5 --all

   is given.

   Data structure choices:

   (a) List representing a partial solution so far;
       plus one local extant variable (in each activation) to keep track of which direction to try next.

   (b) List representing a partial solution so far (so it can be printed);
       plus data maintained in the maze array (only ONE array!) used to speed up the algorithm;
       plus one local extant variable (in each activation) to keep track of which direction to try next.

   (c) Data maintained in the maze array used to speed up the algorithm and print the solution,
       plus one local extant variable (in each activation) to keep track of which direction to try next.
See details and a simple example below.

2. Find and print within the maze one **depth-first search tree (DFS)** that is rooted at a given square (10 points). The maze size, one given square, and the maze itself is read and the DFS tree is printed when the command is “`proj5 --depthtree`”. Here’s an example. On input

```
3 0 0
0 0 1
0 0 0
1 0 1
```

the command **proj5 --depthtree** might print:

```
 3 0 0
 0 0 1
 0 0 0
 1 0 1
  X v 1
  ^ < <
  1 ^ 1
```

This can use the data structures:

- Data maintained in the maze array used to speed up the algorithm and print the solution tree inside the maze,
  plus one local extant variable (in each activation) to keep track of which direction to try next.

3. Find and print within the maze one **breadth-first search tree (BFS)** that is rooted at a given square (10 points). The maze size, one given square, and the maze itself is read and the BFS tree is printed when the command is “`proj5 --breadthtree`”. Here’s an example. On input

```
3 0 0
0 0 1
0 0 0
1 0 1
```

the command **proj5 --breadthtree** might print:

```
3 0 0
0 0 1
0 0 0
1 0 1
```
This can use the data structures:

- Data maintained in the maze array used to speed up the algorithm and print the solution tree inside the maze;
- plus a queue to help choose which square to process next in order to try to make the solution tree bigger.

Besides these 80 points for functionality, 10 points will go for writing pre-conditions and post-conditions for each function you write, and 10 points for program structure, the build script, use of RCS, etc.

3 Part 1

The description below is based on using data structure choice (1) above, which is closest to the Main and Savitch’s slide and textbook presentation. You may choose either to follow these directions, or to solve the problem in a different way perhaps as outlined above.

Once you get Part 1 solved, you should be familiar enough with the problem domain to move on to Parts 2 and 3. Lectures and an additional handout will be provided next week to support those.

The 8 (or “n”) queens problem is defined and a non-recursive, stack-based solution outline is given in problem (11) at the end of DSO Ch. 7 on stacks; and will be covered upcoming lecture. Details of this assignment and strategies for its solution will begin in Lecture 23 of April 25.

The key solution function will use a recursive call inside a loop (or simply 4 recursive calls). It will be used to search a maze and print every possible path from the start to the destination. The input will come from the terminal (standard input file) and follow example:

```
4 0 0 3 3
0 0 1 1
0 0 0 0
1 0 1 1
1 0 0 0
```
The maze is an \( n \) by \( n \) grid of squares, with \( n \leq 24 \). The first input number \( n \), in decimal, will signify the number of rows and columns in the maze. The next two numbers will indicate the start square in terms of its row (how far down) and column (how far across) indicies in that order. The last two numbers will indicate the destination square similarly.

The remaining input numbers define the shape of the maze in the following way. These numbers will consist of \( n \) 0s or 1s in each row, with a total of \( n \) rows (giving \( n^2 \) 0s or 1s). They will correspond position by position to the squares of the maze.

We will imagine a maze-traversing “robot” who starts in the start square and tries to find various routes to the destination square.

If 1 appears in a square, that square is “filled”, so the maze exploring robot can not move into or through that square. If 0 appears, the square is empty and the robot can move through it. It (he, she, ??) can’t move through corners of squares: a square has at most four neighbors, left and right (in the same row) and upper (in the row above, with index 1 less), and lower (in the row below).

Each route, or solution is a graph-theoretic simple path. A simple path from start to destination must begin at the start square, end at the destination square, and have no more than one appearance of each square. In other words, when he follows one solution, the robot never goes back again to a square he (she, it ??) had already visited.

The only test inputs for grading will have the correct number of 0s and 1s, and the start and destination will indicate empty squares in the maze. Some test inputs will have no solutions, some will have one, and others will have many.

Your program must first print a diagram of the maze it read by printing the first 5 numbers first (separated by spaces) followed by rows of space separated 0s and 1s.

Your program should then print each path from the start to the destination square. Each path should however be printed inside a maze diagram like the one above, preceded by a blank line, but with the characters X, <, >, ^, or v in each square of the path. Character X will indicate the start square.

Characters <, >, ^, or v in a square will indicate the direction (left, right, up, down) of the first step the robot would make in order to go from the square BACK to the start square along the solution path.

Finally, after printing zero or more diagrams each showing a different path, it must print a blank line and a line like that in the example below with the number of solutions it found. The program should then exit.

For example, if the input is

\[
\begin{array}{c}
3 & 0 & 0 & 1 & 2 \\
0 & 0 & 1 \\
0 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
\]

your program should print:

\[
\begin{array}{c}
3 & 0 & 0 & 1 & 2 \\
0 & 0 & 1 \\
0 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
\]

4
Your program might print the solutions in a different order than ours, since the order will depend on the organization of the four recursive calls described below and in the lecture. We will test your program mainly by checking that it finds the correct number of solutions for various maze inputs. Note that it must properly handle mazes that have no solutions, for example, the one completely filled except for the start and destination squares.

Use a two-dimensional array to represent the maze. To represent the path with a list, use a linked list of pairs of integers:

```c
struct PathListNode
{
    int row; // alternative: use a structure type member
    int column; // perhaps struct {int row; int column} data;
    PathListNode* next;
};
```

Each pair represents the row and column of one square on the path. It is convenient to keep the path in reverse order (store a newly found square in it like a stack): the last square on the path, which is your present location, is the first square on the list, and therefore the easiest to access.

Write the recursive member function function
```c
void PrintSolutions(PathListNode * pLast),
```
(or make it return the count integer) which finds, prints and counts all paths which extend the path ending with the square in `*pLast` to a path that reaches the destination. The base case detects that the present location is equal to the destination square. Therefore, print the diagram displaying the list back from `pLast`, increment the solution counter and return. In the recursive case, use a loop (or simply four statements) to consider all four neighbor squares. If a neighbor square (1) is not filled, and (2) is inside the maze and (3) not already in `*pLast` or any of its predecessors, that square is OK. For each OK square, add that square to the path, forming the node `*pNew`, and call
```c
PrintSolutions( pNew )
```
It is essential that a square already on the given path is not OK – otherwise the algorithm will go in circles.

The program reads the maze and forms a single node path containing only the start square. Then it calls `PrintSolutions( ... )` on that path. So this call to `PrintSolutions( ... )` will print all the paths that extend from the start square to the destination. Easy!

Your program should be structured so the crucial functions, the maze array, perhaps the solution counting variable, etc. are members of a class named `Maze`, coded with a separate header and implementation file, plus a separately compiled “main” module. The linked list node type might be declared outside or inside this class.
Other requirements of previous projects (use of RCS for any credit at all, a build script, no core/object/executable files, written pre- and postconditions to document functions, etc.) remain in force and will affect your grade.

Submit your work to project name Proj5

4 Project Management Suggestions/Recommendations

1. Study and master the problem and solution algorithm: Try it out on paper, etc.

2. Design and then implement the code to read the input, create an array representation of the maze, and print the maze. Test and debug it!

3. Design, implement and test the code to declare structures and classes dealing with paths; including a test driver that will read number pairs you type in and create the corresponding path. If you feel lazy, just hard-code function calls to build some paths so you can test your path builder and printer. Plan to throw that test driver away! But first, use it to debug the code that prints a path in the maze diagram. You might want to program invariant testing assertions in your path building functions to catch bugs that result from including a filled square in a path.

4. Develop one or more member functions to help tell if a given number pair is a square, an empty square, in the current maze; whether or not that square is already in a given path, and to help find a square’s neighbors.

5. AFTER you are sure you code to print paths in the maze works, then work on implemention and testing the actual solution search algorithm.

Notice that some or all of the aspects under items 3 and 4 might be mixed up. However, item 1 and item 2 should definitely be done first and item 5 be done last.

5 Remember...

All previous project requirements are still in force and will be graded: use of RCS, build scripts, prohibition of object, executable and ESPECIALLY core files in submissions, separation of interfaces (class definitions), implementations, and uses of functions and classes by means of included header files and separately compiled implementation files, documentation using pre- and post-conditions, etc. (Sorry for the repetition..)

The executable build script named build.sh should build an executable file named mazer. The submission should be a single directory submitted using the commands

```
turnin-csi310 -c csi310 -p Proj5 ...
turnin-csi310 -c csi310 -p Proj5 -v
```

were ... denotes the directory name.
5.1 Data Structure Alternatives

If you want to, you may use more efficient alternative to searching the linked list for testing if a square is on the path found so far. In this alternative data structure strategy, the recursive procedure `PrintSolutions()` writes a value different from 0 or 1 an array elements when the corresponding square becomes part of the path found so far. Also, the procedure changes that element’s value back to 0 when the procedure backtracks. For this strategy to be more efficient than searching the linked list, it must not copy the array. If the array is copied for each recursive activation, the program will be much less efficient than if the linked list is searched. (Tip: The final exam will have a question that addresses these issues.)

Another alternative, which will be used for parts 2 and 3, will be to make a 2 dimensional array of characters to represent the maze plus record which squares are in the current path. The first square, the start square, will contain 'X'. When a square is included in the current path, the '0' in it is replaced by a direction character '<', '>', '^', or 'v'. When that square is removed from the current path, the direction character is replaced by the '0'. It is easy to create the function that distinguishes a square containing '0' from a square containing a different character.

Etc, Acknowledgement

You might want to play with a version that only prints the original maze and the number of solutions, so you can see how many solutions there are, and how long it took the computer to find them all, for various sizes of mazes.

A research oriented project would be to generate random mazes, and see how the number of solutions varies with the density (ie., probability) of empty squares.

This assignment is a revision of one designed by Prof. Andy Haas, Computer Science Department of the University at Albany.