Notice: No more lab assignments!

Your assignment is to write a program that uses a recursive function and a data structure suitable for managing a list to find all solutions to a puzzle similar to 8 queens problem. The 8 (or “n”) queens problem is defined and a non-recursive, stack-based solution outline on page 376; and it was covered in a lecture.

Details of this assignment and strategies for its solution are given in Lecture 23, of April 22.

The key solution function will use a recursive call inside a loop. The function will 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 will look like this:

```
5
0 0 0 1 1
0 1 0 0 1
0 1 1 0 1
0 0 0 1 1
1 1 0 0 0
```

The maze is an n by n grid of hexagons, with n < 64. n will come first in the input.

If 1 appears in a hexagon, that hexagon is “filled”, so the maze exploring robot can not move into or through it. If 0 appears, the hexagon is empty and the robot can move through it. You can’t move through corners of hexagons: a hexagon has at most six neighbors, upper left, upper right, right and left on the same row, and lower left and lower right.

Start is the top left hexagon, finish is the bottom right hexagon. The corresponding numbers will be 0 for all test inputs we will use for grading.

Your program must first print a diagram of the maze it read using correctly indented (as shown above) rows of 0’s and 1’s.

For each path from start to finish, it must then print a diagram like the one above preceded by a blank line, but with the letter P on each hexagon of the path.

Finally, after printing zero or more diagrams showing a solution path, it must print a blank line and the line shown in the example below with (of course) the number of solutions it found.

For example, if the input is

```
2
0 0
0 0
```

your program should print:

```
0 0
0 0
```

```
P P
O P
```
Number of solutions = 4

Note that the 2nd and 3rd diagram show the same hexagons are in their paths, even though the paths are different.

Your program will probably have print the solutions in a different order than ours, since the order will depend on the organization of the six-fold loop 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 upper left and lower right hexagons.

Use a two-dimensional array to represent the maze. To represent the path, 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;
    struct PathListNode* next;
};
```

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

Write the recursive member function function void PrintSolutions(PathListNode * pLast), which finds and prints all paths which extend pPath1 until it reaches the finish. In the base case, the present location is equal to finish. Show the diagram displaying the list back from pLast and return. In the recursive case, use a loop that considers all six neighbor hexagons. If a neighbor hexagon is not filled, or off the edge of the maze, or already in *pLast or its predecessors, that hexagon is OK. For each OK hexagon, add that hexagon to the path, forming the node *pNew, and call PrintSolutions( pNew )

It is essential that a hexagon already on given 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 hexagon (0, 0). Then it calls PrintSolutions( ... ) on that path.

As outlined in the lecture, your program should be structured so the crucial functions, the maze array, 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 requirements (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 proj6

If you want to, you may use possibly a possibly efficient alternative to the linked list to represent the hexagons on the path found so far. Similarly, 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 hexagons.

Acknowledgement

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