1 Assignment

Your assignment is to write a program that uses a recursive methods, non-recursive methods and various of data structures to find paths.

1. Up to 60 points will be awarded for a program to find and print one path (if any) from a given start square to a given destination square using the marking and backtracking method explained in the lecture. Another up to 20 points will be added if your program prints the path as the letters A, B, C, ... IN THE MAZE printout, instead of the sequence of coordinates like (0,0), (1,0), (1,1), (1,2).

The Java file named findPath.java (together with other .java files you submit) must compile to produce an application program so that when findPath runs, it reads the problem from standard input and prints a solution as detailed in the lecture notes.

2. Another 10 points will be awarded for another (separate) program that labels each empty square with the minimum number of steps in a path from the start square to it. To do this, the program should use a queue data structure to implement breadth-first graph traversal. The number of steps is called the length of the path. The program should be named findDists.

3. Another 10 points will be awarded for extending the findDists so that it prints all the minimum length paths from the start square to the destination square.

4. Submit your work to project name Proj 08 in one directory.

2 findPath

For example, when the input is:

```
3 3
0 0
2 1
0 0 1
0 0 0
1 0 1
```

the output should be:

```
Input maze:
0 0 1
0 0 0
1 0 1
```

```java
3 3
0 0
2 1
0 0 1
0 0 0
1 0 1
```
One solution:
A B 1
0 C 0
1 D 1

(Your solution might be different because the order of trying cases is up to you.)

When there is no solution, print that conclusion, as with input:

3 4
0 0
2 1
0 0 1 0
0 1 0 0
1 0 0 0

Print:
Input Maze:
0 0 1 0
0 1 0 0
1 0 0 0

No solutions.

3 Labelling

The algorithm uses the operation “mark the square visited”. You should finish the Maze class design from the lecture, so that the maze data can be shared among methods belonging to the class. Therefore, the Maze class should have a data member (a partially filled 2-dimensional array) for inputting and holding the maze itself, plus another partially filled 2-dimensional array for holding the “marks” that record that a square has been visited.

4 findDists

The algorithmic approach which calculates marking values when a square (more generally, graph vertex) is visited is called “labelling”.

The algorithm that should be used for solving the findDists problem is a non-recursive labelling algorithm in which the squares to be processed are stored in a Queue. (A Queue is an ADT that stores a sequence of entries, where one end, called the tail is used for inserting an entry, and the other end, called the head is used to read or delete an entry.)

The idea is to begin by putting the start square in the queue, labelling it start square by 0, and label all the others by a value that signifies “not visited yet”. Then, we progress by labelling the squares by the minimum number of steps from the start square as we visit them.

The square $S$ at the head of the queue is processed as follows:

1. Remove $S$ from the queue.
2. $S$ should have been labelled; let $D$ be the value of its label. Calculate $D + 1$.

3. Process $S$ as follows: Check each neighbor $N$ of $S$.

- If $N$ is not labelled, label $N$ by $D + 1$ and insert $N$ at the tail of the queue.
- If $N$ is labelled, do nothing with $N$.

The algorithm is finished when the queue becomes empty.

DONT PROGRAM IT NOW!!!! Draw a maze, and try out the algorithm on paper. Compare to this example:

```
3 5
0 0
2 1
0 0 1 1 0
0 0 0 1 0
1 0 1 0 0
```

the output should be:

```
Input maze:
0 0 1 1 0
0 0 0 1 0
1 0 1 0 0
```

```
Distance labels:
1 2 X X U
2 3 4 X U
X 4 X U U
```
Note:

1. The goal square input is ignored.

2. The display of the labels within the maze uses the characters X to denote a “filled square”, U to denote the empty square whose that never got labelled, and an integer to denote a label value.

You don’t have to (for full credit) make the output look pretty when there are label values greater than 9, but you can if you wish.

**Extending findDists**

After the labels are computed and printed, run a recursive backtracking and printing algorithm that prints ALL the paths with a minimum number of steps from the start square to the goal square. The above example continued:

**Minimum length solutions:**

A B 1 1 0  
0 C 0 1 0  
1 D 1 0 0  

A 0 1 1 0  
B C 0 1 0  
X D 1 0 0

The number of minimum length solutions is 2

Tip: Every step in a minimum length solution must go from a square to another square with label is exactly 1 more.

**5 Notes**

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 goal 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 by printing rows of space separated 0s and 1s.

After printing either a solution path expressed with coordinates, or one diagram with a solution, or the message that there are no solutions, the program should then exit.

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++
class PathNode {
    int x;
    int y;
    PathNode prev;
};
```

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 `void PrintSolutions(PathNode 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 `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`.

6 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 methods 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.

**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 projects given by Prof. Carrano, and other textbook writers or faculty such as Prof. Andy Haas, Computer Science Department of the University at Albany.