CSE 373 -- Autumn 1999 -- Programming Assignment #2

(*due Monday, October 25*)

Introduction: For this assignment, you will write a program that finds shortest paths through mazes. This program will give you practice with Stacks and Queues (and probably Lists as well), and is similar to search algorithms that might be used by AI programs to search for legal moves in games or for specific information on the web.

We will supply you with a C++ Maze class that allows you to read a maze from a file and move around it a square at a time. Your goal will be to find the *shortest* path (in terms of the number of moves) from the starting position in the maze to the goal. There may be multiple paths from start to goal, so once you find the goal you will probably still need to do more searching to determine whether a shorter path exists.

Search Algorithms: You will need to implement two search algorithms: depth-first search (using the Stack ADT, *not* recursion), and breadth-first search (using a Queue). For both cases, you will need to make several design decisions. For example:

- What information should be stored on the Stack/Queue to allow backtracking and ensure that every path is eventually considered?
- How will you decide which way to move when several options are available?
- How will you prevent yourself from walking around in a circle indefinitely or from checking the same path multiple times?
- How will you save path(s) so that you can print out the shortest path once you're done running?

There are several reasonable answers to each of these questions, though some may be simpler, faster, or more elegant than others.

The Maze Class: All mazes will be rectangular where each position is represented by a coordinate pair: (row, col). Each position can contain an empty space, a wall (which you can't move into), or the goal. The border of the maze will be walls to prevent you from falling out of the maze accidentally. For example, the following is an ASCII representation of a simple maze:

```
#########
###    X#     S = starting point
### # ###     X = goal
#S  # ###     # = wall
#####   #
#########
```

Our Maze class supports the following methods. See the C++ header file for details:

- **`Maze(string filename)`** -- this constructor takes in a filename and reads the maze description from the specified file. It also sets the current position to the start of the maze.

- **`Size(int& numrows,int& numcols)`** -- this lets you query the number of rows and columns in the maze.

- `Position(int& row, int& col)` -- this lets you query the row and column of the current position in the maze.

- `Goal(int& row, int& col)` -- this lets you query the goal's position in the maze.

- `Look(direction dir)` -- this looks in the specified direction (north, south, east, or west) and tells whether or not that direction is blocked.

- `Move(direction dir)` -- this moves you one step from the current position in the specified direction (north, south, east, or west) *if* it does not contain a wall. A value is returned indicating whether or not the move was legal.

- `Jump(int row, int col)` -- this moves the current position to the specified row and column *if* it is a position that you've moved to before. Otherwise it leaves it where it was. A value is returned indicating whether or not the jump was legal.

- `~Maze()`-- the maze's destructor prints out statistics indicating the number of moves and jumps you've made since calling the constructor, for your own benefit.

Note that you will not necessarily want or need to use all of these methods; they are merely provided for you if you want them. Your program needs to work with the Maze class *as provided* (that is, you may not rely on methods or behavior that we do not provide). If you believe that there are serious problems with the interface as it exists (or find bugs in the implementation), please contact the course staff.

Strategy: One approach to this problem is to try every possible path from the starting point to the goal in order to find the shortest one. This is known as *exhaustive search*, because in performing it you exhaust all the possibilities for getting from one point to the other. This is a perfectly reasonable approach for this program and will get you full credit (as long as you use the Stack and Queue in a reasonable manner, and avoid pitfalls like walking in circles or checking the same path multiple times).

If you want to, however, you may try to make your program smarter about how it finds the shortest path. For example, one simple strategy is that if you ever are on a path that is longer than the shortest path you've found so far, there's no way it will become shorter, so it cannot be the shortest path. Make sure that any strategies you implement do not affect the correctness of your program. It should still find the shortest path for any legal input maze. Making your program smarter will not get you any additional credit (other than the staff's admiration), but is something you can use to challenge yourself for fun.

Competition: If there is enough interest, students who think they've implemented particularly clever strategies can request to race their program against other students in the class (where the primary measurement of "speed" will be the amount of moving and jumping you did in order to find the shortest path). If you decide during the assignment that you want to enter the race, send Brad email indicating your interest and containing a maze that you would like to have used as one of the race day mazes. Assuming that there aren't 80 participants, we'll race every participant on each of the submitted mazes and see who does best overall. Neither participating in the race nor the outcome of the race will affect your grade -- it's just for fun and to vent those macho feelings that you've been stifling while writing crystal clear C code.

Turn-in: For this assignment, follow the usual turn-in guidelines. In your introductory paragraph(s) on the cover page, please address the design decision questions mentioned in the *Search Algorithms* section above, as well as any other interesting design decisions you made (including strategy, if applicable). Make note of any changes to the List, Stack, and Queue ADTs that you have made (none should be necessary, but perhaps you found some change convenient).

In the hardcopy of your code, please mark:

1. The type of your Stack for the depth-first search
2. The type of your Queue for the breadth-first search
3. Where you store away the shortest path when you find it
4. An indication of where you avoid walking in circles
5. An indication of where you avoid walking the same path multiple times
6. Any other pieces of code that are particularly relevent for your implementation

I would also like you to include two mazes: one that represents a case where your depth-first search will find the shortest path before the breadth-first search, and a second in which the breadth-first search will find the shortest path before the depth-first searh. Give an explanation of why these mazes work better with one algorithm than the other (This identification of best/worst cases is an important skill in any programming task, which is why I ask for it here).