Homework 9, CS161, Spring 2012

Due Monday, April 30th, 10:00 a.m.

We have decided to grade your maze solutions by hand, rather than using a script. You will get credit if it meets the postconditions specified in the MazeInterface.java file. We only ask that you do it in a certain way, below, to help us with grading.

1 CharMatrix.java

Look at `CharMatrixInterface.java`. You should write a program, `CharMatrix.java` that implements it. In other words, your class declaration should be obtained by filling in the following:

```
class CharMatrix implements CharMatrixInterface
{
}
```

In addition to the methods listed in the interface, you should provide it with a default constructor and a constructor that is obtained by filling in the following:

```
    /*********************************************************************/
    // Constructor that reads a character matrix from a file.  The parameter
    //  to the constructor is a String object indicating the name of the file.
    //  The file is a text file, and must be formatted as follows:
    //
    // - The first line of the file is an integer n that indicates the number
    //    of rows of the matrix;
    // - The second line of the file is an integer that indicates the number
    //    m of columns of the matrix;
    // - For each i from 3 to n+2, line i contains the characters for row
    //    i-3 of the matrix.  (The rows are numbered 0 through n-1 and the
    //    columns are numbered 0 through m-1.)  The line must contain exactly
    //    m characters, not counting the newline character.
    //
    // Example:

    // 3
    // 4
    // abcd
    // acdb
    // baad

    // The .dat files provided with the assignment give other examples.
```

```
// You should open the file in a try block, and you should
// handle errors resulting from bad files specified by the user
// using exception handling.
```

Once you have done this, in your program for testing, you should be able to do the following:

```
CharMatrixInterface matrix = new CharMatrix (filename);
```

This establishes a variable of type `CharMatrixInterface`. Because `CharMatrix` implements `CharMatrixInterface`, you can assign it to a `CharMatrixInterface` variable. The variable can be used just like a class variable, except that you are only allowed to call the `CharMatrixInterface` methods on it, even if you wrote other public methods for `CharMatrix`.

2 Maze.java

Next, you need to write Maze.java, which navigates a maze.

```
public class Maze extends CharMatrix implements MazeInterface
{
}
```

Notice that it extends `CharMatrix`. This is because we can think of a maze as a special case of a character matrix. See `maze1.dat` for an example. Look in `MazeInterface.java` to see what methods you should implement in it. Because it implements `MazeInterface`, you may refer the constants OPEN, MARKED, GOAL, etc., that are defined in that class. In your code, refer to all character values by these names, which communicate to others the roles of the different character values you put into a maze.

It should have a default constructor and the following constructor:

```
/*********************************************************************/
// Constructor that reads a maze from a file.  The file must be in
// the file format of the CharMatrix class; see comments preceding
// the constructor in that class.  In addition, the borders of the matrix
// must consist exclusively of WALL characters (see constant definitions
// above) and all characters must have values from the set
// {WALL, OPEN, GOAL, MARKED, PATHMARK}.
/*********************************************************************/
public Maze(String filename)
{
}
```

You should find this to be trivial to write, because the class extends the `CharMatrix` class, so you can call `super(String)`.

Let us summarize some of the methods:

`countCellsFrom` is a recursive method that works by induction on the number of open cells. That is, the measure of the "size" of the problem that it is given is the number of open cells in the maze. This means that we can assume by induction, when designing the method, that if we make a recursive call on a maze that has fewer open cells than the one we were given, the recursive call will work correctly. To meet this condition, the first thing we do is replace the cell at `(curRow, curColumn)` with the MARKED character, reducing the number of open cells by 1.

Then, check the cells immediately above, immediately to the right, immediately below, and immediately to the left of `(curRow, curColumn)`. If the neighboring cell is not empty, move to the next neighboring cell in this order. If it is open , make a recursive call on the neighbor. Because we can assume by induction that the recursive call will work, it will mark and count all cells reachable through that open neighbor.

Once you've done this for all four neighbors, you've marked and counted all cells reachable from `(curRow, curColumn)` through an open neighbor. **That's all cells reachable initially from `(curRow, curColumn)`, since the path to any such cell has to go to an open neighbor as its first step.**

Don't forget to use the returned values of the recursive calls to figure out how many empty cells were initially reachable from `(curRow, curColumn)`, **counting** `(curRow, curColumn)`.

Once you have finished this, think about how to modify the strategy slightly in order to implement `findGoal`. If you don't finish all the methods, include a stub for the method that satisfies the interface, and have it print out "Not implemented" before returning. That way, we can still run your program on the methods you got working.

2.1 Debugging

When a program has a bug, it's because your inductive proof that it works has a flaw. If you can't figure out where you've gone wrong, try to use the computer to smoke out the error in your thinking.

To do this, try to find the smallest example of the problem that elicits the bug. Spend some time playing around with examples to find one. Suppose that in doing so, you discover running `countCellsFrom` starting at $(2, 2)$ messes up in the following small example:

```
  01234
0|XXXXX
1|XX XX
2|X   X
3|XX XX
4|XXXXX
```

If it's a smallest example where your method fails, then the recursive calls don't fail. Verify that they don't using the `eclipse` debugger. **Step over the recursive method calls, not into them. Check whether they have worked by verifying that their preconditions have been met and checking whether they have met their post-conditions. If their preconditions haven't been met, you've discovered a bug.**

Otherwise, if they have met their postconditions, they have been exonerated as the culprits for the bug you're trying to track down. You didn't have to step into them to exonerate them.

The reason we emphasize this is that stepping into recursive calls using a debugger is a good way to get hopelessly lost. Use the bugger in a way that's consistent with induction, the trick whereby we can understand the most complicated recursive algorithms using only our mortal minds.

Suppose you find out that one of them has failed. **Then you've found an even smaller example, so you should redo your maze to be this smaller example and start over on it.**

For example, suppose it messes up on the recursive call on the neighbor at $(1, 2)$. What maze was handed to this recursive call? It's this:

```
  01234
0|XXXXX
1|XX XX
2|X . X
3|XX XX
4|XXXXX
```

So put the dot in the middle of your maze in your input file and rerun the method, starting at $(1, 2)$. Using the debugger, it should be obvious what's going wrong, since its only job is to put a dot in that square and return 1.

On the other hand, suppose all of the recursive calls have been exonerated. Then the code for the inductive case in the main call is the culprit. Use the debugger to step through statements of the main call, comparing what happens at each step with what you expected would happen. Since the final outcome is wrong, and you didn't expect this, there must be a first statement that does something you didn't expect. There's your culprit statement. Fix it and repeat the exercise until no culprits remain, which means that your method will works – at least on the example you've chosen.

When debugging a recursive method, work from the smallest recursive calls to the larger ones, exonerating them as you go, until you find the culprit call. Then step through the statements of the culprit call to identify the culprit statement.