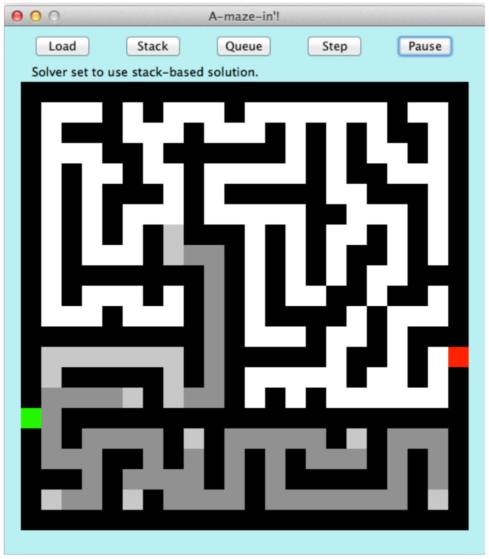
Software Design II (CS 220), Spring 2016
Assignment 06 (35 points)
Due by 5:00 PM, Wednesday 14 December 2016



Solver set to use stack-based solution.

For this assignment, you will be using stacks and queues to solve a maze in different ways. You are supplied with code to start you off. When run, it opens a window, and allows you to choose various maze files, stored as text. Some mazes are provided, and you can make your own, too.

(a) (*5 points*) First, complete the `MazeMaker` class. The constructor for this class takes in the name of a maze-file. These files are written in plain-text format. Some have been supplied for you, but you can write your own, using:

- Character '`X`' for walls.
- Character '`_`' for empty space.
- Character '`S`' for the starting location.
- Character '`G`' for a goal location.

Mazes need not be square, although they will probably look better graphically if they are. They can be practically any size, although there are limits to how big they can be, given the limited space in the window. Your code should read in such a file, and fill in a **two-dimensional array of characters** to match it, row by row and column by column. When you are done, loading a maze-file will cause it to be displayed in the window by the existing graphics code that is supplied as part of the downloaded project.

Note: to make sure files load properly, remember that they should be placed inside the main level of the Eclipse project (if you are using Eclipse). That is, don't put them inside the `src` or `bin` folders, put alongside them, at the same directory level.

(b) (*5 points*) When you solve the maze, you will use either a stack or queue to do so, giving different results depending upon which you choose. To accomplish this, you will modify the given stack and queue classes so they implement a common interface. This will allow your solver to use a variable of the common interface type, and easily switch between the two. You should modify the `Stack` and `Queue` classes now, so that each implements the common generic `StackOrQueue` interface. That is, each should supply all of the required methods in that interface, using their own particular methods. So, if you are writing code in the `Stack` class, you should make sure that the methods to insert and access the data are implemented in a stack-like way (using LIFO access); for the `Queue` class, those same method should be implemented in a queue-like way (using FIFO access).

When this is complete, you can write code like the following:

```
    StackOrQueue<Square> list;
    list = new Stack<Square>();
    Square s = new Square();
    list.insert( s );
    list = new Queue<Square>();
    list.insert( s );
```

This code first instantiates the `list` variable using a stack; when `insert()` is called, it will thus work in stack-like fashion, inserting objects on the **top** of the stack. When `list` is replaced with a queue, however, the same call to `insert()` will now place the object at the **back** of the list, in queue-like fashion.

(c) (*5 points*) Now that you have implemented the common interface, you should modify the `Solver` class to use objects of that kind. The class should have three instance variables, one for each of the parameters passed to the constructor (which should save them as usual), and a list object of generic type `StackOrQueue<Square>`, that will be used for solving the maze. You should also fill in the two methods `setStack()` and `setQueue()` which are called when a user presses the **Stack** or **Queue** buttons. Each method should instantiate the list so that it is the appropriate specific type. After setting the list to the right type, your code should then find the starting square for the maze (the one that appears in `green`), and insert it into the list so that search can begin.

(d) (*15 points*) You will now fill in the `solve()` method to solve the maze. This method will implement a single step of a search, using the list of `Square` objects, as follows:

(i) If the list is empty, then we stop, as there is no possible path to the goal. The code should run the `fail()` method from the `MazeWindow` at that point.

(ii) If the list is not empty, then we remove the first square, S, from the list.

(iii) If we have *already visited* square S, then we can ignore it.

(iv) If square S is the goal square (the `red` one), then we are done: we have found a path. The code should run the `succeed()` method from the `MazeWindow` at that point.

(v) If we *have not* visited square S before, then we explore as follows:

- Look at the squares around S (in any of the four cardinal directions).
- For each of these new squares, if it is not a wall (`black`, add it to the list.

(vi) Keep track of the fact that we explored square S, so we won't re-explore it later.

To finish this part, we want some graphical sign of progress for the search. Add some more code to your solution method so that it marks the empty squares it sees. In particular, if it explores an empty white square (i.e., any square except the walls, start, or goal), it should mark it by setting its color to something new (don't use one of the colors already used in the maze). Secondly, in step (iii), if it sees a square more than once, it should change its color again, so that when we are done, we will be able to tell which squares were visited once, and which were visited multiple times. The exact choice of colors is up to you.

When you are done, the method will be called whenever the user hits the **Step** button, or whenever the animating timer is activated. Then, when you press those buttons, after loading a maze, and choosing either a stack or queue, you should see the progress of the solver, either one step at a time, or in animated fashion.

Run your solver on the various mazes to test that it works. It should find the goal in all the supplied mazes, except for `maze03`, which has no possible solution. Run the two different kinds of searches: why do they behave differently? (This is just something to think about, not something you have to answer for this assignment.) In particular, look at the performance on the simple files `maze04` and `maze05`. Which method works better in each case? Why do you think this is, exactly?

(e) (*5 points*) Finally, you will improve the graphical display and overall program performance. Amend your existing code so that whenever the **Stack** or **Queue** buttons are pressed, and the program creates the new list to store objects, all squares in the maze should go back to their original color, so that if we do multiple searches, each one resets the maze first.

Handing in your work: Your work will be handed in by creating a compressed (zipped or otherwise) folder containing **all and only your Java source files**. Once the files are compressed, you can upload them via the class D2L site, using the dropbox link for this assignment.

Coding conventions: Each step of the program has a points total, given above. For full points, you should complete all those components, and observe the basic coding conventions as follows:

- Comments should appear at the start of any code-file you write or change, with your name, the date of your work, and a short explanation of what the code does.
- Each method should be preceded by a blank line, followed by at least one line of comments explaining what the method does.
- Methods should be `private` if possible, and `public` only if necessary.
- Class variables should be local to methods if possible, and should only appear as global variables if they appear in more than one method of the class.
- Unless absolutely necessary, global instance variables should be `private`.
- Code can be broken up by blank lines, but keep this to a low level. There should be no more than a single blank line separating pieces of code. Within a line of code, white space should not be too wide, for clarity.
- Code should be properly indented and separated into lines.