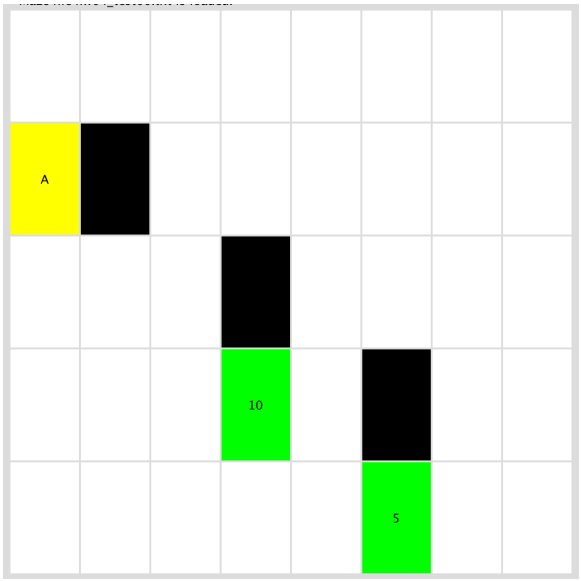
Artificial Intelligence (CS 452/552), Fall 2017
Assignment 04 (75 points)
Due before class, Wednesday, 15 November 2017

You will write a program that reads in an input file; this file will describe an $(n \times m)$ grid-world maze problem containing:

- A set of *obstacles* that block movement in the grid.
- A set of *targets* each with a unique utility value, $U > 1$.
- A starting location $(row, column)$ for an AI agent.

For example, we might have a (5×8) grid as in the figure below, with three obstacles (black squares), two targets (green circles, with utility U shown), and agent at starting location $(1, 0)$:



In this case, the input would be given in a text-file in the following format, beginning with the size of the grid in $(row \times column)$ layout, and using a value of 0 for any open square, a value of 1 for the agent location, a value of -1 for any obstacle, and a value $k > 1$ for any target location:

```
5 8
0  0  0  0  0  0  0  0
1 -1  0  0  0  0  0  0
0  0  0 -1  0  0  0  0
0  0  0 10  0 -1  0  0
0  0  0  0  0  5  0  0
```

You can assume that the grid is no larger than (15×15) in size; it will contain at least 1, and no more than 5, targets. Each will have a distinct value U, $1 < U <= 10$, i.e., no two targets have the same value. There will always be at least one path from the the agent's starting location to any target (there may be more than one such path, and some paths may pass through other targets).

Elementary specifications:

1. Your program will run from a class named `Solver`, which contains your `main()` method.

2. The `Solver` class will implement the interface `MazeSolver`, included with the downloaded document you are reading.

3. When executed, the program will create an instance of the `Solver` class, and then create an instance of the `MazeWindow` class, which will generate the GUI for the program. The constructor for the latter class requires that your `Solver` be given as input, in order that the classes communicate; it also includes a `public` method, `makeWindow()` that will generate your user interface when called. That is, you should have code something like the following:

```
Solver solver = new Solver();
MazeWindow window = new MazeWindow( solver );
window.makeWindow();
```

   Note: your code need not look exactly like this. For example, you may choose other variable names, or use instance rather than local variables. It does need to create the `Solver` and `MazeWindow` objects, and execute the `makeWindow()` method.

4. As part of implementing the `MazeSolver` interface, your class must contain a method called `getMaze()` that takes in a `File` object, corresponding to a maze specification file, and returns a 2-dimensional array of integers corresponding to that file. For example, if we opened the file described on the previous page, the returned array of integers would be of size `[5][8]`, and contain all of the integer values seen in the text-file, in the same order as seen in that file. When successfully completed, selecting a maze specification file using the `Load` button in the GUI will cause it to be displayed for you to see.

5. As part of implementing the `MazeSolver` interface, your class must contain a method called `getSolution()` that takes in an integer and returns a solution to the resulting problem instance, as described below. The solution will be in the form of a list of `String` objects that describe the optimal policy. The format of these objects is described in the interface description of the method. When this is complete, pressing the `Solve` button in the GUI will display the policy that your code has returned, using the value entered into the text-field as the agent's carrying capacity (this value is set to `1` by default, but can be set to any value from `1` to `10` on repeated runs for the same maze. (Attempts to set it out of bounds will simply result in it being either the minimal or maximal feasible value.)

Solution technique:

The program will use the dynamic programming methods shown in class (iterative policy evaluation and update) to generate an optimal policy for the agent, where we define this as: the shortest path from the starting location to the set of n target objects of largest value, where n is the minimum of carrying capacity c and the total number of targets. That is to say: if the carrying capacity c is less than the number of objects in the maze, then the shortest path to the top c objects, in any

order, should be generated; if the capacity c is greater than or equal to the total number of objects, then the shortest path that takes the agent to the location of every object should be generated.

You will initially assume that all movements by the agent are deterministic; the agent can move either up, down, left, or right, and this will either succeed or fail each time. The agent can't move through obstacles, or leave the grid in any direction; every other movement will always succeed. If the agent passes through a target that is among the n most valuable ones, then it picks that target up; if the agent passes through a target that is not among the most valuable ones, it ignores it. The process can then terminate as soon as the agent has visited the squares containing each of the n most valuable targets.

The output of the program, returned for visualization by the `getSolution()` method will be a list describing the set of steps that the final policy, $\pi^\star$, gives for the agent to move from the starting location to its final target. For example, if we ran the program on the input described above, with carrying capacity $c = 2$, we would get back output including a path to both targets, such as:

```
[Down, Down, Right, Right, Right, Down, Right, Right]
```

If, on the other hand, we used the same input file, but only had carrying capacity $c = 1$, the final output would be different, since the agent would only pick up the single target with value $U = 10$:

```
[Down, Right, Down, Right, Right]
```

Once your program has found a proper policy and returned it, the GUI will show that policy on-screen for verification.

Note: your output could be different. There are generally many equivalent paths through the grid, so the path may not be the same, and can even be different on different runs of the algorithm (although it ***should always*** have the same, shortest-possible length).

You will hand in source-code for the program. The instructor will compile it and test it by giving it different input files. Some sample files are provided, and you can create your own for testing, using the same format as given above. Submit the work on D2L by the due-date and time (before class on the date given at the start of the assignment). Your submission should consist of an archive containing the source code and instructions for using it (if that code runs in any way other than from the `main` method in the `Solver` class). Follow basic software engineering principles; that is, your code should be clean and well-structured, with comments explaining each method or function, and the usual format conventions to make it readable.