Project 3 – Maze Runner!

Due: 10/22/2018 11:59:59 pm

Goal. In this project, you will create an efficient solver for two-dimensional mazes. Each maze consists a grid of positions. Someone moving through the maze can generally move in one of the four compass directions (North, South, East, West) to advance to an adjacent position; however, walls may block some of these possibilities. There is also a start position, and an exit position from the maze. For a given maze, starting from the start position, your program should either return a solution (i.e. a path to the exit) or determine that no solution exists. ***Your program should also run faster than all our single-threaded solvers*** when run on a multi-core machine. The implementations details for how to solve the maze are left up to you, subject to certain restrictions given below, and you may use any default Java library classes you wish.

Contest. This project will also include a contest component by which you may earn some extra credit, depending on the execution speed of your solution. In the contest, your program will be run against programs submitted by other students in the class on a contest server; the results will be posted to a shared leaderboard. This board will detail every submission's running time on various undisclosed test mazes. Extra credit will be awarded based on the performance of your program in comparison with others. Submission to the submit server will automatically queue your program to run on the contest server, once the contest server is activated. Further details regarding the contest server will be posted to Piazza.

Maze Restrictions. For the purposes of this project you may assume the following about the mazes your code will be run on.

- Mazes will contain at most one solution.
- Mazes will not contain any cycles or loops.
- Mazes will fit into a grid measuring 20,000 cells by 20,000 cells.

Project skeleton code. The code we will provide you contains the following class files. You may make changes to the following.

- `StudentMTMazeSolver.java`. This class will contain your solution, and it is the one we will test. It extends the class `MazeSolver`. *You must ensure that your class is a subclass of the* `MazeSolver` *class!* Here are more details about your class.
  - Constructor `StudentMTMazeSolver()`. The constructor takes a `Maze` object to solve and stores it for later reference.
  - `List<Direction> solve()`. This method computes and returns a solution to the maze object stored in the instance field, if one exists. The solution should consist of a list of directions taken at every cell beginning from the start and leading to the endpoint. If no solution exists, this method should return null.
- `Main.java`. This class is intended to facilitate testing your code and can be modified as desired. It will not be run by the submit server, and any changes you make to it will

be overwritten on the contest server, so please ensure that there is nothing in this file that the rest of your program depends on. Specific methods in this class include the following.

- `read()`. This method reads a `Maze` object from a file and stores it in private instance variable `maze`.
- `solve()`. This method executes the solvers indicated and prints out the results.
- `initDisplay()`. This method displays a graphical representation of a maze for debugging purposes. Your maze solver can also be configured to draw its path on the display to make tracing easier.
- `main()`. This method runs opens a `Maze` from a file, runs the given solvers, and displays the results alongside a picture of the `Maze` if requested.

The following files are also provided, and you may make use of them. ***You should not make any modifications to these files****, however!* Our testing will overwrite these files in your submission with the original ones from the skeleton.

- `Maze.java`. Represents a two-dimensional maze as an `AtomicIntegerArray` and provides methods returning information about the maze. The method `checkSolution()` can be used to verify results. For this project all mazes are read and deserialized from files.
- `Position.java`. Represents a cell in the `Maze`.
- `Direction.java`. Represents the four compass directions, used to refer to neighboring `Positions`.
- `Move.java`. Represents a move by storing a `Position` along with the `Direction` of the move. Also has a reference to the previous `Move` for traversal-building.
- `Choice.java`. Represents movement possibilities for a `Position` in a `Maze` as part of a traversal. In addition to `Position`, it stores the `Direction` of the previous `Position` as well as valid `Directions` to proceed in. A wall at a given `Position` is represented by a lack of `Choice` in a given `Direction`.
- `MazeSolver.java`. Superclass of all maze solvers. Whatever your implementation is, it MUST subclass this class.
- `SkippingMazeSolver.java`. `MazeSolver` that for each move, follows a `Direction` and progresses through the `Maze` until a `Choice` is reached, allowing for a more concise representation of a traversal. This partial `Direction` list can be converted to a full `Direction` list upon returning. A method to color `Positions` may be used in debugging.

In addition, we have included several fully-implemented single-threaded solvers – `STMazeSolverBFS.java`, `STMazeSolverDFS.java` and `STMazeSolverRec.java` – for you to compare times against and/or use as a basis for your solution.

Your implementation. You should implement your solution in the file `StudentMTMazeSolver.java`, which must be a subclass of `MazeSolver`. You may also

modify `Main.java`, if you wish, and add additional class files beyond those in the skeleton code if this will help with your solution. Please do not modify any of the other skeleton files beyond `StudentMTMazeSolver.java` and `Main.java`, however. You should also ensure that nothing in your solution depends on anything in `Main.java`.

Testing. Among the tests we will use for this project are races where we will compare your solver's time against a single-threaded solver's time. For full credit, your program should be able to consistently win these races. We will, of course, also test your output for correctness. For offline testing, we have provided several maze files for you to run, as well as some single-threaded solvers you can use as benchmarks. Because of the provided files and the presence of the contest server, we will NOT have public tests on the submit server. **Discussion for this project is encouraged**, although as usual you are not allowed to share code with your colleagues.

Submission. Submit a .zip file containing your project files to the CS submit (submit.cs.umd.edu). You may use the same approach outlined in Project 0 to create this from inside Eclipse. ***Please do not include any maze files in your project submission!*** Failure to adhere to this request may hinder your program's ability to run.