Homework 6: A-MAZE-ing Race (based on a UMd project)

Goal: In this project you will create an efficient solver for two-dimensional mazes. Each maze is a grid of positions, which may have walls between them. 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. There is also a start position, and an exit position from the maze. For a given maze, your program should be able to either return a solution (path from the start position to the exit) or find that there is no solution. Your program must be faster than any of the single-threaded solvers, when run on a multi-core machine. The implementation details for how to solve the maze are left up to you, and you may use any default Java library classes you need. Your program also needs to count the number of choices your search performs (i.e., the number of moves from one position to the next). Note that the number of the moves is not equal to the length of the path that your program returns.

Contest: This project will include a contest component in which your program will be run against programs submitted by other students in the class. The results will compare every submission's running time on various undisclosed test mazes, and *extra credit will be awarded based on the performance of your program*.

Mazes: The following will hold true of all the mazes we will use to test:

- Mazes will contain at most one solution.
- Mazes will not contain cycles and/or loops.
- Mazes will be no larger than 20,000x20,000 cells.

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

- `StudentMTMazeSolver.java`: This class is where we will check for your implemented solution. 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 was provided to facilitate testing your code and can be modified as desired. We will not run this class during evaluation; any changes made to it will be overwritten. Ensure there is nothing in this file that the rest of your program depends on. Methods in this file are:
  - `read(filename)` This method reads in a `Maze` object from a file and stores it in private variable `maze`.
  - `solve()` Executes the indicated solvers and prints out the results.
  - `initDisplay()` 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()` Opens a `Maze` from a file, runs the given solvers, and displays the results alongside a picture of the `Maze` if requested.

Additionally, you SHOULD NOT store the maze files in the project directory when submitted since they increase the file size. This may jeopardize your program's runs.

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`, `STMazeSolverDFS`, and `STMazeSolverRec`, 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 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. We will also assess your program in the contest, as described above. Discussion for this project is **encouraged**.

Submission: Firsts submission: Pre-, Post-conditions and invariants. Pseudo code optional. Second submission: Submit a .zip file containing your project files to Blackboard.