Project 3: Maze Solver

Goals
You will write a parallel maze solver. The solver will take as input a two-dimensional rectangular maze with at most one solution (this means that the maze doesn't have any loops, and that it doesn't necessarily have a solution). The solver will return the solution, if it exists, or null otherwise. This project admits a great deal of flexibility. The aim is to obtain a solution that makes optimal use of parallel resources.

Your programs will be evaluated on the efficiency of your solution, in comparison with two others, that run in a single threaded manner.

Getting started
First, import the project into Eclipse. It contains the source files, javadoc documentation, and the file generate.jar, which we'll use to generate random mazes. To run this maze generation program, follow the below steps:

- In Eclipse, go to Run > External Tools > External Tool Configurations...
- In the upper left, hit the button to make a new configuration (a paper with a plus on it).
- For the name, type Maze Generation
- For location, type ${system_path:java} (or alternatively the complete path to your java binary)
- For Working Directory, type the physical location of your project. (For example, C:\eclipse\workspace\MazeSolver, or ${workspace_loc:/MazeSolver} )
- Finally, for arguments, type:
  - jar ${workspace_loc:/MazeSolver/generate.jar} <width> <height> <filename> solvable
    o replace ${workspace_loc:/MazeSolver/generate.jar} with the physical location of the included file generate.jar
The four arguments after the path are the maze width, height, file name of the generated maze, and whether it should be solvable or unsolvable. Replace these with anything you like when you test your code.

- Once you run this configuration, it will generate the file in your main project directory. To see, select the project in Eclipse and hit F5 to refresh the working directory and see the file.

Before you’re ready to run your code for any maze size (that is, any size that will be used for grading), you need to set up the JVM so that it allows large stack and heap sizes for very large mazes:

- In Eclipse, for each of the files you intend to run (files that contain either static void main() or JUnit tests):
  - right-click the file and choose Properties.
  - In Run/Debug Settings, choose New… > Java Application
  - In the Arguments tab, for VM arguments, type the following:
    -Xms4096m -Xmx4096m -Xss4096m
    This will set the heap and stack sizes to 4096 MB (do this only if you have at least 4GB of memory on your machine).

If you don’t do this, then for large mazes (like 5000x5000 or more), your program will crash.

### Single Threaded Solvers

The code in the Eclipse project includes five single-threaded example solvers, meant to get you familiarized with the code, and to give you an example of what everything does. They also use different algorithms and different implementations of the same algorithm.

We included several different implementations for DFS. All of them use optimized depth-first search involving so-called choice points. There is some discussion of this idea below.

**STMazeSolverDFS**

This is the best performing single-thread solver. Along with STMazeSolverBFS, it will be used for grading. See below.

**STMazeSolverDFSRec** and **STMazeSolverDFSIt**

These two implementations of the same algorithm are the base for the grader solution (in other words, our solution is a combination of these two, written in a multi-threaded manner described briefly below). They are provided solely as examples of how the code could be written. You may use any part of them, or nothing at all, if you want to attack the problem in a different way.

**STMazeSolverRec**
This is the simplest: a recursive search procedure, which performs slightly worse than the previous three. There is a danger, when using this approach on a very large maze, of running out of stack space, which is why you should specify the above mentioned JVM arguments. This one is provided just as an example.

**STMazeSolverBFS**
This solver performs a breadth-first search. This solution is more complicated that the others. However, since breadth-first search involves exploring from several points of the search space simultaneously, this solver can make a nice starting point for a multi-threaded approach. **Along with STMazeSolverDFS, this class will be used for grading.**

**Running the solvers on a maze**

To run these solvers on the maze you just generated, build the project in Eclipse and then run the Main class, providing the filename as an argument (don’t forget to specify the corresponding JVM arguments as well). This will run all defined solvers on the given maze file. The next section explains how to add your own solvers to the list. To see a graphical depiction of the maze, run with arguments: `<filename> display`
(Note that you should not attempt to display mazes more than 200 x 200 or so, or else they will be too large to fit on your screen.)

**Initial Code Layout**

We provide several classes in the skeleton code.

**Position** represents a location in the maze.

**Direction** represents the four compass directions.

**Move** stores a given **Position** and the **Direction** chosen from it. It also stores the **Move** before it. (Since the maze contains no loops, the previous move will be unique.)

**Maze** stores a representation of the maze and also provides methods that make querying the maze easier. For example, the **Maze.getMoves()** takes a **Position** and returns the list of **Directions** in which you can move if you are at that position.

The **Main** class supports the loading of mazes from disk and calls the maze solvers. Solvers are stored in the solvers array in the **Main.solve()** method. Each solver in this array is executed on the maze provided as input at the command line when the program is called in solving mode. You can add multiple solvers to the list to easily compare them. **Solvers must be subclasses of the abstract class MazeSolver.**
The **Maze** class also provides means to color cells in the maze. The methods `getColor()` and `setColor()` provide access to these features. You do not need to use these methods, but they may be useful for debugging. If the program is being run with the display option, as mentioned above, then a graphical depiction of the maze will be drawn (use this only for debugging, as drawing may slow down your code!). A solver can color maze cells and then use `Maze.display.updateDisplay()` to update the graphical depiction of the maze. To aid in stepping through your algorithm, you can call `Maze.display.waitForMouse()`, which will block execution until the user clicks on the maze window.

**MTMazeSolver** is the class that will be tested. More specifically, `MTMazeSolver.solve()`.

The class has to inherit from `MazeSolver`. These are the only restrictions to the code. You are allowed to add any number of helper classes to your solution – just make sure that `MTMazeSolver.solve()` returns the correct solution if there is one, or null otherwise.

### Approach

The Eclipse project includes the JCIP puzzle-solving framework. This is for your convenience. You might find it worthwhile to create a class that wraps `Maze`, and implements the `Puzzle` interface (taking advantage of methods already present in `Maze`). By doing so, you can use the puzzle solving framework to solve the maze. As a result, this gives you a way to use the JCIP concurrent puzzle solver to solve small maze instances in a multi-threaded manner.

The JCIP implementation of a concurrent solver makes for a good starting point. However, it is unlikely to beat the single-threaded solvers we have provided: the issue is that the units of work in maze solving are very small: a single move. Completing small units of work in concurrent threads generally results in the concurrency overhead erasing any performance gains. You may try to modify this code to allow bigger chunks of processing per thread.

### Possible Algorithms

There are several algorithms that could produce good parallel solvers. Here is a key observation: Any given choice point (i.e., a position with from which multiple outgoing paths are possible) that does not lie on the solution path is likely to quickly lead to dead ends in all directions. Thus, most times an exploration from a single choice point will quickly run out of work. This suggests that, to ensure that threads are mostly doing useful work, tasks ought to consist of exploring several choice points, to increase the amount of useful work per task.

Another hint is that, since there are no loops in the maze, any two points on the maze (such that one is not on the path from the starting point to the other) are completely independent (not requiring synchronization of data structures between each other).
It may help to think of the maze as a tree structure, where the root is the starting point and only one of the leaves is an end point.

**Now, some ideas for algorithms.** One approach is to do a depth-first exploration until there are several unexplored paths on the stack. At this point, these paths could be grouped and handed to a new thread. Be sure to group together enough of them so that the new thread has enough work to do, and using it doesn’t cause useless overhead.

A related idea is to use the Fork-Join framework and try a divide-and-conquer strategy. For example, each task could return a path from the current position to the end of the maze. When a task reaches a choice point, it could fork new tasks that explore each of the possible branches, and wait from them to return; if any of them return a path, then the current node is added to that path, and it is returned. Consider only creating a child task after some number of recursive calls.

It may be useful to use the thread pool instead of creating your own threads, so that you reduce the overhead. Don’t forget, however, that waiting or sleeping on a thread pool thread blocks the thread for all other scheduled tasks in the pool.

Another approach would be to explore simultaneously from the start and the end, hoping to meet in the middle.

Whatever the approach, make sure that you provide a means to control the number of threads used by the search. If too many threads are spawned, the overhead of thread management will erase any performance gains. Recall from the text that you can figure out the number of processors on the executing machine by calling `Runtime.getRuntime().availableProcessors()`.

**Grading**

Any submission that doesn’t compile will receive 0 credit.

Your solution must use a minimum of two threads, otherwise, it will receive 0 (for example, a submission where the student just changed the name of one of the provided classes, will receive no credit).

To obtain full credit, you must create a parallel solver that, when run with at least two processors on the mazes provided at the top of this page, consistently beats the fastest sequential solver included in the skeleton code (`STMazeSolverDFS`). If you are able to beat `STMazeSolverBFS`, but not the faster one, you will receive 90% of the performance points. The others are provided just as examples, and will not be used in the scoring.
The maximum size mazes that will be used for grading are $20,000 \times 20,000$ (one solvable, one unsolvable). Make sure you test your code on mazes ranging from $10 \times 10$ to $20,000 \times 20,000$.

Even if your solver is slower than both BFS and DFS, as long as they are correct (they return a correct path when the maze is solvable and null otherwise) you will still receive partial credit (we'll decide how much on a case by case basis). The point is that you at least try a few ways of beating the single threaded algorithms.

You can use Linuxlab to test your program if your own machine does not have at least two processors (most of those machines do; the server machines you remotely log into have at least four processors).

We will test your code on a >4 cores cpu, so if you make good use of multi-threading capacities, it will show!

Along with your code, add a short text/pdf file in your repository, called “README.txt” or “README.pdf” where you briefly explain what you did. If your solver didn’t beat the single threaded ones, include a paragraph where you explain what you tried, and why you suspect your tries didn’t succeed.

As usual, grading will take into account your code’s thread-safety.

**Submit your solution**

Follow the same steps described for Project 0 when you’re ready to submit your solution.

Note that you may submit your code changes any number of times before the project deadline. We will only grade on the last submission.

**Questions**

Any questions you have, you should post on the Piazza forum. If you’re unsure if a question is allowed public audience, use the forum’s capabilities of sending private posts to the instructors. We will examine your question, and if we decide it’s a general interest question, we’ll change its status to public.

You’re encouraged to discuss the project with each other, and answer your classmates’ questions, but remember you must not share any part of your solution code online.

Good luck!