Homework 1 - A*-y Night

⚠ As a Homework, this is an individual assignment. While you are free to discuss strategies in groups, there should be no sharing of code or detailed pseudocode.

ℹ Your mission: implement A* graph search for the Maze Pathfinding Problem!

Basically, a way better version of your Classwork 1, but with the training wheels taken off!

In particular, we'll be working on Maze Pathfinder problems with:

- **Non-uniform cost:** in which certain actions have a higher cost associated with them than others.
- **Multiple Sub-Goals:** different ways to solve the problem, some of which may be better than others.
- **Planning:** there is a problem-specific objective that is beyond simple search.
- **Possibly No Solutions:** there may be well-formed problems that simply do not have a solution.

In this section, we will state our assumptions about the problem, and then detail what you will need to accomplish your goal in the following Specifications.

Problem Specification

Maze problems in this scenario will be constructed from a list of strings (a 2D maze of characters) indicating the contents of each cell in the maze.

In particular, the following table describes what constitutes a valid maze in this modified version of our Pathfinder:

Character	Definition	Cost of Movement Onto	Valid Maze Has...
X	An impassable wall -- movement cannot be made onto these tiles.	N/A	...walls along the border, but possibly other walls inside of the maze as well.
I	The initial state (where the agent starts the search)	1 (for stepping back-upon, not incurred at the start)	...exactly 1 initial state.

Character	Definition	Cost of Movement Onto	Valid Maze Has...
.	An open cell where the agent may move.	1	...no constraints on open spaces, except that they may not be found along the border.
M	A "mud" tile into which an agent may move, but at a greater cost.	3	...no constraints on mud tiles, except that they may not be found along the border.
K	A key to the exit! Your solution *must* visit *at least one* of these tiles before finding an exit to have a solution.	1	...*any* number of keys (could be 0, 1, 2, 3, ...).
G	The exit / goal state -- an agent need only navigate to this tile *after finding a key* to habe a solution.	1	...at *most* 1 goal state (could be 0 or 1).

⚙ Each MazeState / location will include **exactly 1** of the above categories, meaning that the initial state cannot also be the goal, a key cannot also be on a mud tile, etc.

Example
🗹 Here is an example, valid maze configuration (more are found in the skeleton's unit tests):

```
Maze elements are indexed starting at (0, 0) = (x, y) = (col, row)
[top left of maze]. E.g.,

    0123456
0 ["XXXXXXX",
1  "X....IX",
2  "X..MXXX",
3  "XGXKX.X",
4  "XXXXXXX"]

Initial State: (5, 1)
Key State:     (3, 3)
Goal State:    (1, 3)
Solution:      ["L", "L", "D", "D", "U", "L", "L", "D"]
Total Cost:    12
```

The Pathfinding game proceeds as follows:

- The MazeProblem is formalized, including the maze layout, initial state, goal state, actions, transitions, **cost function**, and a goal test.
- The Pathfinder agent is provided with the problem (i.e., knows where any keys + goals are).
- The Pathfinder must find a sequence of actions ("U", "D", "L", or "R") that takes it from the initial state, to the key, then to any goal that minimizes total cost.

⚠ Importantly: in any valid solution, the agent cannot reach a goal without first reaching a key.

⚠ If **no solution** exists, the `solve` method should return `null`.

With these details in place, let's look at the specifications.

Specifications

ℹ To successfully navigate mazes as described in the previous section, we will be implementing **A* graph search** between locations.

⚠ This assignment is essentially Classwork 1 with more complex problems! For a review of CW 1, see the link below:

>> Classwork 1 Spec << (../../classwork/cw1/classwork-1.html)

Solution Skeleton

Start with the solution skeleton in-hand! In the following project, I've given you the outline for a Maze Pathfinder's supporting components, including a MazeProblem specification. The rest is up to you to implement the A* Pathfinding functionality to solve the given problem!

>> GitHub Classroom Link (https://classroom.github.com/a/zjlCx_1X) <<

⚙ Note: Parts of your solution to Classwork 1 may be useful here, but there will be substantial differences between CW 1 and HW 1, chiefly:

- The addition of subgoals (the key), optimization for non-uniform cost (the mud tiles), and optimization of possible solutions (multiple paths from multiple keys to goal).
- Your search tree nodes and frontier now have some extra recordkeeping, including the addition of a graveyard and Java implementations of priority.
- The MazeProblem class now defines `getCost` and `getKeys` methods.

Consult the JavaDocs for this assignment in the page located below!

>> pathfinder.uninformed javadocs << (./doc/index.html)

A Note on Comparators

📣 During your implementation, you will likely wish to employ a Priority Queue to store some hand-made data type -- the problem here being: how do you define what priority means?

⚙ The solution: Java provides a `Comparable<T>` interface that, if implemented, defines how objects of the implementing type can be **ranked / ordered** compared to other objects of the generic type `<T>`.

How does it allow you to accomplish this, you ask?

⚙ By implementing the `Comparable` interface, the implementing class must define the `int compareTo(T other);` method, which returns an int denoting whether an instance of the given class should be ranked / prioritized less than, equal to, or greater than the parameter `other` by a negative integer, zero, or a positive integer, respectively.

Check out the `Comparable` interface below:

>> Java Comparable Interface << (https://docs.oracle.com/javase/10/docs/api/java/lang/Comparable.html)

Some notes on the above:

- Java Priority Queues that store objects implementing `Comparable` will be dequeued in ascending order (meaning, the smaller the priority value / rank, the sooner it will be dequeued), which is actually nice / desirable for our application.
- Although the `Comparable` interface suggests that a return from `compareTo` of 0 is usually compatible with the class' `equals` method, we're using it just for ranking in some collection, so you need not adhere to this convention (the above suggests how to document this).
- Need some examples? Do some Googling! (self sufficiency and all). Otherwise, feel free to ping me!

Problem Simplifications

Before we look at the constituent pieces of this assignment, let's make a couple of simplifications.

- For this assignment, assume that all input mazes to your Pathfinder are validly formatted by the description of a valid maze in the table above.
- The initial, keys, and goal states will always be separate tiles and may not overlap, though some maps may have no keys, no goals, or may not be reachable from the initial state (in which there is no solution).

Hints

Some challenges, tips, and hints to consider:

- You've graduated from CMSI 281! Feel free to use any data structures from the Java Collections Framework in pursuit of your task.
- Consider the appropriate data structures from the JCF that can be used in pursuit of the frontier and graveyard; you may need to examine the JCF JavaDocs and see some tutorials to use these data structures! (self-sufficiency is another take-away lesson here, though I'm happy to help if you get stuck).
- Function getting out-of-hand complicated? Remember to use ample helper methods! Be particularly conscious of repeated code if you want full style-points.
- Trying to do too much in a single search operation? Consider breaking the overall goal into multiple search operations!
- Recall that A*'s main improvement over Best-First is its efficiency improvement, so your solution will work (quite quickly) on even large mazes. You thus *must* implement some heuristic to get maximal credit on the full grading tests.
- While you are not allowed to modify any of the existing class' *public* interfaces, you are free to add to the Pathfinders' and SearchTreeNodes' private interfaces at will.

Additionally, here's a good order of tasks to tackle:

1. Review your course notes and make sure you have a solid grasp on how A* Search is meant to operate, at least at a high-level. During this review, envision what fields and data structures may be relevant in your solution, paying special attention to what is recorded in each SearchTreeNode.
2. Read through the documentation and the methods available to you in each class from the classwork skeleton so you can envision how to proceed.
3. You should be able to use large portions of your CW1 Pathfinder to motivate the A* one.
4. Remember that the skeleton's unit tests are only a subset of the final grading tests -- make sure to test your submission adequately!

Start early and ask questions! I'm here to help!

Submission

ℹ You will be submitting your assignments through GitHub Classroom!

What

Complete the required method in `Pathfinder.java` that accomplishes the specification above, ***in the exact project structure and package given*** in the skeleton above.

⚠ You must **NOT** modify any class' *public interface* (i.e., any public class or method signatures) in your submission! You *may* add whatever private members, methods, and nested classes that you wish to complete the assignment, but ONLY in the Pathfinder.java file.

How

To **clone** this assignment (if you need a refresher), consult the guide here:

>> GitHub Classroom Tutorial (../../../../tutorials/github-classroom.html) <<

To **submit** this assignment:

- Simply push your final, submission copy to the GitHub Classroom repository associated with your account.
- Place your name at the top of *all* submitted files (in appropriate JavaDoc commenting fashion) **AND in the accompanying `readme` file**.