Artificial Intelligence Homework 1
Due date: 2/6 by 11:59pm

Search is defined as "The process of looking for a sequence of actions that reaches the goal" (AIML pg. 67). In this assignment, you will implement different search algorithms to solve the 8-puzzle.

This is a programming assignment and you may work with a partner if you choose. Click here to download the starter code. You will notice that the code is arranged in packages:

- **graphics:** This package contains the Java classes responsible for creating the GUI, calling your search algorithm, and graphically displaying the result. The main() method in SlidingPuzzle.java is where your code is called. *You do not need to modify anything in this package*.
- **search:** This package is where you will implement your search algorithm.
- **heuristic:** This package will contain your heuristics for A-star.
- **util:** This directory will contain the data structures necessary for the frontier -- i.e., a stack, queue, and priority queue. You will notice the priority queue is already provided for you as an example.

You should take time to read through the code in the different packages and familiarize yourself with how the classes interact.

Graph Search

The heart of the search process is the `GraphSearch` algorithm. This general-purpose algorithm returns a path (i.e., a sequence of actions) from the initial 8-puzzle board state to the goal state. Depending upon the data structure used for the frontier, `GraphSearch` can give rise to various search algorithms. In this assignment, you will be implementing breadth-first search (BFS), depth-first search (DFS), and A-star with two different heuristics.

You can find the pseudocode for `GraphSearch` in your lecture notes and also on the course webpage. *Note that the pseudocode we are using is different than the pseudocode given in the textbook.*

In the **search** package, you are responsible for implementing:

- The `solvePuzzle()` method in the `GraphSearch` class. This method takes in the puzzle and returns a solution in the form of a String that contains directions (U, D, L, R) for solving the puzzle. The directions specify how the "blank tile" should be moved to achieve the goal state.
- There is a second method `graphSearch()` that you must also implement. It is this method that does the actual searching and, as such, should be called by the `solvePuzzle()` method.
- The `GraphSearch` algorithm requires a node data structure. I have provided an interface (Node.java) that your node class must implement. This interface contains all necessary methods for a node (but you are welcome to add additional private methods to your node class as needed).

The Frontier

The `GraphSearch` algorithm takes in an `OrderedCollection` called the "frontier". Depending upon the data structure used for the frontier, `GraphSearch` can give rise to various search algorithms. If the frontier is a

queue, then `GraphSearch` gives rise to BFS. If the frontier is a stack, then `GraphSearch` gives rise to DFS. If the frontier is a priority queue, then depending upon how the priority of a node is defined, `GraphSearch` can give rise to UCS, greedy best first search, or A-star.

In the **util** package, you are responsible for implementing the data structures necessary for the frontier. In particular, you are responsible for implementing:

- A queue that implements the OrderedCollection interface
- A stack that implements the OrderedCollection interface

Internally, your queue and stack classes can simply use the queue/stack from the Java Collections Class. That is, your queue and stack classes should be a lightweight wrapper around the queue/stack classes provided by Java. You will notice the priority queue has already been implemented.

Heuristics

If the frontier is a priority queue, then depending upon how the priority of a node is defined, `GraphSearch` can give rise to UCS, greedy best first search, or A-star. In this assignment, you will only be implementing the A-star algorithm.

For A-star, an evaluation function is used to order the nodes in the frontier. The evaluation function evaluates nodes by combining:

(the cost from the start to the node) + (an estimate of the cost from the node to the goal)

For the 8-puzzle, the cost from the start to the node is simply the depth of the node -- i.e., the number of moves taken from the start to the current node. To estimate the cost from the node to the goal, however, you must use outside information known as a heuristic. There are 2 well-known heuristics for the 8-puzzle: the number of misplaced tiles, and the Manhattan distance.

In the **heuristics** package, you are responsible for implementing:

- A class (that implements the Heuristic interface) that computes the manhattan distance between the current node's board and the goal.
- A class (that implements the Heuristic interface) that computes the number of misplaced tiles between the current node's board and the goal.

Each class should implement the `evaluate()` method from the Heuristic interface. This method should return back an integer representing the cost from the start to the node plus an estimate of the cost from the node to the goal.

Once you have your heuristics implemented, you can pass the heuristic to the constructor of the priority queue to create a frontier ordered according to the heuristic.

Assignment Writeup

Finally, once your code is complete, there is an assignment writeup that asks you to run and evaluate the results of your `GraphSearch` algorithm. You will notice the starter code contains a Word document. Inside this document are two questions. The first asks you to fill in a table showing the total number of nodes expanded and the length of the solution for each algorithm. The second question asks you to use this information to make conclusions about the relative performance of the different search algorithms: BFS, DFS, A-star with the number of misplaced tiles heuristic, and A-star with the Manhattan distance.

Additional Information/Hints:

- Once you have your GraphSearch class implemented and you have implemented at least 1 heuristic, you should be able to run A-star (using my priority queue class). To run BFS or DFS, you will need to implement the queue and stack classes respectively. To run A-star with different heuristics, you can create different heuristics in the **heuristics** package and pass the heuristic in to the constructor of the priority queue class.

- The goal state is always a puzzle with the numbers 1 through 8 arranged row-wise and the blank tile being the very last spot (i.e. the bottom rightmost spot).

- The **puzzles** directory contains 5 solvable puzzles. If you create your own puzzle file, be sure that the puzzle is in fact solvable. Not all puzzles can be solved.

- It might be useful to give your Node class two constructors: one that creates the node given a board, and one that creates the node from a parent node object along with some specified modification (e.g., given a parent node and a direction to move the blank tile).

- Depth first search is not guaranteed to return the shortest path to the goal. As a matter of fact, it often returns a solution with hundreds (or thousands) of directions. As such, when you find the goal state, if you try to use recursion to reconstruct the path from the start to the goal, you will most likely get a StackOverflow Exception with DFS. Instead, I recommend writing a method in the GraphSearch class that uses a loop (instead of recursion) to reconstruct the path from the start to the goal.

Running Your Code

The SlidingPuzzle class takes care of reading puzzles from file. Puzzle files are passed in to the program via the command line (with the extension .puzz). Each puzzle file contains space delimited tokens: the numbers 1 through the maximum number and a period (.) to represent the blank tile. Here's an example of a puzzle file:

```
8 . 3
1 2 7
6 4 5
```

Puzzles are always 3x3 squares. To run your code, you should pass a filename to the main method of the SlidingPuzzle class via the command line.

Grading and Submission
Your assignment will be graded based upon correctness (i.e. I will run your code on various puzzle files), efficiency (i.e. the time/space complexity of your implementation), as well as its adherence to the Java style guide.

To submit, rename your directory hw1_FirstName_LastName. Then zip your directory and upload it to Moodle. Please remember to rename your directory *before* you zip it.

Last modified: Tue Jul 15 13:34:17 PDT 2014