Com S 472 Spring 2022

Project 1: A* Search for the 8-Puzzle (100 pts)

Due at **11:59pm**

Friday, March 11

1. Problem Description

The task is to solve the 8-puzzle. We want to rearrange a given initial configuration of eight numbered tiles residing on a 3 x 3 board into a given final configuration called the goal state. An example puzzle is given below:

Initial State

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

Goal State

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

The rearrangement proceeds by sliding one of the tiles onto the empty square from an orthogonally adjacent position. There are four possible moves `LEFT`, `RIGHT`, `UP`, and `DOWN`, which apply respectively to the four neighboring tiles, if exist, to the right of, to the left of, below, and above the empty square. Below is a sequence of three moves that solves the above puzzle instance.

```
2   3   RIGHT     2 3     UP     1 2 3    LEFT   1 2 3
1 8 4   -----> 1 8 4   ------>    8 4   -----> 8   4
7 6 5          7 6 5           7 6 5          7 6 5
```

The solution path has length 3.

In this project, the goal state will always have the empty square in the middle and the eight tiles surrounding it with their numbers increasing clockwise from 1 at the upper left corner. Namely, the goal state is

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

2. Heuristics

Our objective is to solve the 8-puzzle by generating the ***smallest*** number of moves. As introduced in the lecture, there are two common heuristics that are used for estimating the proximity of one state to another. The first is the ***number of mismatched tiles***, those by which

the two states differ. We will call this heuristic function h_1. The second is the sum of the (horizontal and vertical) distances of the mismatched tiles between the two states, which we will call heuristic function h_2. The function h_2 is also known as the ***Manhattan distance***. Both h_1 and h_2 are ***admissible*** heuristics, namely, they yield estimates that are ***no greater than the minimum number of moves*** required to reach the goal from the current state. Meanwhile, the heuristic h_2 is better than h_1 because it gives a closer estimate of the minimum number of moves to solve the 8-puzzle.

To estimate the number of moves taking the initial state s_0 to the goal state, we need to include the cost from s_0 to the current state as well. The cost for an intermediate state s is thus defined as

$$f(s) = g(s) + h(s)$$

where $g(s)$ is the smallest number of moves found so far from the initial state s_0 to s, and $h(s)$ is a heuristic function (e.g., h_1 or h_2) that estimates the number of moves from s to the goal state. The value of g at the initial state and the value of h at the goal state are both 0. On the path making the smallest number of moves from s_0 to s, the state immediately preceding s is called its ***predecessor*** state while s is called its ***successor*** state. The predecessor state may change when a shorter path from s_0 to s is found.

3. The A* Algorithm

To solve the 8-puzzle, you are required to use the A* algorithm presented in lectures. Provided below are details of an implementation of this algorithm. The A* algorithm maintains two lists, `OPEN` and `CLOSED`, of 8-puzzle states. The states in the `CLOSED` list have been expanded; that is, they have been examined and their successor states have been generated. The states in the `OPEN` list have yet to be expanded.

The A* algorithm has the following six steps:

1. Put the start state s_0 on `OPEN`. Let $g(s_0) = 0$ and estimate $h(s_0)$.

2. If `OPEN` is empty, exit with failure. This situation will never happen if before executing A* an inversion check has been performed on the puzzle to establish its solvability. (The situation may occur when applying A* to solve a different problem.)

3. The states on the list `OPEN` will be ordered in the ***non-decreasing*** value of f. Remove the first state s from `OPEN` and place on the list `CLOSED`.

4. If s is the goal state, exit successfully and print out the entire solution path (step-by-step state transitions).

5. Otherwise, generate s's all possible successor states in one valid move. For every successor state t of s such that t is ***not*** the `predecessor` of s:

    a. Estimate $h(t)$ and compute $f(t) = g(t) + h(t) = g(s) + 1 + h(t)$.

b. If t is not on `OPEN` or `CLOSED`, then put it on `OPEN`.

c. If t is already on `OPEN`, compare its old and new f values and choose the minimum of the two. In the case that the new f value is less than the old f value, reset the `predecessor` link of t (along the path yielding the lower $g(t)$) to s. Otherwise, no change is made to the link.

d. If t is on `CLOSED` and its new f value is less than the old one, the state may become promising again because of this value decrease. In this case, put t back on `OPEN` and reset its `predecessor` link to s.

6. Go to step 2.

4. Implementation

The names of the valid moves and available heuristics are given in two enum types `Move` and `Heuristic`, respectively.

The class `State` implements the configuration (or state) of the board. It has two constructors for generating the initial state only:

```
public State(int[][] board) throws IllegalArgumentException
public State (String inputFileName) throws FileNotFoundException,
                                           IllegalArgumentException
```

All other states are generated from existing states via one move using the following method:

```
public State successorState(Move m) throws IllegalArgumentException
```

The class is ready to represent a node in a circular doubly-linked list, as it has both `next` and `previous` links. It also has a `predecessor` link for tracing all the moves back to the initial state if `this` state is the final state.

The method `isGoalState()` checks if `this` state is the goal state. The method `solvable()` determines whether moves exist to transform `this` state to the goal state.

In the `State` class, the value of the function g (defined in Section 2) is stored in the public instance variable `numMoves`. The method

```
public int cost() throws IllegalArgumentException
```

evaluates $g + h_1$ or $g + h_2$ (h_1 and h_2 are also introduced in Section 2) depending on the heuristic used. The functions h_1 and h_2 are evaluated respectively by the methods `computeNumMismatchedTiles()` and `computeManhattanDistance()`. The chosen heuristics is stored in the static variable `heu`.

In steps 5c) and 5d of the A* algorithm, when the new f value for the state t is less than its old value, set t`.predecessor = `s and t`.numMoves = `s`.numMoves + 1`.

The `State` class implements the `compareTo()` method, which compares the total cost ($g + h_1$ or $g + h_2$) of `this` state with the argument state based on the heuristic `heu` used. Two states are also compared using the `compare()` method of the `StateComparator` class. This second comparison uses the lexicographic order of the tile number sequence on the board.

The `OrderedStateList` class represents a ***circular doubly-linked list*** with a ***dummy head node***. It can be used to represent either the `OPEN` list or the `CLOSE` list, depending on the value of the `boolean` instance variable `isOpen`. In the `OPEN` list, the nodes are ordered using the `compareTo()` method of the `State` class; and in the `CLOSE` list, they are ordered using the `compare()` method of the `StateComparator` class. Ordering of nodes employs the `compareStates()` method, which chooses one of the above two options for comparison based on `isOpen`. The constructor of `OrderedStateList` accepts a provided heuristic to initialize the static variable `heu` for the `State` class (shared by all objects of `State`) so its `compareTo()` method will perform accordingly.

The class `EightPuzzle` has two static methods `solve8Puzzle()` and `AStar()`. The first method accepts an initial board configuration, checks if the 8-puzzle has a solution, and if so, calls the second method multiple times, each time with a different heuristic, to solve the puzzle. The method `AStar` implements the A* algorithm as described in Section 3.

While you are encouraged to use helper functions wherever needed, please ***do not change*** the signature or access modifiers for already defined methods and variables in the template provided.

5. Double Moves

Suppose you are also allowed to move two adjacent tiles together in one step. The example below illustrates the operator DBL_LEFT:

```
1 2 3  DBL_LEFT   1 2 3
  4 5  ------->   4 5
6 7 8             6 7 8
```

The next example illustrates the operator DBL_DOWN:

```
1 8 2  DBL_DOWN   1   2
7 5 3  ------->   7 8 3
6   4             6 5 4
```

The two other operators DBL_RIGHT and DBL_UP carry out movements of two adjacent tiles to the right and upward, respectively.

When double-move operators are allowed, neither of the heuristic functions h_1 and h_2 bounds from below the number of moves to reach the goal. You are therefore asked to design ***a new heuristic function*** h_3 that is admissible. Have h_3 estimate the number of moves as tightly as

possible. This function can then be employed to solve an 8-puzzle using single and double moves.

6. Input and Output

The initial board configuration is input from a file assuming a default name `8Puzzle.txt` in the following format. The file has three rows, each containing three digits with exactly one blank in between. Every row starts with a digit. The nine digits are from 0 to 8 with ***no duplicates*** and 0 denoting the empty square. You may make the code interactive by repeatedly accepting different puzzle files and print out solutions. (There would be no extra credit for doing this though.)

The method `AStar()` returns a string that represents the solution to an 8-puzzle. It starts with the total number of moves (along with the used heuristic) and continues with a sequence of states that begins with the initial state and ends with the goal state.

- When either of the heuristic functions h_1 and h_2 is used, every intermediate state (the final state included) in the sequence must be generated by a ***single*** move from its preceding state.

- When the heuristic function h_3 is employed, a state may be generated by ***either a single move or a double move*** from its predecessor.

The solution sequence should be printed vertically with every two adjacent states separated by the move causing the state transition.

The method `Solve8Puzzle()` calls `AStar()` three times using different heuristics, and concatenate the three solution strings into one. Below is the printout of the string returned from a sample call to the method `Solve8Puzzle()`. (Your code may generate different solutions with the same minimum number of moves when employing the heuristics h_1 and h_2.) A sample output is given below:

```
9 moves in total (heuristic: number of mismatached tiles)

8 1 2
6 3
7 5 4

RIGHT

8 1 2
6   3
7 5 4

RIGHT

8 1 2
  6 3
7 5 4

DOWN
```

```
  1 2
8 6 3
7 5 4

LEFT

1   2
8 6 3
7 5 4

LEFT

1 2
8 6 3
7 5 4

UP

1 2 3
8 6
7 5 4

UP

1 2 3
8 6 4
7 5

RIGHT

1 2 3
8 6 4
7   5

DOWN

1 2 3
8   4
7 6 5


9 moves in total (heuristic: the Manhattan distance)

8 1 2
6 3
7 5 4

RIGHT

8 1 2
6   3
7 5 4

RIGHT
```

```
8 1 2
  6 3
7 5 4

DOWN

  1 2
8 6 3
7 5 4

LEFT

1   2
8 6 3
7 5 4

LEFT

1 2
8 6 3
7 5 4

UP

1 2 3
8 6
7 5 4

UP

1 2 3
8 6 4
7 5

RIGHT

1 2 3
8 6 4
7   5

DOWN

1 2 3
8   4
7 6 5



6 moves in total (heuristic: double moves allowed)

8 1 2
6 3
7 5 4
```

```
DBL_RIGHT

8 1 2
  6 3
7 5 4

DOWN

  1 2
8 6 3
7 5 4

DBL_LEFT

1 2
8 6 3
7 5 4

DBL_UP

1 2 3
8 6 4
7 5


RIGHT


1 2 3
8 6 4
7   5

DOWN

1 2 3
8   4
7 6 5
```

In case the 8-puzzle has no solution, then the string should print out this information as well as the initial state.

```
No solution exists for the following initial state:

4 1 2
5 3
8 6 7
```

7. Submission

Write your classes in the `edu.iastate.cs472.proj2` package. Turn in a zip file that contains the following:

a) A writeup that clearly describes your admissible heuristic function h_3 employed to solve the 8-puzzle optimally when single moves and double moves are all considered. Include your reasoning why the function is admissible.

b) Your source code.

c) A `README` file (optional) for comments on program execution or other things to pay attention to.

Please follow the discussion forums Project 1 Discussion and Project 1 Clarifications on Canvas. Include the Javadoc tag `@author` in every class source file showing your effort. Your zip file should be named `Firstname_Lastname_proj1.zip`.