Com S 228 Spring 2018

Project 3: The 8-puzzle (250 pts)

Due at **11:59pm**

Friday, March 23

1. Problem Description

The task in this project is to solve the 8-puzzle. Our objective here is 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 is allowed to proceed only 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 describes a sequence of three moves that solves the above example.

```
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
```

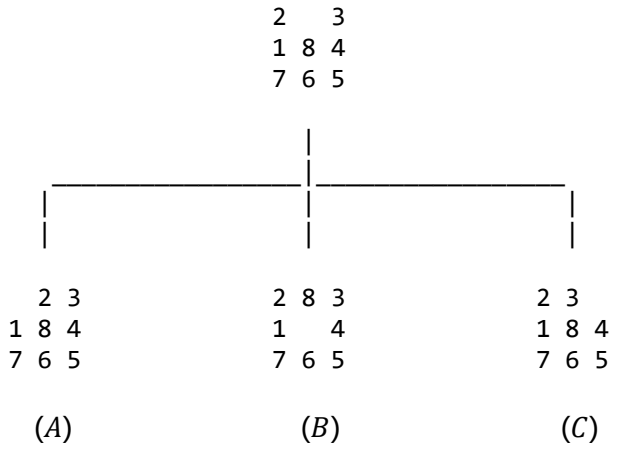
The solution path has length 3.

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. In other words, the goal state is as follows:

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

2. Heuristics

How to solve the 8-puzzle? Examining the same example above, we notice that the initial state has three successor states, that is, states that can be generated from the initial state with one move.



Which of the three alternatives A, B, or C appears most promising? In other words, which will likely lead to a rearrangement consisting of the ***smallest*** number of moves? An exhaustive search of subsequent moves in the puzzle can find out which of the three results in the shortest path to the goal state. However, such a search is utterly impractical as the number of states is destined to "explode" combinatorially in those puzzles where the path length from the initial state to the goal state is large. So we have to rely on some ***heuristics***, or rules of thumb, instead to judge which successor is most promising. In Artificial Intelligence (AI), the use of heuristics in problem solving is a common practice.

One of the judgments is estimating how close a state is to the goal. In the 8-puzzle, there are two very 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. 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***. Below are the estimates for the states A, B, and C from the goal state in the previous example:

$$h_1(A) = 2 \qquad h_1(B) = 3 \qquad h_1(C) = 4$$
$$h_2(A) = 2 \qquad h_2(B) = 4 \qquad h_2(C) = 4$$

Both h_1 and h_2 give estimates that are ***no greater than the minimum number of moves*** required to reach the goal from the current state. The function h_1 says that at least one move is needed to correct each mismatched tile. For instance, tile 2 in the initial state does not match tile 2 in the goal state; so at least one move is needed. Function h_2 is based on the fact that the number of moves involved in correcting a mismatch is no less than the Manhattan distance between the positions of the same tile in the two states. For instance, it requires at least two moves to return tile 8 in state B to its position in the goal state.

Heuristics such as h_1 and h_2 that do not overestimate the minimum solution length for the current state are called ***admissible***. Heuristic h_2 seems to be better than h_1 because it gives a closer estimate of the minimum number of moves to solve the 8-puzzle.

To derive a heuristic that estimates the number of moves taking the initial state to the goal state, we need to include the cost from the initial state to the current state as well. So the cost for an intermediate state s is a function

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

where $g(s)$ is the number of moves from the initial state to s, and $h(s)$ is a heuristic function (like h_1 and 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.

3. The A* Algorithm

To solve the 8-puzzle, you are required to use the so-called A* algorithm, from which all involved search algorithms (including the one that the IBM Deep Blue used to beat World Chess Champion Garry Kasparov in 1997) have more or less evolved. 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 will not happen with an 8-puzzle if before calling A* you use the method from Appendix to check that the puzzle is indeed solvable. (The situation may occur when applying A* to solve a different type of 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 and set their `predecessor` links back to s. For every successor state t of s:

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

    b. If t is not already 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. In the case that the new f value is chosen, reset the `predecessor` link of t (along the path yielding the lowest $g(t)$).

    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. Put t back on `OPEN` and reset its `predecessor` link.

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 `isGoal()` checks if `this` state is the goal state. The method `solvable()` determines whether moves exist to transform `this` state to the goal state, applying the method described in the Appendix.

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 defined 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`.

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 (see Appendix), and if so, calls the second method twice, 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. Input and Output

The initial board configuration is input from a file 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***.

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 which begins with the initial state and ends with the goal state. Every intermediate state (the final state included) in the sequence must result from a ***single*** move from its preceding state. The sequence should be printed vertically with every two adjacent states separated by the move causing the state transition.

The method `Solve8Puzzle()` calls `AStar()` twice, and concatenate the two 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.)

```
10 moves in total (heuristic: the Manhattan distance)

1 2 3
6 5 7
8 4

DOWN

1 2 3
6 5
8 4 7

RIGHT

1 2 3
6   5
8 4 7

UP
```

```
1 2 3
6 4 5
8   7

LEFT

1 2 3
6 4 5
8 7

DOWN

1 2 3
6 4
8 7 5

RIGHT

1 2 3
6   4
8 7 5

RIGHT

1 2 3
  6 4
8 7 5


UP

1 2 3
8 6 4
  7 5

LEFT

1 2 3
8 6 4
7   5

DOWN

1 2 3
8   4
7 6 5
```

10 moves in total (heuristic: number of mismatached tiles)

```
1 2 3
6 5 7
8 4

RIGHT

1 2 3
6 5 7
8   4

DOWN

1 2 3
6   7
8 5 4

LEFT

1 2 3
6 7
8 5 4

UP

1 2 3
6 7 4
8 5

RIGHT

1 2 3
6 7 4
8   5

DOWN

1 2 3
6   4
8 7 5

RIGHT

1 2 3
  6 4
8 7 5
```

```
UP

1 2 3
8 6 4
  7 5

LEFT

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
```

6. Submission

Write your classes in the `edu.iastate.cs228.hw3` package. Turn in the zip file not your class files. Please follow the guideline posted under Documents & Links on Canvas.

You are not required to submit any JUnit test cases. Nevertheless, you are encouraged to write JUnit tests for your code. Since these tests will not be submitted, feel free to share them with other students.

Include the Javadoc tag `@author` in each class source file. Your zip file should be named `Firstname_Lastname_HW3.zip`.

Appendix. Solvability of an 8-puzzle

There is a simple rule for telling if an 8-puzzle is solvable. We refer to the following state where all the tiles appear in the "correct" order (which does ***not*** consider the empty square):

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

The order is that every tile with a smaller number should appear either above or to the left of any tile with a larger number. In our goal state,

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

8 appears before 4. We call this violation of the order an "inversion". The goal state has seven inversions:

6 comes before 5
7 comes before 5
7 comes before 6
8 comes before 4
8 comes before 5
8 comes before 6
8 comes before 7

It can be shown that an 8-puzzle with an initial state and a goal state is solvable if and only if the two states differ by an even number of inversions. In our project, the goal state is fixed, so an 8-puzzle in the project is ***solvable if and only if its initial state also has an odd number of inversions***. For instance, the initial state

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

has five inversions:

4 comes before 1, 2, 3
8 comes before 6, 7

Therefore it is solvable. Swapping the positions of tiles 3 and 5, we have another state

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

with six inversions:

4 comes before 1, 2, 3
5 comes before 3
8 comes before 6, 7

Hence the corresponding 8-puzzle is not solvable.