Algorithms Homework Assignment #4

"8-Puzzle" (3 points)

This homework is adopted from the Princeton University "Algorithms, Part I" by Kevin Wayne, Robert Sedgewick.

Due date: Wednesday, May 9th (Wed) 2018, 9AM
- Submit softcopy on the server.

Write a program to solve the 8-puzzle problem (and its natural generalizations) using the A* search algorithm.

Template code for the homework assignment, as well as some testing code, will be provided to you;

- "Board.java" and "Solver.java" are what you need to implement.
- "Queue.java", "Stack.java", and "MinPQ.java" are provided for you. ➔ (This is already done for you. Nothing to do in these files.)
- You can download these files from e-Class website.

The problem: The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in order, using as few moves as possible. You are permitted to slide blocks horizontally or vertically into the blank square. The following shows a sequence of legal moves from an initial board (left) to the goal board (right).

```
    1  3        1     3        1  2  3        1  2  3        1  2  3
4   2  5   ➔    4  2  5   ➔    4     5   ➔    4  5     ➔    4  5  6
7   8  6        7  8  6        7  8  6        7  8  6        7  8
initial         1 left          2 up          5 left         goal
```

Best-first search: Now, we describe a solution to the problem that illustrates a general artificial intelligence methodology known as the A* search algorithm. We define a search node of the game to be a board, the number of moves made to reach the board, and the previous search node. First, insert the initial search node (the initial board, 0 moves, and a null previous search node) into a priority queue. Then, delete from the priority queue the search node with the minimum priority, and insert onto the priority queue all neighboring search nodes (those that can be reached in one move from the dequeued search node). Repeat this procedure until the search node dequeued corresponds to a goal board. The success of this approach hinges on the choice of priority function for a search node. We consider the following priority functions:

Read this to understand Mantattan Priority Function

Manhattan priority function. The sum of the Manhattan distances (sum of the vertical and horizontal distance) from the blocks to their goal positions, plus the number of moves made so far to get to the search node.

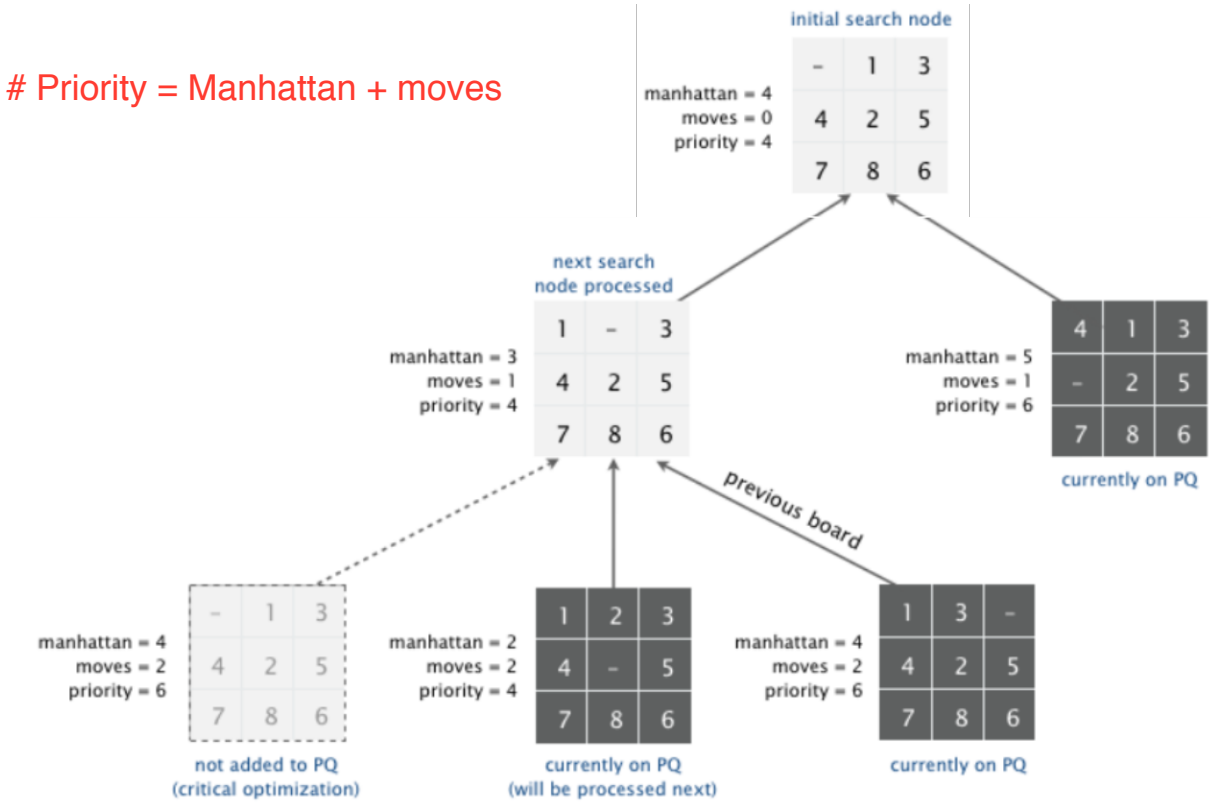
For example, the Manhattan priorities of the initial search node below is 10.

```
8  1  3        1  2  3        1 2 3 4 5 6 7 8
4     2        4  5  6        ---------------
7  6  5        7  8           1 2 0 0 2 2 0 3
initial        goal           Manhattan = 10 + 0
```

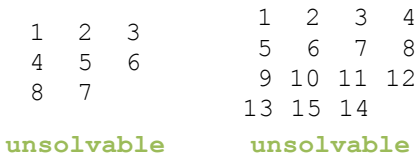
We make a key observation: To solve the puzzle from a given search node on the priority queue, the total number of moves we need to make (including those already made) is at least its priority, using the Manhattan priority function. (This is true because each block must move its Manhattan distance from its goal position. Note that we do not count the blank square when computing the Manhattan priorities.) Consequently, when the goal board is dequeued, we have discovered not only a sequence of moves from the initial board to the goal board, but one that makes the fewest number of moves.

A critical optimization. Best-first search has one annoying feature: search nodes corresponding to the same board are enqueued on the priority queue many times. To reduce unnecessary exploration of useless search nodes, when considering the neighbors of a search node, don't enqueue a neighbor if its board is the same as the board of the previous search node.

Game tree. One way to view the computation is as a game tree, where each search node is a node in the game tree and the children of a node correspond to its neighboring search nodes. The root of the game tree is the initial search node; the internal nodes have already been processed; the leaf nodes are maintained in a priority queue; at each step, the A* algorithm removes the node with the smallest priority from the priority queue and processes it (by adding its children to both the game tree and the priority queue).



Detecting unsolvable puzzles. Not all initial boards can lead to the goal board by a sequence of legal moves, including the two below:



To detect such situations, use the fact that boards are divided into two equivalence classes with respect to reachability: (i) those that lead to the goal board and (ii) those that lead to the goal board if we modify the initial board by swapping any pair of adjacent (non-blank) blocks in the same row. To apply the fact, run the A* algorithm simultaneously on two puzzle instances—one with the initial board and one with the initial board modified by swapping a pair of adjacent blocks in the same row, called '**twin**'. Exactly one of the two will lead to the goal board.

Board and Solver data types. Organize your program by creating an immutable data type Board with the following API:

```
public class Board {
     public Board(int[][] blocks)       // construct a board from an N-by-N array of blocks
                                        // (where blocks[i][j] = block in row i, column j)
     public int dimension()             // board dimension N
     //public int hamming()               // number of blocks out of place
     public int manhattan()             // sum of Manhattan distances between blocks and goal
     public boolean isGoal()            // is this board the goal board?
     public Board twin()                // a board that is obtained by exchanging
                                        // two adjacent blocks in the same row
     public boolean equals(Object y)    // does this board equal y?
     public Iterable<Board> neighbors() // all neighboring boards
     public String toString()           // string representation of this board
                                        // (in the output format specified below)
     public static void main(String[] args) // unit tests (do not modify)
}
```

Corner cases. You may assume that the constructor receives an N-by-N array containing the N^2 integers between 0 and $N^2 - 1$, where 0 represents the blank square.

Performance requirements. Your implementation should support all Board methods in time proportional to N^2 (or better) in the worst case.

Also, create an immutable data type Solver with the following API:

```
public class Solver {
     public Solver(Board initial)           // find a solution to the initial board
                                            // (using the A* algorithm)
     public boolean isSolvable()            // is the initial board solvable?
     public int moves()                     // min number of moves to solve initial board;
                                            // -1 if unsolvable
     public Iterable<Board> solution()      // sequence of boards in a shortest solution;
                                            // null if unsolvable
     public static void main(String[] args) // solve a slider puzzle (given below)
}
```

To implement the A* algorithm, you must use the MinPQ data type from MinPQ.java for the priority queue(s). ➔ MinPQ.java is given to you. Nothing to do!

Corner cases. The constructor should throw a java.lang.NullPointerException if passed a null argument. ➔ Already done for you!

Solver test client. Use the given test client in ***main()*** method of Solver.java to read a puzzle from a file (specified as a command-line argument) and print the solution to standard output. ➔ Already done for you!

Input and output formats. The input and output format for a board is the board dimension N followed by the N-by-N initial board, using 0 to represent the blank square. As an example,

Other requirements:

- Your program should work correctly for arbitrary N-by-N boards (for any $2 \le N < 128$), even if it is too slow to solve some of them in a reasonable amount of time.
  - if you run out of default Java memory, you can try "`java -Xmx3096m Solver test/puzzle47.txt`"
- Your program must run on the course Linux server at nsl2.cau.ac.kr.
- Your code must include your name and student ID at the beginning of the code as a comment.
- Your code should be easily readable and include sufficient comments for easy understanding.
- Your code should not include any Korean characters.

Example input file and Solver output:

```
% more puzzle04.txt
3
  0 1 3
  4 2 5
  7 8 6
% java Solver puzzle04.txt
Minimum number of moves = 4
3
  0 1 3
  4 2 5
  7 8 6
3
  1 0 3
  4 2 5
  7 8 6
3
  1 2 3
  4 0 5
  7 8 6
3
  1 2 3
  4 5 0
  7 8 6
3
  1 2 3
  4 5 6
  7 8 0
% more puzzle-unsolvable3x3.txt
3
  1 2 3
  4 5 6
  8 7 0
% java Solver puzzle3x3-unsolvable.txt
No solution possible
%
```

For puzzles that required more memory than default amount of memory, you can run using the command

```
java -Xmx3096m Solver test/puzzle50.txt
```

which increases the max heap size to 3GB. However, this should not affect the correctness of your code.

What and how to submit

- You must submit softcopy of "Board.java" and "Solver.java" file;
- Here are the instructions on how to submit the softcopy files :
  ① Login to your server account at nsl2.cau.ac.kr.
  ② In your home directory, create a directory "submit_alg/submit_<student ID>_hw4"
  (ex> "/student/20149999/submit_alg/submit_20149999_hw4")
  ③ Put all your files, including of "Board.java" and "Solver.java", in that directory together with all the files that you downloaded from e-Class (we need those files to compile your program).

Grading criteria:

- You get 3 points
  - if all of your programs work correctly, AND
  - if you meet all of above requirements, AND
  - if your code handles all the exceptional cases that might occur.
- Otherwise, partial deduction may apply.
- No delayed submissions are accepted.
- Copying other student's work will result in negative points.
- Code that does not compile or code that does not run will result in negative points

FAQ

- **Is 0 a block?** No, 0 represents the blank square. Do not treat it as a block when computing either the Hamming or Manhattan priority functions.

- **I'm a bit confused about the purpose of the twin() method**. You will use it to determine whether a puzzle is solvable: exactly one of a board and its twin are solvable. A twin is obtained by swapping two adjacent blocks (the blank does not count) in the same row. For example, here is a board and its 5 possible twins. Your solver will use only one twin.

- **Can I terminate the search as soon as a goal search node is enqueued (instead of dequeued)?** No, even though it does lead to a correct solution for the slider puzzle problem using the Manhattan priority functions, it's not technically the A* algorithm (and will not find the correct solution for other problems and other priority functions).

- **Input files**. The directory 'testinput/' contains many sample puzzle input files.
  - ✓ The shortest solution to puzzle[T].txt requires exactly T moves.
  - ✓ The shortest solution to puzzle4x4-hard1.txt and puzzle4x4-hard2.txt are 38 and 47, respectively.
  - ✓ Warning: puzzle36.txt is especially difficult.

- For your understanding, here are the contents of our priority queue (sorted by priority) just before dequeueing each node when using the Manhattan priority function on puzzle04.txt.

```
Step 0:    priority  = 4
           moves     = 0
           manhattan = 4
           3
            0  1  3
            4  2  5
            7  8  6

Step 1:    priority  = 4    priority  = 6
           moves     = 1    moves     = 1
           manhattan = 3    manhattan = 5
           3                3
            1  0  3          4  1  3
            4  2  5          0  2  5
            7  8  6          7  8  6

Step 2:    priority  = 4    priority  = 6    priority  = 6
           moves     = 2    moves     = 1    moves     = 2
           manhattan = 2    manhattan = 5    manhattan = 4
           3                3                3
            1  2  3          4  1  3          1  3  0
            4  0  5          0  2  5          4  2  5
            7  8  6          7  8  6          7  8  6

Step 3:    priority  = 4    priority  = 6    priority  = 6    priority  = 6    priority  = 6
           moves     = 3    moves     = 3    moves     = 2    moves     = 3    moves     = 1
           manhattan = 1    manhattan = 3    manhattan = 4    manhattan = 3    manhattan = 5
           3                3                3                3                3
            1  2  3          1  2  3          1  3  0          1  2  3          4  1  3
            4  5  0          4  8  5          4  2  5          0  4  5          0  2  5
            7  8  6          7  0  6          7  8  6          7  8  6          7  8  6

Step 4:    priority  = 4    priority  = 6    priority  = 6    priority  = 6    priority  = 6    priority  = 6
           moves     = 4    moves     = 3    moves     = 4    moves     = 2    moves     = 3    moves     = 1
           manhattan = 0    manhattan = 3    manhattan = 2    manhattan = 4    manhattan = 3    manhattan = 5
           3                3                3                3                3                3
            1  2  3          1  2  3          1  2  0          1  3  0          1  2  3          4  1  3
            4  5  6          0  4  5          4  5  3          4  2  5          4  8  5          0  2  5
            7  8  0          7  8  6          7  8  6          7  8  6          7  0  6          7  8  6
```