Programming Assignment 4: 8 Puzzle

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

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 predecessor search node. First, insert the initial search node (the initial board, 0 moves, and a null predecessor 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 two priority functions:

- *Hamming priority function.* The number of blocks in the wrong position, plus the number of moves made so far to get to the search node. Intuitively, a search node with a small number of blocks in the wrong position is close to the goal, and we prefer a search node that have been reached using a small number of moves.
- *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 Hamming and Manhattan priorities of the initial search node below are 5 and 10, respectively.

```
8  1  3       1  2  3     1  2  3  4  5  6  7  8     1  2  3  4  5  6  7  8
4     2       4  5  6     -----------------------     -----------------------
7  6  5       7  8        1  1  0  0  1  1  0  1     1  2  0  0  2  2  0  3

initial          goal          Hamming = 5 + 0          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 either the Hamming or Manhattan priority function. (For Hamming priority, this is true because each block that is out of place must move at least once to reach its goal position. For Manhattan priority, 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 Hamming or 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. (Challenge for the mathematically inclined: prove this fact.)

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 predecessor search node.

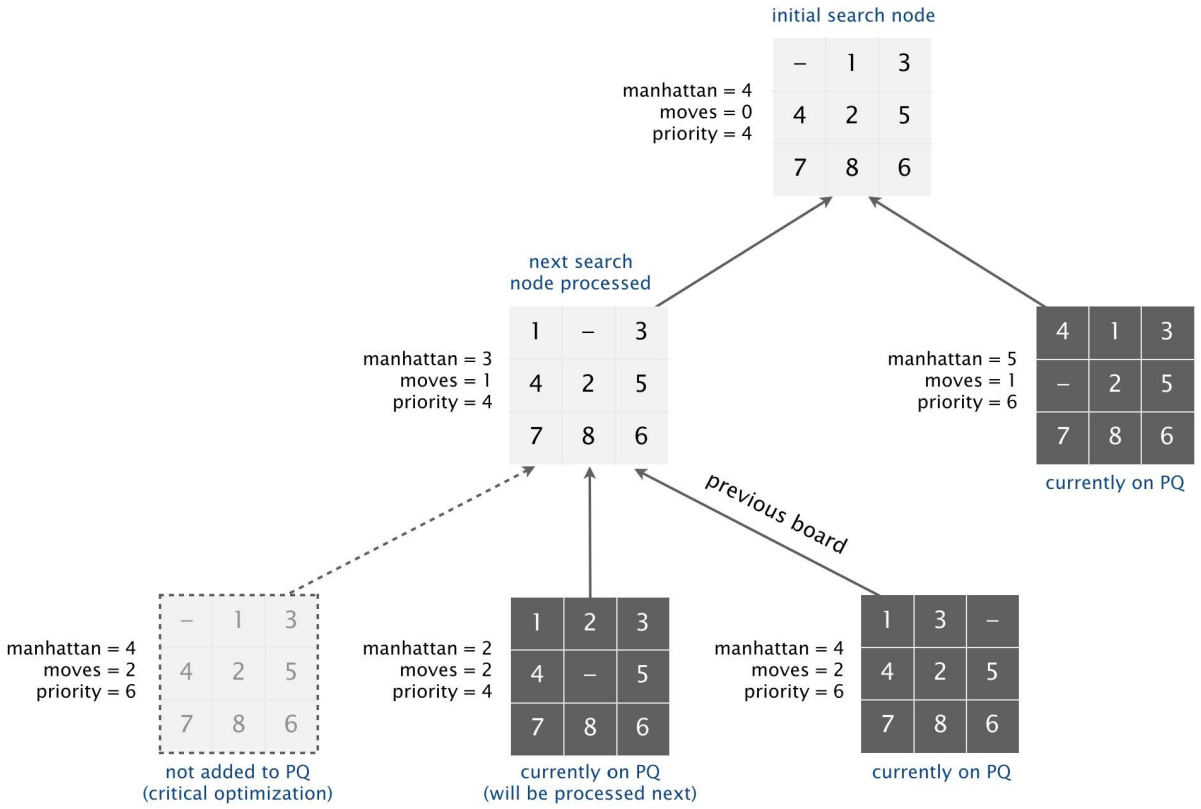
```
8  1  3       8  1  3       8  1          8  1  3       8  1  3
```

```
 4     2       4  2        4  2  3       4     2       4  2  5
 7  6  5       7  6  5     7  6  5       7  6  5       7  6

predecessor   search node   neighbor      neighbor      neighbor
                                         (disallow)
```

A second optimization. To avoid recomputing the Manhattan priority of a search node from scratch each time during various priority queue operations, pre-compute its value when you construct the search node; save it in an instance variable; and return the saved value as needed. This caching technique is broadly applicable: consider using it in any situation where you are recomputing the same quantity many times and for which computing that quantity is a bottleneck operation.

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:

```
 1  2  3          1  2  3  4
 4  5  6          5  6  7  8
 8  7             9 10 11 12
                 13 15 14
unsolvable

                 unsolvable
```

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 blocks (the blank square is not a block). (Difficult challenge for the mathematically inclined: prove this fact.) To apply the fact, run the A* algorithm on *two* puzzle instances—one with the initial board and one with the initial board modified by swapping a pair of blocks—in lockstep (alternating back and forth between exploring search nodes in each of the two game trees). 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 any pair of blocks
    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 (not graded)
}
```

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 `MinPQ` from algs4.jar for the priority queue(s).*

Corner cases. The constructor should throw a `java.lang.IllegalArgumentException` if passed a null argument.

Solver test client. Use the following test client to read a puzzle from a file (specified as a command-line argument) and print the solution to standard output.

```
public static void main(String[] args) {

    // create initial board from file
    In in = new In(args[0]);
    int n = in.readInt();
    int[][] blocks = new int[n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            blocks[i][j] = in.readInt();
    Board initial = new Board(blocks);

    // solve the puzzle
    Solver solver = new Solver(initial);

    // print solution to standard output
    if (!solver.isSolvable())
        StdOut.println("No solution possible");
    else {
        StdOut.println("Minimum number of moves = " + solver.moves());
        for (Board board : solver.solution())
            StdOut.println(board);
    }
}
```

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,

```
% more puzzle04.txt
3
 0  1  3
 4  2  5
 7  8  6

% java-algs4 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 puzzle3x3-unsolvable.txt
3
 1  2  3
 4  5  6
 8  7  0

% java-algs4 Solver puzzle3x3-unsolvable.txt
No solution possible
```

Your program should work correctly for arbitrary *n*-by-*n* boards (for any 2 ≤ *n* < 128), even if it is too slow to solve some of them in a reasonable amount of time.

Deliverables. Submit only the files `Board.java` and `Solver.java` (with the Manhattan priority). We will supply `algs4.jar`. You may not call any library functions other those in `java.lang`, `java.util`, and `algs4.jar`. You must use `MinPQ` for the priority queue(s).