COS 226 Programming Assignment

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. 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 position (left) to the goal position (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                                                      goal
```

Best-first search. We now describe an algorithmic solution to the problem that illustrates a general artificial intelligence methodology known as the A* search algorithm. We define a *state* of the game to be the board position, the number of moves made to reach the board position, and the previous state. First, insert the initial state (the initial board, 0 moves, and a null previous state) into a priority queue. Then, delete from the priority queue the state with the minimum priority, and insert onto the priority queue all neighboring states (those that can be reached in one move). Repeat this procedure until the state dequeued is the goal state. The success of this approach hinges on the choice of *priority function* for a state. 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 state. Intutively, a state with a small number of blocks in the wrong position is close to the goal state, and we prefer a state that have been reached using a small number of moves.
- *Manhattan priority function*. The sum of the 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 state.

For example, the Hamming and Manhattan priorities of the initial state 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 oberservation: to solve the puzzle from a given state 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 tile when computing the Hamming or Manhattan priorities.)

Consequently, as soon as we dequeue a state, we have not only discovered a sequence of moves from the initial board to the board associated with the state, but one that makes the fewest number of moves. (Challenge for the mathematically inclined: prove this fact.)

A critical optimization. After implementing best-first search, you will notice one annoying feature: states corresponding to the same board position are enqueued on the priority queue many times. To prevent unnecessary exploration of useless states, when considering the neighbors of a state, don't enqueue the neighbor if its board position is the same as the previous state.

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

    previous       state       disallow
```

Your task. Write a program `Solver.java` that reads the initial board from standard input and prints to standard output a sequence of board positions that solves the puzzle in the fewest number of moves. Also print out the total number of moves and the total number of states ever enqueued.

The input will consist of the board dimension *N* followed by the *N*-by-*N* initial board position. The input format uses 0 to represent the blank square. As an example,

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

```
% java Solver < puzzle04.txt
    1  3
 4  2  5
 7  8  6

 1     3
 4  2  5
 7  8  6

 1  2  3
 4     5
 7  8  6

 1  2  3
 4  5
 7  8  6

 1  2  3
 4  5  6
 7  8

Number of states enqueued = 10
Minimum number of moves = 4
```

Note that your program should work for arbitrary *N*-by-*N* boards (for any *N* greater than 1), even if it is too slow to solve some of them in a reasonable amount of time.

Detecting infeasible puzzles. Not all initial board positions can lead to the goal state. Modify your program to detect and report such situations.

```
% more puzzle-impossible3x3.txt
3
 1  2  3
 4  5  6
 8  7  0

% java Solver < puzzle3x3-impossible.txt
No solution possible
```

Hint: use the fact that board positions are divided into two equivalence classes with respect to reachability: (i) those that lead to the goal position and (ii) those that lead to the goal position if we modify the initial board by swapping any pair of adjacent (non-blank) blocks. There are two ways to apply the hint:

- 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 (non-blank) blocks. Exactly one of the two will lead to the goal position.
- Derive a mathematical formula that tells you whether a board is solvable or not.

Board and Solver data types. Organize your program in an appropriate number of data types. At a minimum, you are required to implement the following APIs. You are permitted to add additional methods or data types, such as `State`.

```
public class Board {
   public Board(int[][] tiles)        // construct a board from an N-by-N array of tiles
   public int hamming()               // return number of blocks out of place
   public int manhattan()             // return sum of Manhattan distances between blocks and goal
   public boolean equals(Object y)    // does this board position equal y
   public Iterable<Board> neighbors() // return an Iterable of all neighboring board positions
   public String toString()           // return a string representation of the board
}


public class Solver {
   public Solver(Board initial)        // find a solution to the initial board
   public boolean isSolvable()         // is the initial board solvable?
   public int moves()                  // return min number of moves to solve initial board; -1 if no solution
   public Iterable<Board> solution()   // return an Iterable of board positions in solution
}
```

The following `main()` for `Solver` demonstrates the APIs: it reads a puzzle instance from standard input and prints the solution to standard output.

```
public static void main(String[] args) {
   int N = StdIn.readInt();
   int[][] tiles = new int[N][N];
   for (int i = 0; i < N; i++)
       for (int j = 0; j < N; j++)
          tiles[i][j] = StdIn.readInt();
    Board initial = new Board(tiles);
    Solver solver = new Solver(initial);
    for (Board board : solver.solution())
       System.out.println(board);
```

```
        if (!solver.isSolvable()) System.out.println("No solution possible");
        else System.out.println("Minimum number of moves = " + solver.moves());
    }
```

Deliverables. Submit `Board.java`, `Solver.java` (with the Manhattan priority) and any other helper data types that you use (excluding those in `stdlib.jar` and `adt.jar`). Finally, submit a readme.txt file and answer the questions.