# Data Structures and Algorithms in Java

Assignment 6 (Eight Puzzle) Discussion

## Introduction

The purpose of this project is to write a program to solve the 8-puzzle problem (and its natural generalizations) using the $A^*$ search algorithm

## Introduction

Before you write any code, make sure you thoroughly understand the concepts that are central to this assignment. Compute the following for the two initial boards $A$ and $B$ shown below:

$A$

| | | |
|---|---|---|
| 4 | 1 | 3 |
| | 2 | 6 |
| 7 | 5 | 8 |

$B$

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 6 | 5 |
| 7 | 8 | |

1. Hamming distance of the board to the goal board
2. Manhattan distance of the board to the goal board
3. Neighboring boards of the board
4. Row-major order of the board
5. Position of the blank tile (in row-major order) in the board
6. Number of inversions (excluding the blank tile) for the board
7. Is the board solvable? Explain why or why not
8. A shortest solution for the board, if one exists

## Problem 1 (Board Data Type)

Implement an immutable data type called `Board` to represent a board in an $n$-puzzle, supporting the following API:

| Board | |
|---|---|
| `Board(int[][] tiles)` | constructs a board from an $n \times n$ array; `tiles[i][j]` is the tile at row $i$ and column $j$, with 0 denoting the blank tile |
| `int size()` | returns the size of this board size |
| `int tileAt(int i, int j)` | returns the tile at row `i` and column `j` |
| `int hamming()` | returns Hamming distance between this board and the goal board |
| `int manhattan()` | returns the Manhattan distance between this board and the goal board |
| `boolean isGoal()` | returns `true` if this board is the goal board, and `false` otherwise |
| `boolean isSolvable()` | returns `true` if this board solvable, and `false` otherwise |
| `Iterable<Board> neighbors()` | returns an iterable object containing the neighboring boards of this board |
| `boolean equals(Object other)` | returns `true` if this board is the same as `other`, and `false` otherwise |
| `String toString()` | returns a string representation of this board |

## Problem 1 (Board Data Type)

- Instance variables:
  - Tiles in the board, `int[][] tiles`.
  - Board size, `int n`.
  - Hamming distance to the goal board, `int hamming`.
  - Manhattan distance to the goal board, `int manhattan`.
  - Position of the blank tile in row-major order, `int blankPos`.
- `private int[][] cloneTiles()`
  - Return a defensive copy of the tiles of the board.
- `Board(int[][] tiles)`
  - Initialize the instance variables `this.tiles` and `n` to `tiles` and the number of rows in `tiles` respectively.
  - Compute the Hamming/Manhattan distances to the goal board and the position of the blank tile in row-major order, and store the values in the instance variables `hamming`, `manhattan`, and `blankPos` respectively.
- `int size()`
  - Return the board size.
- `int tileAt(int i, int j)`
  - Return the tile at row `i` and column `j`.
- `int hamming()`
  - Return the Hamming distance to the goal board.
- `int manhattan()`
  - Return the Manhattan distance to the goal board.

## Problem 1 (Board Data Type)

- `boolean isGoal()`
  - Return `true` if the board is the goal board, and `false` otherwise.
- `boolean isSolvable()`
  - Create an array of size $n^2 - 1$ containing the tiles (excluding the blank tile) of the board in row-major order.
  - Use `Inversions.count()` to compute the number of inversions in the array.
  - From the number of inversions, compute and return whether the board is solvable.
- `Iterable<Board> neighbors()`
  - Create a queue `q` of `Board` objects.
  - For each possible neighbor of the board (determined by the blank tile position):
    - Clone the tiles of the board.
    - Exchange an appropriate tile with the blank tile in the clone.
    - Construct a `Board` object from the clone, and enqueue it into `q`.
  - Return `q`.
- `boolean equals(Board other)`
  - Return `true` if the board is the same as `other`, and `false` otherwise.

## Problem 2 (Solver Data Type)

Implement an immutable data type called `Solver` that uses the $A^*$ algorithm to solve the 8-puzzle and its generalizations. The data type should support the following API:

| Solver | |
|---|---|
| `Solver(Board board)` | finds a solution to the initial `board` using the $A^*$ algorithm |
| `int moves()` | returns the minimum number of moves needed to solve the initial board |
| `Iterable<Board> solution()` | returns a sequence of boards in a shortest solution of the initial board |

## Problem 2 (Solver Data Type)

- Instance variables:
  - Minimum number of moves needed to solve the initial board, `int moves`.
  - Sequence of boards in a shortest solution of the initial board, `LinkedStack<Board> solution`.
- `Solver :: SearchNode` (represents a node in the game tree)
  - Instsance variables:
    - The board represented by this node, `Board board`.
    - Number of moves it took to get to this node from the initial node, `int moves`.
    - The previous search node, `SearchNode previous`.
  - `SearchNode(Board board, int moves, SearchNode previous)`
    - Initialize instance variables appropriately.
  - `int compareTo(SearchNode other)`
    - Return a comparison of the search node with other , based on the sum: Manhattan distance of the board in the node plus the number of moves to the node (from the initial search node).

## Problem 2 (Solver Data Type)

- `Solver(Board initial)`
  - Create a `MinPQ<SearchNode>` object `pq` and insert the initial search node into it
  - As long as `pq` is not empty:
    - Remove the smallest node (call it `node`) from `pq`.
    - If the board in `node` is the goal board, extract from the node the number of moves in the solution and the solution and store the values in the instance variables `moves` and `solution` respectively, and break.
    - Otherwise, iterate over the neighboring boards of `node.board`, and for each neighbor that is different from `node.previous.board`, insert a new `SearchNode` object into `pq`, constructed using appropriate values.
- `int moves()`
  - Return the minimum number of moves needed to solve the initial board.
- `Iterable<Board> solution()`
  - Return the sequence of boards in a shortest solution of the initial board.