Project 4 (8 Puzzle)

Clarifications and Hints

Prologue

Project goal: write a program to solve the 8-puzzle problem (and its natural generalizations) using the $A^\star$ search algorithm

The zip file (`http://www.swamiiyer.net/cs210/8_puzzle.zip`) for the project contains

- project specification (`8_puzzle.pdf`)
- starter files
    - `Board.java`
    - `Solver.java`
- test script (`run_tests.py`)
- visualization client (`SolverVisualizer`)
- report template (`report.txt`)

Problems

Problem 1 (*Board Data Type*) Create an immutable data type `Board` with the following API:

method	description
`Board(int[][] tiles)`	construct a board from an N-by-N array of tiles
`int tileAt(int i, int j)`	tile at row i, column j (or 0 if blank)
`int size()`	board size N
`int hamming()`	number of tiles out of place
`int manhattan()`	sum of Manhattan distances between tiles and goal
`boolean isGoal()`	is this board the goal board?
`boolean isSolvable()`	is this board solvable?
`boolean equals(Board that)`	does this board equal $that$?
`Iterable<Board> neighbors()`	all neighboring boards
`String toString()`	string representation of this board

Hints

- 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`

Problems

- Helper method `int blankPos()`
  - Return the position (in row-major order) of the blank (zero) tile; for example, if `N = 3` and the blank tile is in row `i = 1` and column `j = 2`, the method should return 5
- Helper method `int inversions()`
  - Return the number of inversions
- Helper method `int[][] cloneTiles()`
  - Clone and return `this.tiles`
- `Board(int[][] tiles)`
  - Initialize the instance variables `this.tiles` and `this.N` to `tiles` and the number of rows in `tiles` respectively
  - Calculate the Hamming and Manhattan distances of `this` board and the goal board, in the instance variables `hamming` and `manhattan` respectively
- `int tileAt(int i, int j)`
  - Return the tile at row `i` and column `j`
- `int size()`
  - Return the board size

Problems

- `int hamming()`
  - Return the Hamming distance to the goal board
- `int manhattan()`
  - Return the Manhattan distance to the goal board
- `boolean isGoal()`
  - Return `true` if `this` board is the goal board, and `false` otherwise
- `boolean isSolvable()`
  - Return `true` if `this` board is solvable, and `false` otherwise
- `boolean equals(Board that)`
  - Return `true` if `this` board equals `that`, and `false` otherwise
- `Iterable<Board> neighbors()`
  - For each possible neighboring board (determined by the position of the blank tile), clone the tiles of `this` board, exchange the appropriate tile with the blank tile in the clone, make a `Board` object from the clone, and enqueue it into a queue of `Board` objects
  - Return the queue

Problem 2 (*Solver Data Type*) Create an immutable data type `Solver` with the following API:

method	description
`Solver(Board initial)`	find a solution to the initial board (using the $A^\star$ algorithm)
`int moves()`	the minimum number of moves to solve initial board
`Iterable<Board> solution()`	sequence of boards in a shortest solution

Hints

- Instance variables
  - Sequence of boards in a shortest solution, `LinkedStack<Board> solution`
  - Minimum number of moves to solve the initial board, `int moves`
- Helper `SearchNode` type representing a node in the game tree
  - Instance variables: the board represented by this node, `Board board`; number of moves it took to get to this node from the initial node (containing the initial board), `int moves`; and the previous search node, `SearchNode previous`
  - `SearchNode(Board board, int moves, SearchNode previous)`: initialize instance variables appropriately

Problems

- Helper `int HammingOrder.compare(SearchNode a, SearchNode b)`
  - Return a comparison of the `a.board.hamming() + a.moves` and `b.board.hamming() + b.moves`
- Helper `int ManhattanOrder.compare(SearchNode a, SearchNode b)`
  - Return a comparison of the `a.board.manhattan() + a.moves` and `b.board.manhattan() + b.moves`
- `Solver(Board initial)`
  - Create a `MinPQ<SearchNode>` object `pq`, initialize `solution`, and insert initial search node into `pq`
  - As long as `pq` is not empty
    - Remove the minimum (call it `node`) from `pq`
    - If the board in `node` is the goal board, obtain `moves` and `solution` from it and break
    - Otherwise, iterate over the neighboring boards, and for each neighbor `board` that is different from the previous, insert a new `SearchNode` object into `pq`, built using appropriate arguments
- `int moves()`
  - Return the minimum number of moves to solve the initial board
- `Iterable<Board> solution()`
  - Return the sequence of boards in a shortest solution

Epilogue

The `data` directory contains a number of sample input files representing boards of different sizes; the input (and output) format for a board is the board size N followed by the N-by-N board, using 0 to represent the blank square

```
$ more data/puzzle04.txt
3
 0  1  3
 4  2  5
 7  8  6
```

The visualization client `SolverVisualizer` takes the name of a file as command-line argument, and

- Uses your `Solver` and `Board` data types to solve the sliding block puzzle defined by the input file
- Renders a graphical animation of your program's output
- Uses the `Board.manhattan()` to display the Manhattan distance at each stage of the solution

```
$ java SolverVisualizer data/puzzle04.txt
```

Epilogue

Your project report (use the given template, `report.txt`) must include

- time (in hours) spent on the project
- short description of how you approached each problem, issues you encountered, and how you resolved those issues
- acknowledgement of any help you received
- other comments (what you learned from the project, whether or not you enjoyed working on it, etc.)

Before you submit your files

- make sure your programs meet the input and output specifications by running the following command on the terminal

  ```
  $ python run_tests.py -v [<problems>]
  ```

- make sure your programs meet the style requirements by running the following command on the terminal

  ```
  $ check_style <program>
  ```

- make sure your report isn't too verbose, doesn't contain lines that exceed 80 characters, and doesn't contain spelling/grammatical mistakes