COS 226

2048 Assignment

/* created by colton wolk */

Write a program that solves the modified version of the game 2048 (but only up to the 128 tile) using the A* search algorithm.

The modified game: 2048, created by Gabriele Cirulli, is a sliding block game played on a 4x4 grid consisting of blank tiles and numbered tiles. *This assignment uses a modified version of the game.* The player may "swipe" the tiles horizontally or vertically, causing each tile to slide in the given direction until it is stopped by either another tile or the border of the grid. If two tiles that have the same value collide during a move, they are combined to form a new tile that is the sum of the previous tiles. Every turn, a new tile (with a value of 2) spawns in the same position on the board (the upper right corner), which is necessarily unoccupied or else the game ends. The objective is to combine tiles with the same value until the 128 tile is created, without running out of space on the board.

Board data type. To begin, create a data type that models a *L*-by-*L* board with sliding tiles. Implement an immutable data type Board with the following API:

```
public class Board {
    // create a board from a L x L array of tiles,
    // where tiles[row][col] = tile at (row, col)
    public Board(int[][] tiles)
    // string representation of this board
    public String toString()
    // tile at (row, col) or 0 if blank
    public int tileAt(int row, int col)
    // number of non-zero tiles on the board
    public int size()
    // return the largest tile in the board
    public int largestTile()
    // the total weighted value of the board
    public int weightedScore()
    // snake value of the board
    public int snakeScore()
    // penalty for having similar values far away
    public int penalty()
    // does this board have GOAL_TILE?
    public boolean hasWon()
    // are there no more possible swipes?
    public boolean gameFinished()
    // does this board equal y?
    public boolean equals(Object y)
    // all neighboring boards
    public Iterable<Board> neighbors()
    // unit testing (required)
    public static void main(String[] args)
}
```

Merges. You should implement a private method `mergeTiles()` to handle all merges. For up and down swipes, this method should iterate through every *column* and find adjacent tiles that have the same value. For left and right swipes, this method should iterate through every *row* and find adjacent tiles that have the same value. *If there are three adjacent tiles with the same value, you must combine the two tiles closest to the border in the direction of motion. An example is shown below.*

```
0  0  0  0                    0  0  0  0
0  2  2  2      swipe left    4  2  0  0
0  0  0  0    ——————>         0  0  0  0
4  4  4  0                    8  4  0  0
```

```
0  0  0  0                    0  0  0  0
0  2  2  2      swipe right   0  0  2  4
0  0  0  0    ——————>         0  0  0  0
4  4  4  0                    0  0  4  8
```

Swipes. You should also implement a private `swipeTiles()` method which, after performing any merges, shifts the numbered tiles in the direction of the swipe. You may also think about it as shifting the blank tiles (zero tiles) in the opposite direction of the swipe. However, if there are no blank tiles in the given row or column, do not modify it. One idea is to maintain a pointer that moves in the opposite direction of the swipe. In this method, you may save the position of the first zero the pointer comes across and swap that zero with the first numbered tile seen by the pointer. Then, reset the pointer to the position after the swapped numbered tile. *Importantly, you must not change the order of the non-zero tiles. Therefore, simply sorting the tiles will not work.*

```
2  0  4  0                    0  0  2  4
4  2  0  0      swipe right   0  0  4  2
0  0  0  0    ——————>         0  0  0  0
2  8  4  2                    2  8  4  2
```

Adding tiles. Lastly, you should implement a private `addTile()` method which adds a 2 in the upper right corner. This method should be called once *after* each swipe is completed. If there is no room for the tile to be added—if the position in the upper right corner is occupied—update gameFinished to true.

Constructor. You may assume that the constructor receives a L-by-L array containing numbered tiles of powers of 2, and at least one blank tile represented by 0. Though 2048 is played on a board where $L = 4$, you should *not* hardwire this value into the program. You may assume $2 \le L \le 32{,}768$.

String representation. The `toString()` method returns a string composed of $L + 1$ lines. The first line contains the number of elements n; the remaining L lines contains the L-by-L grid of tiles in row-major order, using 0 to designate blank tiles.

Tile extraction. Throw a `java.lang.IllegalArgumentException` in `tileAt()` unless both `row` and `col` are between 0 and $L - 1$.

Comparing two boards for equality. Two boards are equal if they are have the same size and number elements and their corresponding tiles are in the same positions. The `equals()` method is inherited from `java.lang.Object`, so it must obey all of Java's requirements.

Neighboring boards. The `neighbors()` method returns an iterable containing the neighbors of the board. A board can have 1, 2, 3, or 4 neighbors, depending on the number of valid moves a player has. A move is considered valid if it changes the state of the board, as in it changes the location of one or more tiles or a merge is performed. You may find it helpful to add a private method to determine if swiping in a given direction is a valid move. For example, in the board shown below, swiping up, left, or right does not change the state. The only valid move is to swipe down, so only 1 neighbor exists.

16 8 4 2		16 8 4 2
0 0 0 0	*swipe up,*	0 0 0 0
0 0 0 0	*left, or right*	0 0 0 0
0 0 0 0	——————>	0 0 0 0
16 8 4 2		0 0 0 0
0 0 0 0	*swipe down*	0 0 0 0
0 0 0 0	——————>	0 0 0 0
0 0 0 0		16 8 4 2

Unit testing. Your `main()` method must call each public method directly and help verify that they works as prescribed (e.g., by printing results to standard output).

Performance requirements. In the worst case, your implementation must support `size()` and `tileAt()` in constant time. The constructor, `weightedScore()`, `snakeScore()`, `penalty()`, `hasWon()`, `gameFinished()`, `equals()`, `toString()`, and `neighbors()` must be supported in time proportional to L^2 (or better).

Reducing duplicate code. To promote maintainability, to shorten code, and to make bugs easier to fix, it may be helpful to create private helper methods—especially for the merge and swipe operations.

A* search. Now, we describe a solution to achieve the 128 tile 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 the goal board.
The efficacy of this approach hinges on the choice of *priority function* for a search node. We will examine the *weighted score*, *snake score*, and *penalty*, though we will only use the *weighted score* in the function.

- The *weighted score* is the product of the board multiplied by an array of weights. The weighted array should have a high number in the bottom left column, zero in the upper right column, and decrease by a constant factor diagonally. This way, higher numbers will naturally gravitate to the bottom left, away from the position where new tiles are added. An example is the following:

Weighted Array	*Board*
3 2 1 0	0 0 0 2
4 3 2 1	0 0 0 0
5 4 3 2	2 0 0 0
6 5 4 3	8 0 4 0

$$\sum W_{i,j} B_{i,j} = (2 \times 0) + (2 \times 5) + (4 \times 4) + (8 \times 6) = 10 + 16 + 40 = 66$$

- The *snake score* examines the tiles above, below, to the left, and to the right of the largest tile. The closer each tile is to the previous one in the given row or column, the higher the value that is added.
- The *penalty* is the sum of the differences between each tile and every one of its neighbors. A high penalty is bad—this happens when large numbered tiles are scattered around the grid.

Solver data type. In this part, you will implement A* search to solve *n*-by-*n* slider puzzles. Create an immutable data type `Solver` with the following API:

```
public class Solver {
    // find a solution to the initial board (using the A* algorithm)
    public Solver(Board initial)
    // min number of moves to solve initial board
    public int moves()
    // sequence of boards in a shortest solution
    public Iterable<Board> solution()
    // test client (see below)
    public static void main(String[] args)
}
```

Implementation requirement. To implement the A* algorithm, you must use the `MinPQ` data type for the priority queue. It may seem more intuitive to use MaxPQ, but this would violate A* search principles.

Corner case. Throw a `java.lang.IllegalArgumentException` in the constructor if the argument is `null`.

Test client. Your test client should take the name of an input file as a command-line argument and print the number of moves to solve the puzzle and a corresponding solution. The input file contains the board size *L*, followed by the *L*-by-*L* grid of tiles, using 0 to designate the blank tile.

Optimization. To speed up your solver, implement the following optimization:

- *Caching the largest tile and weighted score*. To avoid recomputing the largest tile and the weighted score of a board from scratch each time during various priority-queue operations, precompute their values in the `Board` constructor; save them in instance variables; and return the saved values 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.

Challenge for the bored. Implement a better priority function that reaches 128 in the minimum number of moves necessary.

Another challenge. Implement a better priority function that is capable of reaching values higher than 128, or prove mathematically that this cannot be accomplished with the current insert invariants.

Deliverables. Submit the files `Board.java` and `Solver.java`. We will supply `algs4.jar`. Your may not call any library functions other than those in `java.lang`, `java.util`, and `algs4.jar`. You must use `MinPQ` for the priority queue. Finally, submit a readme.txt file and answer the questions.

Written by Colton Wolk, adapted from 8-Puzzle. Special thanks to Maia Ginsburg.