Homework 4: 8 Puzzle

Getting the Skeleton Files

As usual, run `git pull skeleton master` to get the skeleton files.

Video Introduction

A video that I produced a couple of years ago for this assignment can be found at this link. Some notable differences for our semester:

- You do not have to write `Board.neighbors`.
- `Board.toString` is provided.
- You do not have to write `Board.isSolvable`.

Introduction

In this assignment, we'll be making our own puzzle solver! 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 tiles labeled 1 through 8 and a blank square. Your goal is to rearrange the tiles so that they are in order, using as few moves as possible. You are permitted to slide tiles 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
4   2  5   =>   4  2  5   =>   4     5   =>   4  5
7  8  6         7  8  6         7  8  6         7  8
```

Homework 4 Navigation

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 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 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 tiles 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 tiles 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 tiles 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
 4     2        4  5  6     ----------------------
 7  6  5        7  8        1  1  0  0  1  1  0  1

initial           goal          Hamming = 5 + 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 tile that is out of place must move at least once to reach its goal position. For Manhattan priority, this is true because each tile 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.)

Optimizations

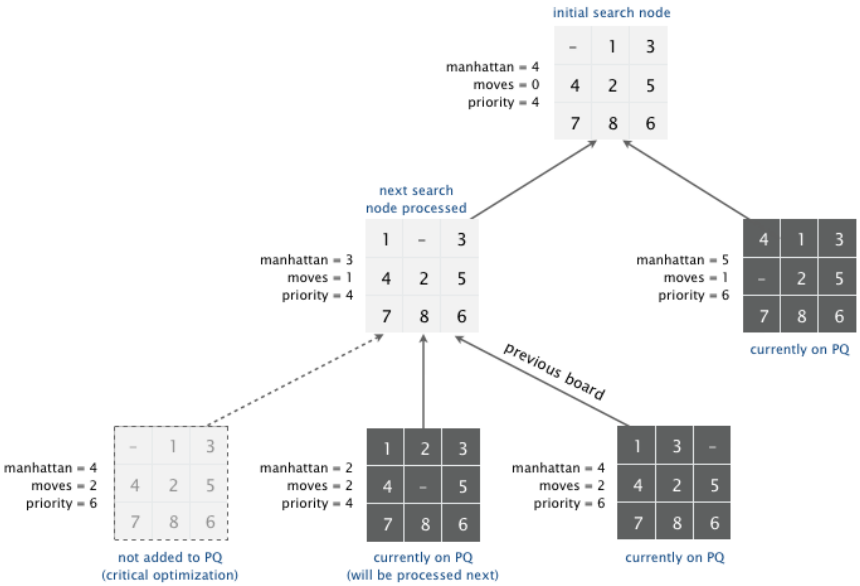
A critical optimization: Best-first search has one annoying feature: search nodes corresponding to the same board are enqueued on the priority queue many times. To reduce unnecessary exploration of useless search nodes, when considering the neighbors of a search node, don't enqueue a neighbor if its board is the same as the board of the previous search node.

A second optimization: To avoid recomputing the Manhattan distance of a board (or, alternatively, the Manhattan priority of a solver node) from scratch each time during various priority queue operations, compute it at most once per object; save its value 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

Homework 4 Navigation

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).



Board

Organize your program by creating an immutable Board class with the following API:

```
public class Board {
  public Board(int[][] tiles)
  public int tileAt(int i, int j)
  public int size()
  public int hamming()
  public int manhattan()
  public boolean isGoal()
  public boolean equals(Object y)
  public String toString()
}
```

Where the methods work as follows:

```
Board(tiles): Constructs a board from an N-by-N arr
              tiles[i][j] = tile at row i, column j
tileAt(i, j): Returns value of tile at row i, colum
size():       Returns the board size N
hamming():    Hamming priority function defined abo
```

```
equals(y):    Returns true if this board's tile val
              position as y's
toString():   Returns the string representation of
              method is provided in the skeleton
```

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. The tileAt() method should throw a `java.lang.IndexOutOfBoundsException` unless both i and j are between 0 and N − 1.

Performance requirements: Your implementation should support all Board methods in time proportional to N^2 (or faster) in the worst case.

Solver

Before moving on, note that you are provided a **BoardUtils.class** file, which supports a `public static Iterable<Board> neighbors(Board b)` method. You may find this method useful for this part.

Create an *immutable* Solver class with the following API:

```
public class Solver {
    public Solver(Board initial)
    public int moves()
    public Iterable<Board> solution()
}
```

Where the methods work as follows:

```
Solver(initial): Constructor which solves the puzzl
                 everything necessary for moves() a
                 not have to solve the problem agai
                 puzzle using the A* algorithm. Ass
moves():         Returns the minimum number of move
                 initial board
solution():      Returns the sequence of Boards fro
                 to the solution.
```

To implement the A* algorithm, **you must use the MinPQ class from edu.princeton.cs.algs4 for the priority queue**.

Solver.

Hint: Recall the search node concept mentioned above for using your PQ.

Solver Test Client

We've provided some basic code in **Solver.java** for you to test your solver against an input file. **Do not modify** this method. Puzzle input files are provided in the *input* folder.

The input and output format for a board is the board size N followed by the N-by-N initial board, using 0 to represent the blank square. An example of an input file for N = 3 would look something like this:

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

Your program should work correctly for arbitrary N-by-N boards (for any 1 < N < 32768), even if it is too slow to solve some of them in a reasonable amount of time. Note N > 1.

To test against input, run the following command from the hw4 directory after compiling:

```
java hw4.puzzle.Solver [input file]
```

So, if I tested against an input file `input/test01.in` with the following contents:

```
2
1  2
0  3
```

I should get the following output:

```
$ java hw4.puzzle.Solver input/test01.in
Minimum number of moves = 1
2
1  2
0  3

2
```

Homework 4 Navigation

Submission

Submit a zip file containing just the folder for your hw4 package (similar to hw3). It should contain **Board.java**, and **Solver.java**. Due to technical limitations of this autograder, it should contain no other .java files. If you have auxiliary java files (e.g. SearchNode.java), please move these classes into `Board` or `Solver`. It's OK if you also include `BoardUtils.class`.

FAQ

The autograder is complaining that graderhw4.Board objects can't be converted to Board or something like that.

The first step of the AG swaps out any usage of `Board` with `graderhw4.Board` in your `Solver.java`. However, it is not smart enough to find other classes (yet). For now, move your `SearchNode.java` class inside of `Solver.java`.

Likewise, if you have style errors, it will also fail. For example, if your code says

```
`neighbors=BoardUtils.`
```

instead of

```
`neighbors = BoardUtils`
```

My string matching code will fail.

Why am I getting cannot resolve symbol 'BoardUtils'?

You are probably compiling from the wrong folder. Compile from the "login/hw4" directory, not "login/hw4/hw4/puzzle".

Homework 4 Navigation

What if I'm using Intellij?

File -> Project Structure -> Libraries -> (+) sign to add new Java Library -> Select your **login/hw4** directory **DO NOT USE** login/hw4/hw4/puzzle -> OK -> OK -> OK.

These are the steps needed for Macs. I suspect there won't be big differences for other operating systems.

Is BoardUtils.neighbors working? It looks like it only returns the initial board.

It works, but it does depend on the board being immutable.

How do I know if my Solver is optimal?

The shortest solution to puzzle4x4-hard1.txt and puzzle4x4-hard2.txt are 38 and 47, respectively. The shortest solution to "puzzle*[T].txt" requires exactly T moves. Warning: *puzzle36.txt*, *puzzle47.txt*, and *puzzle49.txt*, and *puzzle50.txt* are relatively difficult.

I run out of memory when running some of the large sample puzzles. What should I do?

You should expect to run out of memory when using the Hamming priority function. Be sure not to put the JVM option in the wrong spot or it will be treated as a command-line argument, e.g.

```
java -Xmx1600m hw4.puzzle.Solver input/puzzle36.txt
```

My program is too slow to solve some of the large sample puzzles, even if given a huge amount of memory. Is this OK?

You should not expect to solve many of the larger puzzles with the Hamming priority function. However, you should be able to solve most (but not all) of the larger puzzles with the Manhattan priority function.

corresponding to the same board. Should I try to eliminate these?

In principle, you could do so with a set data type such as java.util.TreeSet or java.util.HashSet (provided that the Board data type were either Comparable or had a hashCode() method). However, according to Kevin Wayne at Princeton, almost all of the benefit from avoiding duplicate boards is already extracted from the critical optimization and the cost of identifying other duplicate boards will be more than the remaining benefit from doing so. In short, you're spending tremendous amounts of memory for a relatively small runtime optimization.

Is it OK if I try to eliminate them anyway by creating a big set of all the Boards ever seen?

Maybe. Make sure your code is able to complete the puzzles below when given only 128 Megabytes of memory (see below for how to test).

What size puzzles are we expected to solve?

Here are the puzzles you are explicitly expected to solve:

```
input/puzzle2x2-[00-06].txt
input/puzzle3x3-[00-30].txt
input/puzzle4x4-[00-30].txt
input/puzzle[00-31].txt
```

The puzzles work fine on my computer, but not on the AG. I'm getting a GC overhead limit exceeded error, or just a message that the "The autograder failed to execute correctly."

Your computer is probably more powerful than the autograder. Notably, the AG has much less memory. You should be able to complete puzzles 30 and 31 in less than a second, and they should also work if you use only 128 megabytes of memory. To run your code with only 128

command.

```
java -Xmx128M hw4.puzzle.Solver ./input/puzzle30.tx
java -Xmx128M hw4.puzzle.Solver ./input/puzzle31.tx
java -Xmx128M hw4.puzzle.Solver ./input/puzzle4x4-3
```

If your code is taking longer, by far the **most likely issue is that you are not implementing the first critical optimization properly**. Another possiblity is that you are creating a hash table of every board ever seen, which may cause the AG computer to run out of memory.

It is not enough to simply look at your code for the optimization and declare that it is correct. Many students have indicated confidence in their optimization implementation, only to discover a subtle bug. Use print statements or the debugger to ensure that a board never enqueues the board it came from.

Situations that cover 98% of student performance bugs:

- Recall that there is a difference between `==` and `equals`.
- Recall also that the optimization is that you should not "enqueue a neighbor if its board is the same as the board of the **previous** search node". Checking vs. the current board does nothing. In other words, no Node should ever enqueue its parent.
- Recall that the optimization is that a board should not enqueue its own parent! This is different than checking that it is different from the board that was dequeued two iterations of A* ago.

How do I ensure my Board class immutable?

The most common situation where a Board is not immutable is as follows:

- Step 1: Create a 2D array called cowmoo.

constructor.

- Step 3: Change one or more values of cowmoo.

If you just copy the reference in the Board constructor, someone can change the state of your Board by changing the array. You should instead make a copy of the 2D array that is passed to your board constructor.

Why can't Gradescope compile my files even though I can compile them locally?

Due to the nature of the autograder, you cannot use any public Board and Solver methods that were not mentioned in the spec. Consider moving the logic into one file.

The AG is reporting a bug involving access$ or some kind of null pointer exception. What's going on?

It's important that your `moves` and `solutions` methods work no matter the order in which they are called, and no matter how many times they are called. Failing the mutability test, or failing only `moves` but not `solutions` tests are sure signs of this issue.

Credits

This assignment originally developed by Kevin Wayne and Bob Sedgewick at Princeton University.