Programming Assignment 4 Checklist: 8 Puzzle

Frequently Asked Questions

Can I use different class names, method names, or method signatures from those prescribed in the API? No, as usual, your assignment will not be graded if it violates the API.

Is 0 a block? No, 0 represents the blank square. Do not treat it as a block when computing either the Hamming or Manhattan priority functions.

Can I assume that the puzzle inputs (arguments to the Board constructor and input to Solver) are valid? Yes, though it never hurts to include some basic error checking.

Do I have to implement my own stack, queue, and priority queue? You must use either `MinPQ` or `MaxPQ` for your priority queue (because we will intercept calls in order to do performance analysis). For the other data types, you may use versions from either `algs4.jar` or `java.util`.

How do I return an `Iterable<Board>`? Add the items you want to a `Stack<Board>` or `Queue<Board>` and return that. Of course, your client code should not depend on whether the iterable returned is a stack or queue (because it could be some any iterable).

How do I implement `equals()`? Java has some arcane rules for implementing `equals()`, discussed on p. 103 of Algorithms, 4th edition. Note that the argument to `equals()` is required to be `Object`. You can also inspect [Date.java] or [Transaction.java] for online examples.

Must I implement the `toString()` method for `Board` exactly as specified? Yes. Be sure to include the board dimension and use 0 for the blank square. Use `String.format()` to format strings—it works like `StdOut.printf()`, but returns the string instead of printing it to standard output. For reference, our implementation is below, but yours may vary depending on your choice of instance variables.

```
public String toString() {
    StringBuilder s = new StringBuilder();
    s.append(n + "\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            s.append(String.format("%2d ", tiles[i][j]));
        }
        s.append("\n");
    }
    return s.toString();
}
```

Should the `hamming()` and `manhattan()` methods in `Board` return the Hamming and Manhattan priority functions, respectively? No, `hamming()` should return the number of blocks out of position and `manhattan()` should return the sum of the Manhattan distances between the blocks and their goal positions. Recall that the blank square is not considered a block. You will compute the priority function in `Solver` by calling `hamming()` or `manhattan()` and adding to it the number of moves.

I'm a bit confused about the purpose of the twin() method. You will use it to determine whether a puzzle is solvable: exactly one of a board and its twin are solvable. A twin is obtained by swapping any pair of blocks (the blank square is not a block). For example, here is a board and several possible twins. Your solver will use only one twin.

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

  board      twin       twin       twin       twin       twin
```

How do I reconstruct the solution once I've dequeued the goal search node? Since each search node records the previous search node to get there, you can chase the pointers all the way back to the initial search node (and consider them in reverse order).

Can I terminate the search as soon as a goal search node is enqueued (instead of dequeued)? No, even though it does lead to a correct solution for the slider puzzle problem using the Hamming and Manhattan priority functions, it's not technically the A* algorithm (and will not find the correct solution for other problems and other priority functions).

I noticed that the priorities of the search nodes dequeued from the priority queue never decrease. Is this a property of the A* algorithm? Yes. In the language of the A* algorithm, the Hamming and Manhattan distances (before adding in the number of moves so far) are known as *heuristics*. If a heuristic is both *admissible* (never overestimates the number of moves to the goal search node) and *consistent* (satisfies a certain triangle inequality), then this noticed property is guaranteed. The Hamming and Manhattan heuristics are both admissible and consistent. You may use this noticed property as a debugging clue: if the priority of the search node dequeued from the priority queue decreases, then you know you did something wrong.

Even with the critical optimization, the priority queue may contain two or more search nodes corresponding to the same board. Should I try to eliminate these? In principle, you could do so with a set data type such as `SET` in `algs4.jar` or `java.util.TreeSet` or `java.util.HashSet` (provided that the `Board` data type were either `Comparable` or had a `hashCode()` method). However, 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.

Can I put the logic for detecting whether a puzzle is infeasible in Board instead of Solver? There is a elegant algorithm for determining whether a board is solvable that relies on a parity argument (and occasionally we change our API to require this solution). However, the current API requires you to detect infeasiblity in `Solver` by using two synchronized A* searches (e.g., using two priority queues).

I run out of memory when running some of the large sample puzzles. What should I do? Be sure to ask Java for additional memory, e.g., `java -Xmx1600m Solver puzzle36.txt`. We recommend running from the command line (and not from the DrJava interaction pane). 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 Solver -Xmx1600m puzzle36.txt`.

My program can't solve some of the 4-by-4 puzzles, even if I give it a huge amount of space. What am I doing wrong? Probably nothing. The A* algorithm (with the Manhattan priority function) will struggle to solve even some 4-by-4 instances.

Testing

Input files. The directory 8puzzle contains many sample puzzle input files. For convenience, 8puzzle-testing.zip contains all of these files bundled together.

- The shortest solution to `puzzle[T].txt` requires exactly *T* moves.
- The shortest solution to `puzzle4x4-hard1.txt` and `puzzle4x4-hard2.txt` are 38 and 47, respectively.
- Warning: `puzzle36.txt` is especially difficult.

Test client. A good way to automatically run your program on our sample puzzles is to use the client PuzzleChecker.java.

Priority queue trace.

- Here are the contents of our priority queue (sorted by priority) just before dequeueing each node when using the Manhattan priority function on `puzzle04.txt`.

```
Step 0:    priority  = 4
           moves     = 0
           manhattan = 4
           3
            0  1  3
            4  2  5
            7  8  6

Step 1:    priority  = 4    priority  = 6
           moves     = 1    moves     = 1
           manhattan = 3    manhattan = 5
           3                3
            1  0  3          4  1  3
            4  2  5          0  2  5
            7  8  6          7  8  6

Step 2:    priority  = 4    priority  = 6    priority  = 6
           moves     = 2    moves     = 1    moves     = 2
           manhattan = 2    manhattan = 5    manhattan = 4
           3                3                3
            1  2  3          4  1  3          1  3  0
            4  0  5          0  2  5          4  2  5
            7  8  6          7  8  6          7  8  6

Step 3:    priority  = 4    priority  = 6    priority  = 6    priority  = 6    priority  = 6
           moves     = 3    moves     = 3    moves     = 2    moves     = 3    moves     = 1
           manhattan = 1    manhattan = 3    manhattan = 4    manhattan = 3    manhattan = 5
           3                3                3                3                3
            1  2  3          1  2  3          1  3  0          1  2  3          4  1  3
            4  5  0          4  8  5          4  2  5          0  4  5          0  2  5
```

```
                   7  8  6         7  0  6         7  8  6         7  8  6         7  8  6

      Step 4:    priority  = 4   priority  = 6   priority  = 6   priority  = 6   priority  = 6   priority  = 6
                 moves     = 4   moves     = 3   moves     = 4   moves     = 2   moves     = 3   moves     = 1
                 manhattan = 0   manhattan = 3   manhattan = 2   manhattan = 4   manhattan = 3   manhattan = 5
                 3               3               3               3               3               3
                  1  2  3         1  2  3         1  2  0         1  3  0         1  2  3         4  1  3
                  4  5  6         0  4  5         4  5  3         4  2  5         4  8  5         0  2  5
                  7  8  0         7  8  6         7  8  6         7  8  6         7  0  6         7  8  6
```

There were a total of 10 search nodes enqueued and 5 search nodes dequeued. In general, the number of search nodes enqueued and dequeued may vary slightly, depending the order in which the search nodes with equal priorities come off the priority queue, which depends on the order in which `neighbors()` returns the neighbors of a board. However, for this input, there are no such ties, so you should have exactly 10 search nodes enqueued and 5 search nodes dequeued.

- The contents of our priority queue (sorted by priority) just before dequeueing each node when using the Hamming priority function on `puzzle04.txt` turns out to be identical to the results above: for this input file, throughout the A* algorithm, a block is never more than one position away from its goal position, which implies that the Hamming function and the Manhattan functions are equal.

- Write the class `Solver` that uses the A* algorithm to solve puzzle instances.

Enrichment

How can I reduce the amount of memory a `Board` uses? For starters, recall that an n-by-n `int[][]` array in Java uses about 24 + 32n + 4n^2 bytes; when n equals 3, this is 156 bytes. To save memory, consider using an *n*-by-*n* `char[][]` array or a length *n*^2 `char[]` array. You could use a more elaborate representation: since each board is a permutation of length n^2, in principle, you need only about lg ((n^2)!) bits to represent it; when n equals 3, this is only 19 bits.

Any ways to speed up the algorithm? Yes there are many opportunities for optimization here.

- Use a 1d array instead of a 2d array (as suggested above).
- Cache either the Manhattan distance of a board (or Manhattan priority of a search node). It is waste to recompute the same quantity over and over again.
- Exploit the fact that the difference in Manhattan distance between a board and a neighbor is either −1 or +1.
- Use only one PQ to run the A* algorithm on the initial board and its twin.
- When two search nodes have the same Manhattan priority, you can break ties however you want, e.g., by comparing either the Hamming or Manhattan distances of the two boards.
- Use a parity argument to determine whether a puzzle is unsolvable (instead of two synchronous A* searches). However, this will either break the API or will require a fragile dependence on the `toString()` method, so don't do it.

Why are the boards divided into exactly two equivalence classes with respect to reachability? Here is one proof by Aaron Archer.

Is there an efficient way to solve the 8-puzzle and its generalizations? Finding a shortest solution to an *n*-by-*n* slider puzzle is NP-hard, so it's unlikely that an efficient solution exists.

What if I'm satisfied with any solution and don't need one that uses the fewest number of moves? Yes, change the priority function to put more weight on the Manhattan distance, e.g., 100 times the Manhattan distance plus the number of moves made already. This paper describes an algorithm that guarantees to perform at most N^3 moves.

Are there better ways to solve 8- and 15-puzzle instances using the minimum number of moves? Yes, there are a number of approaches.

- Use the A* algorithm with a better admissible priority function:
  - *Linear conflict*: add two to the Manhattan priority function whenever two tiles are in their goal row (or column) but are reversed relative to their goal position.
  - *Pattern database*: For each possible configuration of 4 tiles and the blank, determine the minimum number of moves to put just these tiles in their proper position and store these values in a database. The heuristic value is the maximum over

all configurations, plus the number of moves made so far. This can reduce the number of search nodes examined for random 15-puzzle instances by a factor of 1000.

- Use a variant of the A* algorithm known as IDA* (for iterative deepening). [This paper] describes its application to the 15-slider puzzle.
- Another approach is to use [bidirectional search], where you simultaneously search from the initial board to find the goal board and from the goal board to find the initial board, and have the two search trees meet in the middle. Handling the stopping condition is delicate.

Can a puzzle have more than one shortest solution? Yes. See `puzzle07.txt`.

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

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

In such cases, you are required to output any one such solution.