COS 226 Programming Assignment 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, you will receive a serious deduction for violating the API.

Is 0 a tile? No, 0 represents the absence of a tile. Do not treat it as a tile 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? No, use the versions in algs4.jar.

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`. For online examples, see Date.java or Transaction.java.

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.

```
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 ", tileAt(i, j)));
        }
        s.append("\n");
    }
    return s.toString();
}
```

Should the `hamming()` and `manhattan()` methods in `Board` return the Hamming and Manhattan priority functions? No, you should return the sum of the Hamming and Manhattan distances from the tiles to their goal positions. Also, recall that the blank square is not considered a tile. You will compute the priority function in `Solver` by calling `hamming()` or `manhattan()` and adding to it the number of moves.

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 property is guaranteed. The Hamming and Manhattan heuristics are both admissible and consistent. You may use this property as a debugging clue: if it is ever violated, 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` (provided that the `Board` data type were `Comparable`). However, almost all of the benefit from avoiding duplicate boards is already extracted from the critical optimization and the cost to identify other duplicate boards exceeds the benefit from doing so.

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-algs4 -Xmx1600m PuzzleChecker puzzle36.txt`. If your program is unable to solve certain instances, document that in your `readme.txt` file. 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-algs4 PuzzleChecker -Xmx1600m 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. Also, be sure to *execute from the command line (and not DrJava)*.

Testing

Input files. The ftp directory contains many sample puzzles. 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.

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

Visualization client. You can also use SolverVisualizer.java, which takes the name of a puzzle file as a command-line argument and animates the solution.

Sample trace. The program defines two different data structures on the set of search nodes —the *game tree* and the *priority queue*. Below is a detailed trace of each data structure during the solution to `puzzle04.txt`.

- The game tree represents relationships obtained by executing valid moves in the slider game. If a search node *X* is the parent of another search node *Y*, then the boards in *X* and *Y* are neighbors (and one move apart). If the board in a search node equals the board in its grandparent, it is rejected by the critical optimization and its subtree is not explored; we assign a lowercase letter as the ID for such a search node. If the board in a search node equals the goal board, we denote it by brackets in the game tree. The path from the root to such a node provides a sequence of moves to solve the puzzle.

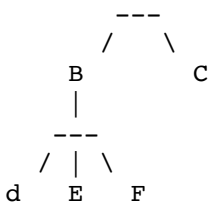
- We also show the contents of the priority queue (assumed to be a binary heap) after each priority queue operation. We use the symbol `'-'` to denote an empty entry in an array.

For brevity, we assign each search node a single–letter ID. The following table provides the relevant information for each search node (the ID, the board, its Manhattan distance from the goal board, the number of moves to reach the search node, the Manhattan priority function, and the previous search node).

```
ID  Board   Priority function     Parent in game tree
================================================================================

A   0 1 3   Manhattan:        4   Previous: null
    4 2 5   Moves:            0
    7 8 6   Priority: 4 + 0 = 4

B   1 0 3   Manhattan:        3   Previous: A
    4 2 5   Moves:            1
    7 8 6   Priority: 3 + 1 = 4

C   4 1 3   Manhattan:        5   Previous: A
    0 2 5   Moves:            1
    7 8 6   Priority: 5 + 1 = 6

d   0 1 3   Manhattan:        4   Previous: B
    4 2 5   Moves:            2
    7 8 6   Priority: 4 + 2 = 6

E   1 2 3   Manhattan:        2   Previous: B
    4 0 5   Moves:            2
    7 8 6   Priority: 2 + 2 = 4

F   1 3 0   Manhattan:        4   Previous: B
    4 2 5   Moves:            2
    7 8 6   Priority: 4 + 2 = 6

g   1 0 3   Manhattan:        3   Previous: E
    4 2 5   Moves:            3
    7 8 6   Priority: 3 + 3 = 6

H   1 2 3   Manhattan:        3   Previous: E
    0 4 5   Moves:            3
    7 8 6   Priority: 3 + 3 = 6

I   1 2 3   Manhattan:        3   Previous: E
    4 8 5   Moves:            3
    7 0 6   Priority: 3 + 3 = 6

J   1 2 3   Manhattan:        1   Previous: E
    4 5 0   Moves:            3
    7 8 6   Priority  1 + 3 = 4

K   1 2 0   Manhattan:        2   Previous: J
    4 5 3   Moves             4
    7 8 6   Priority  2 + 4 = 6
```

```
l   1 2 3   Manhattan:        2   Previous: J
    4 0 5   Moves:            4
    7 8 6   Priority: 2 + 4 = 6

M   1 2 3   Manhattan:        0   Previous: J
    4 5 6   Moves:            4
    7 8 0   Priority: 0 + 4 = 4
```

```
################################################################################
Step 0
================================================================================

Game Tree
--------------------------------------------------------------------------------

                                    A

Priority Queue
--------------------------------------------------------------------------------

pq = new MinPQ();   0 1 2 3 4 5 6 7 8 9
                    - - - - - - - - - -

pq.insert(A);       0 1 2 3 4 5 6 7 8 9
                    - A - - - - - - - -


################################################################################

Step 1
================================================================================

Game Tree
--------------------------------------------------------------------------------

                                    A
                                    |
                                   ---
                                  /   \
                                 B     C

Priority Queue
--------------------------------------------------------------------------------

pq.delMin();        0 1 2 3 4 5 6 7 8 9
// returns A        - - - - - - - - - -

pq.insert(B);       0 1 2 3 4 5 6 7 8 9
                    - B - - - - - - - -

pq.insert(C);       0 1 2 3 4 5 6 7 8 9
                    - B C - - - - - - -


################################################################################

Step 2
================================================================================

Game Tree
--------------------------------------------------------------------------------

                                    A
                                    |
```



Priority Queue

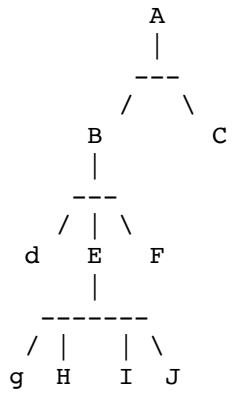
```
pq.delMin();        0 1 2 3 4 5 6 7 8 9
// returns B        - C - - - - - - - -

pq.insert(E);       0 1 2 3 4 5 6 7 8 9
                    - E C - - - - - - -

pq.insert(F);       0 1 2 3 4 5 6 7 8 9
                    - E C F - - - - - -
```

##########

Step 3

==========

Game Tree



Priority Queue

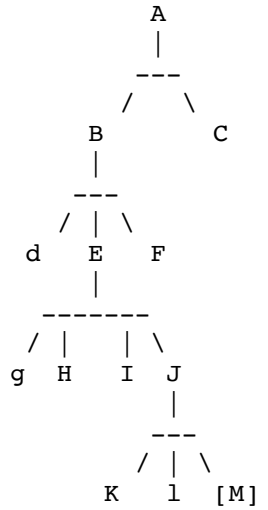
```
pq.delMin();        0 1 2 3 4 5 6 7 8 9
// returns E        - F C - - - - - - -

pq.insert(H);       0 1 2 3 4 5 6 7 8 9
                    - F C H - - - - - -

pq.insert(I);       0 1 2 3 4 5 6 7 8 9
                    - F C H I - - - - -

pq.insert(J);       0 1 2 3 4 5 6 7 8 9
                    - J F H I C - - - -
```

##########

Step 4

==========

Game Tree



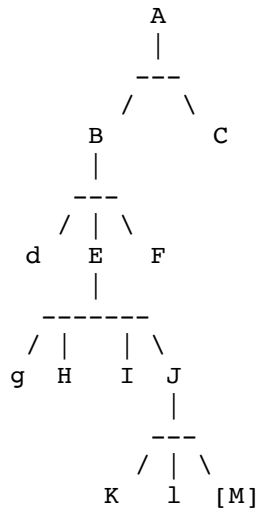
Priority Queue

```
pq.delMin();        0 1 2 3 4 5 6 7 8 9
// returns J        - C F H I - - - - -

pq.insert(K);       0 1 2 3 4 5 6 7 8 9
                    - C F H I K - - - -

pq.insert(M);       0 1 2 3 4 5 6 7 8 9
                    - M F C I K H - - -
```

##########

Step 5

Game Tree



Priority Queue

```
pq.delMin();        0 1 2 3 4 5 6 7 8 9
// returns M        - H F C I K - - - -
```

M corresponds to a goal state, return path from root to leaf: A -> B -> E -> J -> M

Possible Progress Steps

These are purely suggestions for how you might make progress. You do not have to follow these steps.

- Download the directory `8puzzle` to your system. It contains a number of sample input files.
- Write the data type `Board` that represents an *N*–by–*N* puzzle board. Be sure to thoroughly test and debug it before proceeding.
- Write a class `SearchNode` that represents a search node of the game (such as the board, the number of moves to reach it, and the previous search node). You can either make this a nested class within `Solver` or make it a stand–alone class. Be sure to include a constructor which initializes the value of each instance variable.
- Write the class `Solver` that uses the A* algorithm to solve puzzle instances. This will include creating a `MinPQ`. You can choose one of two options to determine order:
  - Make it implement the `Comparable<SearchNode>` interface so that you can use it with a `MinPQ`. The `compareTo()` method should compare search nodes based on their Hamming or Manhattan priorities.
  - Create a `Comparator<SearchNode>` for each priority function and initialize the `MinPQ` with it.

A video is provided for those wishing additional assistance. Be forewarned that video was made in early 2014 and is somewhat out of date. For example the API has changed.

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 + 32*N* + 4N^2 bytes; when *N* equals 3, this is 156 bytes. To save memory, consider using a 1D array of length N^2. In principle, each board is a permutation of size N^2, so 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? One useful trick is to *cache* the Manhattan distance of a board (or Manhattan priority of a search node). That is, maintain an extra instance variable; compute and initialize it upon construction (or first usage); afterwards, just return the cached value.

Is there an efficient way to solve the 8–puzzle and its generalizations? Finding a shortest solution to a 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
   7  6     7     6     7  4  6     7  4  6        4  6     4     6     4  5  6
5  4  8     5  4  8     5     8        5  8     7  5  8     7  5  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
   7  6     5  7  6     5  7  6     5     6        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
```

What's the maximum number of moves need to solve any 8–slider puzzle? Any 8–slider puzzle can be solved with at most 31 moves; any 15–slider puzzle can be solved with at most 80 moves.

What's the best known algorithm for determining whether a puzzle is solvable? N^2. There is a mergesort–style algorithm for counting the number of inversions of a permutation of size N in time proportional to N log N. You can also determine the parity of the number of inversions of a permutation in time proportional to N. We note that the best–known algorithm for counting the number of inversions of a permutation is N sqrt(log N)—see Counting Inversions, Offline Orthogonal Range Counting, and Related Problems.