Assignments

Assignment 3

Attached Files: checker.zip (23.341 KB)

8 Puzzle

Write a program to solve the 8-puzzle problem (and its natural generalizations) using the A* search algorithm.

The problem. 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 blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in order, using as few moves as possible. You are permitted to slide blocks 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  3        1  2  3
 4  2  5   =>   4  2  5   =>   4  *  5   =>   4  5  *   =>   4  5  6
 7  8  6        7  8  6        7  8  6        7  8  6        7  8  *

 initial        1 left          2 up          5 left          goal
```

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 an PQ. Define the Node class as a private inner-class of the Solver class (see below), where Node implements the Comparable interface. Note: Node is not generic! Then, delete from the PQ the search node with the minimum priority, and insert into the PQ of all neighboring search nodes (those that can be reached in one move from the removed search node). Repeat this procedure until the search node that has been removed 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 blocks 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 blocks 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 blocks 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    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    1  2  0  0  2  2  0  3

 initial          goal         Hamming = 5 + 0          Manhattan = 10 + 0
```

We make a key observation: To solve the puzzle from a given search node on the PQ, 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 block that is out of place must move at least once to reach its goal position. For Manhattan priority, this is true because each block 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 removed, 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.)

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

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

  previous    search node    neighbor    neighbor     neighbor
                                        (disallow)
```

Detecting infeasible puzzles. Not all initial boards can lead to the goal board such as the one below.

```
          1  2  3  4  5  6  8  7

               infeasible
```

To detect such situations, use the fact that boards are divided into two equivalence classes with respect to reachability: (i) those that lead to the goal board and (ii) those that cannot lead to the goal board. Moreover, we can identify in which equivalence class belongs by considering its *parity*.

- *Odd board size.* The parity of the number of inversions of the N^2 - 1 blocks.
-
- *Even board size.* The parity of the number of inversions of the N^2 - 1 blocks plus the row in which the blank square plus one.

Board and Solver data types. Organize your program by creating an immutable data type Board with the following API:

```
public class Board {
    public Board(int[][] blocks)           // construct a board from an N-by-N array of blocks
                                           // (where blocks[i][j] = block in row i, column j)

    public int dimension()                 // board dimension N

    public int hamming()                   // number of blocks out of place
    public int manhattan()                 // sum of Manhattan distances between blocks and goal

    public boolean isGoal()                // is this board the goal board?
    public boolean isSolvable()            // is the board solvable?

    public boolean equals(Object y)        // does this board equal y?

    public Iterable<Board> neighbors()     // place all neighboring boards into your iterable Queue (assignment 1)

    public String toString()               // string representation of the board (in the output format specified below)
}
```

and a data type Solver with the following API:

```
public class Solver {
    public Solver(Board initial)               // find a solution to the initial board (using the A* algorithm)

    public boolean isSolvable()                // is the initial board solvable?

    public int moves()                         // min number of moves to solve initial board

    public Iterable<Board> solution()          // sequence of boards in a shortest solution

    public static void main(String[] args){}  // solve a slider puzzle (given below)
}
```

Throw an java.lang.IllegalArgumentException if the initial board is not solvable. To implement the A* algorithm, you must use the PQ<T> data type for the priority queues.

Solver test client. Use the following test client to read a puzzle from a file (specified as a command-line argument) and print the solution to standard output.

```
public static void main(String[] args) {
                              // create initial board from file
                                In in = new In(args[0]);
                                int N = in.readInt();
                                int[][] blocks = new int[N][N];

                                       for (int i = 0; i < N; i++)
                                   for (int j = 0; j < N; j++)
                                     blocks[i][j] = in.readInt();

                                         Board initial = new Board(blocks);      // solve the puzzle
                                Solver solver = new Solver(initial);    // print solution to standard output

                                         if (!initial.isSolvable())
                                  System.out.println("No solution possible");
                                  else {
                                  System.out.println("Minimum number of moves = " + solver.moves());

                                          for (Board board : solver.solution())
                                   System.out.println(board);
                                }
                              }
```

Input and output formats. The input and output format for a board is the board dimension *N* followed by the *N*-by-*N* initial board, using 0 to represent the blank square. As an example,

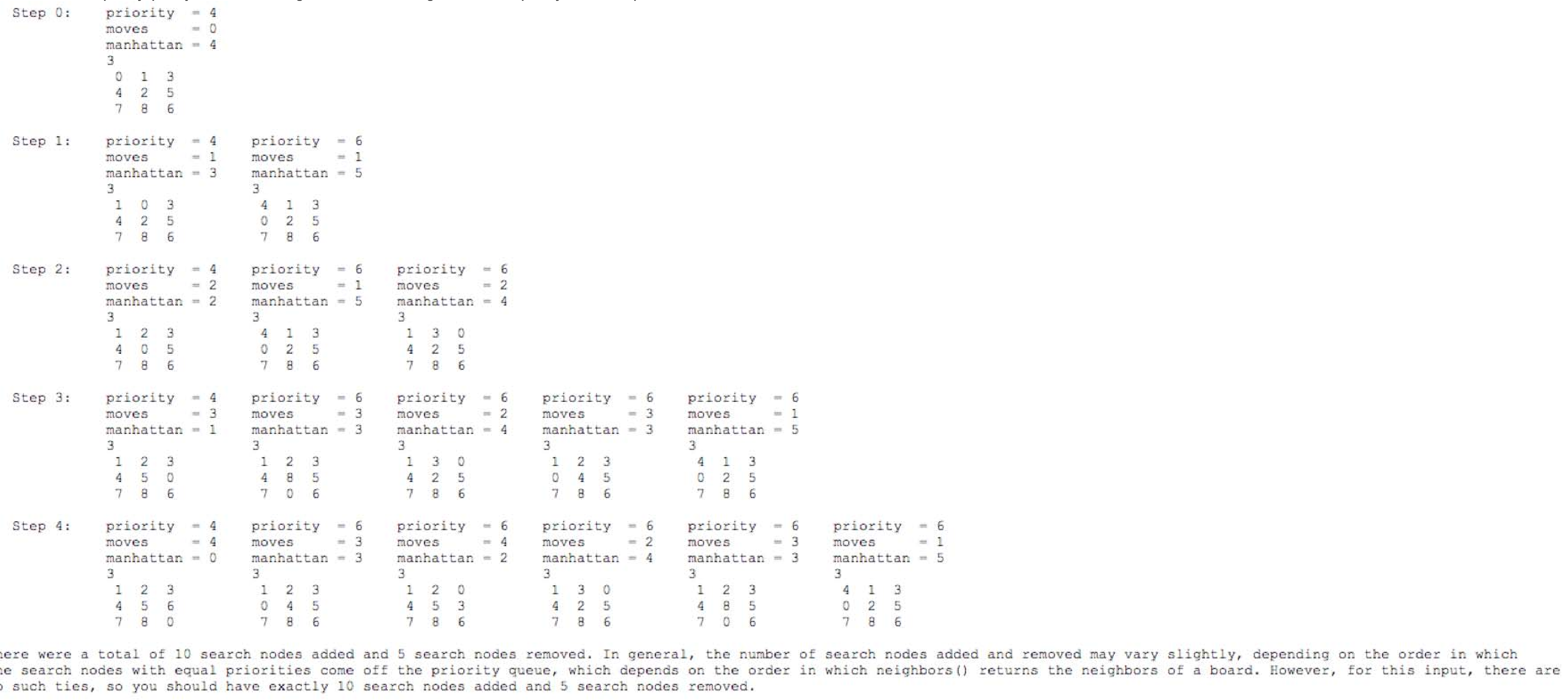
```
% more puzzle04.txt

3
 0  1  3
 4  2  5
 7  8  6

 % java Solver puzzle04.txt

 Minimum number of moves = 4

 3
  0  1  3
  4  2  5
  7  8  6

 3
  1  0  3
  4  2  5
  7  8  6

 3
  1  2  3
  4  0  5
  7  8  6

 3
  1  2  3
  4  5  0
  7  8  6

 3
  1  2  3
  4  5  6
  7  8  0


 %  more puzzle-unsolvable3x3.txt

 3
 1  2  3
 4  5  6
 8  7  0


 %  java Solver puzzle3x3-unsolvable.txt
 No solution possible
```

Your program should work correctly for arbitrary *N*-by-*N* boards (for any 1 ≤ *N* < 128), even if it is too slow to solve some of them in a reasonable amount of time.

Deliverables

Submit the files Board.java and Solver.java (with the Hamming priority). Your may not call any library functions other than those in java.lang or java.util. You must use the PQ<T> data type for the priority queue.

Here are the contents of the priority queue just before removing 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 added and 5 search nodes removed. In general, the number of search nodes added and removed may vary slightly, depending on 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 added and 5 search nodes removed.

Thank you Kevin Wayne!

Rubric:

public Board(int[][] tiles) - 20 points
public int hamming() - 15 points
public int manhattan() - 15 points
public Iterable<Board> neighbors() - 30 points
public String toString() - 10 points
public boolean equals(Object y) - 10 points
public boolean isGoal() - 2.5 points
public boolean isSolvable() - 2.5 points

public Solver(Board initial) - 20 points
public Iterable<Board> solution() - 20 points
main() - 10 points
correct ouput for Checker 45 points