

Artificial Intelligence

Assignment 1 - 8 Puzzle

Eugene Kenny

Objectives

To write a program to solve the 8 puzzle problem (and its natural generalisations) 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 position (left) to the goal position (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	
																		goal

A* Search

We describe a solution to the problem that illustrates a general artificial intelligence methodology known as the A* search algorithm. We define a state of the game to be a board position, the number of moves made to reach the board position, and the previous state. First, insert the initial state (the initial board, 0 moves, and a null previous state) into a priority queue. Then, delete from the priority queue the state with the minimum priority, and insert onto the priority queue all neighboring states (those that can be reached in one move). Repeat this procedure until the state dequeued is the goal state. The success of this approach hinges on the choice of priority function for a state. 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 state. Intuitively, a state with a small

number of blocks in the wrong position is close to the goal state, and we prefer a state 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 state.

For example, the Hamming and Manhattan priorities of the initial state below are 5 and 10, respectively.

8	1	3		1	2	3
4		2		4	5	6
7	6	5		7	8	
initial				goal		

1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	1	0	0	1	1	0	1	1	2	0	0	2	2	0	3
Hamming = 5 + 0								Manhattan = 10 + 0							

To solve the puzzle from a given state 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 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 tile when computing the Hamming or Manhattan priorities.) Consequently, as soon as we dequeue a state, we have not only discovered a sequence of moves from the initial board to the board associated with the state, but one that makes the fewest number of moves.

After implementing A*, you will notice one annoying feature: states corresponding to the same board position are enqueued on the priority queue many times. To reduce unnecessary exploration of useless states, when considering the neighbours of a state, don't enqueue a neighbour if its board position is the same as the previous state.

8	1	3	8	1	3	8	1	3
4		2	4	5		4		2
7	6	5	7	6	5	7	6	5
previous			state			disallow		

Your task Write a program Solver.java that reads the initial board from standard input and prints to standard output a sequence of board positions that solves the puzzle in the fewest number of moves. Also print out the total number of moves.

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

```

% more puzzle04.txt

0  1  3
4  2  5
7  8  6

% java Solver < puzzle04.txt
    1  3
4  2  5
7  8  6

1      3
4  2  5
7  8  6

1  2  3
4      5
7  8  6

1  2  3
4  5
7  8  6

1  2  3
4  5  6
7  8

Number of states enqueued = 10
Minimum number of moves = 4

```

Note that your program should work for arbitrary N-by-N boards (for any N greater than 1), even if it is too slow to solve some of them in a reasonable amount of time.

Board and Solver data types

Organize your program in an appropriate number of data types. At a minimum, you are required to implement the following APIs. You are permitted to add additional methods or data types, such as State.

```

public class Board {
    public Board(int[] [] tiles)           // construct a board from an N-by-N
                                           // array of tiles
    public int hamming()                   // return number of blocks out of place
    public int manhattan()                 // return sum of Manhattan distances between

```

```

// blocks and goal
public boolean equals(Object y) // does this board position equal y
public Iterable<Board> neighbors() // return an Iterable of all neighboring
// board positions
public String toString() // return a string representation of
//the board
}

public class Solver {
    public Solver(Board initial) // find a solution to the initial board
    public boolean isSolvable() // is the initial board solvable?
    public int moves() // return min number of moves to solve
// initial board; -1 if no solution
    public Iterable<Board> solution() // return an Iterable of board
// positions in solution
}

```

The following main() for Solver demonstrates the APIs: it reads a puzzle instance from standard input and prints the solution to standard output.

```

public static void main(String[] args) {
    int N = StdIn.readInt();
    int[][] tiles = new int[N][N];
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            tiles[i][j] = StdIn.readInt();
    Board initial = new Board(tiles);
    Solver solver = new Solver(initial);
    for (Board board : solver.solution())
        System.out.println(board);
    if (!solver.isSolvable()) System.out.println("No solution possible");
    else System.out.println("Minimum number of moves = " + solver.moves());
}

```

Deliverables. Submit Board.java, Solver.java (with the Manhattan priority) and any other helper data types that you use