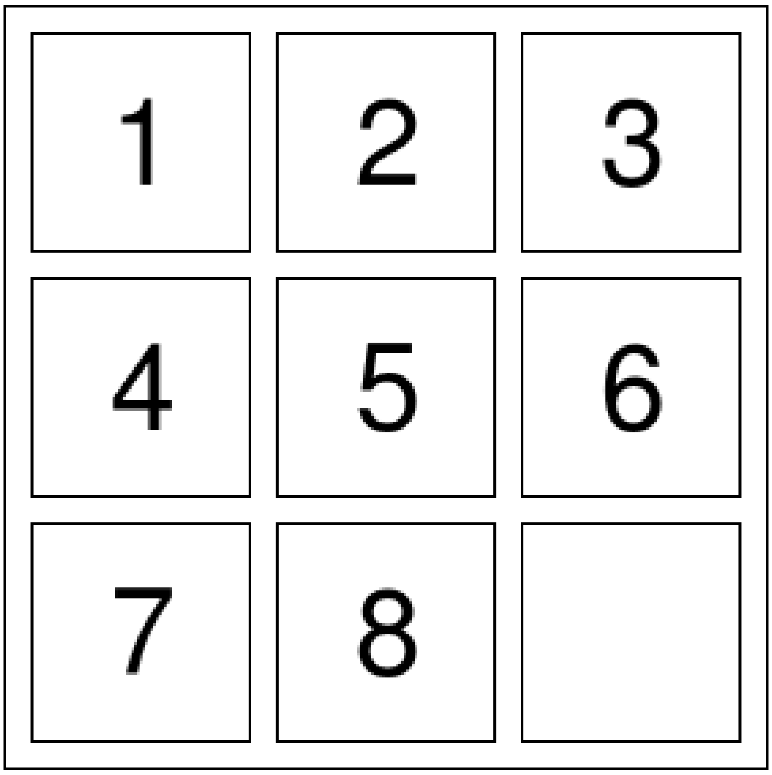
جامعة الملك سعود
King Saud University

The n-puzzle



Prepared by:

Reem alqabbani	437200871
Hanan alaskar	436200769

Second Semester 2018/2019

Introduction:

In our project we took n-puzzle arranged in a random ordering, solve the puzzle by A* algorithm to find the optimal way to the goal, step by step and with the shortest path, we used two heuristic algorithms Manhattan and hamming.

Problem formulation:

Problem formulation is considered as a pre-step for AI search algorithms. To formulate a problem, we need to identify a state representation, the initial state, the goal state, the actions that are possible to do to reach the goal state, and the cost of each action. So, we formulate the problem by identifying them.

> **States**: the states represented as 2-dimensional array of type Integer, while the index reflects the position of a specific tile. This example of 8-puzzle state representation: {[1, 2, 3]، [4, 5, 6]، [7, 8, 0]}
>
> **Initial state**: the initial state is taken from the input file.
>
> **Goal state**: the state goal is a state when the value of tiles is organized increasingly from 1 to n and the last tile equals 0. To illustrate, the goal state of the 3-puzzle is {[1, 2]، [3,0]}.
>
> **Actions**: the possible actions, in general, are moving the blank to left, right, up, and down. But, in some cases, the possible actions reduce to two actions only, to illustrate that in a situation where the blank is in the left-down corner position the possible movements are moving the blank up or right only.
>
> **Path cos**t: the cost equal g(n)+h(n) because the A* algorithm has been used to find the solution.

The evaluation function:

Since we are using A* algorithm to find the solution. A* consider a heuristic algorithm because it uses a heuristic function beside the cost function. In other words, the evaluation function in our program is the sum of cost function and heuristic function. While the cost function equals the total of movement from initial state to n state, and every movement costs 1 which is called g(n). For Heuristic function which calculates the cost from n state to the goal and called h(n). In our program we used Heuristic functions: Hamming distance, and Manhattan distance. Hamming distance calculates the total number of misplaced tiles. However, Manhattan distance calculates the sum of the distances of tiles from their current positions to their goal positions. We found that Manhattan Heuristic dominates Hamming Heuristic because the value of Manhattan Heuristic is greater than the value Hamming Heuristic and they both admissible heuristics.

Design:

We designed our project as 4 classes: Test ,Solver, State and Node. Here a brief about each class.

- **Node Class**

  We used the Node class which is considered as an n-tree data structure. This kind of data structure has the ability to track the child to its root, also n-tree allows the parent to have many children. So, these features help us to find the path from the goal state to the initial state as we save the states into this data structure.

- **State Class**

  The State class is a core class from this program. Insider this class we implement the functions and attributes related to this class.

- **Solver Class**

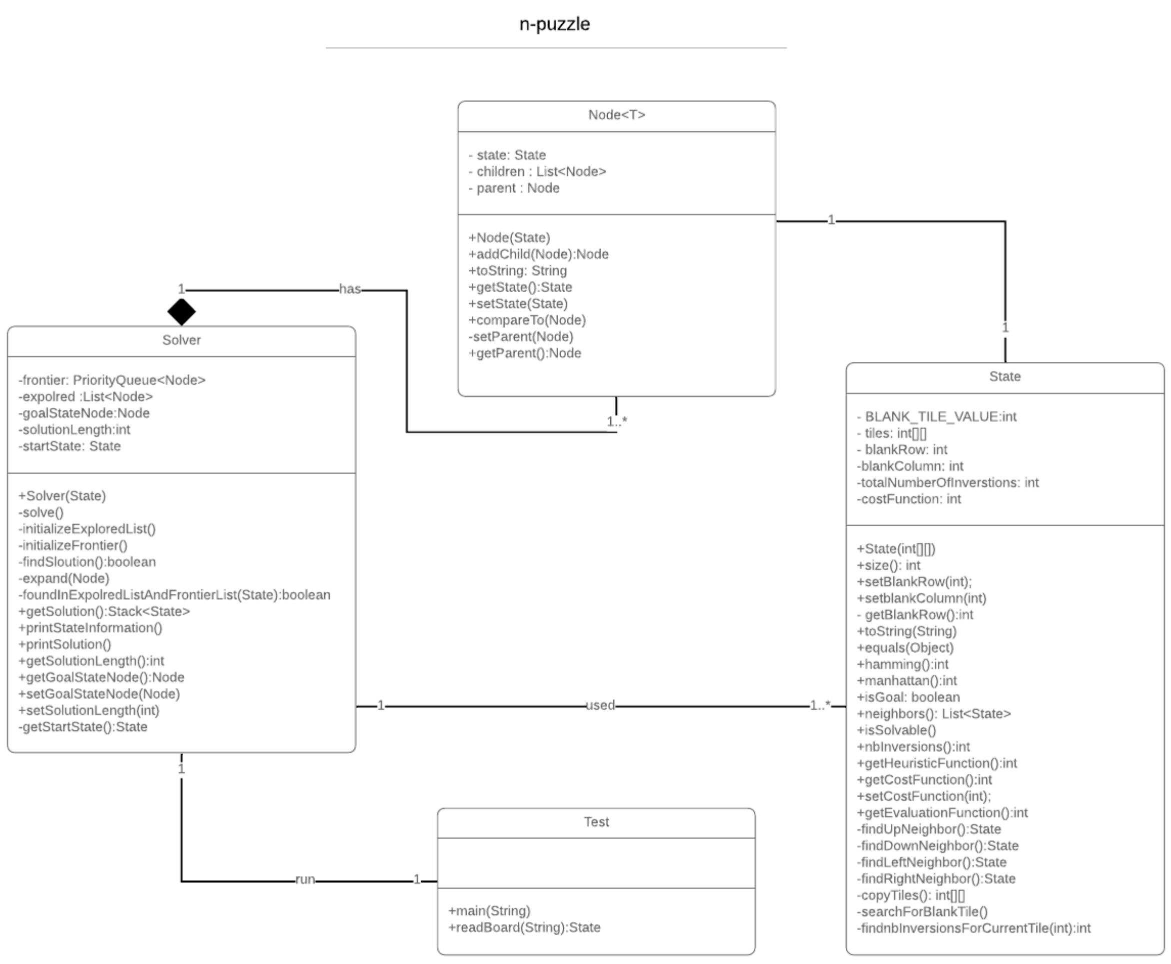
  Solver class has the same value as the State class. The main function of this class is to solve the puzzle that means finding the optimal path from the initial state to the goal state. To do that we used several methods and attributes.

- **Test Class**

  There are two main purposes for creating the Test class. The first one is reading the puzzle from text input and saving it into a state object. The second is creating an object of type Solver and calling several methods in class Solver to solve the puzzle and print information about it. The first purpose has been accomplished in the readBoard(String) method and the second in the main method.

In uml class diagram we represent the relationship between these classes:

UML Class Diagram:



Implementation:

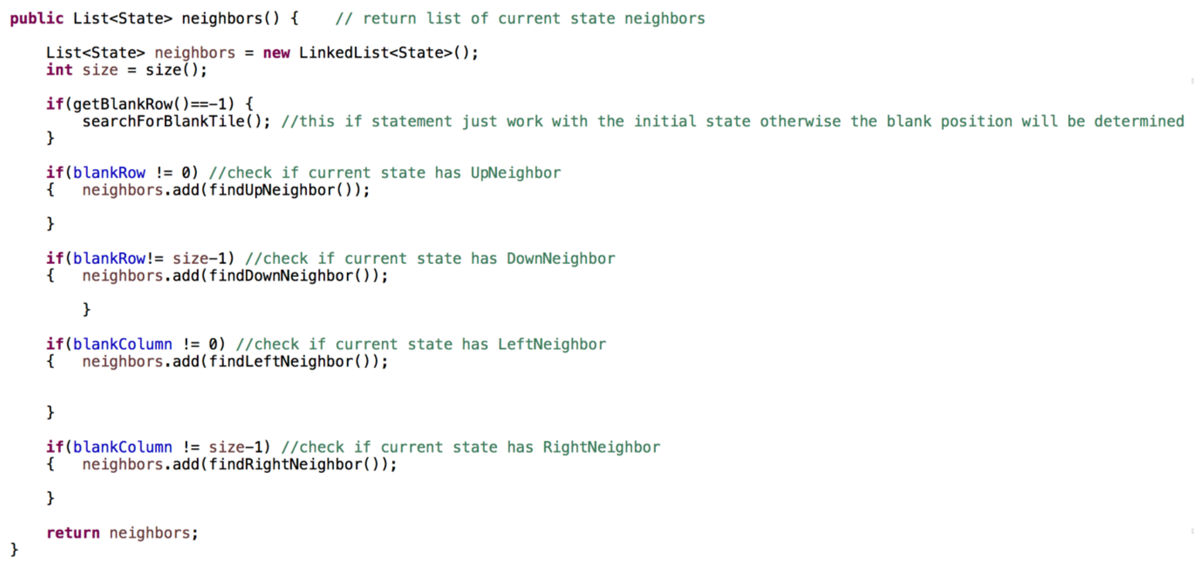
As you read in the design part of our program consists of 4 classes: Node, State, Solver, and Test. In the implementation part, we focus on the two main classes.

- **State Class**

  The main attribute in State class is “Tiles” which is a 2-dimensional array of type integers. we found the “Tiles” is the appropriate representation to the state. Also, State class has two attributes that are responsible for saving the position of blank tile, these attributes named blankRow and blankColumn both of type integer .

Also, one of the important functions in State class is neighbors() function which returns a list of possible neighbors for the current state. In other words, this method returns all the potential next states after doing the possible actions. In this method, we start by creating an empty list of class State, then check if the position of the blank tile in the current state is known, if not call searchForBlnkTile(). After that, the method checks the possibility of having neighbors in each direction before looking for a potential neighbor in a certain direction. The method returns a list that has all potential neighbors after finding them.

below the code of neighbors() method in State class



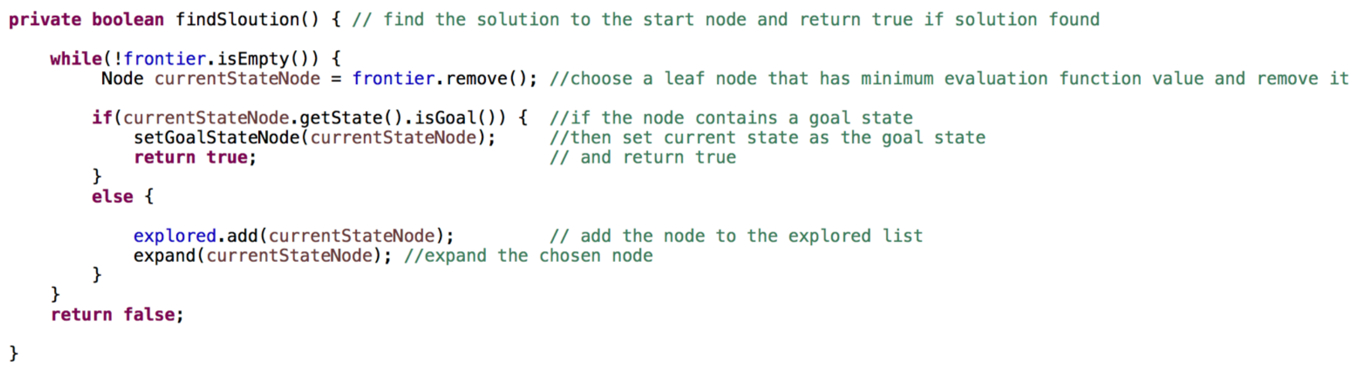
```
public List<State> neighbors() {    // return list of current state neighbors

    List<State> neighbors = new LinkedList<State>();
    int size = size();

    if(getBlankRow()==-1) {
        searchForBlankTile(); //this if statement just work with the initial state otherwise the blank position will be determined
    }

    if(blankRow != 0) //check if current state has UpNeighbor
    {   neighbors.add(findUpNeighbor());

    }

    if(blankRow!= size-1) //check if current state has DownNeighbor
    {   neighbors.add(findDownNeighbor());

        }

    if(blankColumn != 0) //check if current state has LeftNeighbor
    {   neighbors.add(findLeftNeighbor());


    }

    if(blankColumn != size-1) //check if current state has RightNeighbor
    {   neighbors.add(findRightNeighbor());

    }

    return neighbors;
}
```

- **Solver Class**

The key attributes in Solver class are frontier (priority queue of State object) and explored (list of State object). The frontier contains the states that are waiting to inspect if it is the goal state, the order of these states in the frontier depends on their evaluation function value which is the return value of getEvaluationFunction() method . Moreover, the explored contains the states that have been inspected. These attributes help us to implement A* graph algorithm search that we used to find the optimal solution.

The essential method in Solver class is findSloution() that returns true if the solution is found and false otherwise. This method is called, after adding the initial state node into the frontier. This method starts by while loop which its condition is the frontier is not empty, then inside the while, it starts by deleting the first state in frontier and saves it into the current state node variable, then inspect if the current state is the goal state or not, and returns true if the current state equals the goal state. In the other scenario, it adds the current state to explored, and calls the expand(currentStateNode) method which is responsible for adding the neighbors of the current state to the frontier in case they don't exist in the frontier or in the explored. In case, the while loop stops before returning true the method returns false.

below the code of findSloution() method in Solver class



```
private boolean findSloution() { // find the solution to the start node and return true if solution found

    while(!frontier.isEmpty()) {
         Node currentStateNode = frontier.remove(); //choose a leaf node that has minimum evaluation function value and remove it

        if(currentStateNode.getState().isGoal()) {  //if the node contains a goal state
            setGoalStateNode(currentStateNode);     //then set current state as the goal state
            return true;                            // and return true
        }
        else {

            explored.add(currentStateNode);         // add the node to the explored list
            expand(currentStateNode); //expand the chosen node
        }
    }
    return false;

}
```

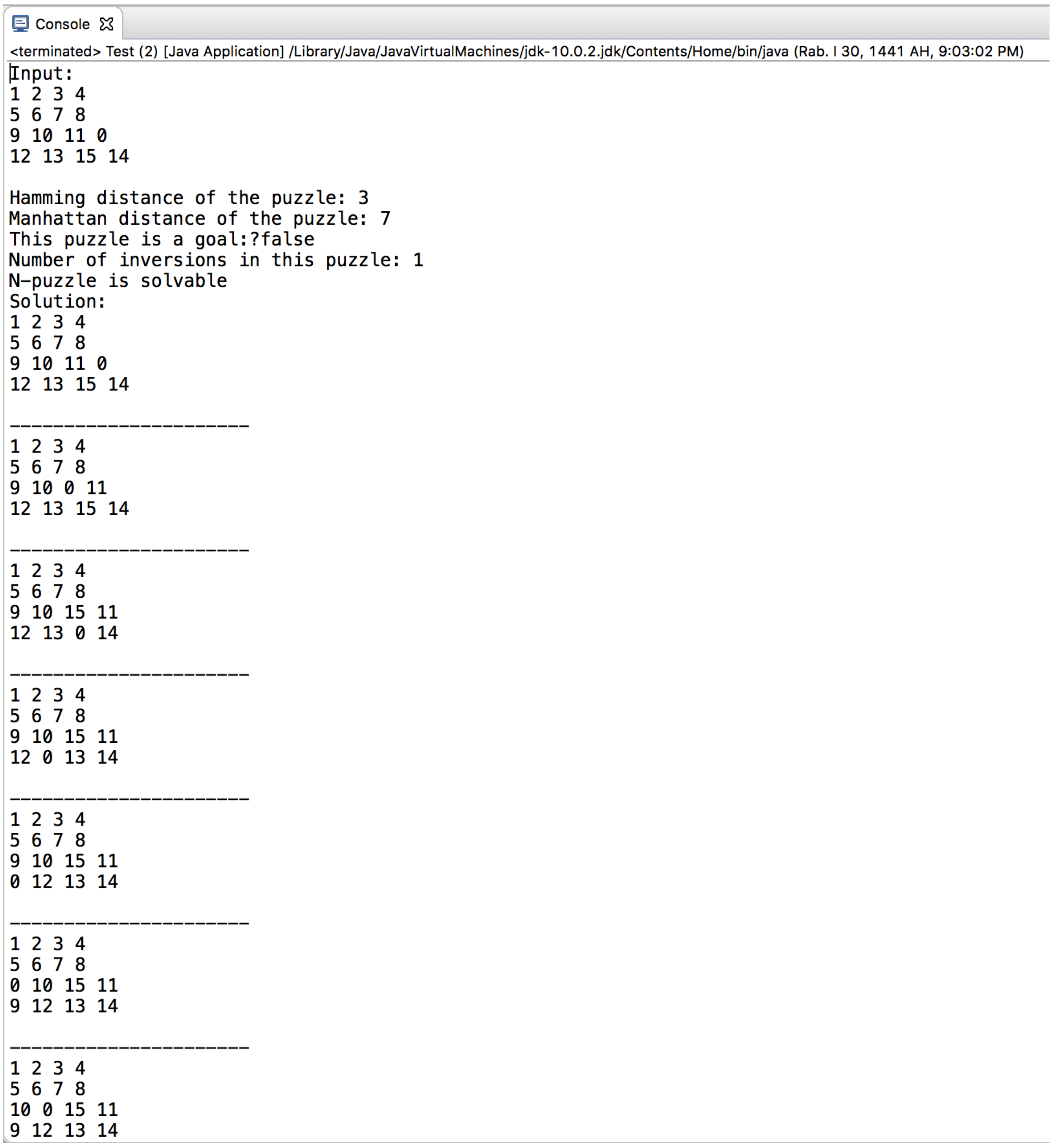
Results:

In this section, we will represent the result from two perspectives. The first perspective is the ability of the program to show the right results for puzzles from different sizes and conditions. The other perspective is testing the two heuristics to find the best one.

The result from the first perspective.

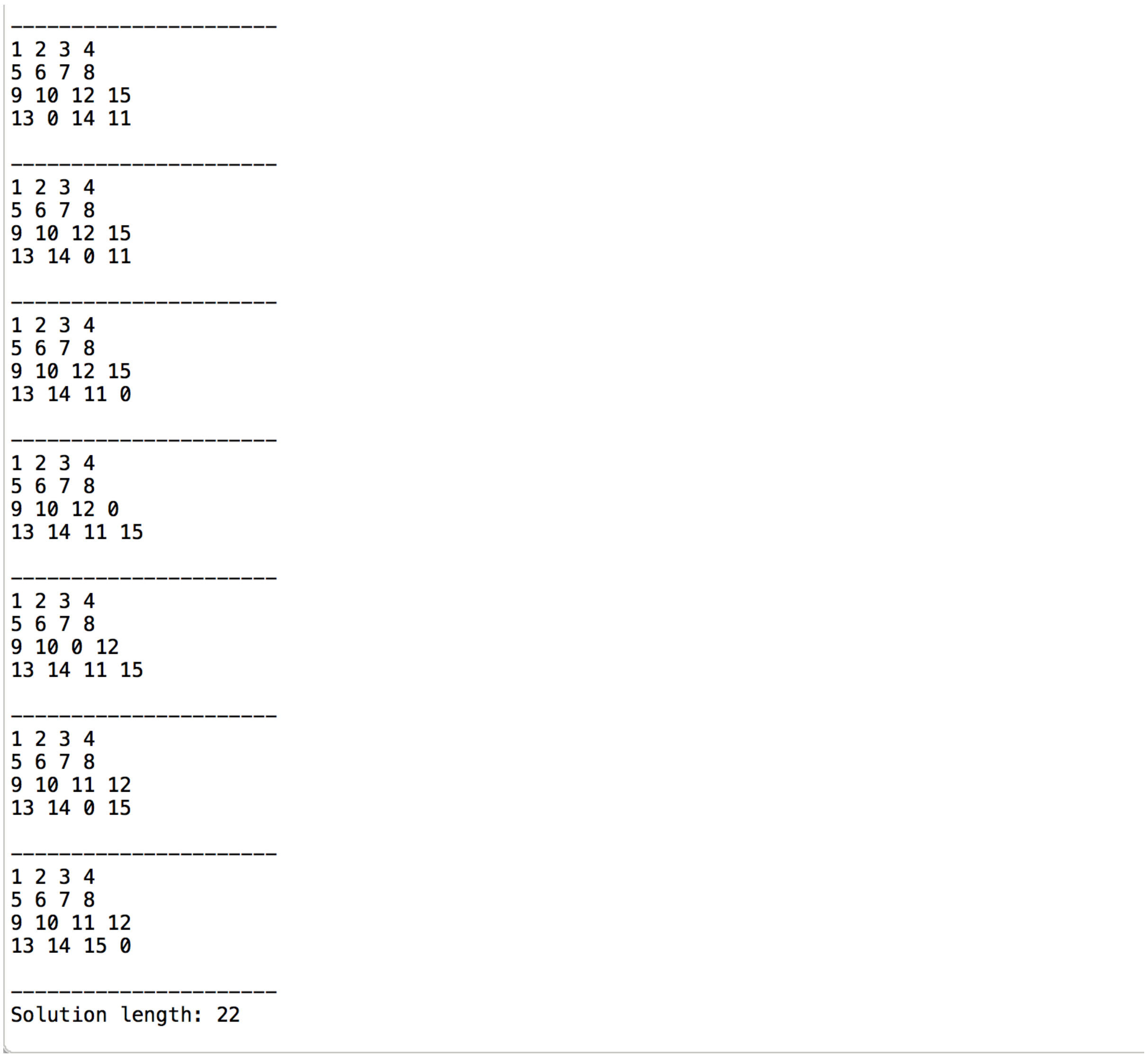
The first puzzle that we try to solve. It is a puzzle of size 15 and a solvable puzzle. The screenshots below show the result after entering this puzzle to our program.

example of solvable solution:



```
<terminated> Test (2) [Java Application] /Library/Java/JavaVirtualMachines/jdk-10.0.2.jdk/Contents/Home/bin/java (Rab. I 30, 1441 AH, 9:03:02 PM)
Input:
1 2 3 4
5 6 7 8
9 10 11 0
12 13 15 14

Hamming distance of the puzzle: 3
Manhattan distance of the puzzle: 7
This puzzle is a goal:?false
Number of inversions in this puzzle: 1
N-puzzle is solvable
Solution:
1 2 3 4
5 6 7 8
9 10 11 0
12 13 15 14

----------------------
1 2 3 4
5 6 7 8
9 10 0 11
12 13 15 14

----------------------
1 2 3 4
5 6 7 8
9 10 15 11
12 13 0 14

----------------------
1 2 3 4
5 6 7 8
9 10 15 11
12 0 13 14

----------------------
1 2 3 4
5 6 7 8
9 10 15 11
0 12 13 14

----------------------
1 2 3 4
5 6 7 8
0 10 15 11
9 12 13 14

----------------------
1 2 3 4
5 6 7 8
10 0 15 11
9 12 13 14
```

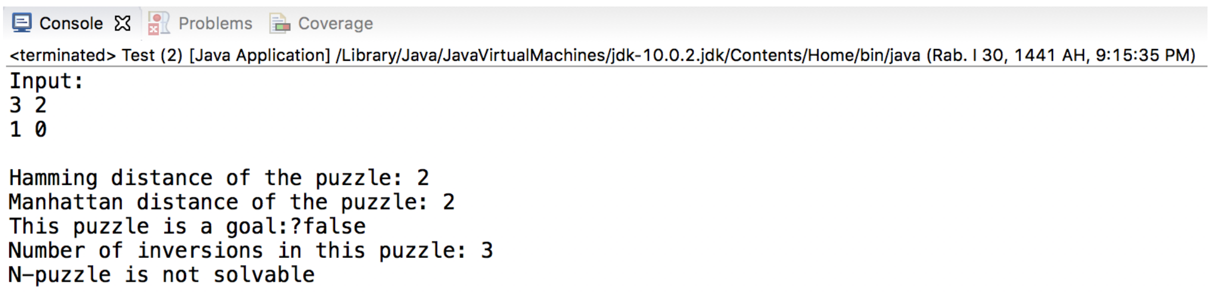
```
----------------------
1 2 3 4
5 6 7 8
10 12 15 11
9 0 13 14

----------------------
1 2 3 4
5 6 7 8
10 12 15 11
9 13 0 14

----------------------
1 2 3 4
5 6 7 8
10 12 15 11
9 13 14 0

----------------------
1 2 3 4
5 6 7 8
10 12 15 0
9 13 14 11

----------------------
1 2 3 4
5 6 7 8
10 12 0 15
9 13 14 11

----------------------
1 2 3 4
5 6 7 8
10 0 12 15
9 13 14 11

----------------------
1 2 3 4
5 6 7 8
0 10 12 15
9 13 14 11

----------------------
1 2 3 4
5 6 7 8
9 10 12 15
0 13 14 11

----------------------
```



```
----------------------
1 2 3 4
5 6 7 8
9 10 12 15
13 0 14 11

----------------------
1 2 3 4
5 6 7 8
9 10 12 15
13 14 0 11

----------------------
1 2 3 4
5 6 7 8
9 10 12 15
13 14 11 0

----------------------
1 2 3 4
5 6 7 8
9 10 12 0
13 14 11 15

----------------------
1 2 3 4
5 6 7 8
9 10 0 12
13 14 11 15

----------------------
1 2 3 4
5 6 7 8
9 10 11 12
13 14 0 15

----------------------
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0

----------------------
Solution length: 22
```

example of not solvable solution:

The second puzzle that we try to solve. It is a puzzle of size 3 and an unsolvable puzzle. The screenshot below shows the result after entering this puzzle to our program.



```
<terminated> Test (2) [Java Application] /Library/Java/JavaVirtualMachines/jdk-10.0.2.jdk/Contents/Home/bin/java (Rab. I 30, 1441 AH, 9:15:35 PM)
Input:
3 2
1 0

Hamming distance of the puzzle: 2
Manhattan distance of the puzzle: 2
This puzzle is a goal:?false
Number of inversions in this puzzle: 3
N-puzzle is not solvable
```

To make the program able to classify the entered puzzle if it solvable or not, we need to implement a specific method that depends on the number of inversions on the entered puzzle.

The result from the second perspective.

We compare the given two heuristics depending on time and we conclude the manhattan distance is faster than hamming. The table below shows puzzles from different sizes and execution times using two different heuristics.

the heuristic algorithm we used it in our project :

Example of n-puzzle	Manhattan **Time complexity**	hamming **Time complexity**
n=8 Input: 1 2 3 4 6 5 8 7 0	sloution length in manhattan: 17 Execution time in milliseconds : 86	sloution length in hamming: 17 Execution time in milliseconds : 130
n=24 Input: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 24 22 23 0	sloution length in manhattan: 19 Execution time in milliseconds : 328	sloution length in hamming: 19 Execution time in milliseconds : 32992
n=48 Input: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 48 46 47 0	sloution length in manhattan: 19 Execution time in milliseconds : 1100	sloution length in hamming: 19 Execution time in milliseconds : 28959

Conclusion:

There are a lot of ways to solve n-puzzles, but we are concerned about optimal ways depending on time. We try to reduce the number of loops and the complexity as much as possible to make a good performance, so that it has a short execution time, as we have shown in the table previously. We choose the Manhattan heuristic because this heuristic gives the right solution with less time.