INTELLIGENT SYSTEMS LAB

ÁLVARO ÁNGEL-MORENO PINILLA
CARLOS CÓRDOBA RUIZ
ROBERTO PLAZA ROMERO

Index:

SUBTASK 1

1. Requirements of the application

This application represented a living puzzle, made by a single image divided in as subimages as pieces of the puzzle. To move any piece from a place to another, every image must be stored in a correct way.

In order to develop it, we could choose any programming language, using the correct libraries. In this case, we chose Java, with some libraries that will be explained later.

The requirements that are established for this problem are:

- An artifact to save the whole image (including the subimages when we need to crop it).
- Detect the referential point (the black space) in any position of the puzzle.
- Move that referential point in any possible position and direction (getting exceptions in any case that is not possible).

2. Libraries and most used commands

As we explained before, we are using Java programming language to develop this application. But simple java libraries aren't enough to process and work with images. That's why we are using different kinds of libraries:

- Javax.swing: Allows us to create frames and labels to show the image in a correct way whenever we want.
- Java.awt: With this library, we can use a data type for images, called BufferedImage. It has a specific way of working that will be discussed later.
- Javax.imageio.ImageIO: With this library, we can work with the different kinds of images (.jpg, png…) and print those at the screen (write method).

We could have used some different libraries, but as we decided to work pixel by pixel, the best libraries that we could use are these. As the Buffered Images work with RGB combinations, the best thing to do is work with this pixels, that can be compared as RGB combinations as well.

3. Solution of Problems

In this section, we are going to explain the most important problems that we have to deal with in the development of the program. In addition, there's an explanation of how did we solve all of them.

- Define each position of the misplaced puzzle: Splitting both images (the original and the misplaced) in the same number of subimages, we can compare them one by one. After storing both group of subimages in arrays, each subimage of the misplaced subimages will get a value with the comparison. That value (integer) is saved into a different array called pos [][], with the same size as the subimages array.
  - Each move will be updated into the pos [][] array as well.
- How moves are made: We have defined the black space as the origin of the movements. This means that, for every movement, we have to detect in which position we have the referential point. To do so, we call a method to find the black position before any movement is made. Once we have found it, that subimages change its position with the corresponding one, depending of the direction of the movement.
- Allow the puzzle to do just the valid moves: As the puzzle has its limits (as a square), we have to control it from the programming point of view. Before doing the move, the program checks if the move can be made. In negative case, the movement won't be made.

SUBTASK 2

1. Requirements of the Application

With the result of the previous task, we will get a M x N-1 matrix, with the black one identificated. The objective is to explore the different possibilities with a search tree. The following tree is made of nodes, that represent each of the states that the puzzle can take. The information given by the node is parent, state, cost, action, depth and value (a random integer).

As a frontier, the elements must be represented in an ordered list, based on the value that correspond to each one.

Finally, it is necessary to compare two data structures, using both of them in the creation and store of them. The most important scopes to compare are the times needed to insert in the frontier, and the maximum number of nodes that can be saved.

2. Object Oriented Structure

In this subtask, we started to introduce Object Oriented Programming in order to have reusability, fiability and a code easier to modify and understand.

These are the following Object Classes that exist in the Subtask 2:

- StateSpace: Has all the attributes that defines any state of the puzzle. It is used in the main method every time a new move is made.
- nodeTree: This class can be created with the object State. Another way to create one of them is with a normal constructor (filling each of its attributes).
- ImgProcessor: Contains the creation and comparison of the images of subtask 1, so we can use it in an Object Oriented way in other classes.

3. Libraries and Used Data Structures

In addition to every library used in the last subtask,

- java.util.Random: The random generator to create the value at each node (an integer between 100 and 1000), so we can compare the nodes with that resulting value.
- Priority Queue: This structure can take the nodes as they are created, ordering them, depending on their value, after the iteration in which the come inside the structure.
- Linked List: This list saves the nodes in the order when they are created. To order them by value, it is necessary to go across the list in each iteration, comparing their values to order them in a correct way.

4. Solution of Problems

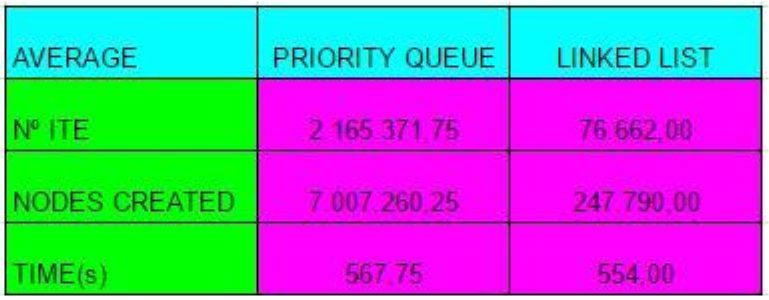
- Selecting two data structures: We decided to compare between a priority queue and a linked list. The current solution has two classes with the same implementation, the only change is how the frontier is implemented and which data structure is being used. That allows us to run them without changing any code.

- Creation of correct values: With the creation of random values at the constructor of node Tree, every sibling with the same parent had the same value. In order to resolve this problem, we create a random value at every iteration where a node tree object is created.

- The method to know the position of the black space: Before, we had a method that is called after each move, to know the new position of the black point. In order to change that, we defined two integer into State class. Those integers are changed as the black point is moved, so we already know the new position.

5. Comparison between structures

This is the comparison between both data structures. We take into account the number of iterations and nodes created at a certain time, when Eclipse tool can’t afford more memory for the structure. The Priority Queue couldn’t afford more memory after 567 seconds, while the Linked List kept running in the program. To compare them, we have established a similar time, comparing their other characteristics.



AVERAGE	PRIORITY QUEUE	LINKED LIST
Nº ITE	2.165.371,75	76.662,00
NODES CREATED	7.007.260,25	247.790,00
TIME(s)	567,75	554,00

These are the conclusions that we have taken from the study of both structures and the comparison that we did.

- By the number of iterations of each structure, we can define that Priority Queue works better in the same conditions and time than a Linked List, almost 3,5 times better.
- The number of nodes created by the Priority Queue is much greater than in the Linked List. This is due to in a similar time, the Priority Queue create them much faster and easier, as the Linked List has to order all the nodes each time a new one is created.
- In conclusion, the **Priority Queue** is much easier to apply, can work longer and creating more nodes, so a Linked List looks out-dated in comparison.

SUBTASK 3

1. Assignment of the application

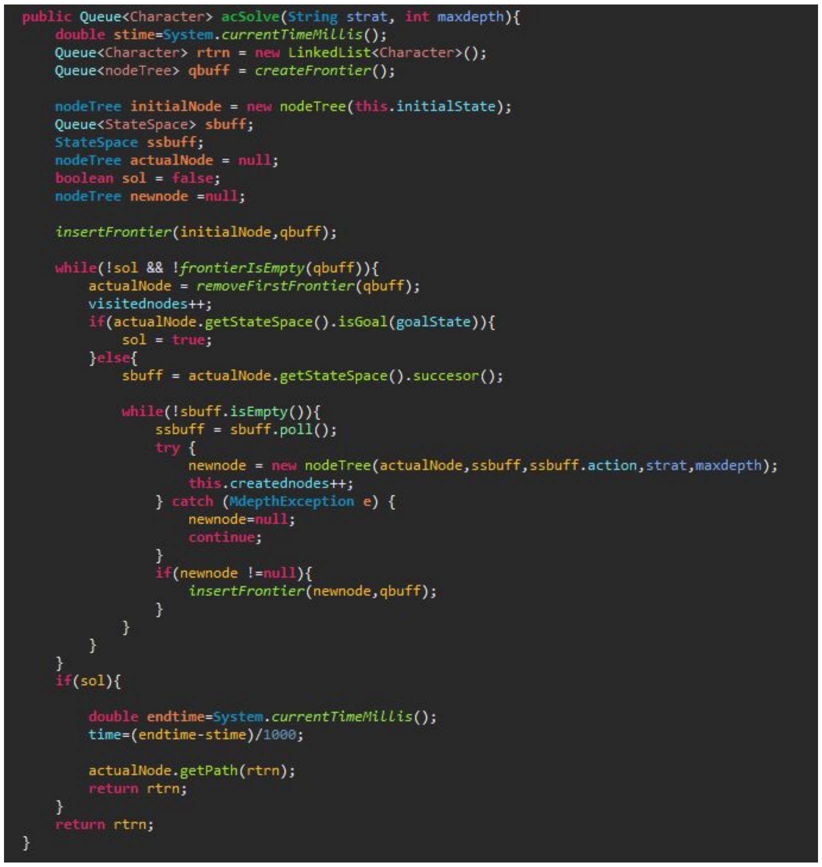
In this subtask we must program a basic version of the search algorithm with three different strategies:

- Breadth-first search.
- Depth-first search.
- Depth-limited search.
- Iterative Deepening Search.
- Uniform cost.

The solution of this assignment is to produce a text file containing a sequence of states corresponding to the path solution, and produce the solution of the puzzle as a sequence of pictures.

2. Code used for search algorithms

In order to do “compact” the 5 different search the first thing we do is to realize that Breadth First Search and Uniform Cost strategies are the same in this scenario, because the only cost that is contemplated is the depth.

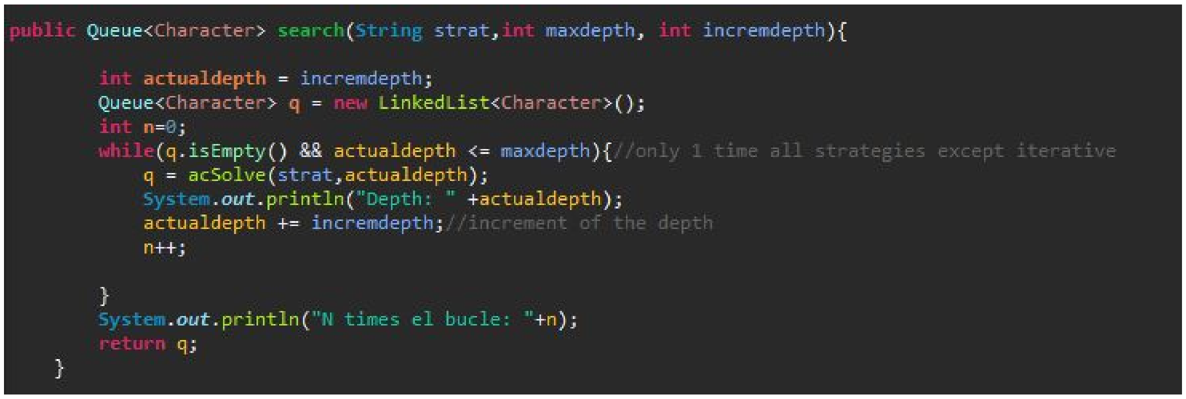


```java
public Queue<Character> acSolve(String strat, int maxdepth){
    double stime=System.currentTimeMillis();
    Queue<Character> rtrn = new LinkedList<Character>();
    Queue<nodeTree> qbuff = createFrontier();

    nodeTree initialNode = new nodeTree(this.initialState);
    Queue<StateSpace> sbuff;
    StateSpace ssbuff;
    nodeTree actualNode = null;
    boolean sol = false;
    nodeTree newnode =null;

    insertFrontier(initialNode,qbuff);

    while(!sol && !frontierIsEmpty(qbuff)){
        actualNode = removeFirstFrontier(qbuff);
        visitednodes++;
        if(actualNode.getStateSpace().isGoal(goalState)){
            sol = true;
        }else{
            sbuff = actualNode.getStateSpace().succesor();

            while(!sbuff.isEmpty()){
                ssbuff = sbuff.poll();
                try {
                    newnode = new nodeTree(actualNode,ssbuff,ssbuff.action,strat,maxdepth);
                    this.creatednodes++;
                } catch (MdepthException e) {
                    newnode=null;
                    continue;
                }
                if(newnode !=null){
                    insertFrontier(newnode,qbuff);
                }
            }
        }
    }
    if(sol){

        double endtime=System.currentTimeMillis();
        time=(endtime-stime)/1000;

        actualNode.getPath(rtrn);
        return rtrn;
    }
    return rtrn;
}
```

This is the code generated by means of traducing to java de pseudocode provided and “enhancing” it by means of adding a timer, implementing the count of created an visited nodes in the search, and the strategy to make the algorithm reject a node if it’s cost is larger than the maximum cost.

The algorithm by itself is an example of a Breadth First Search, knowing from previous assignments that the frontier is a Priority Queue, therefore, we manage to recycle the code by means of add an extra attribute to the class nodeTree, this let us choose the strategy we are using while creating the nodes and therefore increment the value of the depth by one or minus one depending if we want to search by depth or by breadth.

With this code and constraints we can do 4 of the 5 proposed solutions, but the Iterative Deepening Search is a bit more difficult. Therefore this is also a traduction of the pseudocode to java, is used to control the way the IDS works.



```java
public Queue<Character> search(String strat,int maxdepth, int incremdepth){

        int actualdepth = incremdepth;
        Queue<Character> q = new LinkedList<Character>();
        int n=0;
        while(q.isEmpty() && actualdepth <= maxdepth){//only 1 time all strategies except iterative
            q = acSolve(strat,actualdepth);
            System.out.println("Depth: " +actualdepth);
            actualdepth += incremdepth;//increment of the depth
            n++;

        }
        System.out.println("N times el bucle: "+n);
        return q;
    }
```

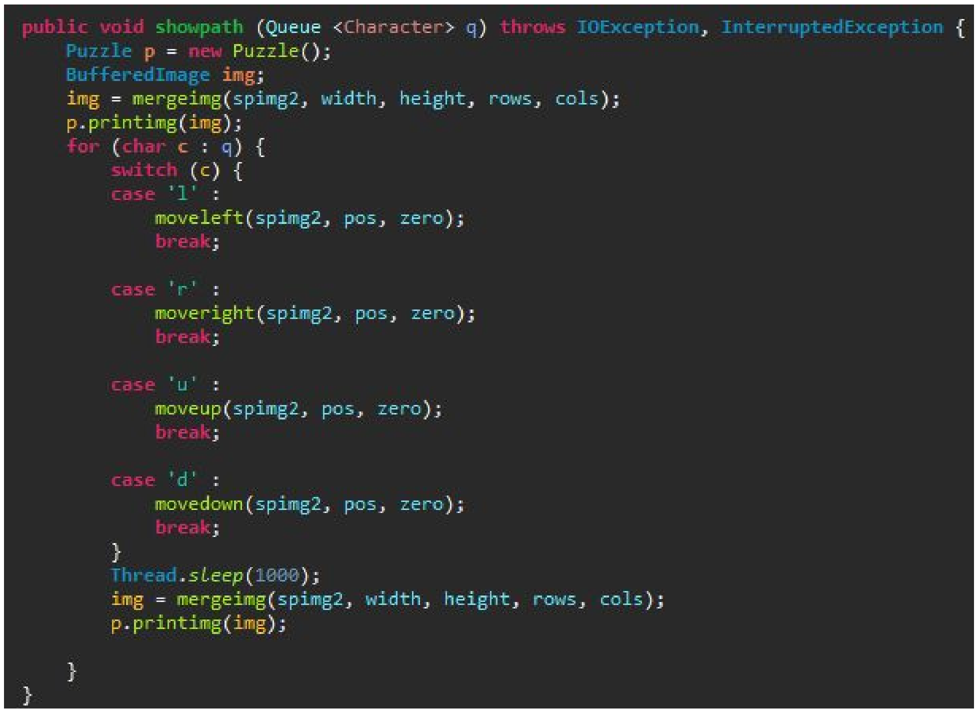
This function is not going to be executed if the search is not an Iterative Deepening Search.

The return value of the methods is always a LinkedList containing the set of movements you have to make in order to achieve the goal state.

3. Representation of the solution

In the main class of this deliverable we have to take some input data from the user, the path of the goal image, the path of the initial image, but in this deliverable we have as default the “AlhambraInicialPuzzle4x4.png” and “IntermedioAlhambra41.png", the number of rows and cols of the image, after having this data the user have to select the type of search we want to solve the problem: Breath-first search (BFS), Depth-first search (DFS), -Depth-limited search(DLS), Iterative Deepening Search(IDS), Uniform Cost Search(UCS).

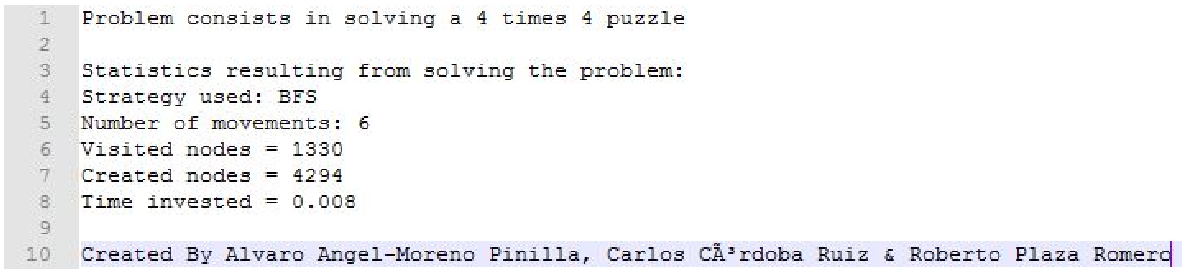
When we select the type of search we will get a queue with the movements we have to make to disordered picture in order to get the goal image, in order to represent graphically the solution we call to the method “showpath” of the ImageProcessor Class:



```java
public void showpath (Queue <Character> q) throws IOException, InterruptedException {
    Puzzle p = new Puzzle();
    BufferedImage img;
    img = mergeimg(spimg2, width, height, rows, cols);
    p.printimg(img);
    for (char c : q) {
        switch (c) {
        case 'l' :
            moveleft(spimg2, pos, zero);
            break;

        case 'r' :
            moveright(spimg2, pos, zero);
            break;

        case 'u' :
            moveup(spimg2, pos, zero);
            break;

        case 'd' :
            movedown(spimg2, pos, zero);
            break;
        }
        Thread.sleep(1000);
        img = mergeimg(spimg2, width, height, rows, cols);
        p.printimg(img);

    }
}
```

In order to see better the solution we introduce the use of “thread.sleep(1000)”, that makes that we wait 1 second until the next movement is make.

After representing graphically the solution we generate a document called "statistics.dat" that have this information:



```
Problem consists in solving a 4 times 4 puzzle

Statistics resulting from solving the problem:
Strategy used: BFS
Number of movements: 6
Visited nodes = 1330
Created nodes = 4294
Time invested = 0.008

Created By Alvaro Angel-Moreno Pinilla, Carlos CÃ³rdoba Ruiz & Roberto Plaza Romero
```

The strategy used, the number of movements in order to achieve the solution, the number of visited nodes, the number of created nodes and the time invested to find the solution.

4. Solution of problems

The most redactable problem that we had during the subtask was the execution with different images:

- At the 10x5 image, that was handled at virtual campus to try out the program, didn’t work from the very beginning. The problem was calculating the width and height of the subimages. They were calculated inside the method, instead of be given by the method. To solve this, we just added the attributes splitWidth and splitHeight to the class ImageProcessor, so we could easily call them into the method that gives us the subimages.
- Another problem is that a 10x5 puzzle doesn’t give a fast solution. With some higher depth (> 10) the execution of the solution can take a lot of time, while the program is calculating new nodes and moves every time.