# Search Algorithms in Java language (N-puzzle)

Luís Miguel Pedrosa de Moura Oliveira Henriques (up201604343)
Mestrado Integrado de Engenharia e Computação, MIEIC
Faculdade de Engenharia da Universidade do Porto
Santa Maria da Feira, Portugal
up201604343@fe.up.pt

**Abstract—In this document it is described a simple game, formulating it as a search problem, solving it with different algorithms, then analysed the results and how we can benefit from them.**

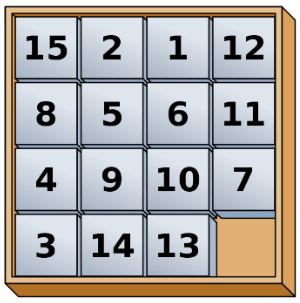
***Keywords—Artificial Intelligence, Search, A* Algorithm, Uniform Cost Algorithm, Greedy Algorithm and Breadth First Algorithm***

## I. Introduction

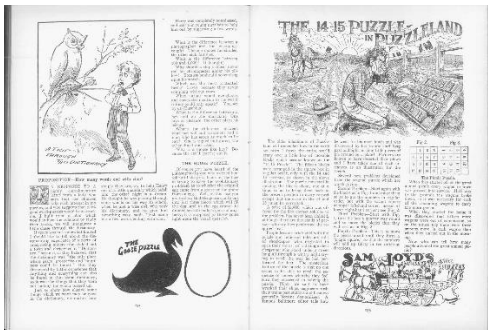
In this project, an application capable of resolving instances of the game "N-Puzzle" was implemented, without any external interaction, using search algorithms of Artificial Intelligence, namely A* Algorithm, Uniform Cost Algorithm, Greedy Algorithm and Breadth First Algorithm.

## II. Problem Description

- The game is played on a WxH board with WxH-1 pieces resulting in N pieces and thus giving the puzzle its name. The most common boards are of sizes 3x3, resulting in 8 pieces, and 4x4, resulting in 15 pieces. Each piece is numbered from 1 to N and can only move to an empty spot. The objective of the game is to move the pieces around until a specific pattern is reached. The challenge lays in the fact that some moves require damaging already finished grids of the board and can lead to complicated states.



- In 1891, Samuel Floyd[1] took credit for inventing the 15-puzzle and, after his death, his son published the book "Cyclopedia of Puzzles" where he described in great detail the puzzle known today. The father also promised a 1000 dollars reward to anyone who could solve a particular instance of it, which is now known to be impossible. Regardless, we know that the first draft of the puzzle was actually done in 1874 by Noyes Palmer Chapman where the goal was to achieve a particular sum in each column and row. The postmaster applied for a patent in 1880 but was rejected for being too similar to an already existing patent owned by Ernest U. Kinsey, although his only described a sliding puzzle mechanism lacking any goal[2].



- Since then, the puzzle was been adapted and distributed by multiple companies and studied by even more mathematicians. Lycée Évariste Galois (October 25th, 1811 – May 31st, 1832; Paris) whose work on Number Theory and Permutations allowed Woolsey Johnson and William E. to publish "Notes on the '15' Puzzle"[3] where they proved not all instances of the puzzle lead to a solvable state, including the one proposed by Samuel Floyd.

## III. Formulation of the problem

State Representation: A board can be represented with a matrix of W width and H height, whose positions range from (0, 0) on the topmost left to (*x*, *y*) on the bottommost right. Every piece can be represented with numbers ranging from "1" to N, N=W*H-1, where each piece occupies a single position on the board. Since the original game includes a board position without a piece, in our implementation this will be represented with a piece numbered with "0". Each piece can only be moved by swapping with the "0" piece.

Initial State: A board will always start with a pseudo-random placing of each piece. It is said pseudo since it was proven that not all random placements lead to solvable states.

Target State: The game ends when the board reaches a target position. In most implementation of the game, this is a state that when traversing the board horizontally, row by row, each board position will have a piece in increasing value. The last position won't have a piece, but in our representation, this will be occupied by the piece labels as "0".

Operators:

xPos – Position of the “0” piece within the horizontal axis.

yPos – Position of the “0” piece within the vertical axis.

N – Number of the previous piece.

| Name: | Pre-conditions: | Effects: | Costs: |
|---|---|---|---|
| MoveUp | yPos > 0; | (xPos, yPos-1) = N;<br>(xPos, yPos) = “0” | 1 |
| MoveDown | yPos < 5; | (xPos, yPos+1) = N;<br>(xPos, yPos) = “0” | 1 |
| MoveLeft | xPos > 0; | (xPos-1, yPos) = N;<br>(xPos, yPos) = “0” | 1 |
| MoveRight | yPos < 5; | (xPos+1, yPos) = N;<br>(xPos, yPos) = “0” | 1 |

## IV. Related Work

The work that most guided me in this project was my group’s first project. It gave me a first contact with search algorithms and gave me a general approach to problems of the same nature. Design solution of other projects were avoided.

## V. Game Implementation

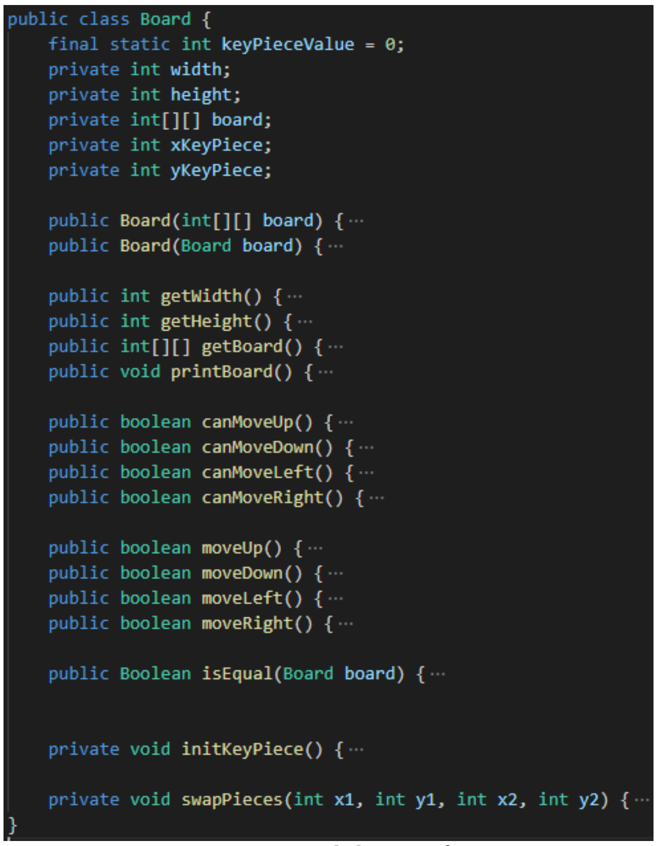
This project was developed using the Java programming language. I chose to implement this game in this specific language, not only because I already felt comfortable using it, but also because it is one of the most used programming languages in the world. Java is also widely used in the Artificial Intelligence context. Therefore, I believed that utilizing this language would be more beneficial than all other languages.

Moving on to the overall organization of the game implementation, we can begin by describing our approach to the illustration of the game board. As specified in the third section, a board is represented by a matrix of variable size. Below, we demonstrate how a 4*4 matrix is represented in this implementation.

```
|05|01|03|04|
|02|00|07|08|
|10|06|11|12|
|09|13|14|15|
```

Figure 4 – Board Representation

In this implementation, each board is represented in the *Board* class. The code below represents the interface of this class.



```
public class Board {
    final static int keyPieceValue = 0;
    private int width;
    private int height;
    private int[][] board;
    private int xKeyPiece;
    private int yKeyPiece;

    public Board(int[][] board) {···
    public Board(Board board) {···

    public int getWidth() {···
    public int getHeight() {···
    public int[][] getBoard() {···
    public void printBoard() {···

    public boolean canMoveUp() {···
    public boolean canMoveDown() {···
    public boolean canMoveLeft() {···
    public boolean canMoveRight() {···

    public boolean moveUp() {···
    public boolean moveDown() {···
    public boolean moveLeft() {···
    public boolean moveRight() {···

    public Boolean isEqual(Board board) {···


    private void initKeyPiece() {···

    private void swapPieces(int x1, int y1, int x2, int y2) {···
}
```

Figure 3 – Board class interface

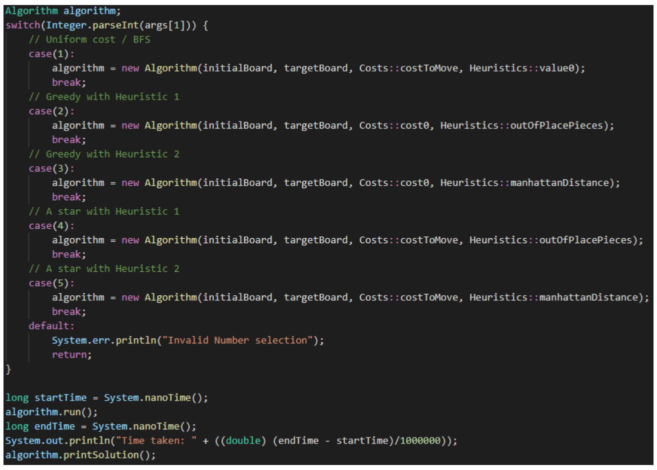
Analysing the code above, the variables *xKeyPiece* and y*KeyPiece* represent the *x* and *y* coordinates of the “0” piece position, *width* and *height* represent the size of the board and the board itself is stored as a 2dimensional array of integers.

The class contains methods create its objects, one from a 2dimensional array and the other from an already existing board. It can also retrieve its internal status as well as printing a formatted board. The *canMove()* and *move()* methods are the ones responsible for changing the internal status of a board.

The *isEqual()* method is used to compare boards and return true if their boards contents is equal regardless if it’s the same object.

## VI. SEARCH ALGORITHMS

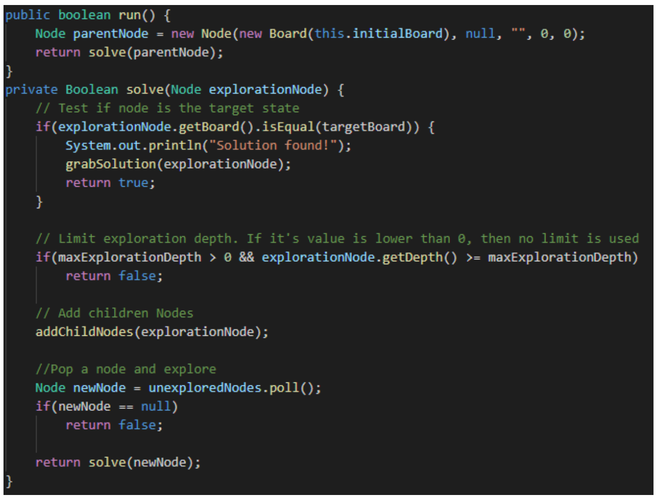
All used algorithms (Uniform, Greedy and A*) follow the same implementation and execution:



```
Algorithm algorithm;
switch(Integer.parseInt(args[1])) {
    // Uniform cost / BFS
    case(1):
        algorithm = new Algorithm(initialBoard, targetBoard, Costs::costToMove, Heuristics::value0);
        break;
    // Greedy with Heuristic 1
    case(2):
        algorithm = new Algorithm(initialBoard, targetBoard, Costs::cost0, Heuristics::outOfPlacePieces);
        break;
    // Greedy with Heuristic 2
    case(3):
        algorithm = new Algorithm(initialBoard, targetBoard, Costs::cost0, Heuristics::manhattanDistance);
        break;
    // A star with Heuristic 1
    case(4):
        algorithm = new Algorithm(initialBoard, targetBoard, Costs::costToMove, Heuristics::outOfPlacePieces);
        break;
    // A star with Heuristic 2
    case(5):
        algorithm = new Algorithm(initialBoard, targetBoard, Costs::costToMove, Heuristics::manhattanDistance);
        break;
    default:
        System.err.println("Invalid Number selection");
        return;
}

long startTime = System.nanoTime();
algorithm.run();
long endTime = System.nanoTime();
System.out.println("Time taken: " + ((double) (endTime - startTime)/1000000));
algorithm.printSolution();
```

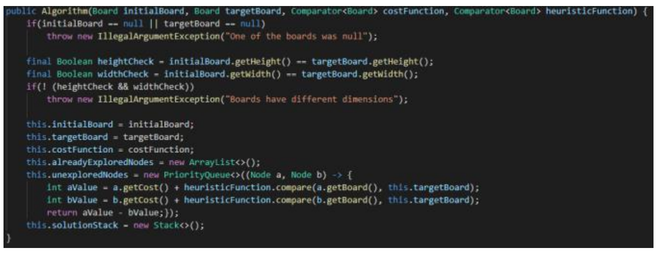
An object *Algorithm* is created with the apropriate arguments, then the *run()* method is called, the solution is generated and *printSolution()* is called to view it.

Exploring the run method, we can see a basic implementation of a search algorithm:  Firstly, generate parent node, then test if it is target state. If not, generate child nodes, then get next node to analyse and repeat until solution is found:



```
public boolean run() {
    Node parentNode = new Node(new Board(this.initialBoard), null, "", 0, 0);
    return solve(parentNode);
}
private Boolean solve(Node explorationNode) {
    // Test if node is the target state
    if(explorationNode.getBoard().isEqual(targetBoard)) {
        System.out.println("Solution found!");
        grabSolution(explorationNode);
        return true;
    }

    // Limit exploration depth. If it's value is lower than 0, then no limit is used
    if(maxExplorationDepth > 0 && explorationNode.getDepth() >= maxExplorationDepth)
        return false;

    // Add children Nodes
    addChildNodes(explorationNode);

    //Pop a node and explore
    Node newNode = unexploredNodes.poll();
    if(newNode == null)
        return false;

    return solve(newNode);
}
```

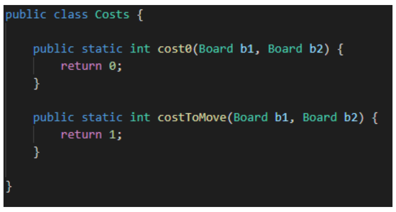
The differences reside in the way the class's constructor is called:



```
public Algorithm(Board initialBoard, Board targetBoard, Comparator<Board> costFunction, Comparator<Board> heuristicFunction) {
    if(initialBoard == null || targetBoard == null)
        throw new IllegalArgumentException("One of the boards was null");

    final Boolean heightCheck = initialBoard.getHeight() == targetBoard.getHeight();
    final Boolean widthCheck = initialBoard.getWidth() == targetBoard.getWidth();
    if(! (heightCheck && widthCheck))
        throw new IllegalArgumentException("Boards have different dimensions");

    this.initialBoard = initialBoard;
    this.targetBoard = targetBoard;
    this.costFunction = costFunction;
    this.alreadyExploredNodes = new ArrayList<>();
    this.unexploredNodes = new PriorityQueue<>((Node a, Node b) -> {
        int aValue = a.getCost() + heuristicFunction.compare(a.getBoard(), this.targetBoard);
        int bValue = b.getCost() + heuristicFunction.compare(b.getBoard(), this.targetBoard);
        return aValue - bValue;});
    this.solutionStack = new Stack<>();
}
```

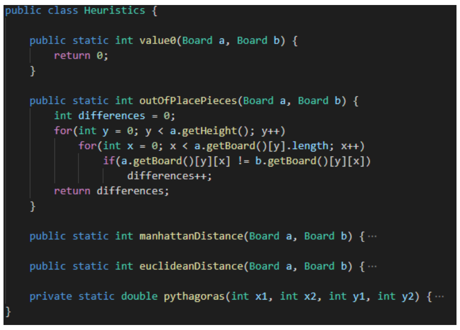
This method takes two Comparator<Board> functions that it uses to generate a priority between nodes. All unexplored nodes are placed in a priority queue and each time a dequeue is done, the node which the highest priority is returned which is equal to the one with the lowest sum of heuristics and cost. If two nodes have the same priority, then they are dequeued in the same order they were placed.

The cost functions used were *cost0()*, where each move's cost doesn't impact the outcome, and *costToMove()*, which, in this game, is always equal to one.



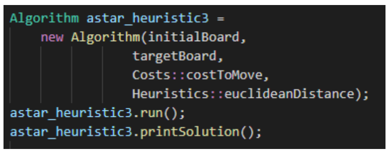
```
public class Costs {

    public static int cost0(Board b1, Board b2) {
        return 0;
    }

    public static int costToMove(Board b1, Board b2) {
        return 1;
    }

}
```

The heuristic functions used were *value0()*, where the heuristic isn't used, *outOfPlacePieces()*, where we count the differences between the two board, and *manhattanDistance()*, where we add up the horizontal and vertical distance of each piece to its correct spot.



```
public class Heuristics {

    public static int value0(Board a, Board b) {
        return 0;
    }

    public static int outOfPlacePieces(Board a, Board b) {
        int differences = 0;
        for(int y = 0; y < a.getHeight(); y++)
            for(int x = 0; x < a.getBoard()[y].length; x++)
                if(a.getBoard()[y][x] != b.getBoard()[y][x])
                    differences++;
        return differences;
    }

    public static int manhattanDistance(Board a, Board b) {···

    public static int euclideanDistance(Board a, Board b) {···

    private static double pythagoras(int x1, int x2, int y1, int y2) {···
}
```

The advatages of this aproch is that the system is easily expandable to include more algorithms with different costs and/or heuristics simply by creating these new functions:



```
Algorithm astar_heuristic3 =
    new Algorithm(initialBoard,
                  targetBoard,
                  Costs::costToMove,
                  Heuristics::euclideanDistance);
astar_heuristic3.run();
astar_heuristic3.printSolution();
```

## VII. Experiences and results

In order to easily visualize the results obtained, each one of the tables below will illustrate the outcome of the different algorithms for the 4 boards proposed. The results were averaged by 100 runs within the same conditions:

Board 1:

| | Nodes Seen | Nodes Generated | Time taken (seconds) |
|---|---|---|---|
| Uniform | 56 | 81 | 0.103 |
| Greedy 1 | 10 | 16 | 0.102 |
| Greedy 2 | 13 | 20 | 0.069 |
| A* 1 | 10 | 16 | 0.023 |
| A* 2 | 13 | 20 | 0.028 |

Board 2:

| | Nodes Seen | Nodes Generated | Time taken (seconds) |
|---|---|---|---|
| Uniform | 219 | 307 | 0.139 |
| Greedy 1 | 15 | 23 | 0.104 |
| Greedy 2 | 19 | 28 | 0.036 |
| A* 1 | 16 | 24 | 0.026 |
| A* 2 | 16 | 23 | 0.032 |

Board 3:

| | Nodes Seen | Nodes Generated | Time taken (seconds) |
|---|---|---|---|
| Uniform | 713 | 1000 | 0.460 |
| Greedy 1 | 965 | 1174 | 0.824 |
| Greedy 2 | 28 | 41 | 0.045 |
| A* 1 | 64 | 91 | 0.038 |
| A* 2 | 30 | 44 | 0.040 |

Board 4:

| | Nodes Seen | Nodes Generated | Time taken (seconds) |
|---|---|---|---|
| Uniform | Ran out of memory before tests were done | | |
| Greedy 1 | 85 | 133 | 0.113 |
| Greedy 2 | 271 | 406 | 0.238 |
| A* 1 | 46 | 69 | 0.037 |
| A* 2 | 79 | 120 | 0.074 |

As we can see from the result, the A* algorithm using the first heuristic, the number of pieces with an offset, always provides the fastest results, requiring, most of the times, the least amount of memory.

The Uniform is almost always the slowest and the one that requires the most amount of memory, which makes sense do to the nature of the algorithm.

We can also see that the execution time decreases when comparing the Greedy algorithms with the corresponding A*. This is do to the fact that the A* algorithms use a more informed search than Greedy one's do.

## VIII. Conclusions and Development Perspectives

This project was certainly interesting to do since it gave me another chance to apply the knowledges acquired from the previous project, but with more experience.

## References

[1] Santos, Carlos Pereira dos; Neto, João Pedro and Silva, Jorge Nuno, "A teoria de grupos + o Puzzle do 15", Single Edition, Norprint, June 2007, ISBN: 978-989612270-6, consulted in April 2019.

[2] Patent owned by Ernest U. Kinsey describing his mechanism for a sliding blocks puzzle. Approved in 1878-08-20, expired in Lifetime status as of 1895-08-20. Available at https://patents.google.com/patent/US207124. Consulted in April 2019. Copy of document was stored in docs/resources for future consult.

[3] Johnson, Woolsey and E., William; "Notes on the '15' Puzzle", published as Article in journal "American Journal of Mathmatics", Vol2, Number4, The Johns Hopkins University Press, December 1879, DOI: 10.2307/2369492 . Avalable at www.jstor.org/stable/2369492. Consulted in April 2019. Copy of document was stored in docs/resources for future consult.

[4] Link to image used as example of 15-puzzle. Available at https://en.wikipedia.org/wiki/File:15-puzzle_magical.svg. Consulted in April 2019. Copy of image was stored in docs/resources for future consult.

[5] Image of Sam Lyod's impossible riddle, screenshot of the son's book, avalable at http://www.mathpuzzle.com/loyd/cop234-235.jpg. Consulted in April 2019. Copy of image was stored in docs/resources for future consult.