**8 Puzzle - Searches**

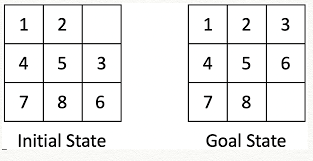
**Introduction**

In this report, I will explain how works the code that I did about the 8 Puzzle solver in python with the algorithm of A\* and the heuristic of Manhattan distance.

**Development**

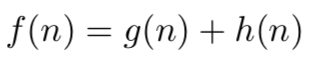
First, I’m going to explain the problem of N-Puzzle and how works in order to reach the goal.

In my example, N is equal to 8. So, the puzzle consists of 8 tiles and one empty space where the tiles can be moved. Start and Goal state of the puzzle are provided. The puzzle can be solved by moving the tiles one by one in the single empty space, in order to reach the goal state.



A\* algorithm

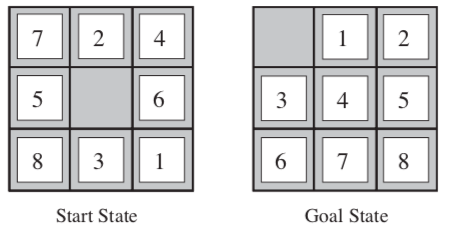
The most widely known form of best-first search is called A∗ search (pronounced “A-star search”). It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal:



Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have f(n) = estimated cost of the cheapest solution through n.

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + h(n). It turns out that this strategy is more than just reasonable: provided that the heuristic function h(n) satisfies certain conditions, A∗ search is both complete and optimal. The algorithm is identical to UNIFORM-COST-SEARCH except that A∗ uses g + h instead of g.

If we want to find the shortest solutions by using A∗, we need a heuristic function that never overestimates the number of steps to the goal. There is a long history of such heuristics for the 15-puzzle; here are two commonly used candidates:



* h1 = the number of misplaced tiles. In the image before, all of the eight tiles are out of position, so the start state would have h1 = 8. h1 is an admissible heuristic because it is clear that any tile that is out of place must be moved at least once.
* h2 = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the city block distance or **Manhattan distance**. h2 is also admissible because all any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in the start state give a Manhattan distance of h2 =3+1+2+2+2+3+3+2=18.

The Manhattan distance is the one that I used, so I will explain it in more detail the example before taking into consideration the goal.

* For the tile 1 to reach the goal destination it has to move 1 space to the left and 2 spaces up. Moves for 1 = 3.

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  | 1 |

* For the tile 2 to reach the goal destination it has to move 1 space to the right. Moves for 2 = 1.

|  |  |  |
| --- | --- | --- |
|  | 2 |  |
|  |  |  |
|  |  |  |

* For the tile 3 to reach the goal destination it has to move 1 space to the left and 1 space up. Moves for 3 = 2.

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  | 3 |  |

* For the tile 4 to reach the goal destination it has to move 1 space to the left and 1 space down. Moves for 4 = 2.

|  |  |  |
| --- | --- | --- |
|  |  | 4 |
|  |  |  |
|  |  |  |

* For the tile 5 to reach the goal destination it has to move 2 spaces to the right. Moves for 5 = 2.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| 5 |  |  |
|  |  |  |

* For the tile 6 to reach the goal destination it has to move 2 spaces to the left and 1 space down. Moves for 6 = 3.

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  | 6 |
|  |  |  |

* For the tile 7 to reach the goal destination it has to move 1 space to the right and 2 spaces down. Moves for 7 = 3.

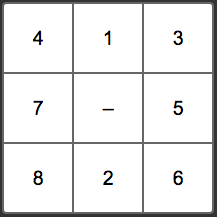
|  |  |  |
| --- | --- | --- |
| 7 |  |  |
|  |  |  |
|  |  |  |

* For the tile 1 to reach the goal destination it has to move 2 spaces to the right. Moves for 8 = 2.

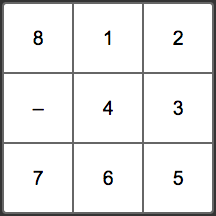
|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
| 8 |  |  |

The summarization of all the moves are 18, so the h(n) in the formula of f(n) is equal to 18.

Another thing to mention is the way I verify if the puzzle is solvable or not at the beginning.



It is not possible to solve an instance of 8 puzzle if the number of inversions is odd in the input state. In the examples given in above image, the first example has 10 inversions, therefore the puzzle is solvable.



The second example has 11 inversions, therefore unsolvable.

**Puzzle Representation**

To explain this, instead of representing the puzzle as a matrix, I will represent it like an array. So, for example, the final state of the puzzle would be [1, 2, 3, 4, 5, 6, 7, 8, 0], where 0 has been used to denote the empty square.

**Inversions and Polarity**

To make sense of the algorithm for determining whether a puzzle state is solvable we need to define two terms:

* **Inversion**. This is any pair of tiles that are not in the correct order.
* **Polarity**. This is the total number of inversions even (solvable) or odd (not solvable).

Consider the following puzzle state which has six inversions:

[1, 3, 4, 7, 0, 2, 5, 8, 6]

Let’s look at the inversions (since the 0 is not taking into consideration):

* 3 > 2 – 1st Inversion: From 3 the following number lesser than 3 is the 2.
* 4 > 2 – 2nd Inversion: From 4 the following number lesser than 4 is the 2
* 7 > 2 – 3rd Inversion: From 7 the following numbers lesser than 7 are the 2, 5 and 6.
* 7 > 5 – 4th Inversion
* 7 > 6 – 5th Inversion
* 8 > 6 – 6th Inversion: From 8 the following number lesser than 8 is 6.

Since there are 6 inversions, 6 is an even number, so this state is solvable. Here’s a state which is not solvable:

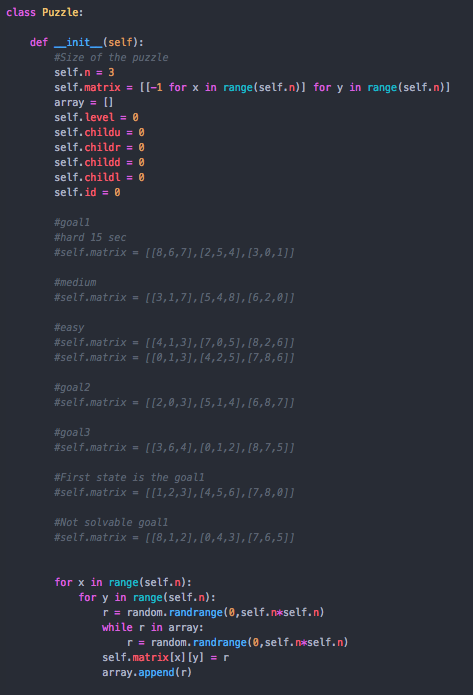
[2, 1, 3, 4, 5, 6, 7, 8, 0]

There’s just a one inversion 2 > 1, 1 is an odd number, so this state is not solvable.

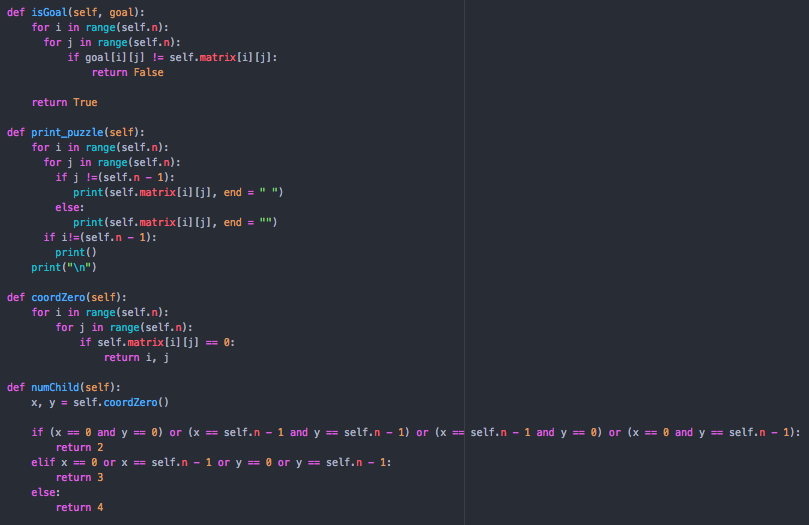
NOTE: This just applies for the goal = [1,2,3,4,5,6,7,8,0]. You can apply this for another goal but you have to identify which number have to be before and after each number.

**Code**

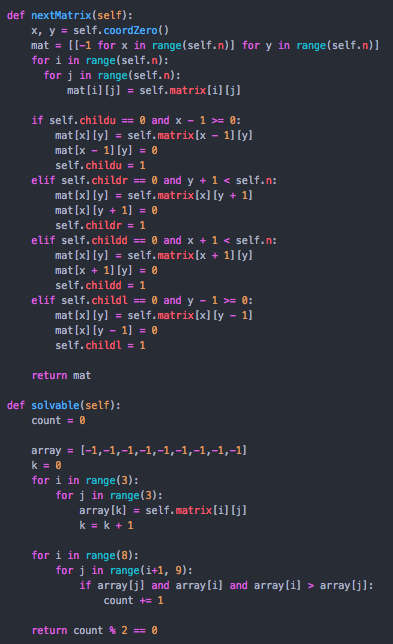
Once you have the importance information about the problem, the algorithm and the heuristic that I used, I will explain how I implement it.



I create 2 classes, the first one “Puzzle” is the one that generates the first instance of the puzzle with the attributes of n that is the amount of rows and columns, matrix which is the random order of the numbers, the level which is the level of the tree, this is the initial state that’s why the level is 0, the childu, childr, childd, childl which is a flag that indicates if the child node of moving the blank space up, down, left and right, is already created and the id which is the identifier of the node. The next part commented are examples of initial state of the matrix. The next for is where the matrix is generated randomly.

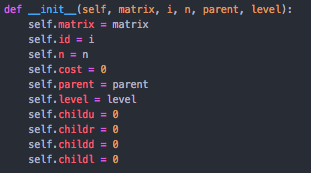


These functions are too simple. The “isGoal()” function received the goal and determine if the puzzle is the already the goal. “Print\_puzzle()” function print the matrix in an specific format. “CoordZero()” returns the coordinates of the matrix where is the blank space (or tile 0). “NumChild()” returns the quantity of child node that the puzzle can have, because is not the same amount of child nodes if the 0 is in a corner than if is in the center of the matrix.

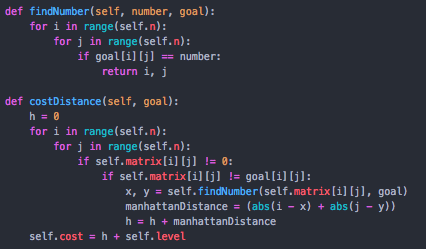


The “nextMatrix()” function detect which is the next child node from the current state of the node. It obtains the coordinates of the blank space (tile 0), and with the flags of childu, childr, childd and childl, the program detects if a node was already created. Finally, the “solvable()” calculates the inversions for the goal = [1,2,3,4,5,6,7,8,0] from the initial state to see if it has a solution.

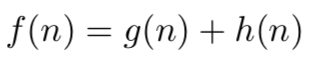
Class “Node” is a class which creates the child nodes and basically has the same attributes of the class “Puzzle”.



The only difference is the attribute parent which is equal to the parent node. There are functions which are the same as in the puzzle class like: “isGoal()”, “print\_node()”, “coordZero()”, “numChild()” and “nextMatrix()”.

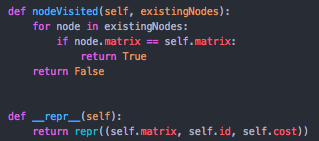


The “findNumber()” functions return the coordinates where is an specific number in the goal matrix in order to make the Manhattan distance calculation. The “costDistance()” is the function that calculates the total cost of the node with the formula that we already see.



Which:

* g(n) is the initial state to the current state. This number is obtained with the level attribute, which is the level of the tree, that indicates how many moves were already did from the initial state.
* h(n) is the summarization of the amounts of moves that each number takes to reach the respective goal coordinates. So, I did this with the “manhattanDistance” variable and that number is adding to the accumulator variable “h”.



The “nodeVisited()” function received the stack where are all the nodes that are already created, the visited nodes, and returns true if the node is already created or false if not.

The “repr()” function will help us to sort the stack of priority that we will use for the algorithm with the attribute of cost.



Now the algorithm. The first thing I do is define the puzzle object and verify if the initial state is the goal, if not it continues to verify if the puzzle is solvable, if not the execution is stopped.

Then I define the stacks 2 stacks, priorityNode that is the stack where I’m going to introduce the new child nodes that I create. And the existingNodes which will have the nodes state that were already created. Then I initialized the id which will be assigned to every node.

Then, the next for is generates the child of the first state which is the “puzzle” variable. It creates the child nodes, calculate the cost distance and then verify if the node already exist in the existing nodes, if not it push that node to the priority stack and also to the existing nodes stack and the id increments 1.

The next thing is to sort the stack, so in order to sort it with the cost attribute I used the next line of code. Here is where the “\_\_repr\_\_” function will work in order to return the cost attribute.

../../../../../Desktop/Captura%20de%20pantalla%202019-11-25%20a%20la(s)%2023.43.02.png

So, I sorted the stack from the lesser cost to the higher, then I put it in reverse in order to have that node in the top and pop it out from the stack. This node I save it in a variable “nodeN”.

Then I used a while loop to generate the rest of the child node until find the goal. So, if the “nodeN” variable is the goal, the while loop ends, if not it makes the same thing as the for loop before.

Once it finds the goal, the while loop ends and I proceed to generate the cheapest cost path. To create this, I used while true loop and in a stack called “path” I introduce the “nodeN” which is the goal and then I assigned to the “nodeN” the parent of that current node in the “nodeN” variable. I did that until I find the node id attribute equal to 0 which means that the node I had is the initial state. Finally, I put it in reverse that stack and printed to see in order from the beginning to the end.

And that it, with this code I solved every puzzle (if it is possible) with the smallest path.

**Referencias**

* Russell S. J., Norvig P. (2010). Artificial Intelligence: A Modern Approach, 3rd Edition, Editorial Pearson.
* Collier, A. B. (2019, 10 April). Sliding Puzzle Solvable? Recuperado el 26 noviembre, 2019, de: <https://datawookie.netlify.com/blog/2019/04/sliding-puzzle-solvable/>
* Sonawane A. (2018, 15 September). Solving 8-Puzzle using A\* Algorithm, Good Audience, Recuperado el 26 noviembre, 2019, de: <https://blog.goodaudience.com/solving-8-puzzle-using-a-algorithm-7b509c331288#:~:targetText=Let's%20start%20with%20what%20I,%E2%80%9C8%2DPuzzle%E2%80%9D%20problem.&targetText=The%20puzzle%20is%20divided%20into,the%20tiles%20can%20be%20moved.>
* <https://www.youtube.com/watch?v=GuCzYxHa7iA>