1

**Path Determination In A ROS Map Image With A\* Algorithm In Python**

|  |  |
| --- | --- |
| Table Of Contents |  |
| 1. What does it do? | 2 |
| 2. Overall Approach and Implementation In Abstract  2a. Fetching terminal commands  2b. Processing the image  2c. Main processing file | 3 |
| 3. Understanding program flow module-by-module  3a. Command Line arguments with **commandline.py**  3b. Processing the map file with **get\_map.py**  3c. Heart of the program with **Astar\_shortest\_path.py** |  |
| 4. Test cases and performance |  |

**1**

**What does it do?**

The goal here is simple and straightforward. Using the A\* algorithm, you’re supposed to find the most optimal path from point A to point B. However, the original goal didn’t go along this way.

The challenge was to utilise the provided .pgm format map file as an input and move your virtual robot through the map successfully by avoiding obstacles. The task also involved using Gazebo and Rviz for world building and visual control.

Since Gazebo does not run on my machine locally and there is hardly any online alternative that provides a similar functionality, the goal was morphened into something else.

Now the goal was to consider the image as a map, assume each pixel to be a node, and then plot your path from one node to another, where the coordinates have been randomly chosen. And the A\* algorithm retained its place, thus also retaining the aspect of challenge intact in the task.

3

**Overall approach and implementation in abstract**

In this section, I talk about how the assignment works from a bird’s eye view. Therefore, I’m just going to skim over the modules and give a casual introduction and an abstract for helping you to get a holistic picuture.

**Fetching terminal commands**

The assignment statement states that the order to insert starting point and goal should involve terminal commands and thus this module is the first thing in the program system to execute.

The module can be considered divided into two parts, namely, defining the input logic and then adding conditional logic in case if one enters irrelevant commands.

Also, the conditional options in the module ask you to refer to the array.txt file that consists of all the coordinate points with value 1. The reason here is to ensure that you don’t enter any coordinate point with a value 0, which signifies black, an obstruction on the map.

The module uses an argument parser module ‘argparse’.

4

**Processing the image**

The provided image was that of a railway station, and perhaps held a position as a cleaning robot simulator input for testing by Pepperming Robotics.

The approach was simple; to convert the image into a threshold version of itself, also called as a ‘binarised image’, in order to differentiate between free spaces and obstructions. The white spaces were the former and black spaces and markings signifying the later.

Binarising the image was only the first step, because the program also required an input expressed as standard ASCII characters. And what else could be a better way than to use nested Python lists consisting of 0s and 1s(although I could have also gone for 255=white and 0=black).

The module returns a map, converted from a thresholded image into 0s and 1s, coordinates of only white space pixels and the image itself.

5

**Heart of the program: The main processing file**

The final module consists not only the main ‘search’ algorithm, but also few other functions complementary to it.

The first function initialises nodes by getting its coordinates as values and then returns the start, goal and current nodes. This output is used as input for the main function and the current node in the function that follows.

The next function defines neighbours and boundaries for pixels on the edge of the binarised image. This function might look redundant or inefficient, but is the only way I could see to handle values that lie on the breadth-wise and length-wise ends of the image.

However, the main goal of this function is to identify and build the eight neighbours or ‘children’ of the target pixel/node so that we can follow with calculating decision making parameters such as f, g and h values.

The next function make the core calculations which define the A\* algorithm. It chooses various neighbouring and not neighbouring nodes based on certain conditions, and calculates their f value by adding the g and h values, while taking into consideration whether a given node was worked upon or not. The node scoring the smallest f value amongst the chosen nodes qualifies as a “successor” to the current node and the process goes on until the f values of the current node and the goal node are the same(the coordinate values of their respective pixels).

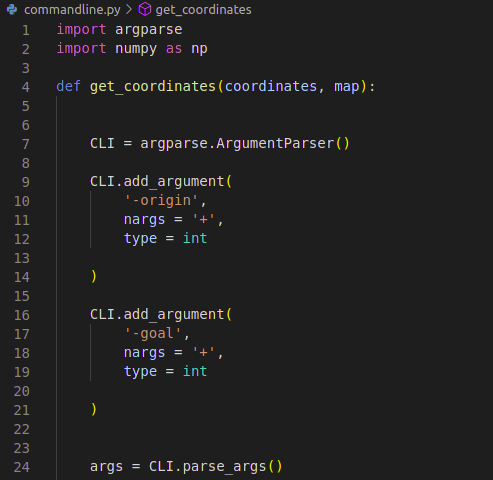
The next function is merely a routine which compares the coordinates of the current and goal nodes in a while loop, essentially doing what’s mentioned above. This function moves in cycles of changing the current node values and then adding and removing them from the ‘visited’ and ‘path’ lists.

The final function implements the graphic rendering of the output with Matplotlib library’s pyplot module.

6

**Understanding program flow module-by-module**

Command Line arguments with **commandline.py**



The above snippet explains the logic behind the creation of command line arguments as input parameters to the program, where the argparse library is used to create the ‘argument’ object.

The **get\_coordinates** function implements the main functionality by taking the coordinate values and a map expressed in binary form, in order to:

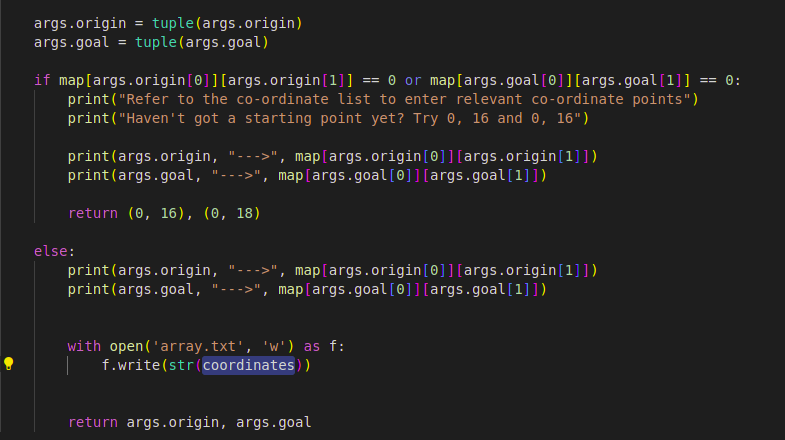
7

1. Collect all legitimate(values with 1) coordinates and write them to a file.

2. Determine whether the input coordinate values are legitimate by comparing user input with the values inside the file.

Next, we create an argument parser object called CLI(command line interface), which enables us to call the built-in add\_argument function and then begin parsing the arguments.

In each add\_argument function definition, the strings in the first lines are identifiers that help us understand the nature/purpose of an argument. The nargs paramter determines the type of argument where ‘+’ meaning any iterable, a tuple in this case. Finally int is the data-type of each object in this tuple.



args.origin and args.goal are two argument objects that have been logically defined.

The conditional statements that follow prompt the user to enter node coordinates that are legitimate by comparing whether their values are 1 or 0. While the former goes ahead with further execution, the later choice

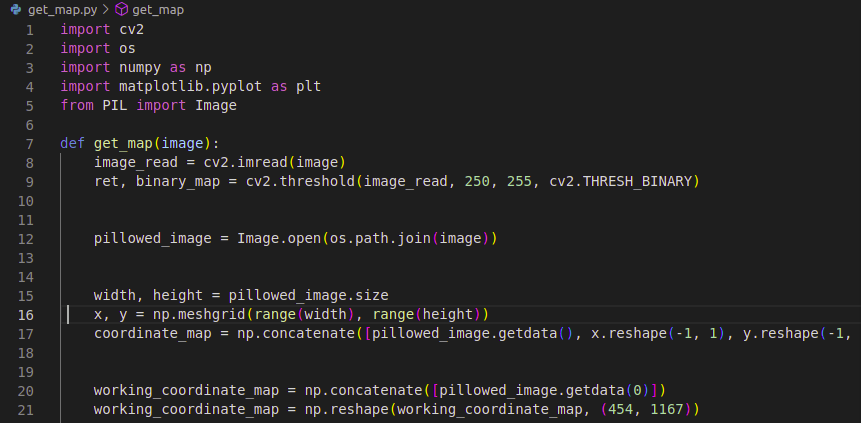
8

shows an ‘error’ message and then inputs start and goal values of (16, 0) and (18, 0) respectively.

9

Processing the map file with **get\_map.py**

I would call this module a preprocessing stage where the image is prepared to be converted into a nested array of 1s and 0s.



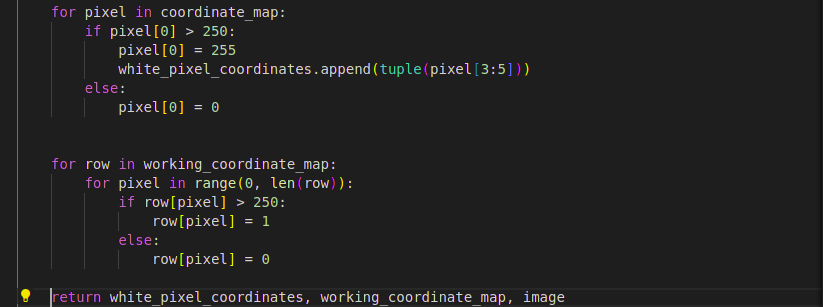
OpenCV reads the image and converts it to a threshold version. A threshold image or a binary image is where the image pixels are either black or white, depending on their colour intensity. To create a binary image, first you need to ‘grayscale’ it.

Then we read the image into variable pillowed\_image, after which we extract its dimensions and create a grid like object’s x and y axes, based on the width and height of the image.

Line 17 creates a list of coordinates from the image, by concatenating the image data(likely pixel intensity values I believe) with reshaped values of the image grid.

Lines 20 and 21 create another numpy array where we fetch and concatenate the same data, in this case, only the pixel value intensities at index position 0 and then reshaping this array as per the shape of our original input image.

10



The first for loop here loops through the coordinate\_map array, sets all values over 255 to 255 and all others, which are almost entirely pixel intensity values below 10. The goal here is to extract pixel intensity values and coordinates of pixels valued as 255 and discard the rest. This helps us group together all pixels that are white.

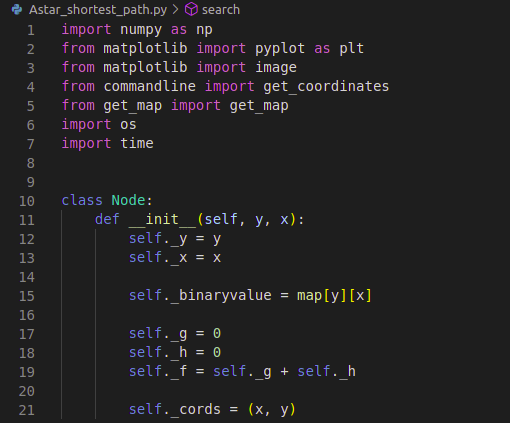
The second loop, does the same thing as above, only using 1 instead of 255, while not discarding the 0 value pixels. This is the main binary representation of our image on which we’ll be working.

We return the coordinate values of white pixels, the array representation of our map and the image itself. The reason we return the image too is because we’re going to need it later to draw the path that has been traced.

11

Heart of the program with **Astar\_shortest\_path.py**

This is the core of our program and thus the final point of understanding it.



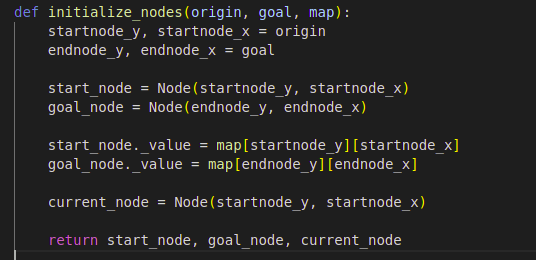
After making the necessary module and function imports, we create a Node class because we need standalone data entities that we can attest to, values that represent each pixel or node.

While x and y coordinates are obvious, we also need to define the binary value of each node which will later help in filtering the wanted nodes while looping.

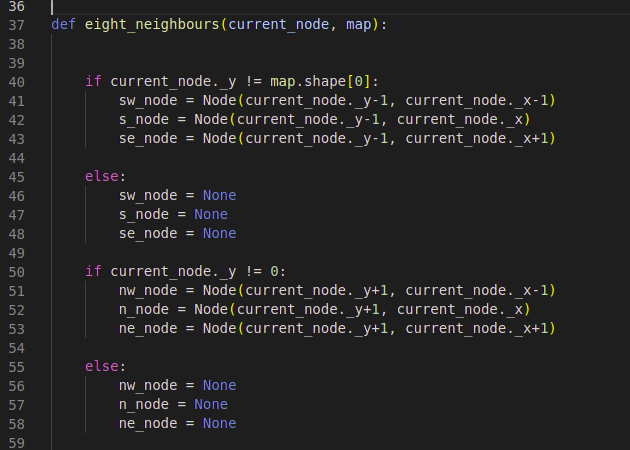
We need f, g and h to calculate the necessary values required for calculating the distance score between start and the current node and from current node to the goal node.

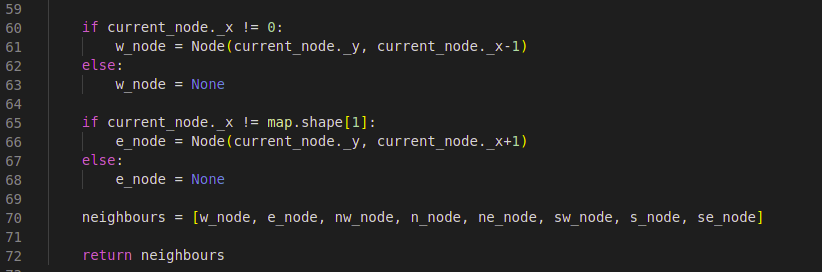
self.\_cords can be used for comparisons where only the x and y coordinate values are required to be displayed or compared.

12



To run the program, we need to create Node entities, which are primarily defined by their positions on the map as well as their pixel intensity values. Once the main entities are defined, we are all set to use them throughout rest of the program.

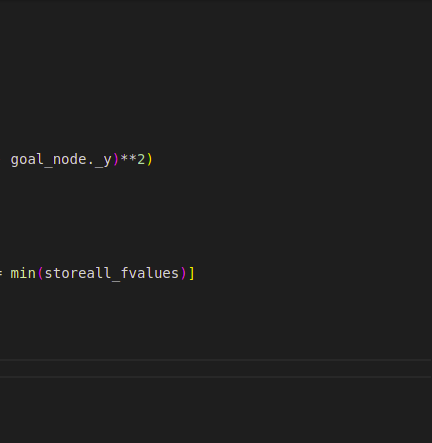
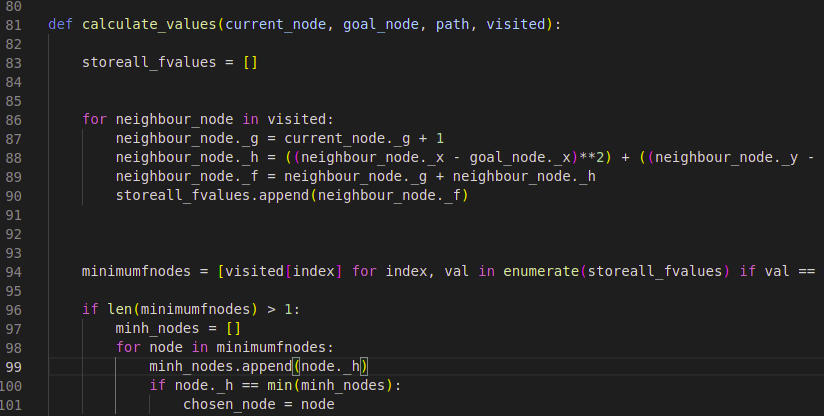




13

The functioning of the A\* algorithm dictates that we should assess the f values of eight neighbouring nodes around the given current node and choose the one that has the lowest f value of them all, as the current node.

The function above is helpful when we have the current node with all its data ready and its just calculating the child nodes that remains to move further.



The above function takes the current node as the pivotal point, the goal node to calculate the f distance, path and visited lists.

Before moving further, I’d like to clarify an important aspect of the algorithm. A\* dictates the use of two lists or array-like structure, generally defined as open and closed(defined as visited and path) respectively.

14

Visited list contains nodes that we have been through i.e. the eight neighbours, while the path list consists of nodes that have become the successor of the previous ‘current node’(and hence became the new current node).

A node that qualifies for the path list, is freed from the visited list.

In the above snippet, we loop through the ‘visited’ list and find the f values of each neighbouring node by calculating the formula

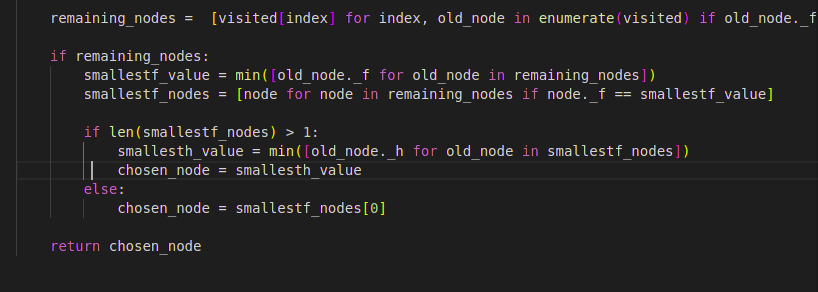
f = g + h, where

g = node per node distance from the starting node

h = The Heuristic distance calculated between the goal node and the current node

For calculating the Heuristic value we have used the Euclidian method, as opposed to Manhattan distance method.

After we are done calculating the f values, we append them in a list and on line 94 we calculate the node with the minimum value amongst all neighbours. However, if ever we have two nodes with the same f value, we have provision from lines 96 to 105 to extract the node with minimum h value and set that as the next ‘current node’.

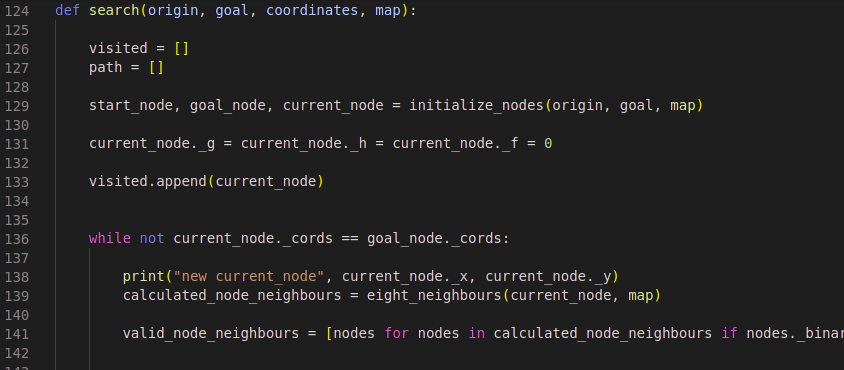


15

The snippets between lines 108 and 120 repeat the above procedure but with the remaining neighbours in our visited list. This is important since it may happen that we have found the next current node, but the visited list has already certain nodes that have a lower f than the new current node?

These ‘remaining neighbours’ we talk of here are the neighbouring nodes of the previous current node.

In addition to that, we do the same thing i.e. calculating the minimal Heuristic or h value if there are two nodes with the same minimal f value. The one with least value will be chosen as the current node.



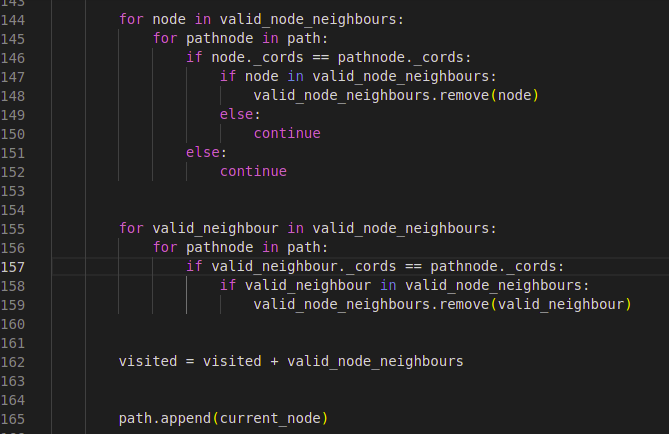
This is the main search function that does the job. All the functions that we had defined before are called here. Note those two empty lists initialised in the beginning; we use the visited list to store the neighbouring nodes of a current node and the path list to store all the subsequent current nodes.

After initialising the start, goal and current nodes, we establish explicitly that the f, g and h values of the current node are 0. Also we add the current node, which is of course the start node, to the visited list.

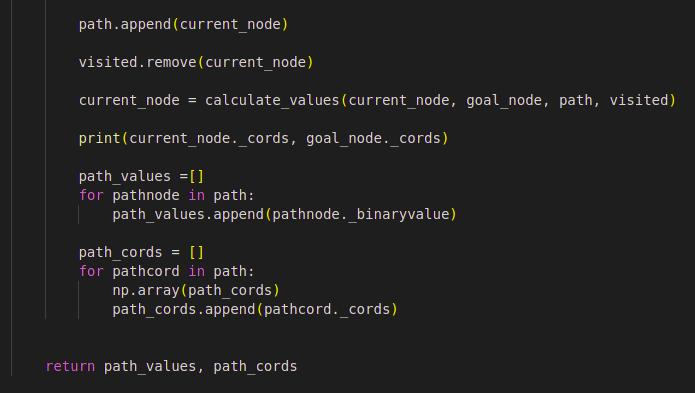
The while continues its execution until the coordinates of the start node are equal to the goal node coordinates.

16

Then we calculate its eight neighbouring nodes and then filter out the ones that have a binary value of 1.



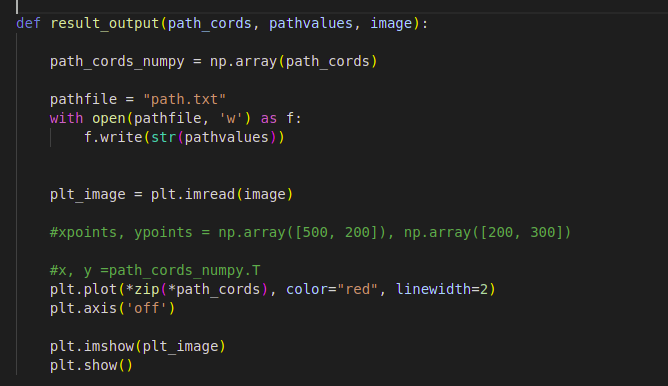
The two for loops in the above ensure that the chosen coordinates are present neither in the visited nor the path lists, so that they can qualify to be new current nodes. Once qualified, we append them to the path list.



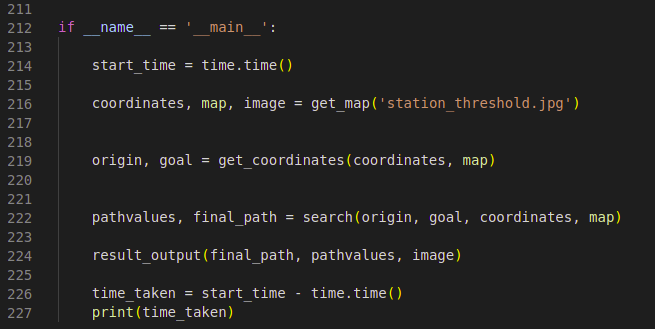
16

Then we call the calculate\_values function to find the next current node and print the coordinates of the current node and the goal node for keeping a track on the terminal.

Lists path\_values and path\_cords are to ensure that our path has purely been only through white pixels as well as getting the coordinates of all the pixels through which our path passess.



In the above function, we write the coordinate path values in the path list to a file path.txt and then proceed to plot the coordinate values in the path\_coords list on the map.

18

In the above and final snippet, we call all the key function where we:

1. use the get\_map function with the target image as the input and get coordinates, binarized map and the image itself.

2. get the coordinates from commandline.py as terminal input

3. implement the core search function and plot the result on a copy of the image.

**19**

**Test cases and performance**

For testing the algorithm evenly, so that there is no performance bias for a narrow set of coordinates on the map, the selection of coordinates was done in following ways.

1. Coordinates from either vertical ends of the map

2. Coordinates from either horizontal ends of the map

3. Coordinates from top and bottom ends of the image map, but not on the edge.

4. Coordinates with single digit value for either x or y of either coordinate values

For the first two cases, the results have been 50% of success since the white pixels on all the edges of the image map, are flanked by black pixels, which make it difficult for the algorithm to trace next white pixel and avoid the obstacle/black pixel.

For cases 3 and 4, the results are 100%.

Out of 20 tests, the algorithm delivered perfect results for 17, with the other 3 being from the edges of the image map.