Path Planning with A* Algorithm

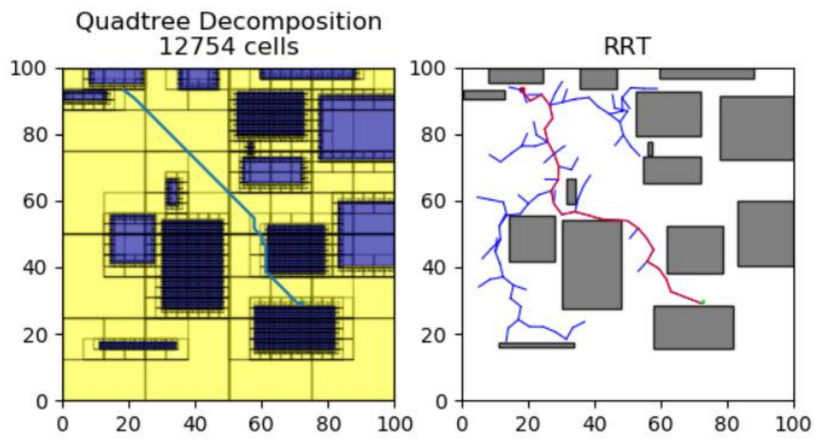
A* search is implemented in aStar.py file.

- **AStarSearch** class is implemented, which is to be used in “fpsb path planning.py” file’s main function.
- **Constructor** of **AStarSearch** takes in “**root**” node of the completed Quadtree, **initial** rectangle and **goal** rectangle.

Functions

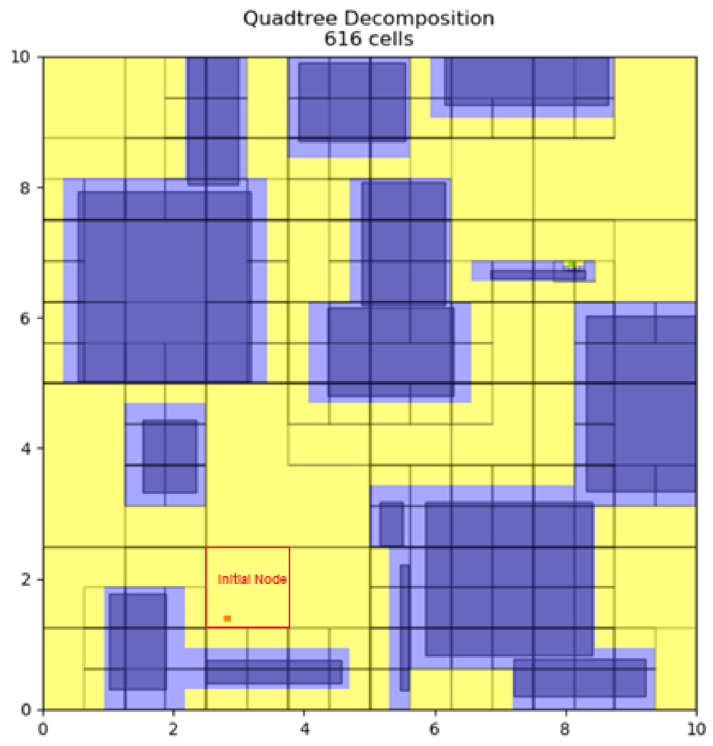
1. find_node(Node curr, Rectange rectangle)
   Finds the node that contains the rectangle and returns that node using A* algorithm
2. calculate_g_cost (Node from, Node to)
   a. Computes the G cost between two nodes.
3. Calculate_h_cost(Node curr, Rectangle rectangle)
   a. Computes the H cost between two nodes.
4. Priority_remove(Node [] node_list)
   a. Finds the node with the lowest F cost (G + H) and returns it.
5. Exists(Node curr, Node [] nodes_list)
   a. Returns true if curr exists in nodes_list. Otherwise false.
6. Reconstruct_path(Node curr)
   a. Returns the path to the goal node in a list
7. Calculate_path_length(Node [] path)
   a. Calculates the length of the given path in cm.
8. Generate_leaf_nodes(Node node)
   a. Takes in a root node of the quadtree and stores all the “free” leaf nodes in self.leaf_nodes list.
9. Generate_neighbours(Node curr)
   a. Takes in the current node (curr position) and finds the neighbouring nodes using AABB algorithm. https://www.youtube.com/watch?v=ghqD3e37R7E. for more information about AABB collision detection.

Output: As obviously observable, the program is finding its path to the goal.

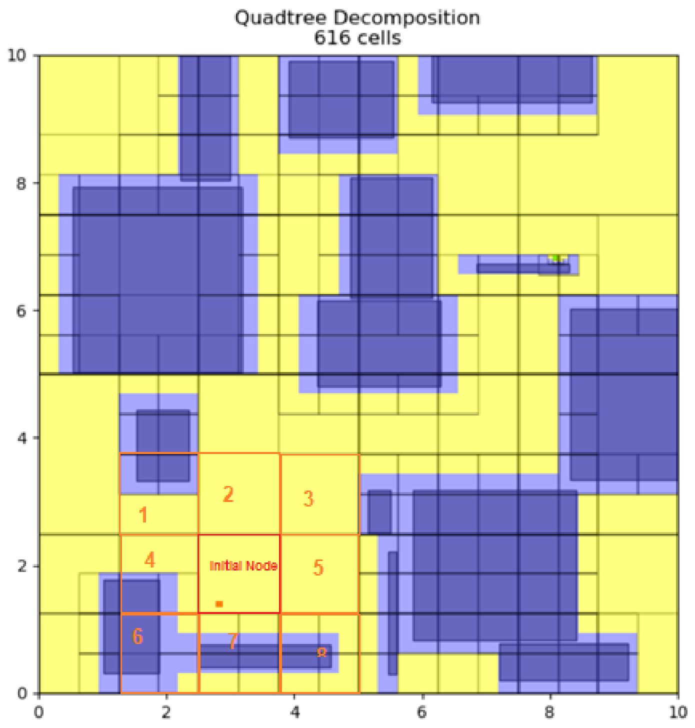


A* Algorithm with Manhattan Distance method

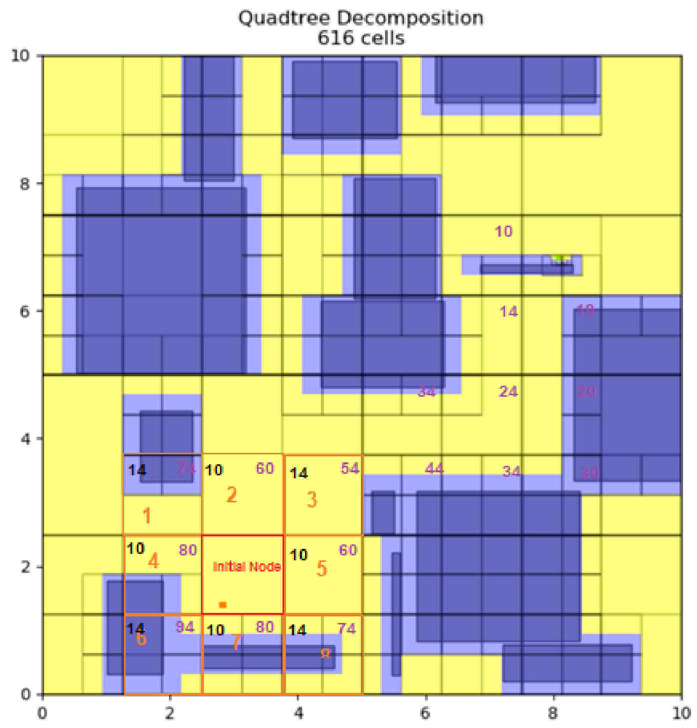
- Detailed explanation with appropriate figure is included.



1. open_nodes = [(0, **free**)] & closed_nodes = []



2. We make the children nodes dynamically around the initial node.

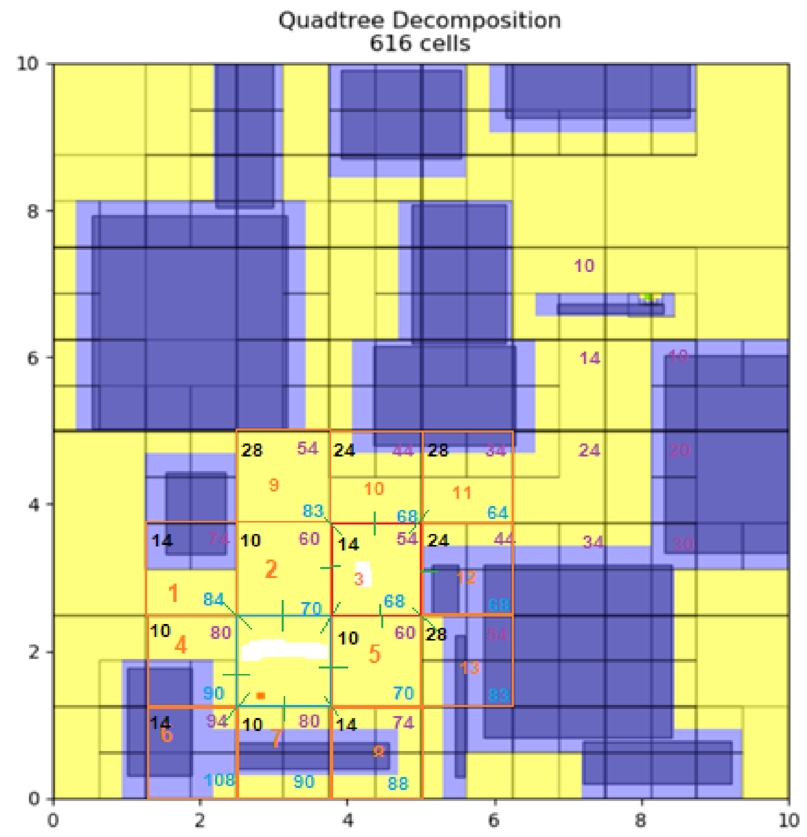


3. Calculate G and H cost. F = G + H. In the next while loop, we are picking the node with the lowest F cost. In here, G cost represents the distance from the starting node and H cost presents the distance to the goal node.

Guess? Yes, we are moving to 3rd children of the initial node.

Then,

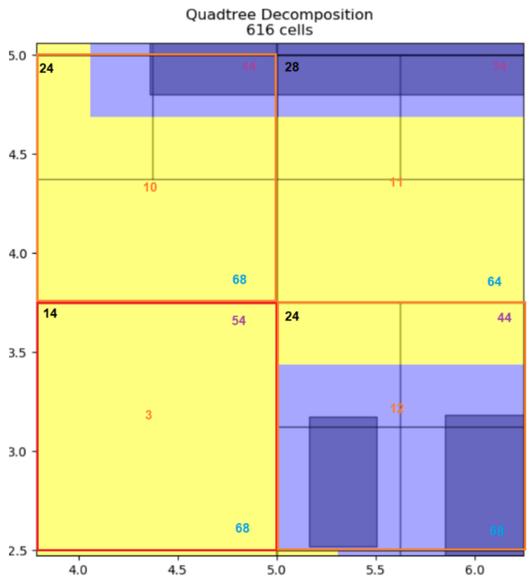
- open_nodes = [(1,**mixed**), (2, **free**), (3, **free**), 4(**mixed**), 5(**free**), 6(**mixed**), 7(**mixed**), 8(**mixed**)]
- closed_nodes = [(0,**free**)]



4. Now that we moved to the previous node's children no.3, we repeat the same process. Calculate g and h cost for each and calculate f cost.

As you see, I have calculated all the costs. And the lowest f cost is 64, which is children in cell 11.

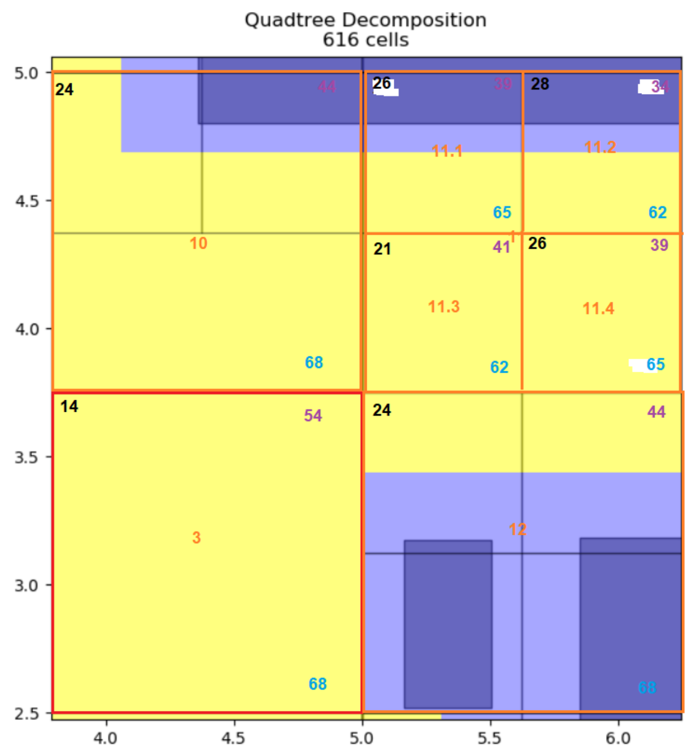
Now this is the special case. In the precious movement, it was a **free** cell, but our next move has to be **mixed** cell as calculated by the Manhattan distance.



Picture on the left side is an enlarged cell in order to describe how we would tackle this special case in detail.

In my understanding, we would have to identify the **free** space within the **cell 11**.

Like what we have done for **quadtree**, let's divide the cell 11 by 4.

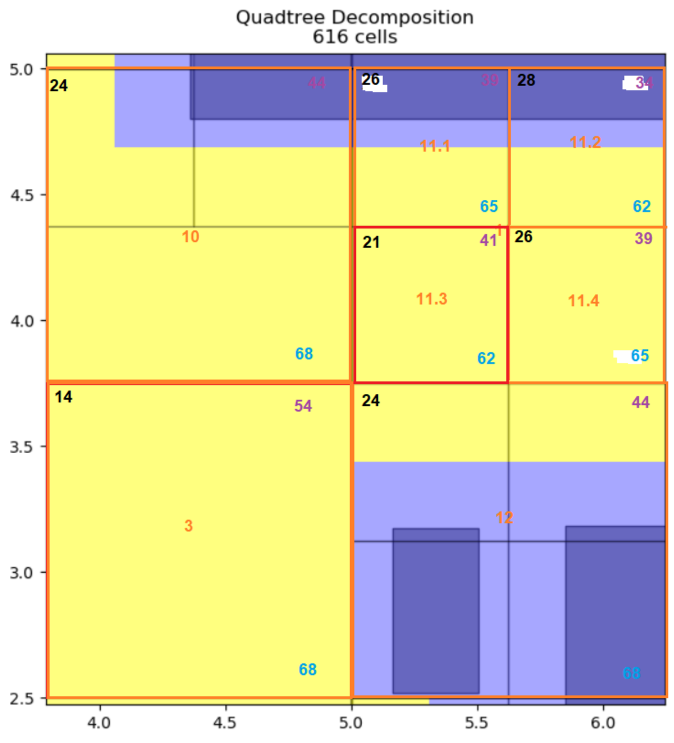


So, if we recursively divide it, we can this. The base case for this recursive function should be

if(curr == “free” && curr.getFCost() is Minimum of all children && get_distance(curr, parentNode) is Minimum of all distances between parentNode and other children)

return curr;

Then, we can move onto 11.3 as it is the **lowest f cost**, **free space and closest to the parent cell 3.**

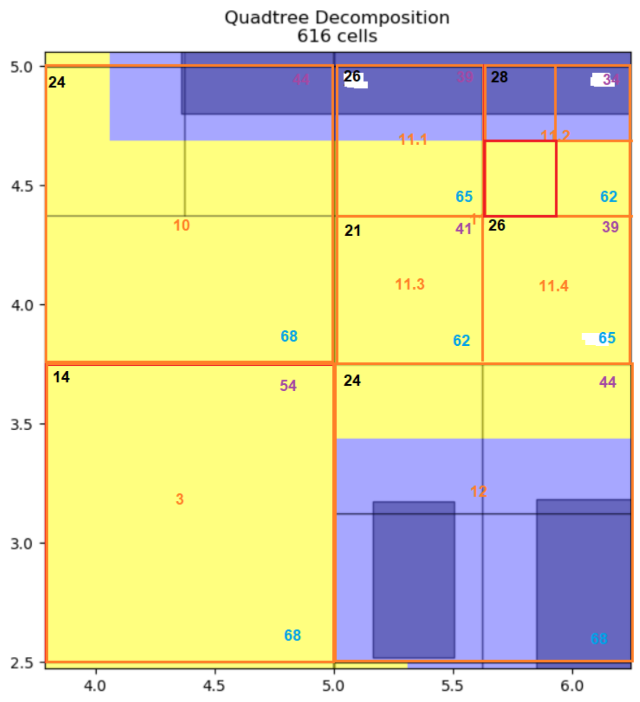


Now we have moved to cell 11.3. Our cell size has become smaller (a quarter). Length and height are now halved.

Since we are still in cell 11, and we already know about the other smaller cells within cell 11, we can move to the cell with the **smallest f cost** within **cell 11** again. We don’t need a further recursion.

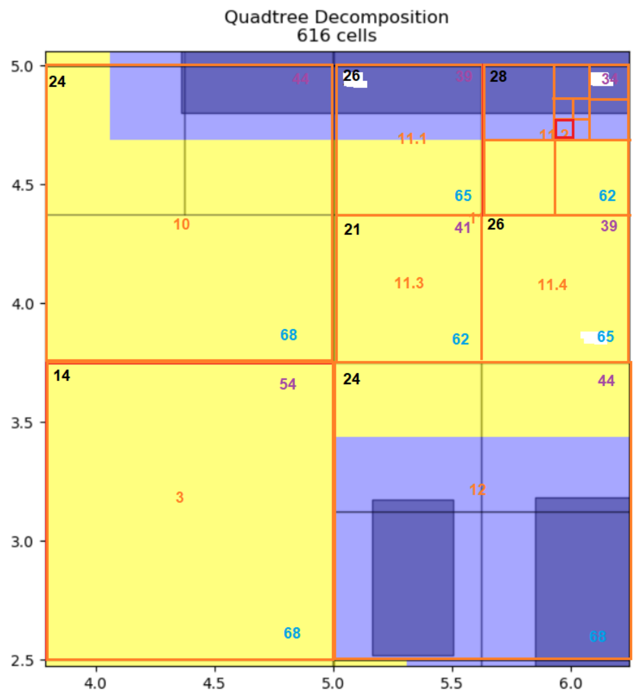
So, we are moving to 11.2 next within cell 11. We don’t even have to consider cell 10 or cell 12 because any f cost within cell 11 will be lower than other neighbouring cells of cell 11.

However, cell 11.2 is a mixed cell. We have to recursively divide the cell again then.



So now we moved to closest cell within 11.2 from 11.3. Guess what happens next? We again move to the most promising movement.

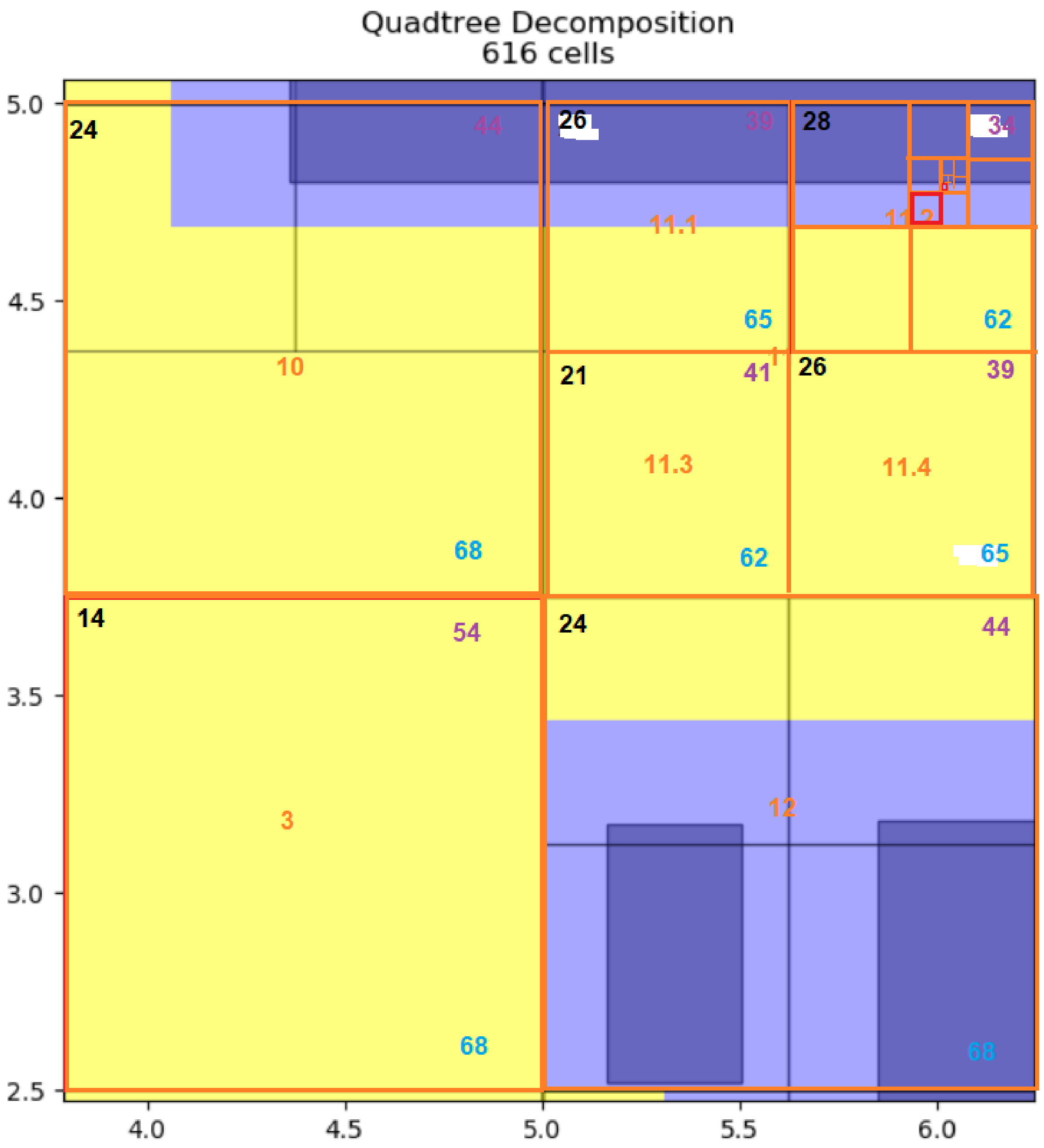
We are moving to the top right cell within 11.2 again. Since that cell is a mixed one, we need to recursively divide until we find a free space



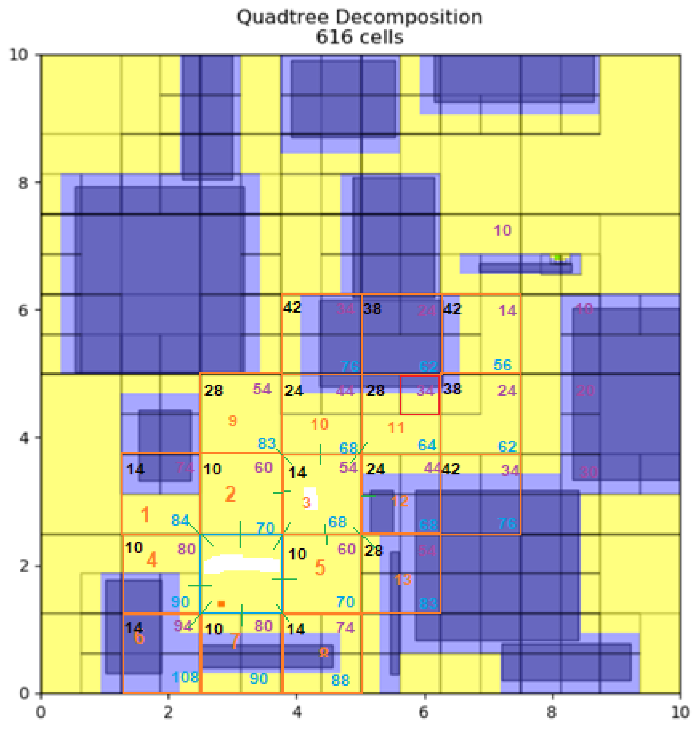
As you see we have moved to the closest cell from 11.23. Then we again make the next move by calculating the comparing the lowest F cost.

Guess?

Yes, we are moving to the top right cell within 11.233.



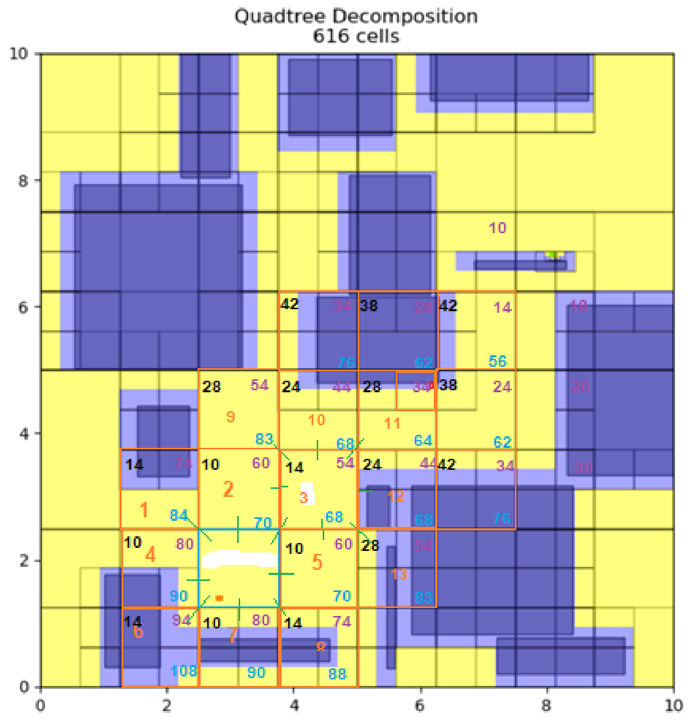
So, we are in the tiny little free space now. I will skip a few steps as it is quite clear what's going to happen next. We are moving to the cell on the right hand side until we reach the border of cell 11. Remember that this all happened in the cell 11



Here comes another special situation. Our current node's size is smaller than the original size (initial node's size we began with). However, we should be able to detect that we are at the edge of the cell 11 because we know the **width** and **length** of the **cell 11**. Thus, we would normally find the neighbours of the cell 11, but not the neighbours of smaller cell in cell 11.

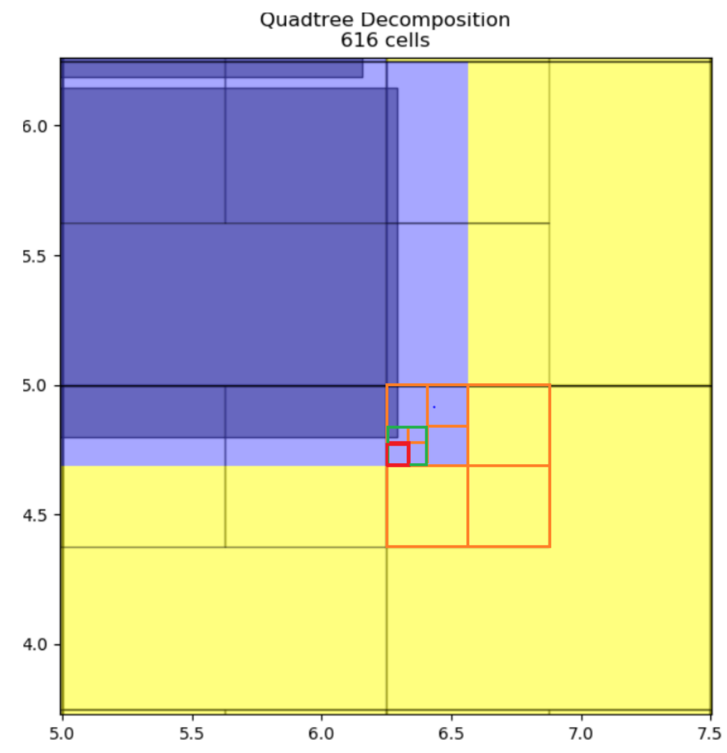
As you see from the figure on the left hand side, you may observe that we need to move to top right as it has the lowest f cost.

We first need to check if we can move this way in this situation. Since there is an obstacle, we cannot move there diagonally. So our next best move is to move one step right.



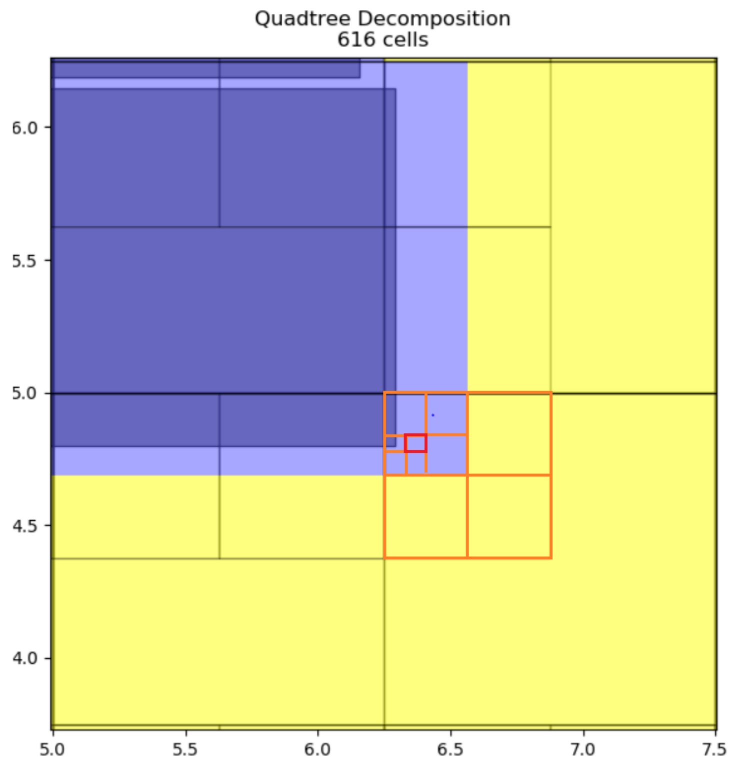
Import question in here is how we detect if we can really move diagonally. In my understanding, checking whether there exists any solid obstacle within top right corner of cell 11 is how we detect it. More in detail, we have recursively divided all the cells within cell 11. We should be able to retrieve in this tree to see if top right corner of cell 11 has a solid "**obstacle**". Otherwise, we can move diagonally. In this case, since there is an obstacle, we can't move diagonally, but only horizontally.

Another problem is cell to the right of cell 11 is a mixed cell. As we did before, we recursively divide it to find the optimal path.



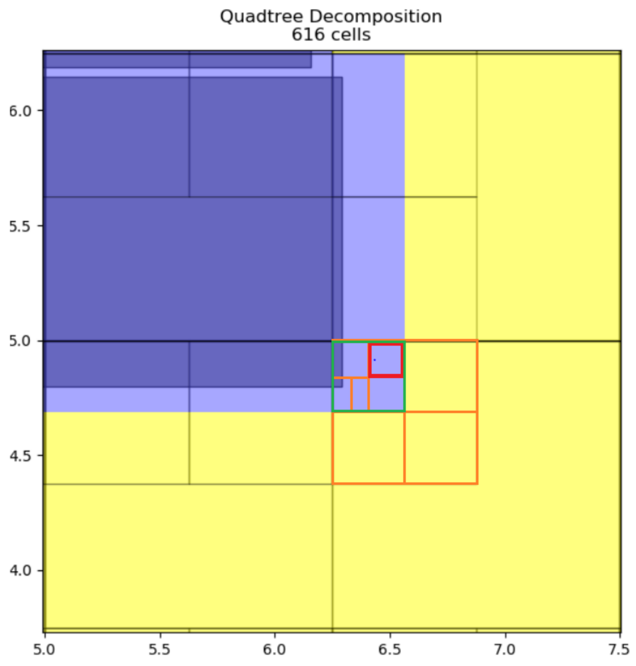
So, as expected, after the recursion, we chose the most promising path. This is complicated, but it should be easy to understand once you see the pattern.

Now what happens next? Let's cell the current cell **"cell 14"**. Since we are with the cell 14 (another recursion function), we are only going to move within cell 14. In the current node (red box), We are going to observe which cell is the most promising. Until we get out of one step larger cell(green one), we can't move straight to another cell within cell 14.



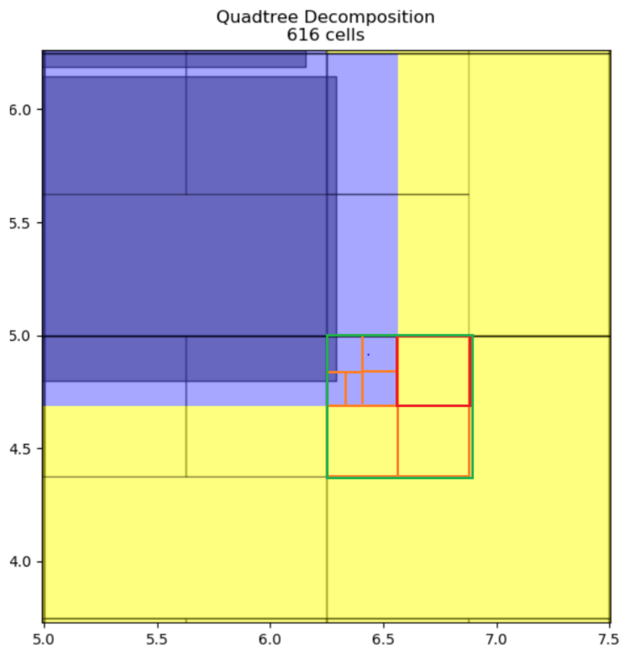
Here is the next promising move within the green box.

Since there is no more promising cell within the green box, we can return to the original recursion and find the next promising move within one size bigger cell.



Now, again since there is no more promising cell within the green cell, we are going to look at the whole cell 14.

As you expect, it will move to cell to the right.



Since there is no more promising cell even in the entire cell 14, we will now move on to the next cell finally. This covers all the special cases.

Downside: If the initial node has a bigger cell size, we would be waste our time to divide all the cells in a smaller size. We need a better heuristic to determine the initial cell size.