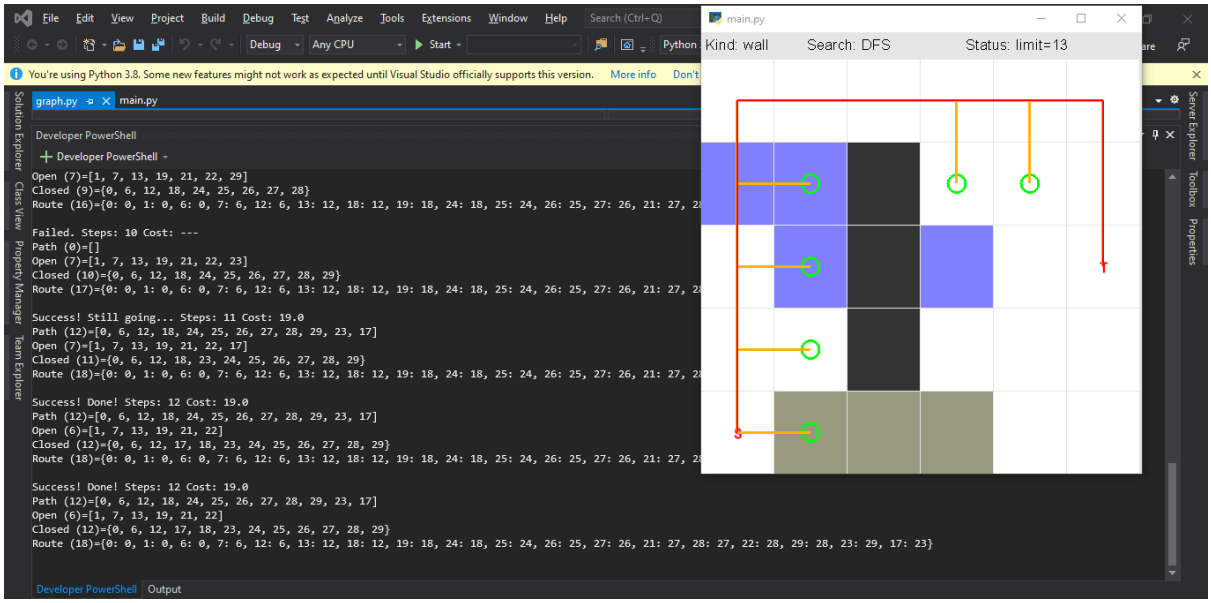
Objective of doing this task:

To understand and utilise Depth First Search (DFS), Breadth First Search (BFS), Dijkstra's and A* Search algorithms applied to a simple "box" based world, while also trying to find which is the most effective at which time or depending in which situation each can be used.

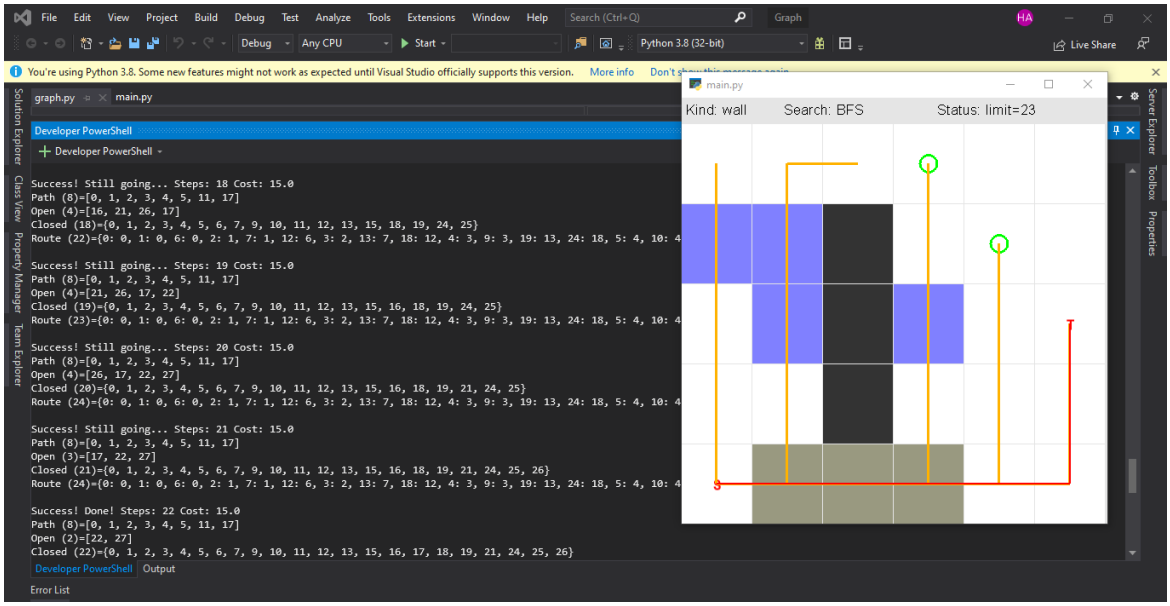
Tasks Carried out for this lab are under as follows:

Running using map1.txt to get the gist of Search algorithm in GUI view which is more appealing to look at than boring CLI.



As it can be seen in the above screen capture that while manually going through each steps using the arrow key to search a path from start position to target position using DFS (Depth First search) while also avoiding the hindered path it determines that the total cost to reach from start node to target node would be 19 and even though the target path was already determined at step 11 the algorithm kept on going to search for more path and at Step 12 concluded no more node left to target.

Now, before we go any further it is important to know what DFS search algorithm means or rather how it even works, so here we go! The DFS (Depth First search) works by going further from the root node to all the way to leaf node and it does that by Traversing one branch fully before moving onto the next one until it reaches the end node. So, what is its benefit? Why we use it? Well honestly it depends but for most part it has lower memory requirements than BFS (breadth-first search) which is another search algorithm which works in a way that it is totally opposite to DFS, so what BFS does is that it traverse the as close as to the root node and what it generally means by that it spreads out uniformly from the start node/root node and grows exponentially, well the thing with BFS is that it has poor time and space complexity, so the solution it produces is not generally the most optimal one and the longer it runs the worse the search gets, similarly with DFS the problem is if the solution is on the right hand side and we start from Left then we're definitely in trouble. Most importantly DFS follows the Stack LIFO while BFS follows the FIFO Queue implementation, and with that BFS will guarantee a result of the shortest possible path length to target, the comparison can be seen below and compared with above DFS.



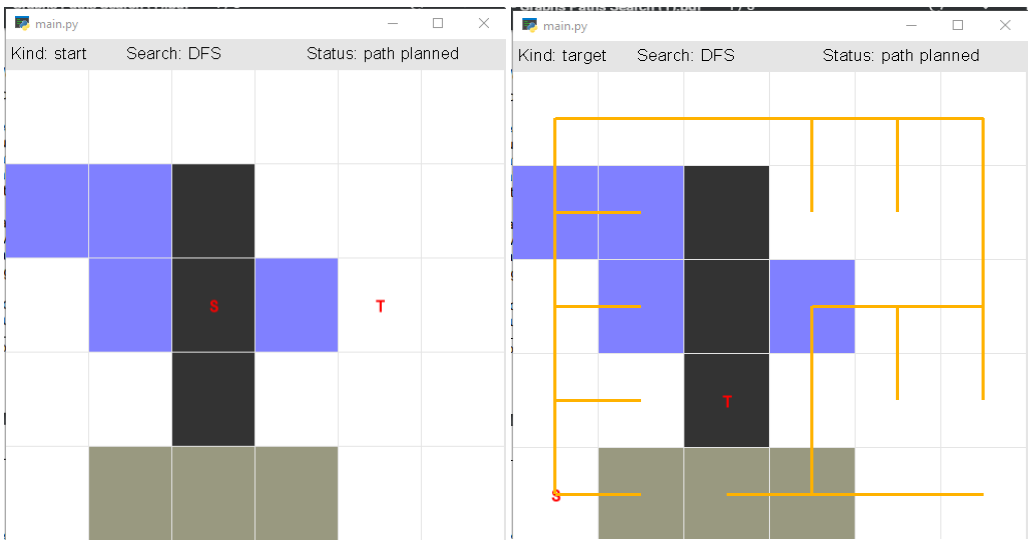
From this picture we can see that BFS has lower cost than DFS, plus BFS was able to find path to target at Step 18 but still kept going on to look for more shorter paths to target while DFS found the target at Step 11 and at Step 12 it concluded no more paths to target.

Based on the Comparison I can deduce that BFS is more better when you have to consider your neighbouring characters or nodes, which is why it can be implemented in a Character game since it is better when target is close to source.

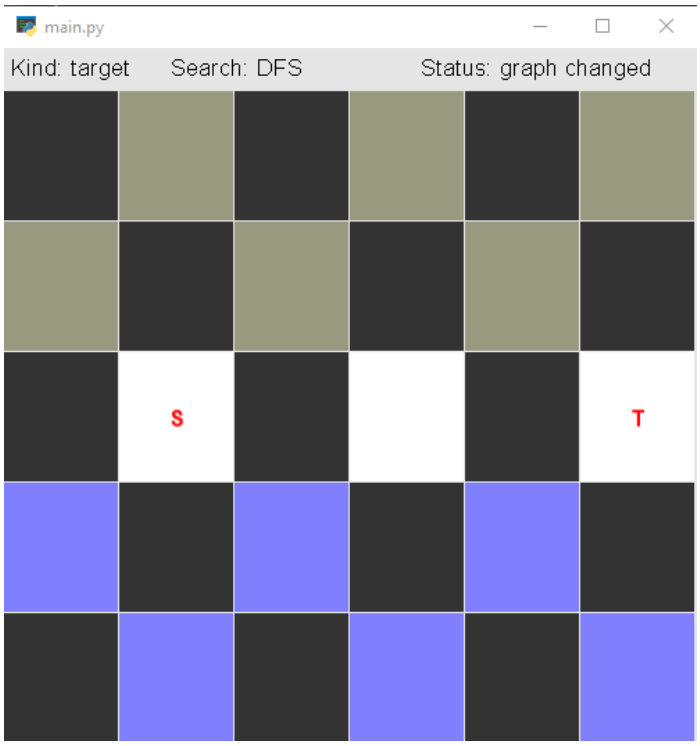
Types of box on the box world:

1. Clear ---- white box
2. Mud ----- grey-brownish box
3. Water ---- blue box
4. Wall ----- black box
5. Start ----- Start ‘S’ position on the grid
6. Targe ---- Target ‘T’ position on the grid

Note: You can edit box or modify world using numeric keys listed above. If set start ’S’ to wall box, no search will be performed and if Target ‘T’ set to wall box no search will be successful.

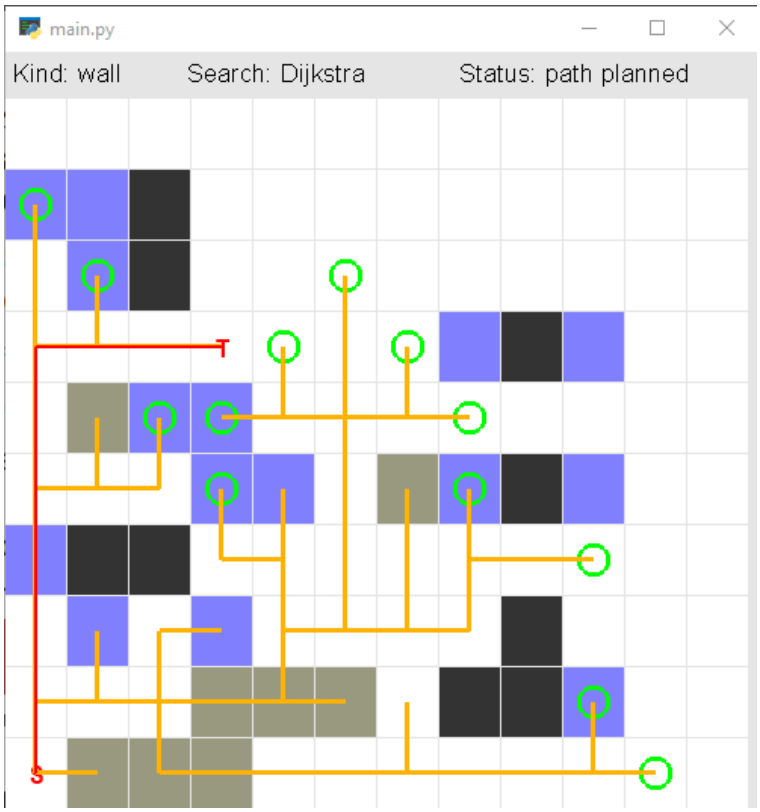


Example of Altered world:

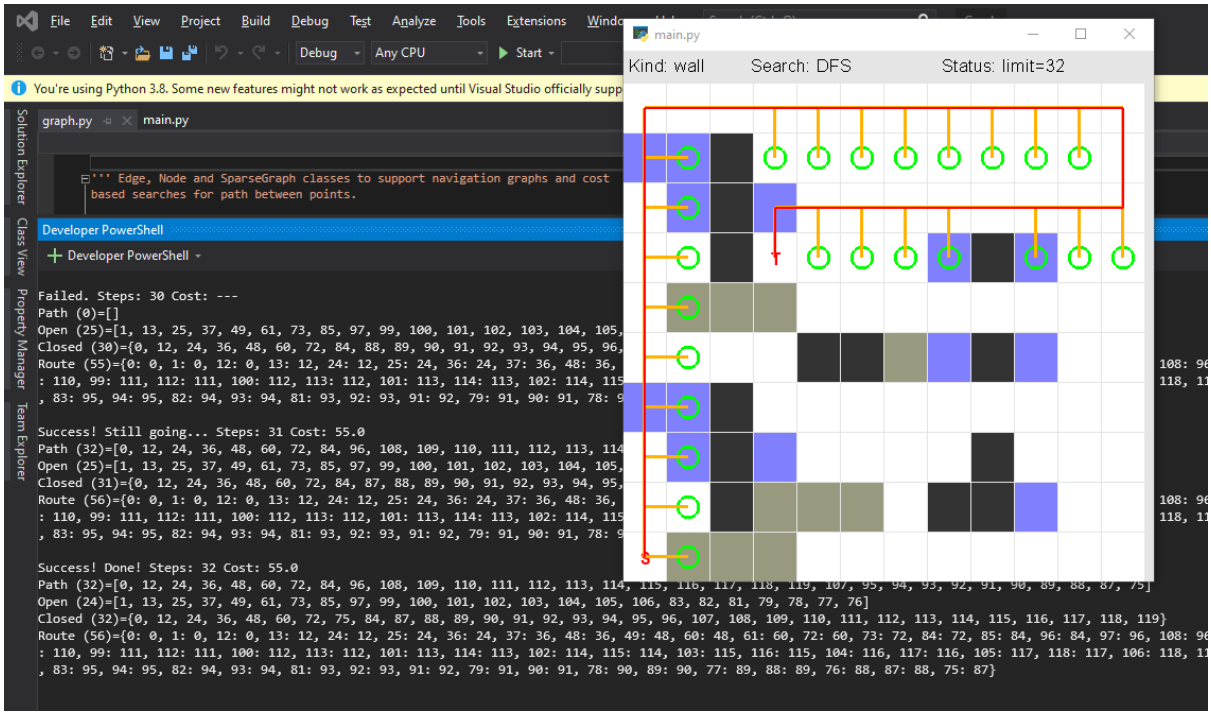


Personally I know it was a dumb move to put wall box everywhere since it would stop the search from even happening now unless we implement edge but this is just an example how weird (appealing) can modifying the world look like.

Using Map2.txt made changes to map of the world and hit Space to get a planned path immediately even after modifying the world. *Using the same Searching algorithm*

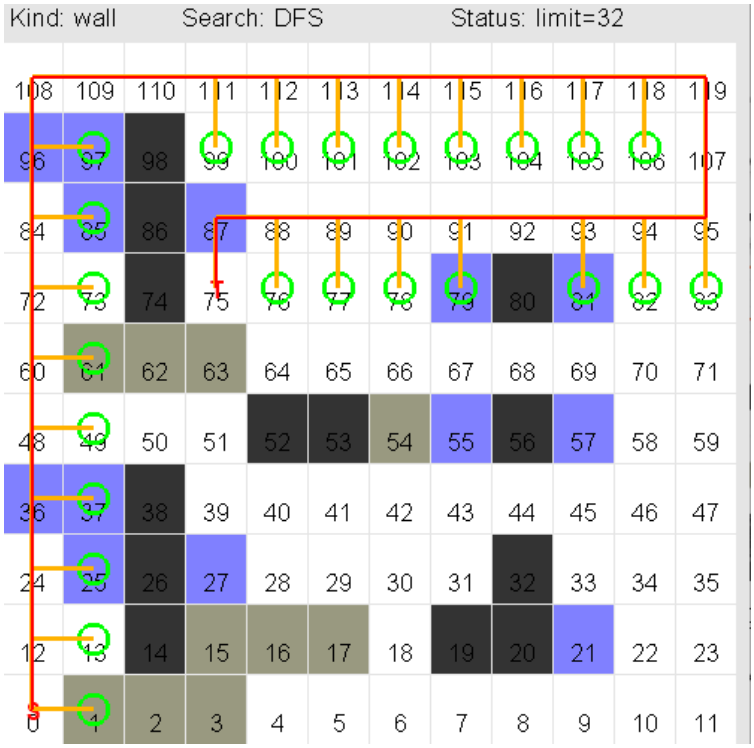


Remove depth/number of search steps taken Limit by using numeric key ‘0’ and then used Up and Down arrow key to toggle through the steps until the red line ‘Path planned’ appeared and as it can be seen that steps take were 32 on map2.



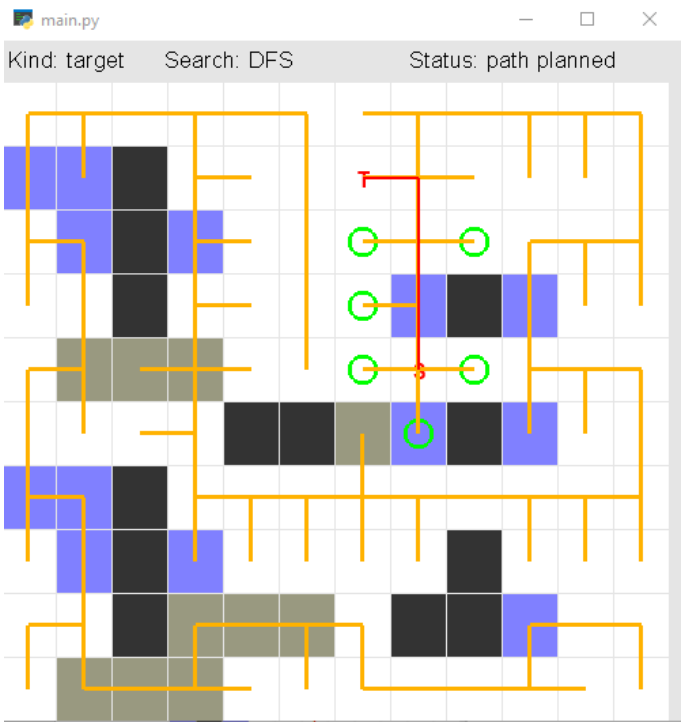
If you were to observe carefully you’ll probably note that even though our searching algorithm found the path to target they don’t say Done but keep on going this is because they look for alternatives like with BFS looking for shortest path.

You will also notice that Open and change list has some numbers which actually represent the box number as you can see below.

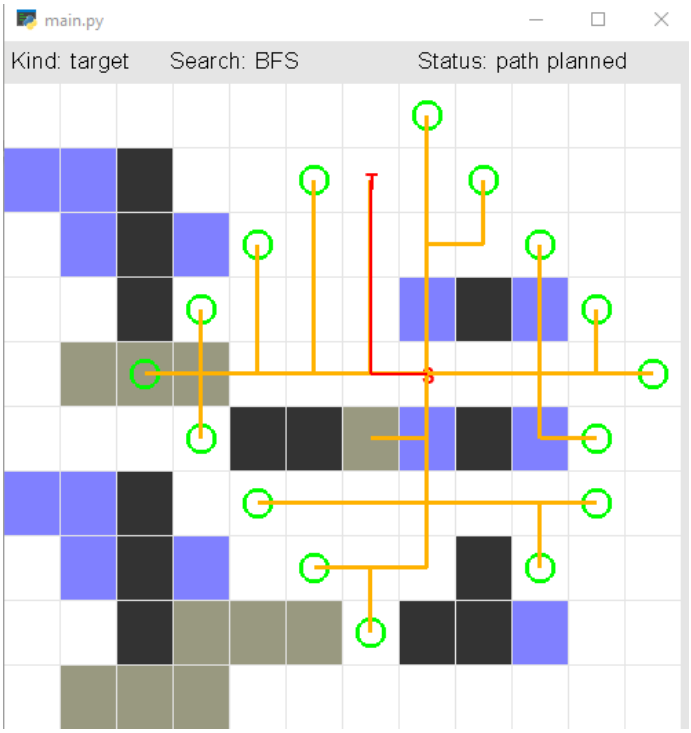


The Open refers to path which are open to take while Close obviously refers to wall, this feature can be accessed using the ‘L’ key.

Notice if you reduce the distance between the Start and the target, how differently the both Algorithm perform including the cost for both of them, this is because BFS finds the shortest path to target which costs low and as mentioned earlier BFS is good for neighbouring searches.




```
Success! Done! Steps: 101 Cost: 12.0
Path (5)=[67, 79, 91, 103, 102]
Open (6)=[55, 66, 68, 78, 90, 92]
Closed (101)={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
 51, 54, 57, 58, 59, 60, 61, 62, 63, 64, 65, 67,
, 108, 109, 110, 111, 112, 113, 114, 115, 116, 1
Route (107)={67: 67, 55: 67, 66: 67, 68: 67, 79:
 119: 118, 107: 119, 95: 107, 83: 95, 94: 95, 82
1: 43, 42: 43, 30: 42, 41: 42, 54: 42, 29: 41, 4
11, 112: 111, 113: 112, 101: 113, 89: 101, 77: 8
, 25: 37, 13: 25, 1: 13, 12: 13, 0: 12, 2: 1, 3:
```

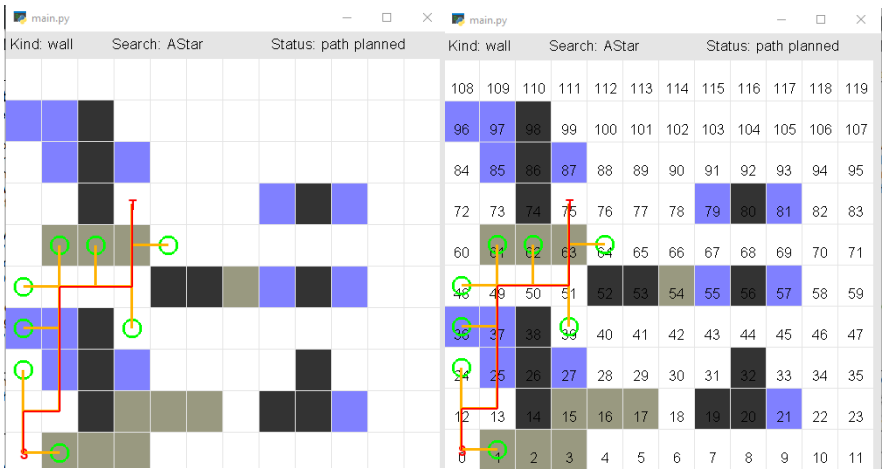



```
Success! Done! Steps: 29 Cost: 4.0
Path (5)=[67, 66, 78, 90, 102]
Open (16)=[58, 71, 82, 93, 104, 115, 18, 29, 40
Closed (29)={30, 31, 41, 42, 43, 44, 45, 54, 55
Route (45)={67: 67, 55: 67, 66: 67, 68: 67, 79:
: 91, 30: 31, 41: 42, 45: 44, 63: 64, 76: 64, 8
6, 101: 89}
```

Trying out Edge based cost search

The meaning behind cost edge search is the distance/edges from source to target with respect to cost, since we're using A* which is actually a priority queue algorithm and by that it means it gives priority to nodes that are supposed to be better than others using a heuristic function whereas Dijkstra explores all possible path so yes you can say that A* is just like Dijkstra but more efficient.

Using the pre-set value in the code:

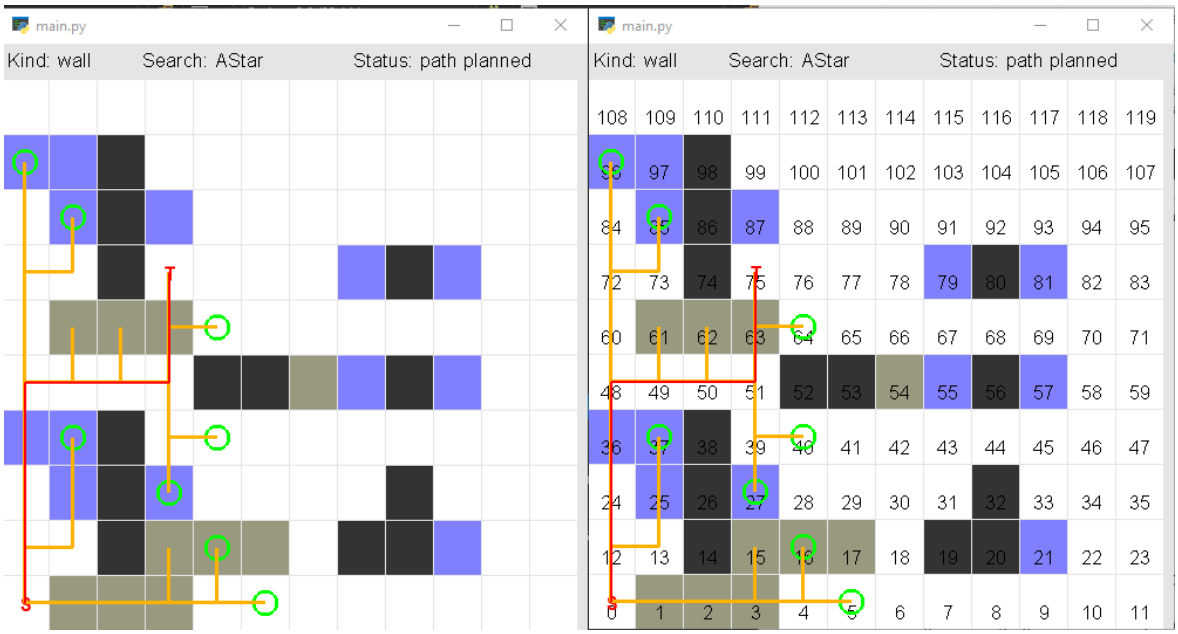


```
PS C:\Users\cools\source\repos\Graph\Graph> python main.py map2.txt
Success! Done! Steps: 10 Cost: 28.0
Path (10)=[0, 12, 13, 25, 37, 49, 50, 51, 63, 75]
Open (8)=pq: [(45.0, 13, 62), (48.0, 16, 64), (54.0, 11, 61), (55.0,
, 24), (73.0, 9, 48), (82.0, 1, 1), (87.0, 7, 36)]
Closed (10)={0, 37, 75, 12, 13, 49, 50, 51, 25, 63}
Route (18)={0: 0, 1: 0, 12: 0, 13: 12, 24: 12, 25: 13, 37: 25, 36: 3
 50: 49, 61: 49, 51: 50, 62: 50, 39: 51, 63: 51, 64: 63, 75: 63}
```

```
min_edge_cost = 10.0 # must be min value for heuristic cost to work
```

Value estimated on basis of comparison is 1.0 which reduces the cost from source to target but increases the step count as it takes on the alternate route from 12 – 24.

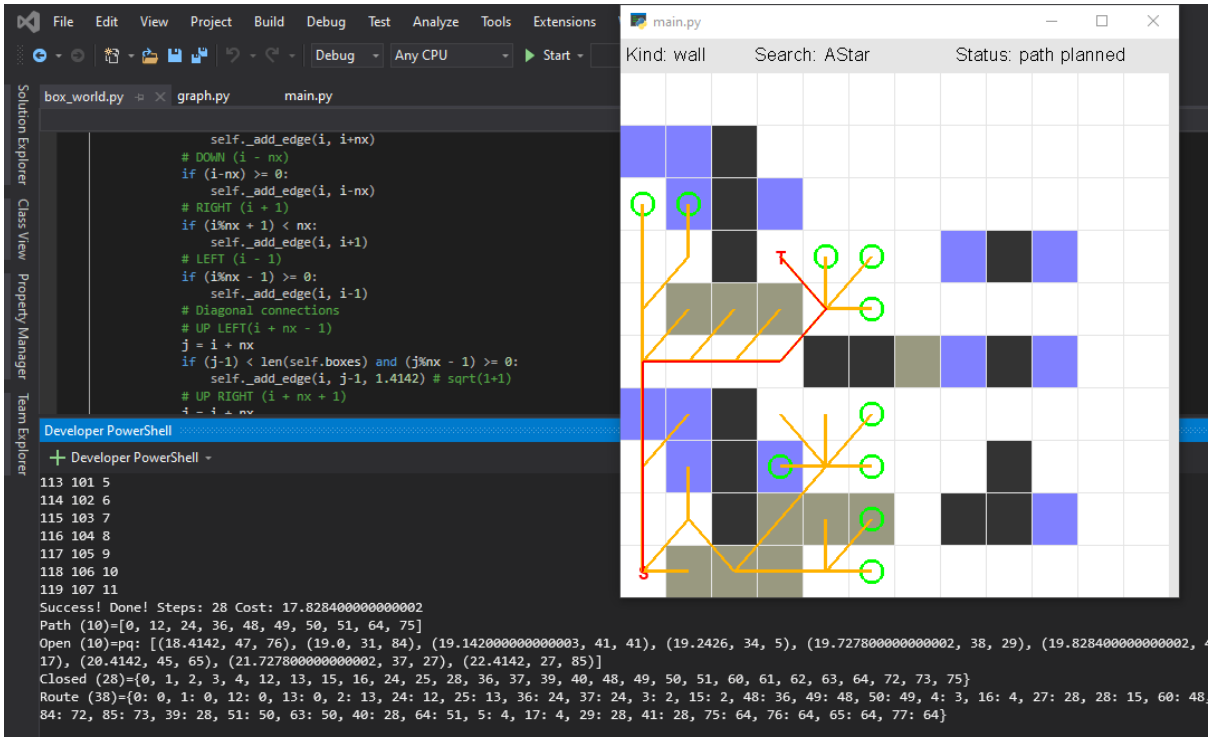
```
min_edge_cost = 1.0  # must be min value for heuristic cost to work
```




```
PS C:\Users\cools\source\repos\Graph\Graph> python main.py map2.txt
Success! Done! Steps: 24 Cost: 19.0
Path (10)=[0, 12, 24, 36, 48, 49, 50, 51, 63, 75]
Open (8)=pq: [(20.0, 28, 16), (21.0, 25, 64), (21.0, 27, 5), (21.0, 3
Closed (24)={0, 1, 2, 3, 4, 12, 13, 15, 24, 25, 36, 39, 48, 49, 50, 5
Route (32)={0: 0, 1: 0, 12: 0, 13: 12, 24: 12, 25: 13, 36: 24, 2: 1,
 63: 51, 85: 73, 64: 63, 75: 63, 5: 4, 16: 4, 27: 39, 96: 84, 40: 39}
```

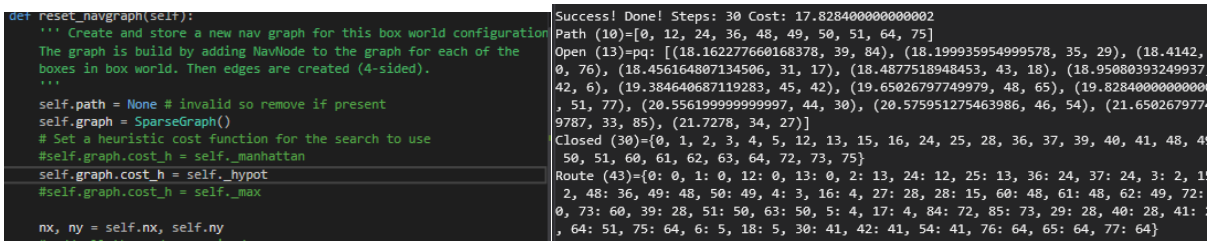
From what I have observed so far, heuristic work as long as the value defined is not an overestimation of the path cost.

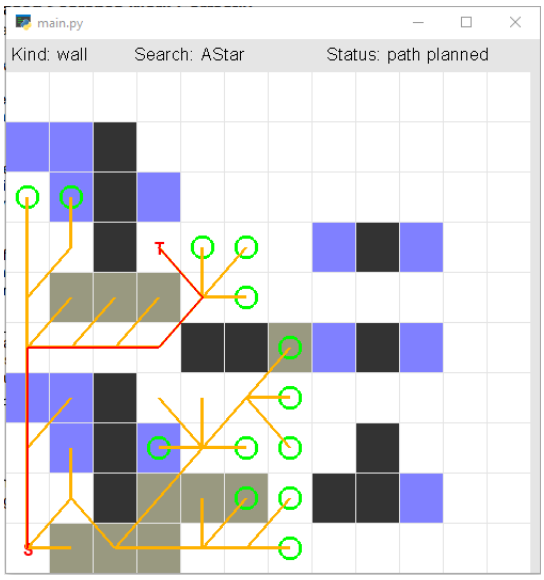
Even though we're using edge cost we don't really have edge marked on the GUI, to change that let's uncomment the code to allow diagonal connections.



Based on the observations so far, and the fact that the cost has been reduced I believe it is correct as it works with cost accuracy. Normally, I've seen Manhattan distance to be calculated in straight line distance but with the diagonal comparison it still has the output with regards to cost accuracy.

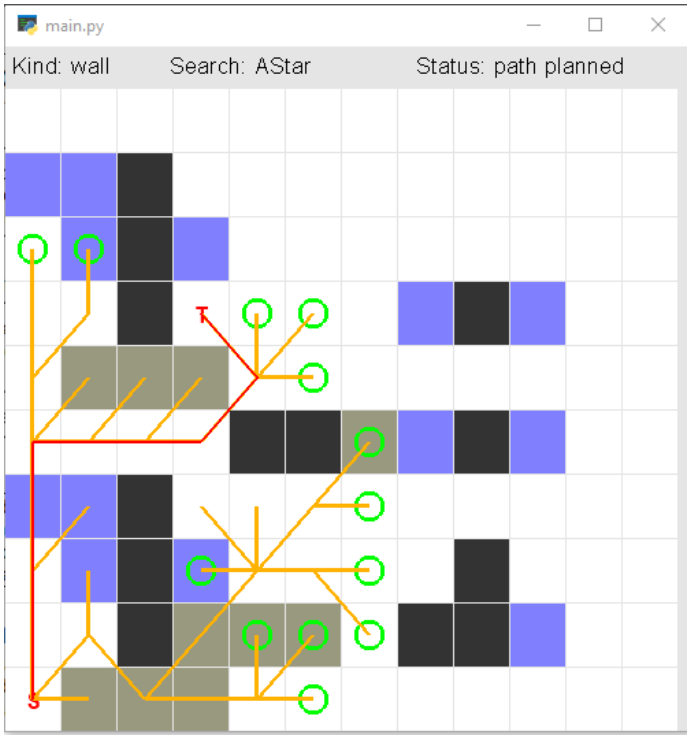
Changed to another heuristic method:





```
def reset_navgraph(self):
    ''' Create and store a new nav graph for this box world configuration
    The graph is build by adding NavNode to the graph for each of the
    boxes in box world. Then edges are created (4-sided).
    '''
    self.path = None # invalid so remove if present
    self.graph = SparseGraph()
    # Set a heuristic cost function for the search to use
    #self.graph.cost_h = self._manhattan
    #self.graph.cost_h = self._hypot
    self.graph.cost_h = self._max
```

```
Success! Done! Steps: 29 Cost: 17.828400000000002
Path (10)=[0, 12, 24, 36, 48, 49, 50, 51, 64, 75]
Open (13)=pq: [(17.2426, 30, 5), (17.2426, 31, 16), (18.0, 41, 84), (18.07099999
8, 32, 17), (18.142000000000003, 44, 42), (18.4142, 48, 76), (18.727800000000002
30), (19.4142, 46, 65), (19.828400000000002, 49, 77), (19.9704, 45, 54), (20.142
000003, 50, 18), (21.4142, 34, 85), (21.727800000000002, 35, 27)]
Closed (29)={0, 1, 2, 3, 4, 12, 13, 15, 24, 25, 28, 29, 36, 37, 39, 40, 41, 48,
, 51, 60, 61, 62, 63, 64, 72, 73, 75}
Route (42)={0: 0, 1: 0, 12: 0, 13: 0, 24: 12, 25: 13, 36: 24, 37: 24, 2: 13, 48:
9: 48, 3: 2, 15: 2, 50: 49, 4: 3, 16: 4, 27: 28, 28: 15, 60: 48, 61: 48, 62: 49,
0, 73: 60, 39: 28, 51: 50, 63: 50, 5: 4, 17: 4, 84: 72, 85: 73, 29: 28, 40: 28,
, 75: 64, 64: 51, 30: 29, 42: 41, 54: 41, 65: 64, 76: 64, 77: 64, 18: 29}
```



Based on the observations and comparison made in the output terminal on costs and steps, it is in fact clear that the heuristic works correctly but with number of searches performed differently, as the number of steps in doing so varies with each heuristic.