# M-PATH-Y

Pathfinding algorithm visualizer built on empathy

ABSTRACT

pathfinding algorithm visualizer made with pygame, which shows users how two different search algorithms calculate the shortest route from one point on the window to another.

Student Name: Jessica Sie – 2502053653 (L1BC)

Algorithms and Programming COMP6047001

This project utilizes and builds on background knowledge on path-finding algorithms, specifically the A* and the Dijkstra’s algorithm.

## A. Project specification

1. Program name:
   em-PATH-y. The name is derived from the word ‘empathy’ for computer science students learning about path-finding algorithms for the first time and aims to aid the understanding of search algorithms by visualization, while emphasizing ‘path’, as in pathfinding algorithms visualizer.

2. Program concept:
   - em-PATH-y is a pathfinding algorithm visualizer made with pygame, which shows users how two different search algorithms calculate the shortest route from one point on the window to another.
   - The search algorithms visualized in this program are the A* pathfinding algorithm and the Dijkstra’s pathfinding algorithm.
   - The program allows users to pick a start and end point on a grid for the algorithm to work out, marking them with different colours.
   - To simulate the algorithms closer to real world applications, the program lets users draw out obstacles which prevents the pathfinder from performing its algorithm there, making the pathfinder go around it.
   - The program lets the user chose either the A* pathfinding algorithm or the Dijkstra’s pathfinding algorithm to find the shortest path from start to end point. The chosen algorithm is run against the set starting point, ending point and obstacles, changing each block to a different colour when calculating its path for visualization. Finally,The program generates a solid colour path to show the shortest way from the start to the end point.
   - The program allows users to clear the window of start/end points and obstacles to reset.

3. Program flow summary
   - User selects a starting point anywhere on the grid
   - User selects an end point on the grid
   - User may choose to draw obstacles for the calculated shortest path to go around
   - The search algorithm of choice is run and calculates the shortest path from start to end point
   - The shortest path from start to end pint around the obstacles is displayed in a single, solid colour

4. Program input
   - Mouse left button - to select start, end and draw obstacles
   - ‘a’ key – to visualize the A* algorithm
   - ‘d’ key – to visualize the dijkstra’s algorithm
   - ‘c’ key – to clear the grid of all points and obstacles

5. Program output
   - The selected block on grid can change colour from white to orange when selected as starting point
   - The selected block on grid can change colour from white to turquoise when selected as end point
   - The selected block(s) can change color from white to black when selected as obstacle(s)
   - The blocks around the start and end point can change colour from white, to red, to green, when the search algorithm is ran.
   - The blocks leading from the starting point to the end point can change color to purple when the shortest path has been found.

6. Program libraries/modules
   - Pygame – to make and run most of the program visually
   - Math – perform calculations for heuristic function of the a star algorithm
   - Colorspy – for color coding the different states of the blocks/nodes
   - queue – to use the priority queue data structure for getting the next node with the lowest cost for the search algorithms

7. Program files
   - main.py - driver file
   - aStarReal.py - functions implementing the A* algorithm
   - dijkstras.py - Functions implementing the Dijkstra's algorithm
   - gridFunctions.py – functions to draw grid on pygame window
   - nodeClass.py – class for nodes/blocks/points

B. Program Interface - how user communicates with the program
   a. Program execution

      1. Run the main.py file on your chosen IDE – window with grids appear

   b. Set start/end points

      1.Left-click a block to set your starting point – block turns orange

      (optional) right-click same block to remove starting point, repeat step to set new point – block turns back to white

      2. Left-click another block to set your end point – block turns blue

      (optional) right-click same block to remove end point, repeat step to set new point – block turns back to white

   c. Set obstacles (optional)

      1.Left-click other block(s) to set obstacles – block turns black

      (optional) right-click same block(s) to remove obstacle, repeat step to set new obstacle – block turns back to white

d. Display algorithm

   Press ‘d’ key to run the Dijkstra’s pathfinding algorithm – blocks around the start and end point turns red or green, shortest path turns purple

   OR

   Press ‘a’ key to run the A* pathfinding algorithm – blocks around the start and end point turns red or green, shortest path turns purple

e. Clear entire window (optional)

   Press ‘c’ key to clear the window – blocks turn back to white

f. Program termination

   1. Click on the close button of the window to terminate program – window closes

C. Solution Design

**Flowchart**

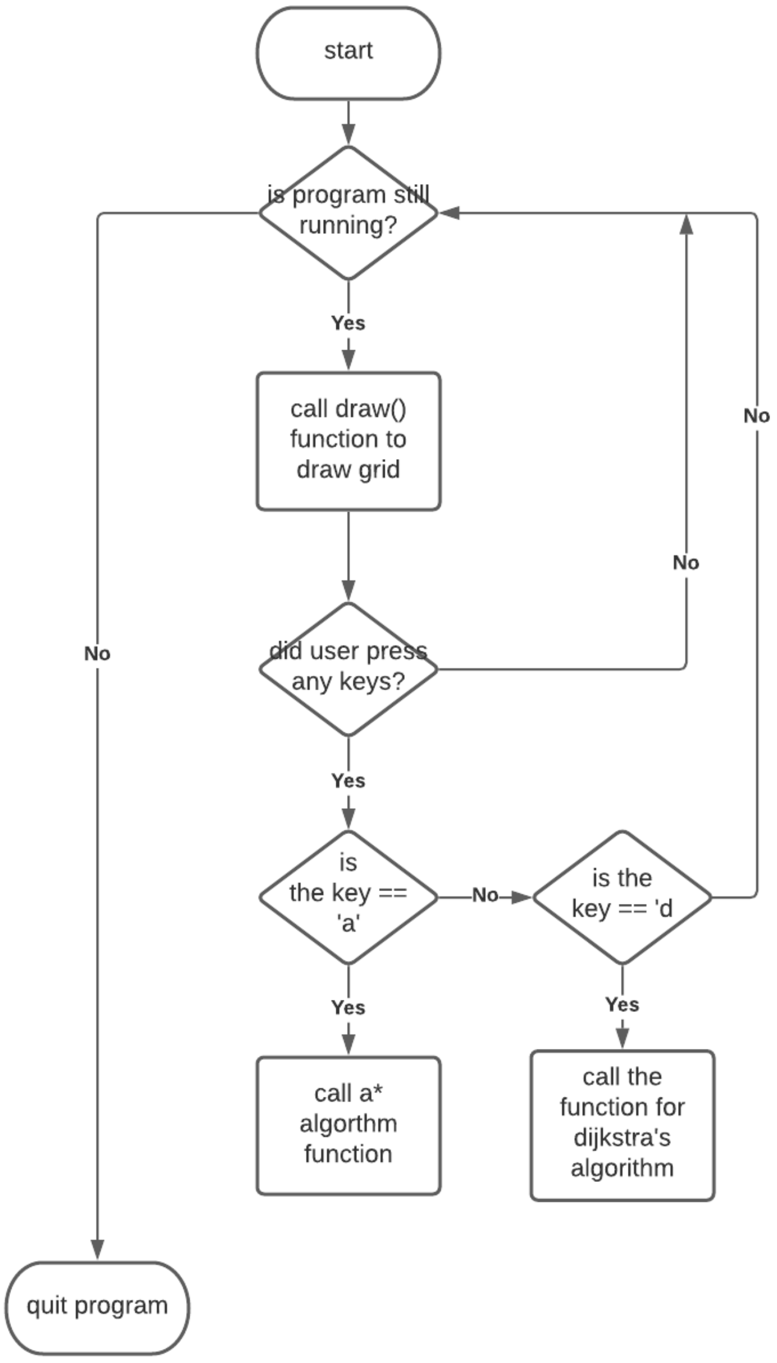

start
is program still running?
Yes
No
call draw() function to draw grid
did user press any keys?
No
No
Yes
is the key == 'a'
No
is the key == 'd
Yes
Yes
call a* algorthm function
call the function for dijkstra's algorithm
quit program

*Figure 1 - flowchart of main program*

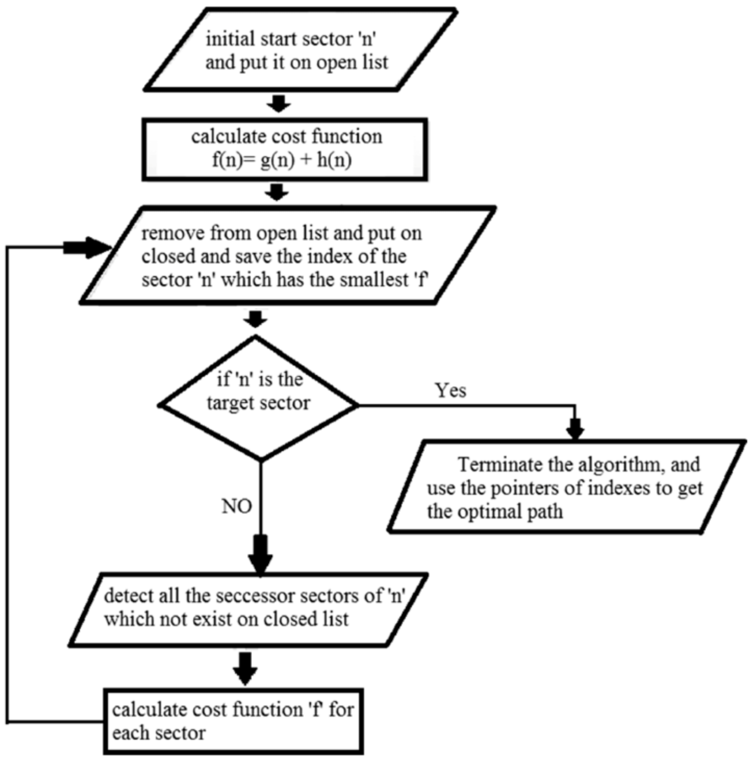

initial start sector 'n' and put it on open list
calculate cost function f(n)= g(n) + h(n)
remove from open list and put on closed and save the index of the sector 'n' which has the smallest 'f'
if 'n' is the target sector
Yes
Terminate the algorithm, and use the pointers of indexes to get the optimal path
NO
detect all the seccessor sectors of 'n' which not exist on closed list
calculate cost function 'f' for each sector

*Figure 2 - flochart of A star search algorithm*

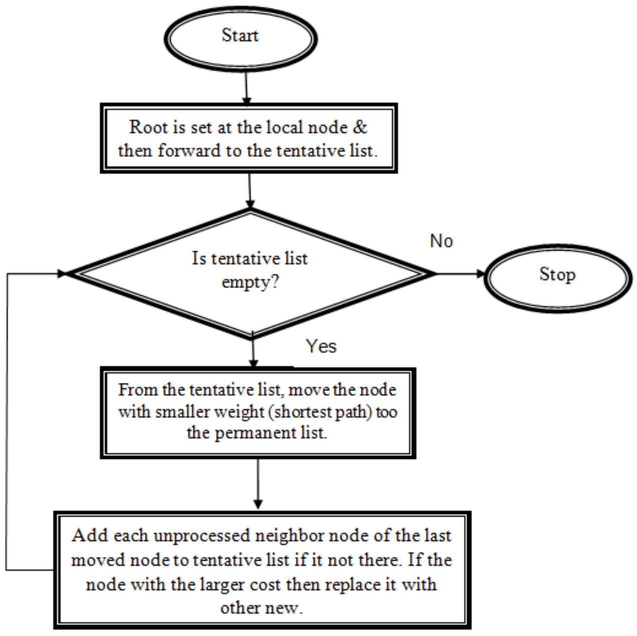
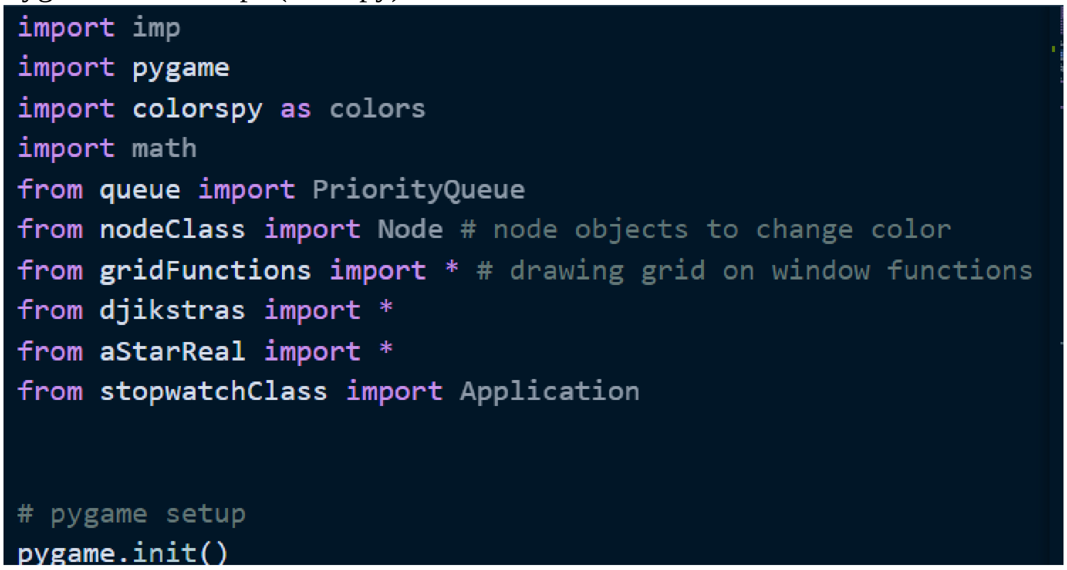

Start
Root is set at the local node & then forward to the tentative list.
Is tentative list empty?
No
Stop
Yes
From the tentative list, move the node with smaller weight (shortest path) too the permanent list.
Add each unprocessed neighbor node of the last moved node to tentative list if it not there. If the node with the larger cost then replace it with other new.

*Figure 3 - flowchart of Dijkstra’s search algorithm*

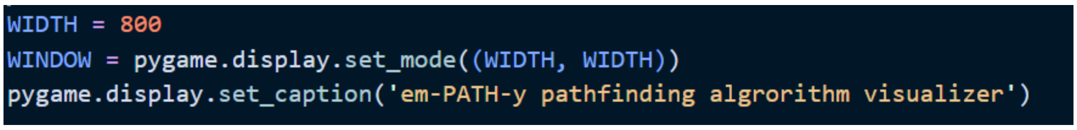
**Source code**

1. Pygame basic setups (main.py)



```
import imp
import pygame
import colorspy as colors
import math
from queue import PriorityQueue
from nodeClass import Node # node objects to change color
from gridFunctions import * # drawing grid on window functions
from djikstras import *
from aStarReal import *
from stopwatchClass import Application


# pygame setup
pygame.init()
```

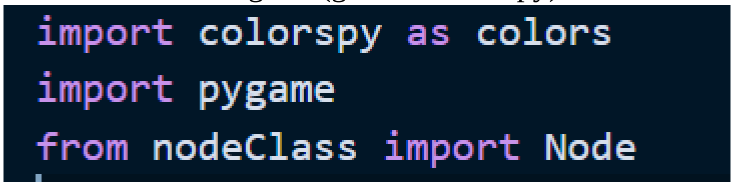
- All imports to the main.py file
- Initializing pygame module by calling pygame.init()



```
WIDTH = 800
WINDOW = pygame.display.set_mode((WIDTH, WIDTH))
pygame.display.set_caption('em-PATH-y pathfinding algroritm visualizer')
```

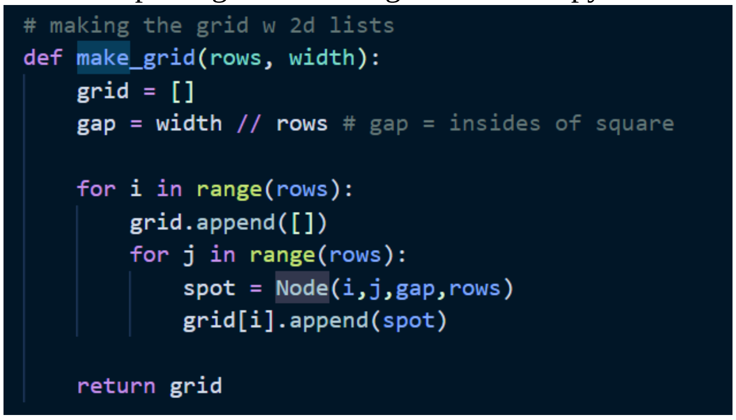
- Setting up the pygame window using a constant ‘WIDTH’ for the same width and height of the window. Caption is set as the program name and a short description of what it does.

2. Functions to make grids (gridFunctions.py)



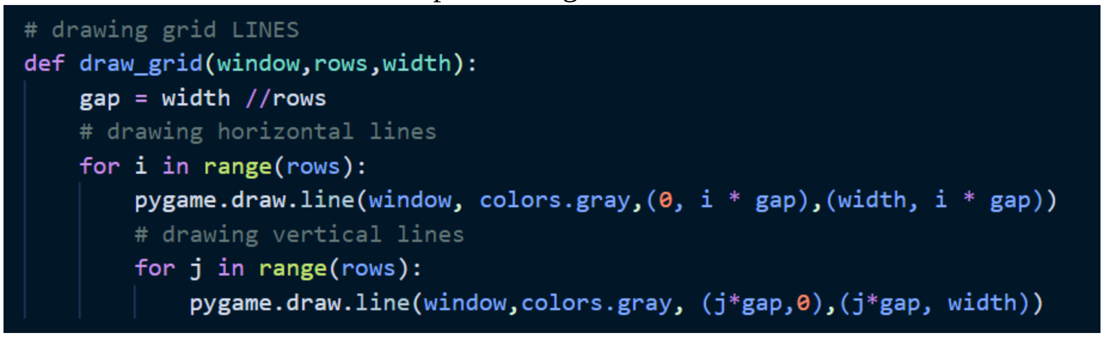
```
import colorspy as colors
import pygame
from nodeClass import Node
```

- Importing modules to gridFunctions.py file



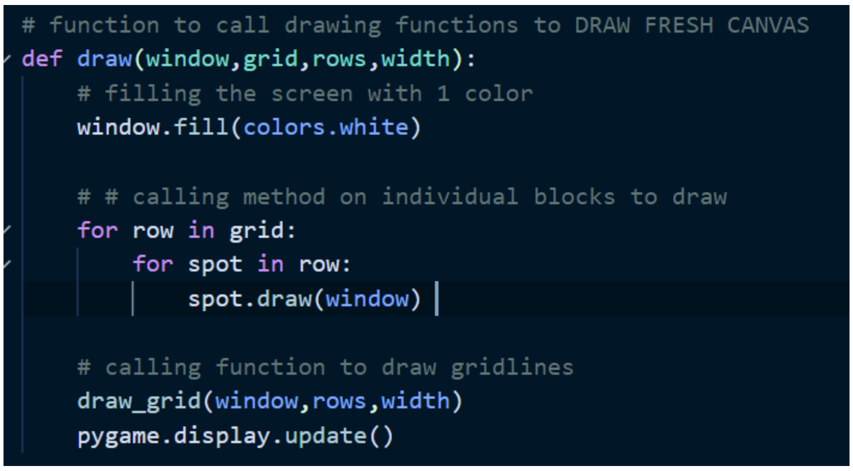
```
# making the grid w 2d lists
def make_grid(rows, width):
    grid = []
    gap = width // rows # gap = insides of square

    for i in range(rows):
        grid.append([])
        for j in range(rows):
            spot = Node(i,j,gap,rows)
            grid[i].append(spot)

    return grid
```

- make_grid() function initializes a 2d list called ‘grid’, where the indexes of each element corresponds to the X and Y coordinates of the grid. This list stores the values of each spot in the grid.



```python
# drawing grid LINES
def draw_grid(window,rows,width):
    gap = width //rows
    # drawing horizontal lines
    for i in range(rows):
        pygame.draw.line(window, colors.gray,(0, i * gap),(width, i * gap))
        # drawing vertical lines
        for j in range(rows):
            pygame.draw.line(window,colors.gray, (j*gap,0),(j*gap, width))
```

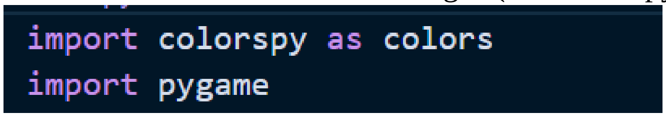
- draw_grid() function creates the horizontal and vertical lines on the window to visually create the grid onscreen. It utilises the pygame method, draw to do this.



```python
# function to call drawing functions to DRAW FRESH CANVAS
def draw(window,grid,rows,width):
    # filling the screen with 1 color
    window.fill(colors.white)

    # # calling method on individual blocks to draw
    for row in grid:
        for spot in row:
            spot.draw(window)

    # calling function to draw gridlines
    draw_grid(window,rows,width)
    pygame.display.update()
```

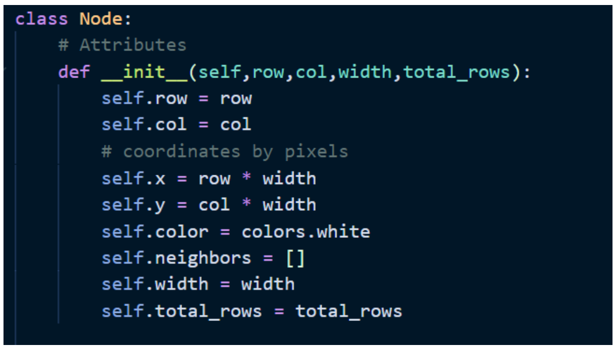
- the draw() function creates a fresh canvas onscreen, it takes in the ‘window’, ‘grid’ 2D list, ‘rows’ and width as parameters
- the function, first makes the background of the entire window white, the colour of null or accessible blocks using the fill method on the window parameter. The function then calls the draw method on each block in the grid to place it on the pygame window. Finally the function calls the draw_grid() function to place grid lines on the widow and calls the display.update() method to display them all on the window

3. Class for individual blocks on the grid(nodeClass.py)



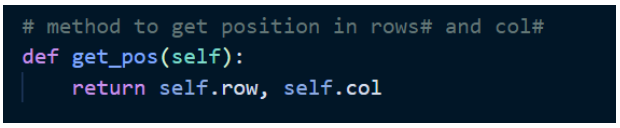
```python
import colorspy as colors
import pygame
```

- Importing modules for nodeClass.py file



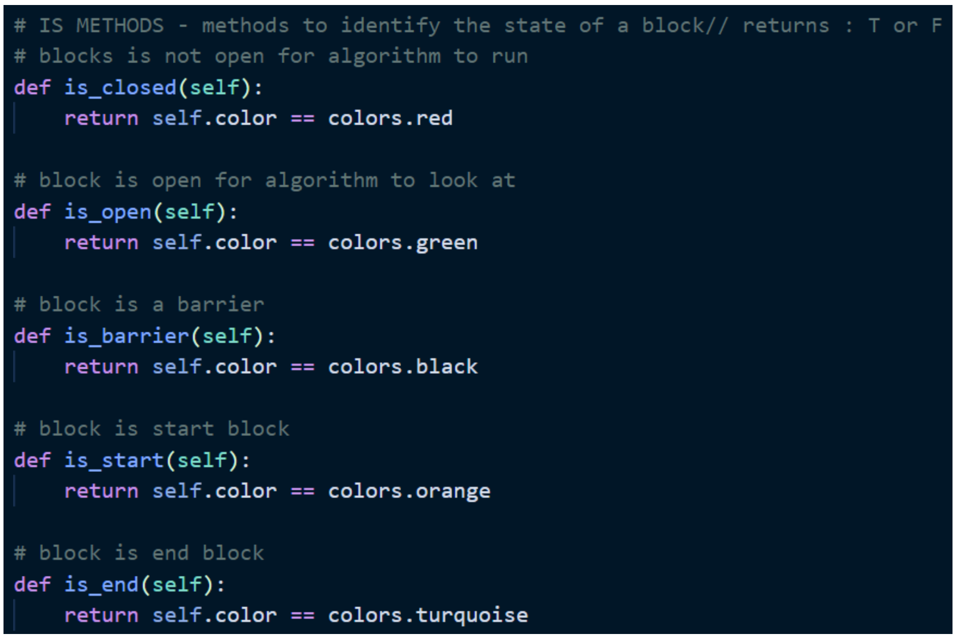
```python
class Node:
    # Attributes
    def __init__(self,row,col,width,total_rows):
        self.row = row
        self.col = col
        # coordinates by pixels
        self.x = row * width
        self.y = col * width
        self.color = colors.white
        self.neighbors = []
        self.width = width
        self.total_rows = total_rows
```

- In the nodeClass.py file, I created a class called 'Node'. It contains the attributes and methods of a node or block in the grid.
- First, I created a variable to store the number of rows in the grid
- The __init__ method basically initializes each block object with a row, col or column, width, and total_rows. This method also initializes other vaiables such as the pixel coordinates of the block, x and y, and an empty neighbours list. The colour attribute is initialized to white, the background colour of the window, with the imported colorspy module as colors.



```python
# method to get position in rows# and col#
def get_pos(self):
    return self.row, self.col
```

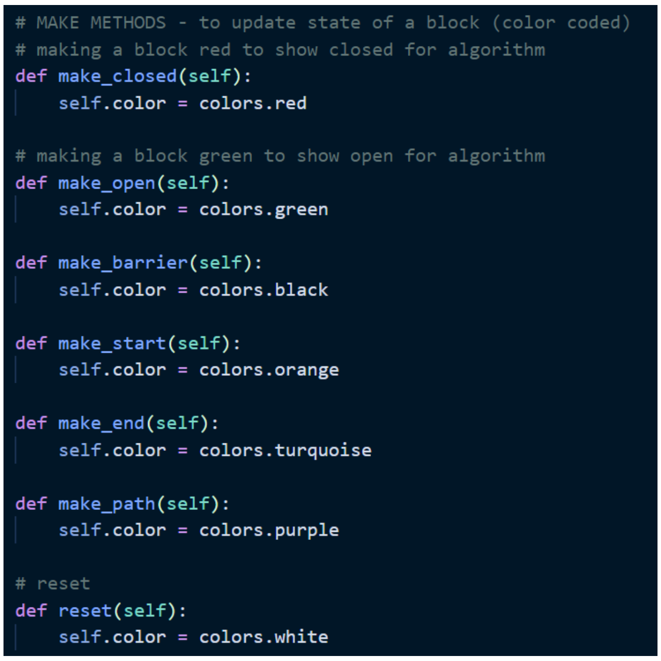
- Then I defined the classes' methods. The get_pos () method to get row and column number of the 'block' or 'spot' object



```python
# IS METHODS - methods to identify the state of a block// returns : T or F
# blocks is not open for algorithm to run
def is_closed(self):
    return self.color == colors.red

# block is open for algorithm to look at
def is_open(self):
    return self.color == colors.green

# block is a barrier
def is_barrier(self):
    return self.color == colors.black

# block is start block
def is_start(self):
    return self.color == colors.orange

# block is end block
def is_end(self):
    return self.color == colors.turquoise
```

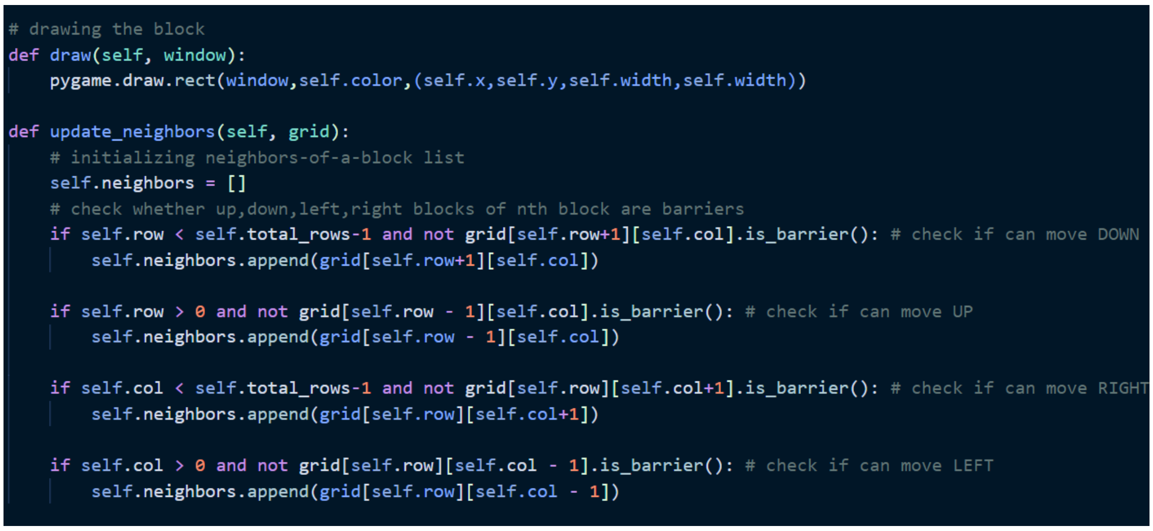
- Then I created these methods starting with 'is' to identify the state of the 'block' object based on its colour and return True or False.
- The module colorspy imported as 'colors' is used for colour code.
- Is_closed() method is called to check if the colour of the block is red.
- Is_opened() method is called to check if the colour of the block is green.
- Is_barrier() method is called to check if the colour of the block is black
- Is_start() method is called to check if the colour of the block is orange

- Is_end() method is called to check if the colour of the block is turquoise



```python
# MAKE METHODS - to update state of a block (color coded)
# making a block red to show closed for algorithm
def make_closed(self):
    self.color = colors.red

# making a block green to show open for algorithm
def make_open(self):
    self.color = colors.green

def make_barrier(self):
    self.color = colors.black

def make_start(self):
    self.color = colors.orange

def make_end(self):
    self.color = colors.turquoise

def make_path(self):
    self.color = colors.purple

# reset
def reset(self):
    self.color = colors.white
```

- Then I created these methods that start with 'make' to update the attribute of the 'block' object by changing their colour with the imported colorspy module as 'colors'.
- The make_closed() module turns the colour attribute of the 'block' object red.
- The make_open() module turns the colour attribute of the 'block' object green.
- The make_barrier() module turns the colour attribute of the 'block' object black.
- The make_start() module turns the colour attribute of the 'block' object orange.
- The make_end() module turns the colour attribute of the 'block' object turquoise.
- The make_path() module turns the colour attribute of the 'block' object purple.
- The reset() module turns the color attribute of the 'block' object back to white.

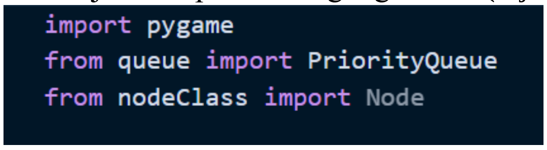


```python
# drawing the block
def draw(self, window):
    pygame.draw.rect(window,self.color,(self.x,self.y,self.width,self.width))

def update_neighbors(self, grid):
    # initializing neighbors-of-a-block list
    self.neighbors = []
    # check whether up,down,left,right blocks of nth block are barriers
    if self.row < self.total_rows-1 and not grid[self.row+1][self.col].is_barrier(): # check if can move DOWN
        self.neighbors.append(grid[self.row+1][self.col])

    if self.row > 0 and not grid[self.row - 1][self.col].is_barrier(): # check if can move UP
        self.neighbors.append(grid[self.row - 1][self.col])

    if self.col < self.total_rows-1 and not grid[self.row][self.col+1].is_barrier(): # check if can move RIGHT
        self.neighbors.append(grid[self.row][self.col+1])

    if self.col > 0 and not grid[self.row][self.col - 1].is_barrier(): # check if can move LEFT
        self.neighbors.append(grid[self.row][self.col - 1])
```

- Then I created the draw() method to display the block on screen with the draw rectangle method from pygame.

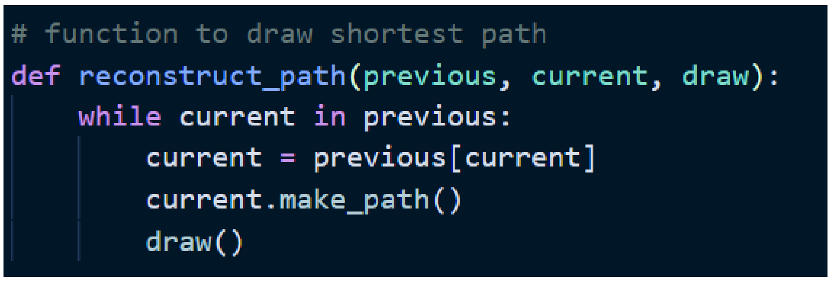
- Finally, I created the update_neighbords() method which checks whether the top, bottom , left and right blocks of the 'block' objects are not barriers. If the condition is satisfied, then add the row and column number of the object's neighbour's list.

4. The Dijkstra's pathfinding algorithm (dijkstras.py)



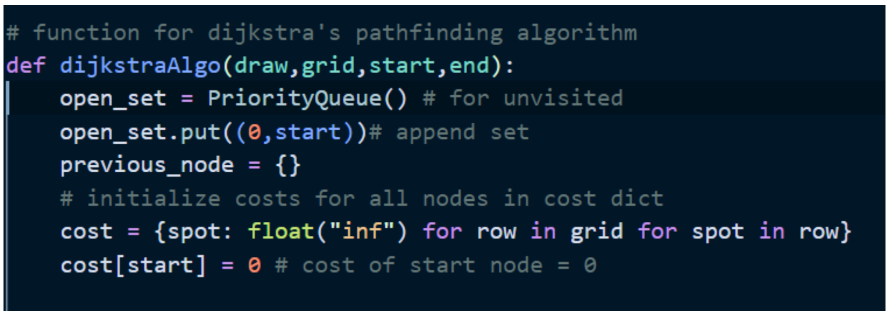
```python
import pygame
from queue import PriorityQueue
from nodeClass import Node
```

- Importing modules for Dijkstras.py
- 'node' class is explained above and priority queue data structure for getting the next node with the lowest cost for the search algorithms



```python
# function to draw shortest path
def reconstruct_path(previous, current, draw):
    while current in previous:
        current = previous[current]
        current.make_path()
        draw()
```

- Creating a function to change colour of the calculated shortest path to a uniform solid colour.
- Function takes in previous, current and draw as parameters and calls make_path() function as explained above, of the current node to turn change its colour attribute to purple



```python
# function for dijkstra's pathfinding algorithm
def dijkstraAlgo(draw,grid,start,end):
    open_set = PriorityQueue() # for unvisited
    open_set.put((0,start))# append set
    previous_node = {}
    # initialize costs for all nodes in cost dict
    cost = {spot: float("inf") for row in grid for spot in row}
    cost[start] = 0 # cost of start node = 0
```

- Creating the function to carry out the Dijkstra's search algorithm which takes in draw, as an anonymous function called with lambda, grid the 2d list, start node and end node.
- Initializing the cost (aka distance) of all nodes from the start node to infinity using float("inf"), to show that cost has not been calculated and so that further calculations with smaller cost can replace the infinite cost
- Initialize the cost of the start node to 0 because the start node is 0 units away from itself
- Initialize the open_set using the priority queue data structure to get the next node with the lowest cost or distance from start node with ease. The open_set variable contains all the unvisited nodes and their cost from start node. The open_set uses the set data structure in python so that duplicate entries of nodes can be ignored.

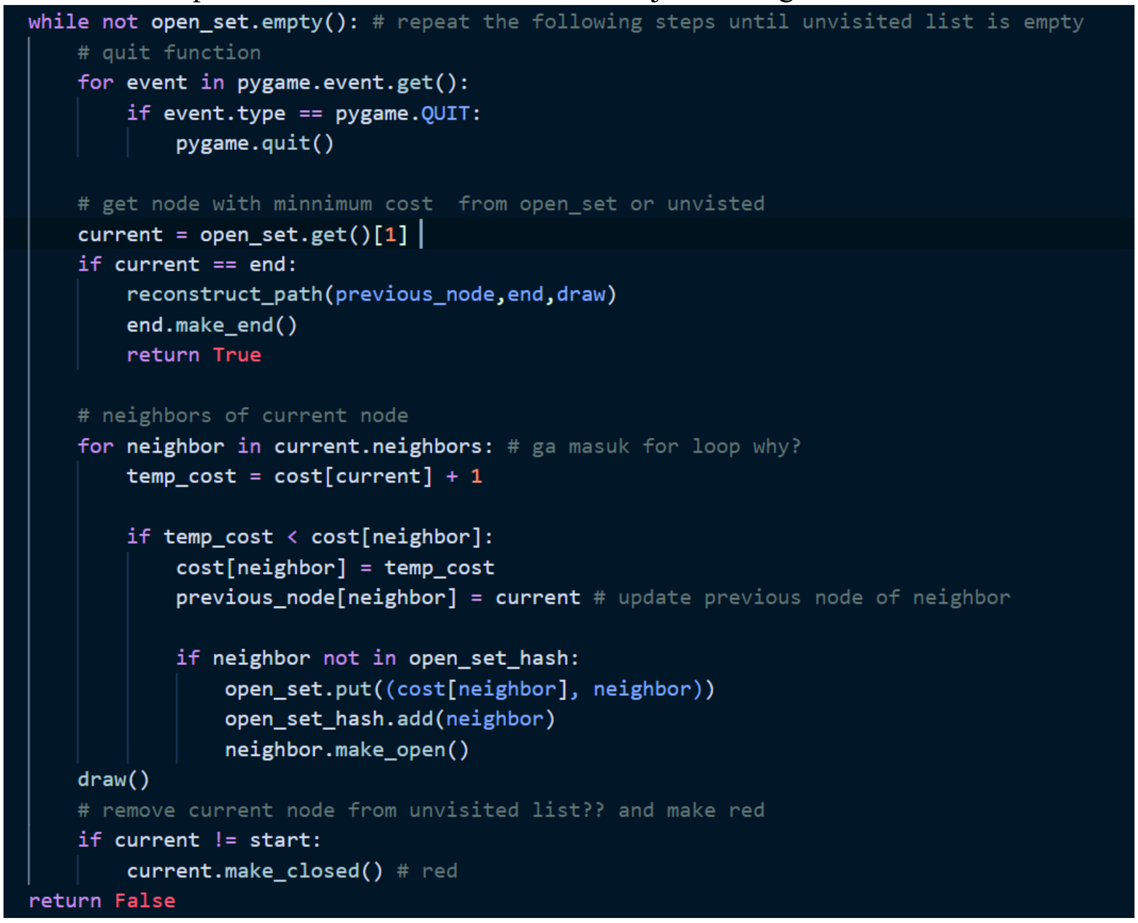
- The cost of the start node in the openset is initialized to 0 with the .put() method for the algorithm to start calculations.

Brief explanation of the dijkstras pathfinding algorithm:

The Dijkstra's algorithm has an unvisited list containing the nodes the algorithm hasn't determine the shortest distance from the start node, their cost, and their previous node. From this list, the node with the smallest cost or distance from start is selected as the current block the algorithm will. The algorithm then calculates the distance of all the node's neighbours from the start point and mutates its cost if it is smaller than the cost in the unvisited list. The algorithm then removes the current node with its cost form the unvisited list and loops by picking a new current node which has the smallest cost (*Isaac computer science*. (n.d.)).

The Dijkstra's algorithm loops until all nodes in the grid have been removed from the unvisited list, and then traces the end node back to the starting node by following the previous node, in order to find the shortest path. However, I only program the algorithm to loop until the end node has been found, instead of all the nodes to save computing time, and power, as to not crash my pc.

Below is an explanation of how I visualized this dijkstra's algorithm:

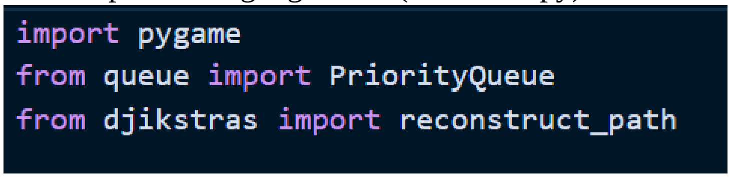


```
while not open_set.empty(): # repeat the following steps until unvisited list is empty
    # quit function
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()

    # get node with minnimum cost  from open_set or unvisted
    current = open_set.get()[1]
    if current == end:
        reconstruct_path(previous_node,end,draw)
        end.make_end()
        return True

    # neighbors of current node
    for neighbor in current.neighbors: # ga masuk for loop why?
        temp_cost = cost[current] + 1

        if temp_cost < cost[neighbor]:
            cost[neighbor] = temp_cost
            previous_node[neighbor] = current # update previous node of neighbor

            if neighbor not in open_set_hash:
                open_set.put((cost[neighbor], neighbor))
                open_set_hash.add(neighbor)
                neighbor.make_open()
    draw()
    # remove current node from unvisited list?? and make red
    if current != start:
        current.make_closed() # red
return False
```

- Created a loop that runs as long as the unvisited list or open_set vaiable is not empty

- Inside the loop I added a quit function, just in case the user wants to quit while the algorithm is running. The quit function uses a for loop to go through the list of pygame events that may happen and if one of them is the quit event, then call the pygame.quit() function
- Get node with the minimum cost from the open set and assign it to the variable 'current', the index [1] allows us to only access the node and not its cost
- An if statement is created inside the while loop to check if the current node that the algorithm is 'looking at' is the end node. If this condition is satisfied, the reconstruct_path() function is called to turn the shortest path into a single, solid colour as explained above, and returns true to quit the while loop as the task using the algorithm to search for the shortest path has been achieved
- Afterwards program checks the neighbours of the current node by using the neighbors method of the Node class. Here the cost of each neighbour is incremented by 1 as it is 1 block further away from the beginning and is assigned to a temporary variable. This temporary cost is then compared with the current cost of the neighbour with an if statement to see if the temporary cost is smaller than the current cost. If this statement is satisfied, then the temporary cost is now assigned as the current cost of the neighbour.
- The neighbour is now made green with make_open method to signify that the algorithm will 'look at' or perform calculations on it next.
- The draw function is called to display the changes in colour of the current node and the neighbouring blocks
- Finally, the current node is removed from the open set or unvisited list and made the colour red by calling the make_cosed() method to show that the algorithm is no longer performing calculations on that node.
- This loop repeats for all the nodes in the open_set until the current node the algorithm is looking at is the end node, or there are no more nodes in the openset, only in these two situations can the search algorithm be terminated

5. The A* pathfinding algorithm (aStarReal.py)



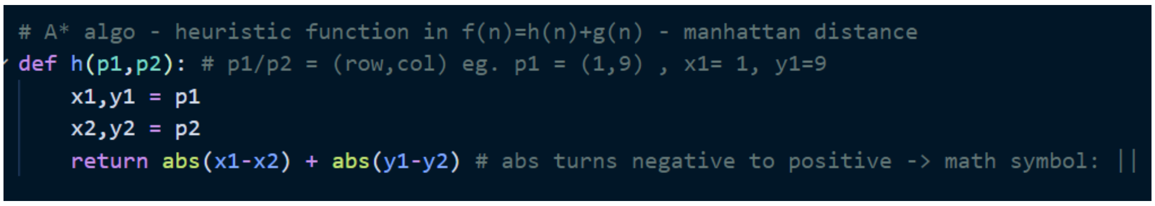
```
import pygame
from queue import PriorityQueue
from djikstras import reconstruct_path
```

- In this file, I implemented the A* pathfinding algorithm to find the shortest path between 2 blocks on the grid.
- Importing modules for aStarReal.py
- Reconstruct_path() function is imported from the dijkstras.py file to avoid repetition. This function serves to traces the end node back to the starting node by following the previous node, to find the shortest path after algorithm has been run.
- The priority queue data structure imported from queue module is for getting the node with the lowest cost for the algorithms to pick out the current node

The A* pathfinding algorithm is an informed search algorithm, which builds on the principles of Dijkstra's shortest path algorithm to provide a faster solution when faced with the problem of finding the shortest path between two nodes. It uses a heuristic function instead of brute force to decide the next node to consider as it moves along the path. The heuristic function provides an estimate of the minimum cost or distance between a given node and the end node. The algorithm will then sum the actual cost from the start node, G(n) , along with the estimated cost to the end node, h(n), to create f(n) for selecting the next node to evaluate. The equation is depicted below.

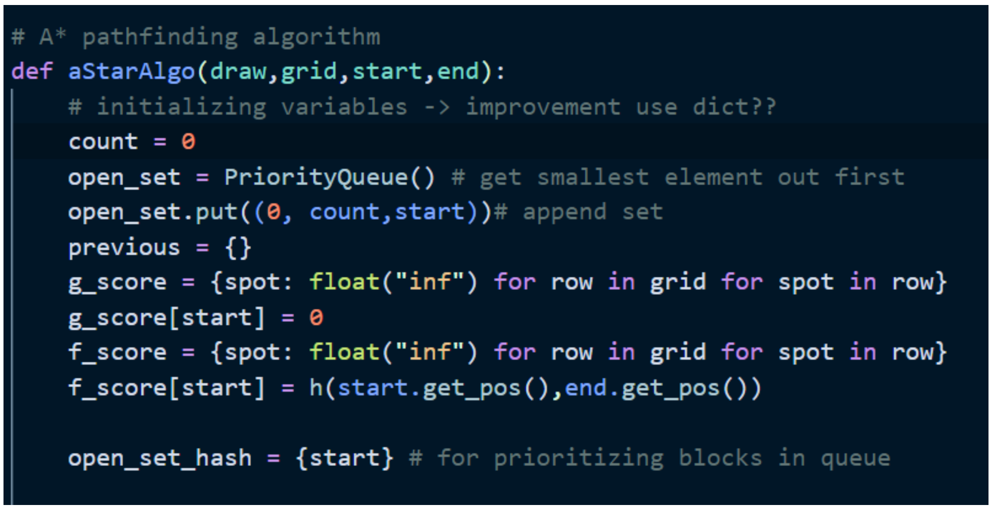
$$f(n) = g(n) + h(n)$$

*Equation 1 - function for calculating estimated cost for selecting the next node to evaluate, where f(n) is estimated cost of the minimum solution, g(n) is the actual cost to reach a certain node from start node, and h(n) is the estimated cost from a certain node to end node(MIT OpenCourseWare.(n.d.))*



```python
# A* algo - heuristic function in f(n)=h(n)+g(n) - manhattan distance
def h(p1,p2): # p1/p2 = (row,col) eg. p1 = (1,9) , x1= 1, y1=9
    x1,y1 = p1
    x2,y2 = p2
    return abs(x1-x2) + abs(y1-y2) # abs turns negative to positive -> math symbol: ||
```

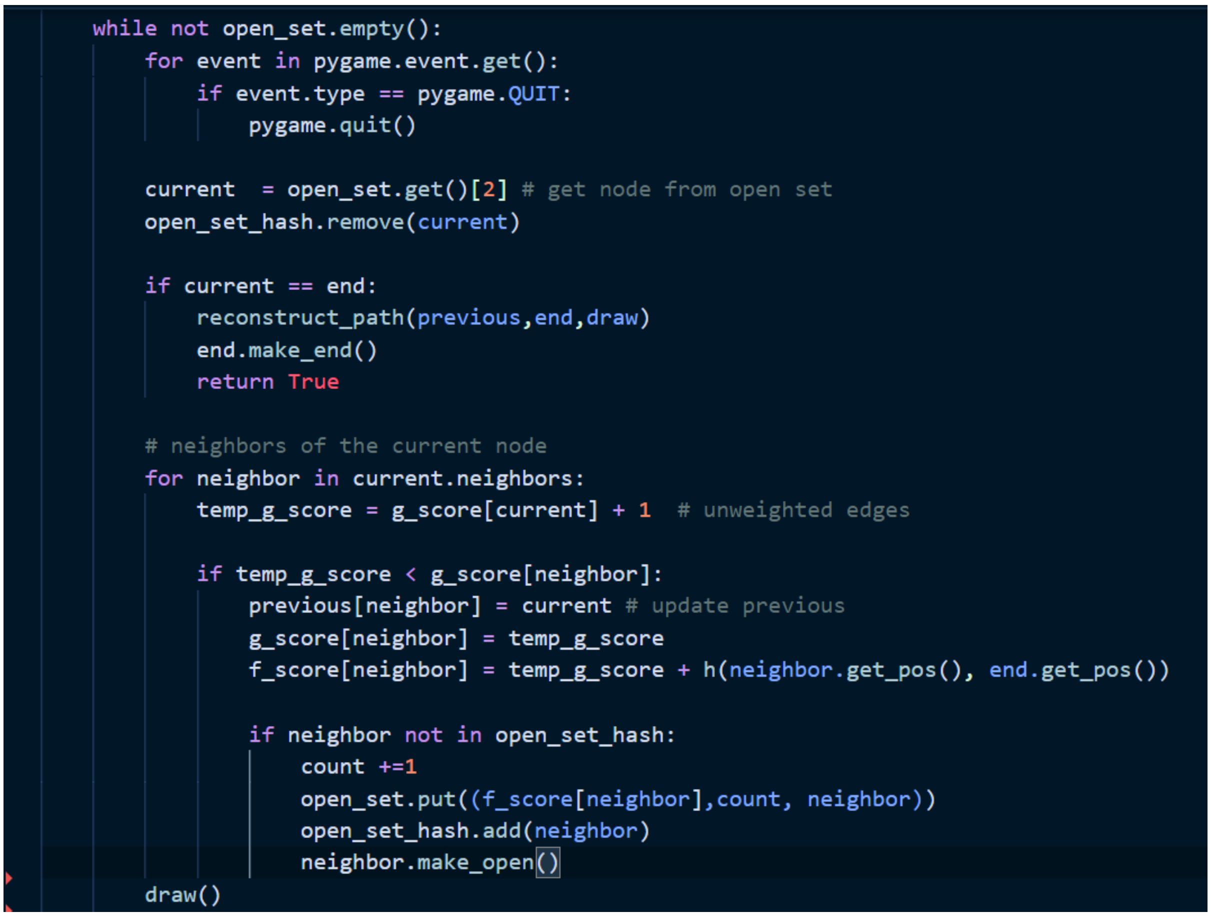
- First I defined a function h() , where 'h' stands for heuristic function. This function calculates the distance from a certain node to the end node by using the Manhattan distance method. The Manhattan distance as defined by Szabo (2015) is "the sum of absolute differences.



```python
# A* pathfinding algorithm
def aStarAlgo(draw,grid,start,end):
    # initializing variables -> improvement use dict??
    count = 0
    open_set = PriorityQueue() # get smallest element out first
    open_set.put((0, count,start))# append set
    previous = {}
    g_score = {spot: float("inf") for row in grid for spot in row}
    g_score[start] = 0
    f_score = {spot: float("inf") for row in grid for spot in row}
    f_score[start] = h(start.get_pos(),end.get_pos())

    open_set_hash = {start} # for prioritizing blocks in queue
```

- Creating the function to carry out the A* search algorithm which takes in draw, as an anonymous function called with lambda, grid the 2d list, start node and end node.
- Initializing the cost or distance of all nodes from the start node to infinity using float("inf"), to show that cost has not been calculated and so that further calculations with smaller cost can replace the infinite cost

- Initialize the cost of the start node to 0 because the start node is 0 units away from itself
- Initialize the open_set using the priority queue data structure to get the next node with the lowest cost or distance from start node with ease. The open_set variable contains all the unvisited nodes and their cost from start node. The open_set uses the set data structure in python so that duplicate entries of nodes can be ignored.
- The cost of the start node in the openset is initialized to 0 with the .put() method for the algorithm to start calculations.
- Open_set_hash stores the

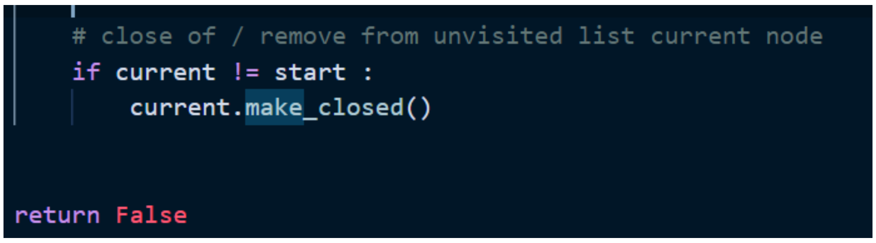


```
while not open_set.empty():
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()

    current  = open_set.get()[2] # get node from open set
    open_set_hash.remove(current)

    if current == end:
        reconstruct_path(previous,end,draw)
        end.make_end()
        return True

    # neighbors of the current node
    for neighbor in current.neighbors:
        temp_g_score = g_score[current] + 1  # unweighted edges

        if temp_g_score < g_score[neighbor]:
            previous[neighbor] = current # update previous
            g_score[neighbor] = temp_g_score
            f_score[neighbor] = temp_g_score + h(neighbor.get_pos(), end.get_pos())

            if neighbor not in open_set_hash:
                count +=1
                open_set.put((f_score[neighbor],count, neighbor))
                open_set_hash.add(neighbor)
                neighbor.make_open()
    draw()
```

- Created a loop that runs while the unvisited list or open_set vaiable is not empty
- Inside the loop I added a quit function, just in case the user wants to quit while the algorithm is running. The quit function uses a for loop to go through the list of pygame events that may happen and if one of them is the quit event, then call the pygame.quit() function
- Get node with the minimum cost from the open set and assign it to the variable ‘current’, the index [2] allows us to only access the node and not its cost or its counter.
- An if statement is created inside the while loop to check if the current node that the algorithm is calculating is the end node. If this condition is satisfied, the reconstruct_path() function is called to trace the end node back to the

starting node by following the previous nodes and turn the shortest path into a single, solid colour as explained above. Then the if statement returns true to quit the while loop as the task using the algorithm to search for the shortest path has been achieved

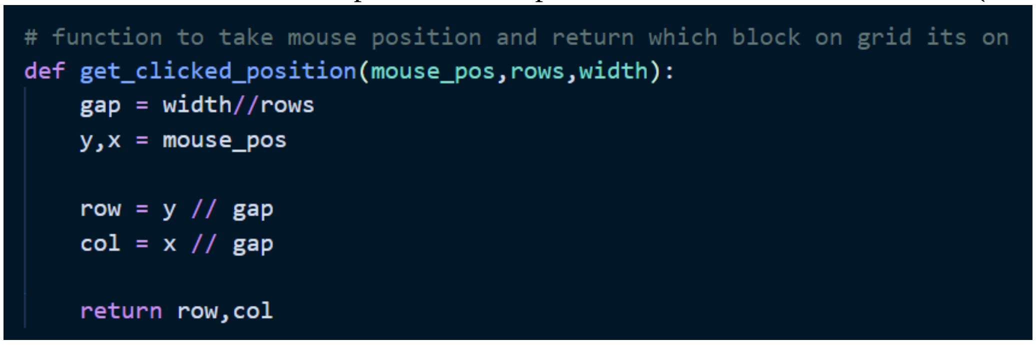
- Afterwards inside the while loop, the program checks the neighbours of the current node by using the neighbors method of the Node class. Here the actual cost from start, or g(n), of each neighbour is incremented by 1 as it is 1 block further away from the beginning and is assigned to a temporary variable.
- This temporary cost is then compared with the current g(n) cost of the neighbour with an if statement to see if the temporary cost is smaller than the current g(n) cost. If this statement is satisfied, then the temporary cost is now assigned to the g(n) cost of the neighbour.
  The previous node of the neighboring node is set to the current node, as they came from it. Then the f(n) cost of the neighbour is calculated using the sum of the g(n) cost and the h(n) cost, which is calculated by calling the h() function explained above.
- Thee cost neighbouring node in the open set or unvisited list is modified using put() method
- The neighbour is now made green with make_open method to signify that the algorithm will ‘look at’ or perform calculations on it next.
- The draw function is called to display the changes in colour of the current node and the neighbouring blocks



```
    # close of / remove from unvisited list current node
    if current != start :
        current.make_closed()


return False
```

- Finally, the current node is removed from the open set or unvisited list and made the colour red by calling the make_cosed() method to show that the algorithm is no longer performing calculations on that node.
- This loop repeats until the current node the algorithm is looking at is the end node. Only in this can the loop performing the search algorithm be terminated.

6. Function to convert mouse position from pixels to row and column number (main.py)



```
# function to take mouse position and return which block on grid its on
def get_clicked_position(mouse_pos,rows,width):
    gap = width//rows
    y,x = mouse_pos

    row = y // gap
    col = x // gap

    return row,col
```

- The get_clicked_pos() function takes in the mouse position, in pixels, as one of its arguments  and returns the index of the block on the gird.
- The calculation is as follows: The space in between each line of the grid, also known as the gap displayed in the figure below, is calculated by dividing width of the window by the number of rows of the grid. With this, the integer quotient of the pixel coordinates, x and y, with the gap is going to give us the index number of the corresponding row or column of the grid, where the mouse pointer is.

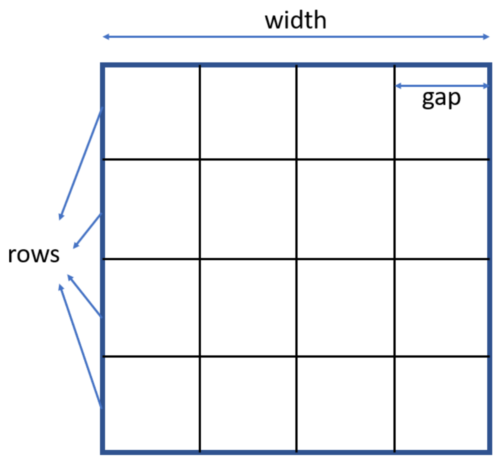
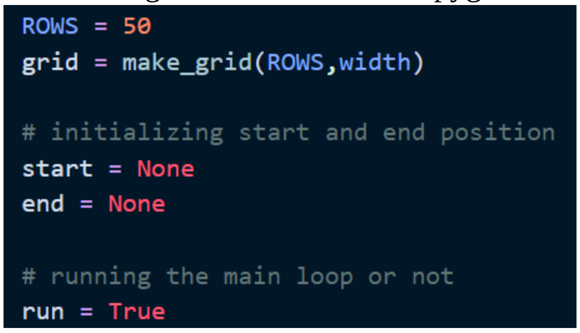


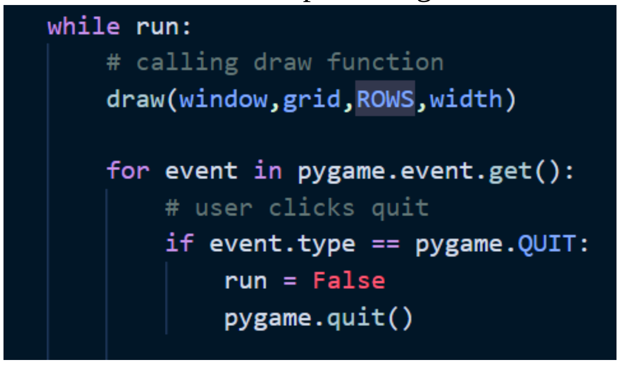
*Figure 4 - program window, the grid, rows and gap in between*

7. The running condition / the main pygame loop (main.py)



```python
ROWS = 50
grid = make_grid(ROWS,width)

# initializing start and end position
start = None
end = None

# running the main loop or not
run = True
```

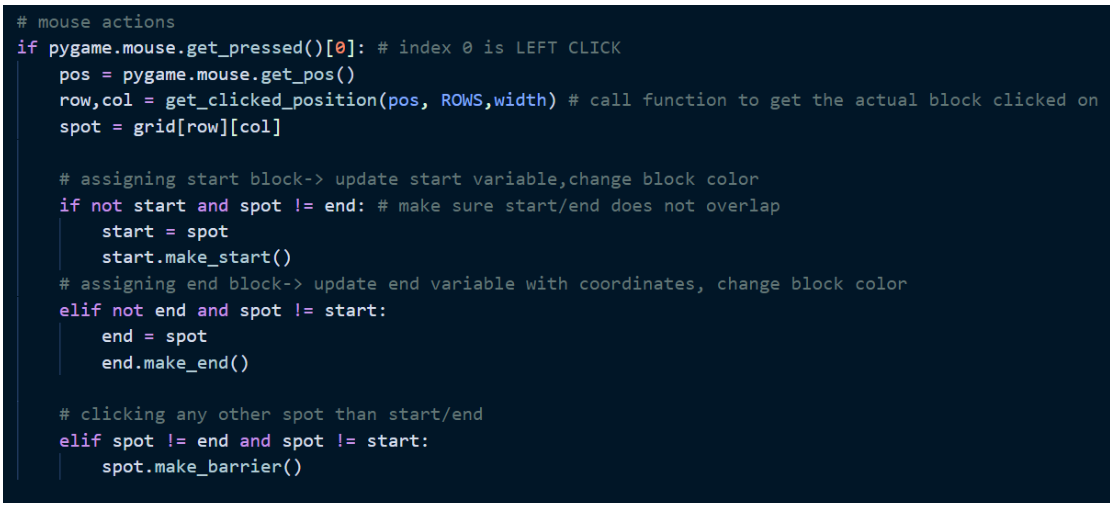
- Initialize some variables before the main program loop runs. 'ROWS' constant is the number of blocks in a row. Start and end position is set as none as none have been selected by user, run is set to True to allow main while loop to run next.
- call make_grid() function to initialize a 2d list, where the indexes of each element corresponds to the X and Y coordinates of the grid. This list stores the values of each spt in the grid.



```python
    while run:
        # calling draw function
        draw(window,grid,ROWS,width)

        for event in pygame.event.get():
            # user clicks quit
            if event.type == pygame.QUIT:
                run = False
                pygame.quit()
```

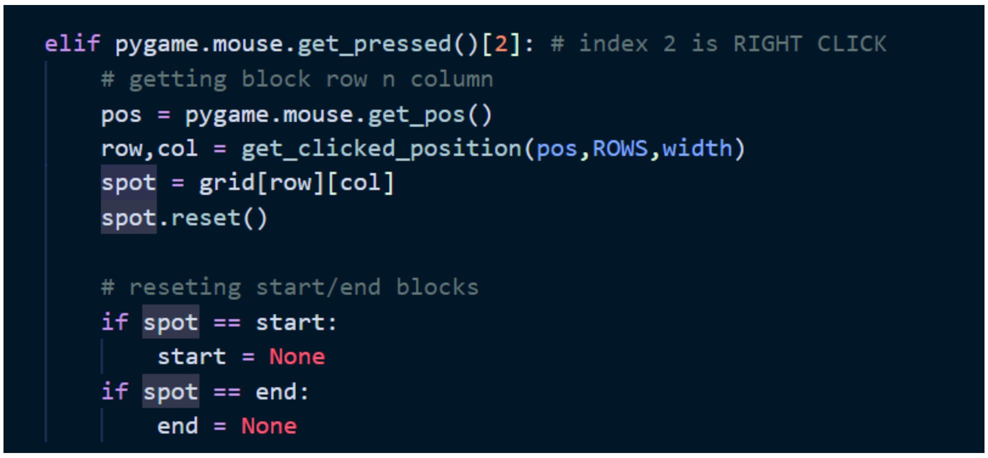
- When the program is first launched, the program enters this while loop. The for loop is just to ensure that if the player quits the game window when the algorithm has started, the program will terminate it and close the widow.

- Inside the main while loop, I called the draw() function for the grids, previously explained in the sections above



```
# mouse actions
if pygame.mouse.get_pressed()[0]: # index 0 is LEFT CLICK
    pos = pygame.mouse.get_pos()
    row,col = get_clicked_position(pos, ROWS,width) # call function to get the actual block clicked on
    spot = grid[row][col]

    # assigning start block-> update start variable,change block color
    if not start and spot != end: # make sure start/end does not overlap
        start = spot
        start.make_start()
    # assigning end block-> update end variable with coordinates, change block color
    elif not end and spot != start:
        end = spot
        end.make_end()

    # clicking any other spot than start/end
    elif spot != end and spot != start:
        spot.make_barrier()
```

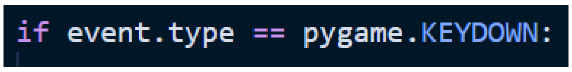
- These group of if statements are to change the blocks that have been clicked by the user into different colors to signify that it is either the chosen start, end or obstacle block(s).
- While main while-loop is running, the if statement in the main while loop checks if the player left-clicks on the window. If the condition is fulfilled, the program gets the position of the mouse click with get_pos() method in pixels. The get_clicked_position() function is then called and returns the row and column number of the grid where the mouse click occurs, as previously explained above.
- The variable ‘spot’ is used to store the current block that has been clicked for easy access an manipulation later on.
- The nested if statements then checks if the start or end points have been chosen. On a clean grid, and as initialized before, the start and end points were set to none, making the condition of ‘not start’ or ‘not end’ True. If these conditions are satisfied, the program calls the make_start() or make_end() method of the imported ‘Node’ class accordingly. These methods change the colour of the selected block/spot to visually show that they have been selected.
- If the first two conditions are not satisfied, the final condition checks if the chosen block is not a start or end point. If condition is satisfied, the block is made into a barrier by calling the make_barrier() method on the spot object.



```
elif pygame.mouse.get_pressed()[2]: # index 2 is RIGHT CLICK
    # getting block row n column
    pos = pygame.mouse.get_pos()
    row,col = get_clicked_position(pos,ROWS,width)
    spot = grid[row][col]
    spot.reset()

    # reseting start/end blocks
    if spot == start:
        start = None
    if spot == end:
        end = None
```

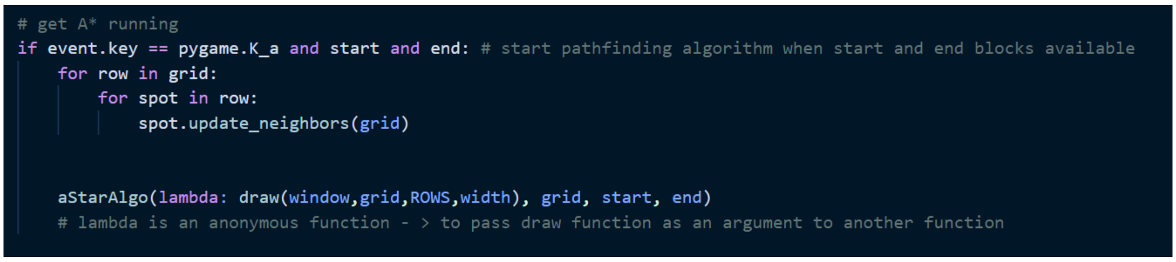
- These other set of elseif statements continued from the one above,  are to deselect the blocks that have been chosen by the user and change their colors back to white.
- The get_pressed() method with [2] as index signifies the right click button, if the condition that the right click button is pressed, the position of the mouse is stored and converted to corresponding block on the grid. That block is then changed to white by calling the reset() method of the Node class. If the block is a start or end node, then it is reset by assigning None value to it.

Starting the pathfinding algorithms



```
if event.type == pygame.KEYDOWN:
```

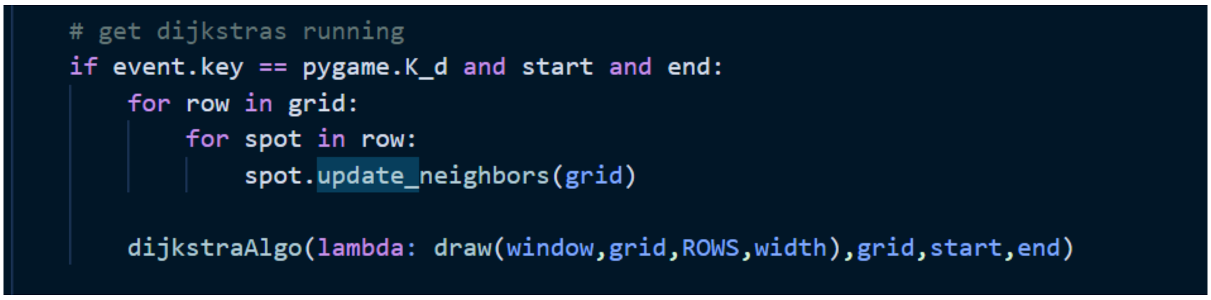
- This if statement in the main while loop checks if the user pressed any key on the keyboard. If statement is satisfied, it goes inside the statement. The statement inside are as follows: if function to start the a star algorithm, if function to start the Dijkstra's algorihm and the if function to reset the entire grid.



```
# get A* running
if event.key == pygame.K_a and start and end: # start pathfinding algorithm when start and end blocks available
    for row in grid:
        for spot in row:
            spot.update_neighbors(grid)


    aStarAlgo(lambda: draw(window,grid,ROWS,width), grid, start, end)
    # lambda is an anonymous function - > to pass draw function as an argument to another function
```

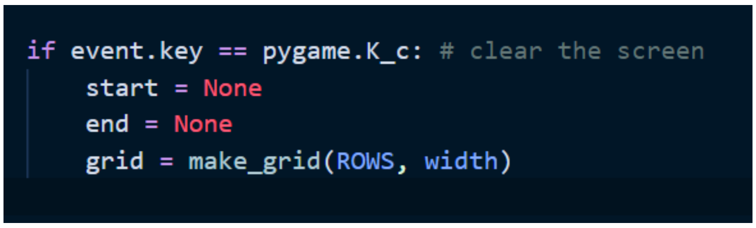
- These few lines run the a star pathfinding algorithm function
- This if statement in the main while loop checks if the user pressed the 'a' key on the keyboard. If condition is satisfied, the program calls the update_neighbors() method from the imported 'Node' class for the individual blocks on the grid. Afterwards, the aStarAlgo() function is finally called, workings of the algorithm is described above. Here, lambda or an anonymous

function is used to pass another function as an argument into the aStarAlgo() function.



```
# get dijkstras running
if event.key == pygame.K_d and start and end:
    for row in grid:
        for spot in row:
            spot.update_neighbors(grid)

    dijkstraAlgo(lambda: draw(window,grid,ROWS,width),grid,start,end)
```

- The few lines of code above, run the Dijkstra’s pathfinding algorithm function, explained before in this report
- This if statement in the main while loop checks if the user pressed the ‘d’ key on the keyboard, ‘d’ is for Dijkstra’s. If the condition is met, the program calls the update_neighbors() method from the imported ‘Node’ class for the individual blocks on the grid. Afterwards, the dijkstraAlgo() function is finally called, workings of the algorithm is described above. Here, lambda or an anonymous function is used to pass another function as an argument into the dijkstraAlgo () function.



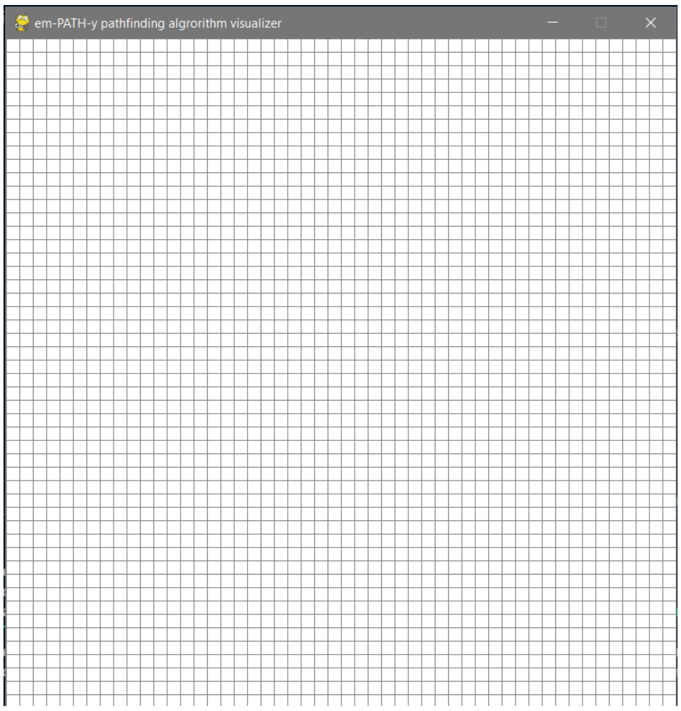
```
if event.key == pygame.K_c: # clear the screen
    start = None
    end = None
    grid = make_grid(ROWS, width)
```

- This if statement in the main while loop checks if the user pressed the ‘c’ key on the keyboard to clear the entire grid. If the condition is met, the program assigns None as values for the start and end node and calls the make_grid() function to initialize a new 2D list.

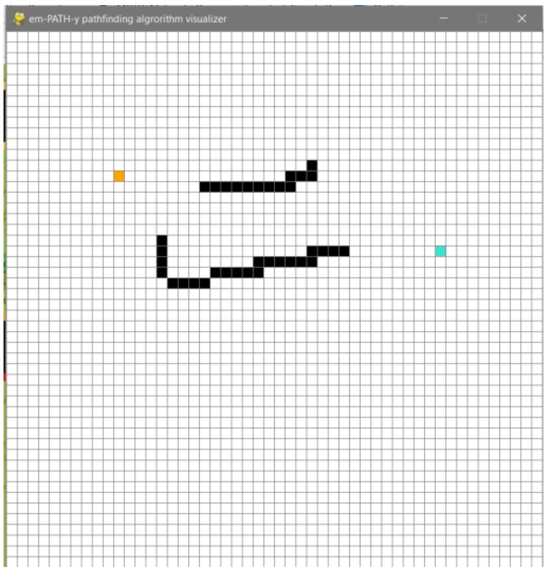
  - ▪ Drawing the grid on window

1. Dijkstra’s algorithm implementation
2. A* algorithm implementation
3.

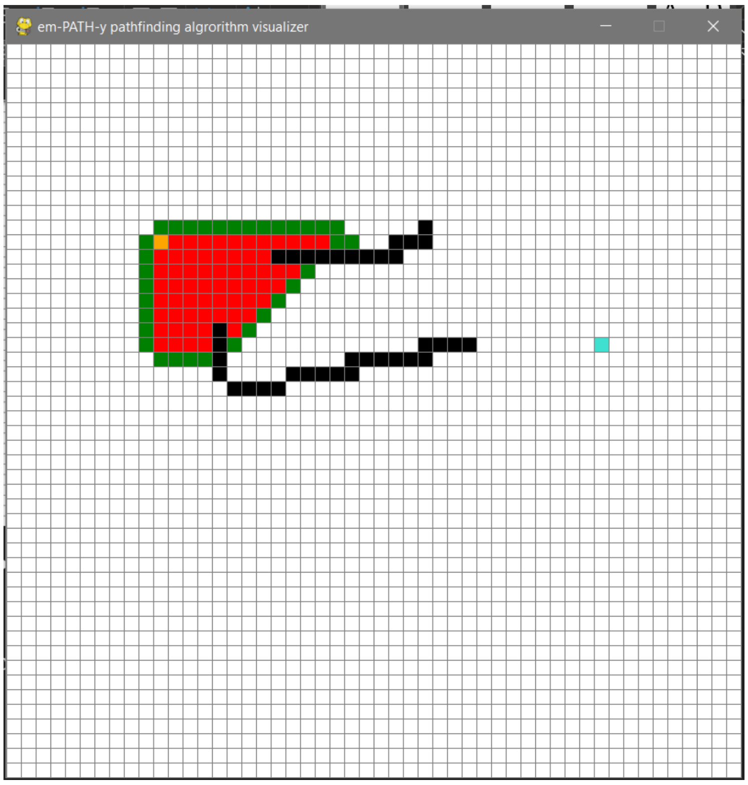
D. Evidence of working program



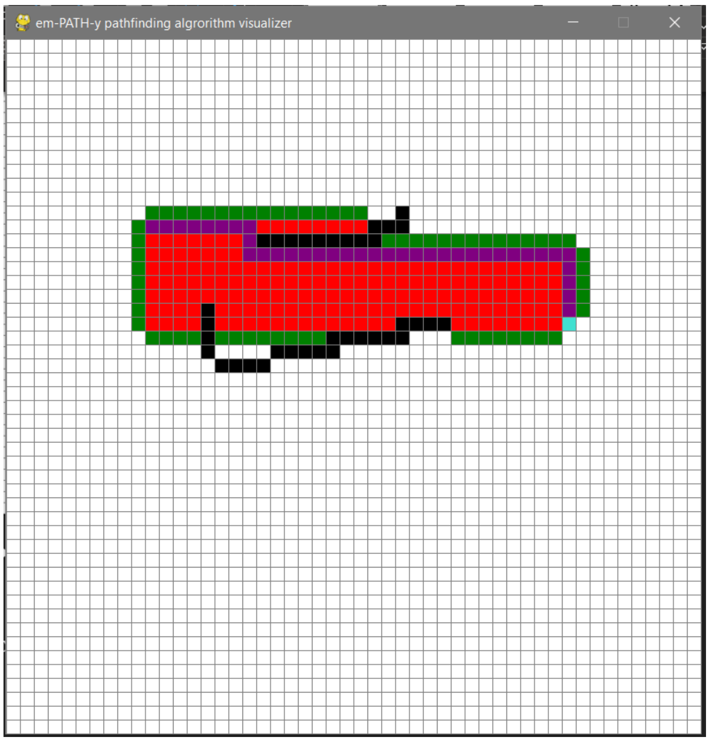
- em-PATH-y Program initial display window



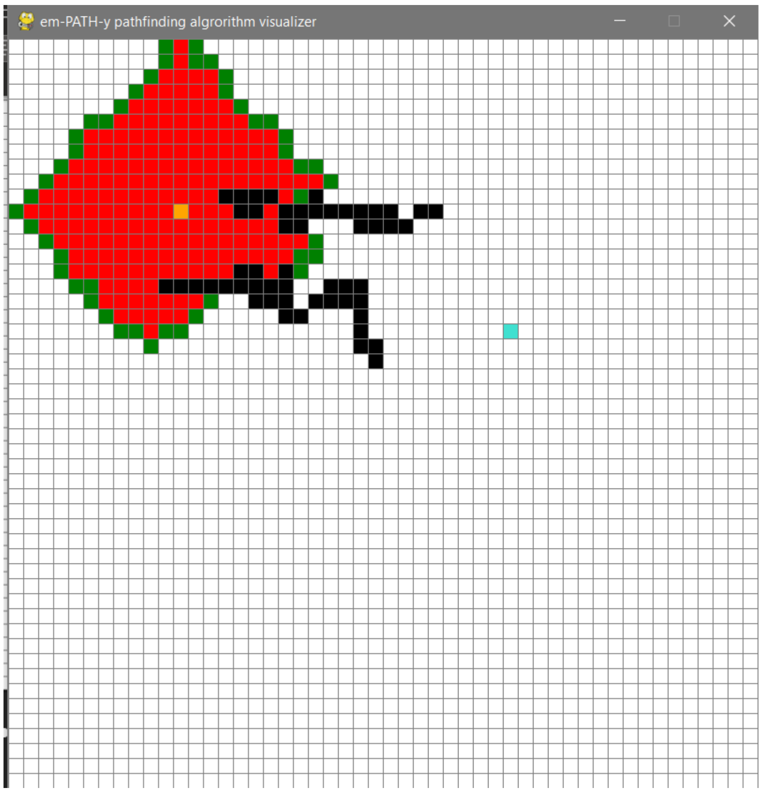
- idle window with user picked start, end and obstacles nodes



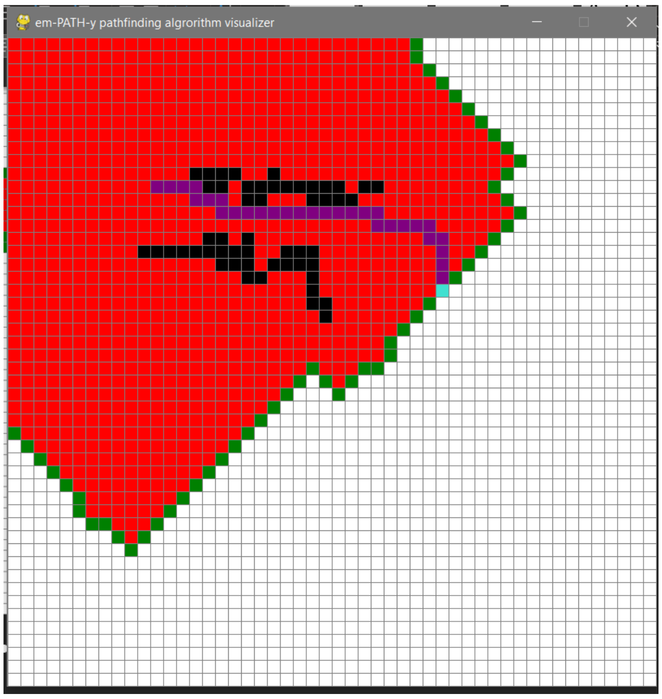
- running  A* (star) algorithm window



- completed A* (star) algorithm window



- running dijkstra's algorithm window



- completed dijkstra's algorithm window

## E. References

*Isaac computer science*. (n.d.). Isaac Computer Science. https://isaaccomputerscience.org/concepts/dsa_search_dijkstra?examBoard=all&stage=all

*Lecture 5: Search: Optimal, branch and bound, a* | Lecture videos | Artificial intelligence | Electrical engineering and computer science | MIT OpenCourseWare*. (n.d.). MIT OpenCourseWare | Free Online Course Materials. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/lecture-videos/lecture-5-search-optimal-branch-and-bound-a/

Szabo, F.E. 2015. The Linear Algebra Survival Guide. Academic Press, Pages 219-233.ISBN 9780124095205. https://doi.org/10.1016/B978-0-12-409520-5.50020-5.