Open Source Development Lab

Python Project

A* Path Finding Visualization

17BIT027 – Harsh Naik

17BIT036 – Meet Katrodiya

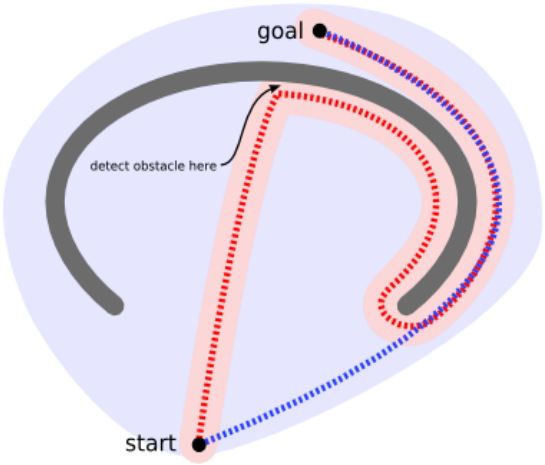
Description:

A* (A star) path finding algorithm is one of the best and popular technique used in path-finding and graph traversals.

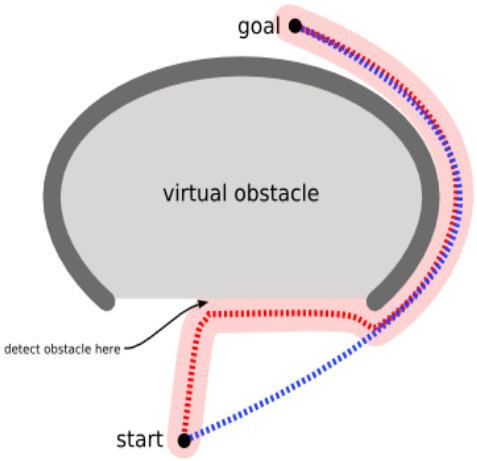
The major benefit for using this algorithm over Dijkstra's algorithm is that, this algorithm takes into account the direction of destination while finding the shortest path (Dijkstra's algorithm considers only the shortest path from the source to the given point). A* algorithm uses heuristic to guide itself in finding the shortest path.

On a map with many obstacles, pathfinding from points A to B can be difficult. A robot, for instance, without getting much other direction, will continue until it encounters an obstacle, as in the path-finding example to the left below.

However, with heuristic used in A* algorithm, essentially planning ahead at each step so a more optimal decision is made. With A*, a robot would rather find a path in a way similar to the diagram on the right below.



Inefficient way to find path



A path finding*

Algorithm explanation:

Like Dijkstra, A* works by making a lowest-cost path tree from the start node to the destination node. What makes A* different and better for many scenarios is that for each node, A* uses a function f(n) that gives an estimate of the total cost of a path using that node. Therefore, A* is a heuristic function, which differs from an algorithm in that a heuristic is more of an estimate and is not necessarily provably correct.

A* expands paths that are already less expensive by using this function:

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

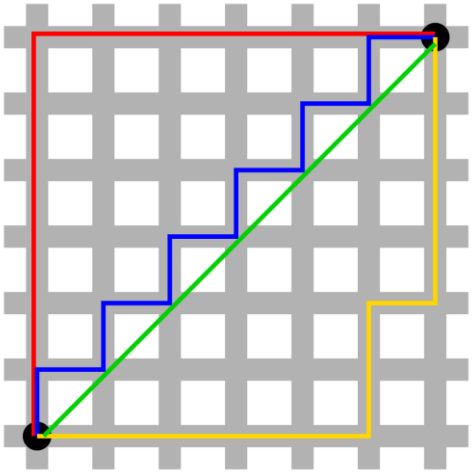
f(n) = total estimated cost of path through node n

g(n) = cost so far to reach node n

h(n) = estimated cost from n to goal. This is the heuristic part of the cost function, so it is like a smart guess. It can be Euclidian distance, Manhattan distance, etc.

A* algorithm begins at the start node, and considers all adjacent nodes. Once the list of adjacent nodes has been populated, it filters out those which are inaccessible (walls, obstacles, out of bounds). It then picks the cell with the lowest cost, which is the estimated f(n). This process is recursively repeated until the shortest path has been found to the destination node. The computation of f(n) is done via a heuristic that usually gives good results.

Here, in the implementation we have used Manhattan distance as a heuristic, which can be considered as L-distance



Manhattan distance between two points (x1,y1) & (x2,y2) is:

$$h = |x1 - x2| + |y1 - y2|$$

Implementation in Python:

Requirements:

- Python3
- pygame
- math

Spot class:

Each cell in a grid is an object of a Spot class. Its members are x, y positions, color, its neighbors, width, total rows and row, column.

Important functions:

h - Calculates Manhattan distance (heuristic) between two points

make_grid – For creating a grid (2D-list) of Spot objects

draw_grid - Used for drawing horizontal and vertical line to surround create cell boundaries.

draw - For drawing the window with grid

main - This function is used for putting start node, end node and obstacles in the grid. It is possible to change the position of nodes with right mouse button.

get_clicked_pos - This function is used for getting position (x, y coordinates) of clicked spot.

algorithm – A* path finding algorithm is implemented here.

update_neighbor – For a given spot update all of its neighbors that are not obstacles.

reconstruct_path - This function creates final shortest path by backtracking from the end node.

Code:

```
1. import pygame
2. import math
3. from queue import PriorityQueue
4.
5. WIDTH = 700
6. WIN = pygame.display.set_mode((WIDTH, WIDTH))
7. pygame.display.set_caption("A* Path Finding Algorithm Visulization")
8.
9. RED = (255, 0, 0)
10. GREEN = (0, 255, 0)
11. BLUE = (0, 255, 0)
12. YELLOW = (255, 255, 0)
```

```
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
PURPLE = (82, 0, 204)
ORANGE = (255, 165 ,0)
GREY = (128, 128, 128)
TURQUOISE = (0, 179, 179)

class Spot:
    def __init__(self, row, col, width, total_rows):
        self.row = row
        self.col = col
        self.x = row * width
        self.y = col * width
        self.color = WHITE
        self.neighbors = []
        self.width = width
        self.total_rows = total_rows

    def get_pos(self):
        return self.row, self.col

    def is_closed(self):
        return self.color == RED

    def is_open(self):
        return self.color == GREEN

    def is_barrier(self):
        return self.color == BLACK

    def is_start(self):
        return self.color == ORANGE

    def is_end(self):
        return self.color == TURQUOISE

    def reset(self):
        self.color = WHITE

    def make_start(self):
        self.color = ORANGE

    def make_closed(self):
        self.color = RED

    def make_open(self):
        self.color = GREEN

    def make_barrier(self):
        self.color = BLACK
```

```
63.
64.     def make_end(self):
65.         self.color = TURQUOISE
66.
67.     def make_path(self):
68.         self.color = PURPLE
69.
70.     def draw(self, win):
71.         pygame.draw.rect(win, self.color, (self.x, self.y, self.width, self.width))
72.
73.     def update_neighbors(self, grid):
74.         self.neighbors = []
75.         if self.row < self.total_rows - 1 and not grid[self.row + 1][self.col].is_barrier(): # DOWN
76.             self.neighbors.append(grid[self.row + 1][self.col])
77.
78.         if self.row > 0 and not grid[self.row - 1][self.col].is_barrier(): # UP
79.             self.neighbors.append(grid[self.row - 1][self.col])
80.
81.         if self.col < self.total_rows - 1 and not grid[self.row][self.col + 1].is_barrier(): # RIGHT
82.             self.neighbors.append(grid[self.row][self.col + 1])
83.
84.         if self.col > 0 and not grid[self.row][self.col - 1].is_barrier(): # LEFT
85.             self.neighbors.append(grid[self.row][self.col - 1])
86.
87.     def __lt__(self, other):
88.         return False
89.
90.
91. def h(p1, p2): #L-distance as heuristic
92.     x1, y1 = p1
93.     x2, y2 = p2
94.     return abs(x1 - x2) + abs(y1 - y2)
95.
96.
97. def reconstruct_path(came_from, current, draw):
98.     while current in came_from:
99.         current = came_from[current]
100.                 current.make_path()
101.                 draw()
102.
103.
104.         def algorithm(draw, grid, start, end):
105.             count = 0
106.             open_set = PriorityQueue()
107.             open_set.put((0, count, start))
108.             came_from = {}
109.             g_score = {spot: float("inf") for row in grid for spot in row}
110.             g_score[start] = 0
111.             f_score = {spot: float("inf") for row in grid for spot in row}
112.             f_score[start] = h(start.get_pos(), end.get_pos())
```

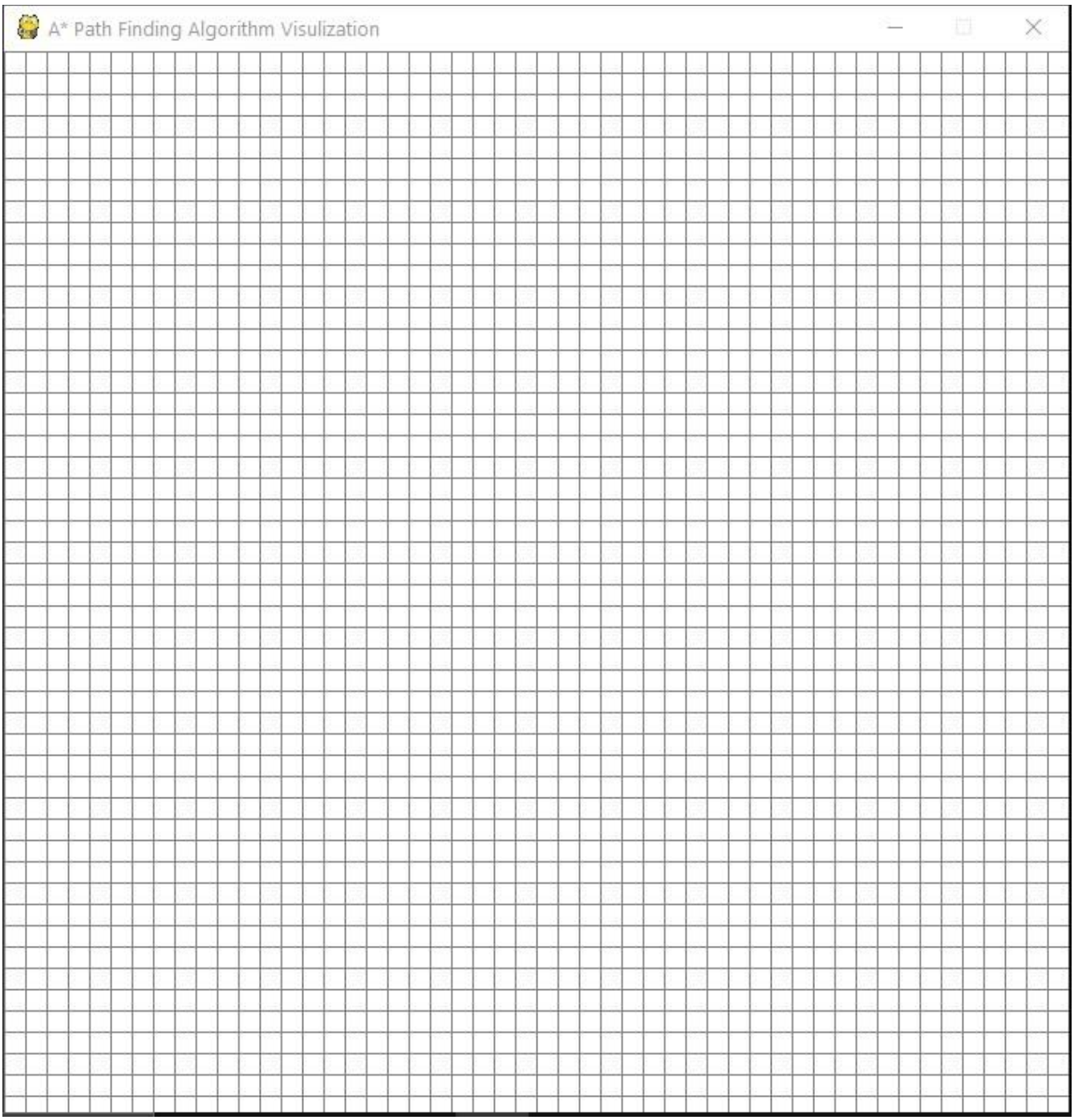
```
113.
114.         open_set_hash = {start}
115.
116.         while not open_set.empty():
117.             for event in pygame.event.get():
118.                 if event.type == pygame.QUIT:
119.                     pygame.quit()
120.
121.             current = open_set.get()[2]
122.             open_set_hash.remove(current)
123.
124.             if current == end:
125.                 reconstruct_path(came_from, end, draw)
126.                 end.make_end()
127.                 start.make_start()
128.                 return True
129.
130.             for neighbor in current.neighbors:
131.                 temp_g_score = g_score[current] + 1
132.
133.                 if temp_g_score < g_score[neighbor]:
134.                     came_from[neighbor] = current
135.                     g_score[neighbor] = temp_g_score
136.                     f_score[neighbor] = temp_g_score + h(neighbor.get_pos(), end.get_pos())
137.                     if neighbor not in open_set_hash:
138.                         count += 1
139.                         open_set.put((f_score[neighbor], count, neighbor))
140.                         open_set_hash.add(neighbor)
141.                         neighbor.make_open()
142.
143.             draw()
144.
145.             if current != start:
146.                 current.make_closed()
147.
148.         return False
149.
150.
151.     def make_grid(rows, width):
152.         grid = []
153.         gap = width // rows
154.         for i in range(rows):
155.             grid.append([])
156.             for j in range(rows):
157.                 spot = Spot(i, j, gap, rows)
158.                 grid[i].append(spot)
159.
160.         return grid
161.
162.
```

```
def draw_grid(win, rows, width):
    gap = width // rows
    for i in range(rows):
        pygame.draw.line(win, GREY, (0, i * gap), (width, i * gap))
        for j in range(rows):
            pygame.draw.line(win, GREY, (j * gap, 0), (j * gap, width))


def draw(win, grid, rows, width):
    win.fill(WHITE)

    for row in grid:
        for spot in row:
            spot.draw(win)

    draw_grid(win, rows, width)
    pygame.display.update()


def get_clicked_pos(pos, rows, width):
    gap = width // rows
    y, x = pos

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

    return row, col


def main(win, width):
    ROWS = 50
    grid = make_grid(ROWS, width)

    start = None
    end = None

    run = True
    while run:
        draw(win, grid, ROWS, width)
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False

            if pygame.mouse.get_pressed()[0]: # LEFT
                pos = pygame.mouse.get_pos()
                row, col = get_clicked_pos(pos, ROWS, width)
                spot = grid[row][col]
                if not start and spot != end:
                    start = spot
                    start.make_start()
```

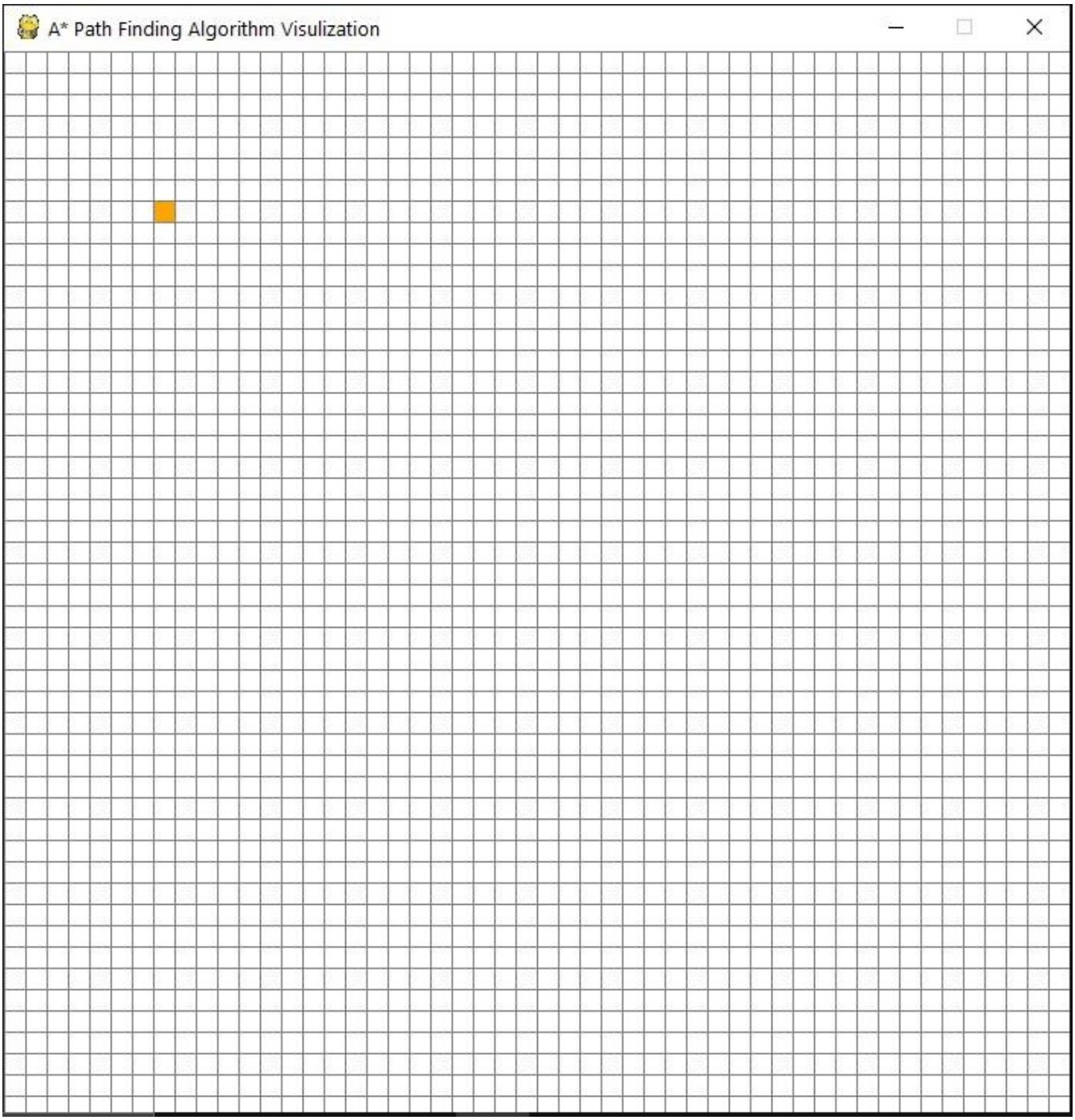
```
213.
214.                 elif not end and spot != start:
215.                     end = spot
216.                     end.make_end()
217.
218.                 elif spot != end and spot != start:
219.                     spot.make_barrier()
220.
221.             elif pygame.mouse.get_pressed()[2]: # RIGHT
222.                 pos = pygame.mouse.get_pos()
223.                 row, col = get_clicked_pos(pos, ROWS, width)
224.                 spot = grid[row][col]
225.                 spot.reset()
226.                 if spot == start:
227.                     start = None
228.                 elif spot == end:
229.                     end = None
230.
231.             if event.type == pygame.KEYDOWN:
232.                 if event.key == pygame.K_SPACE and start and end:
233.                     for row in grid:
234.                         for spot in row:
235.                             spot.update_neighbors(grid)
236.
237.                     algorithm(lambda: draw(win, grid, ROWS, width), grid, start, end)
238.
239.                 if event.key == pygame.K_c:
240.                     start = None
241.                     end = None
242.                     grid = make_grid(ROWS, width)
243.
244.         pygame.quit()
245.
246.     main(WIN, WIDTH)
```

Output:

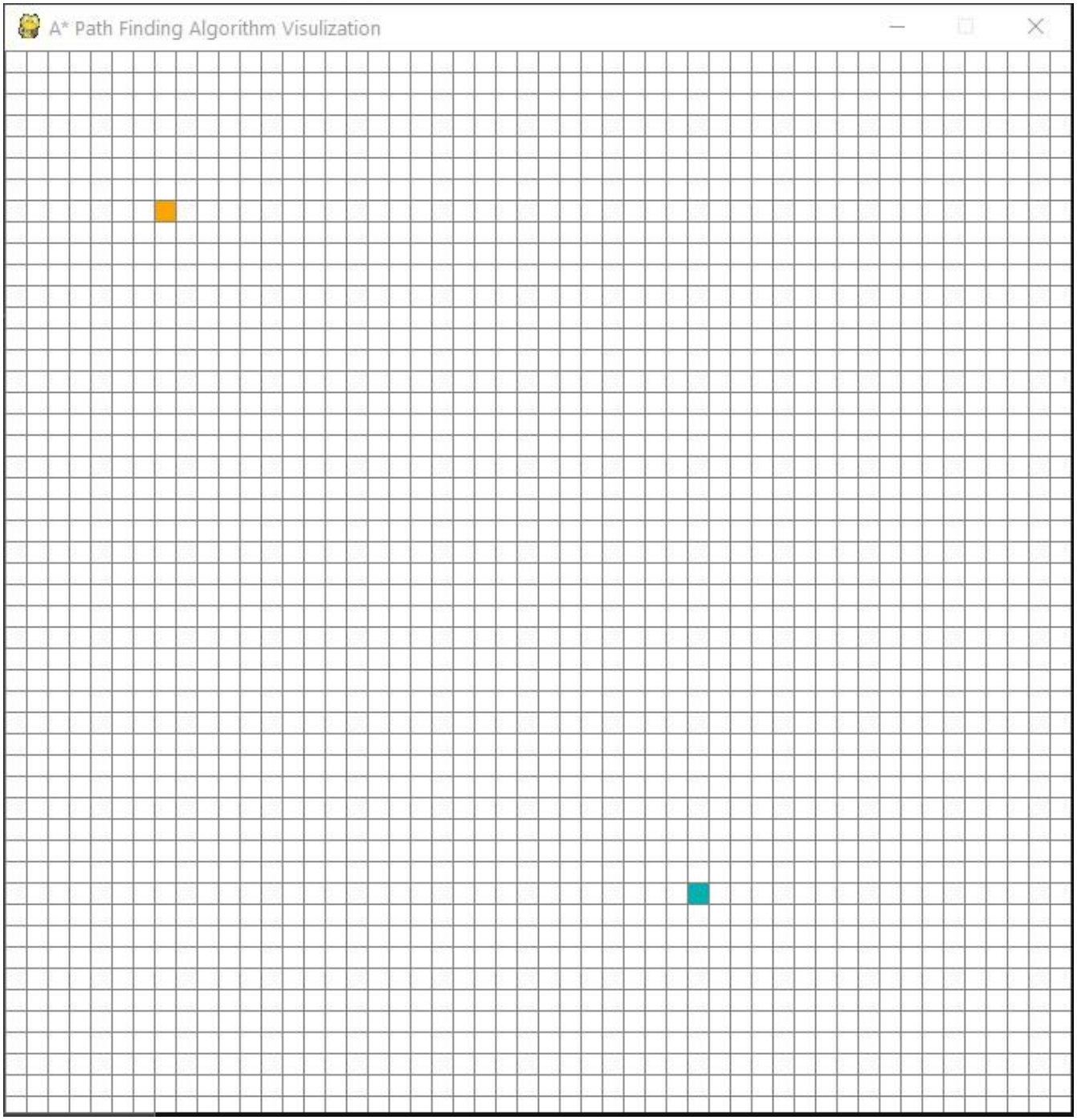
1. Upon executing the above code following pygame window will be opened consisting of 50x50 grid. (At any point in time, clicking ‘c’ key will reset the grid as following)



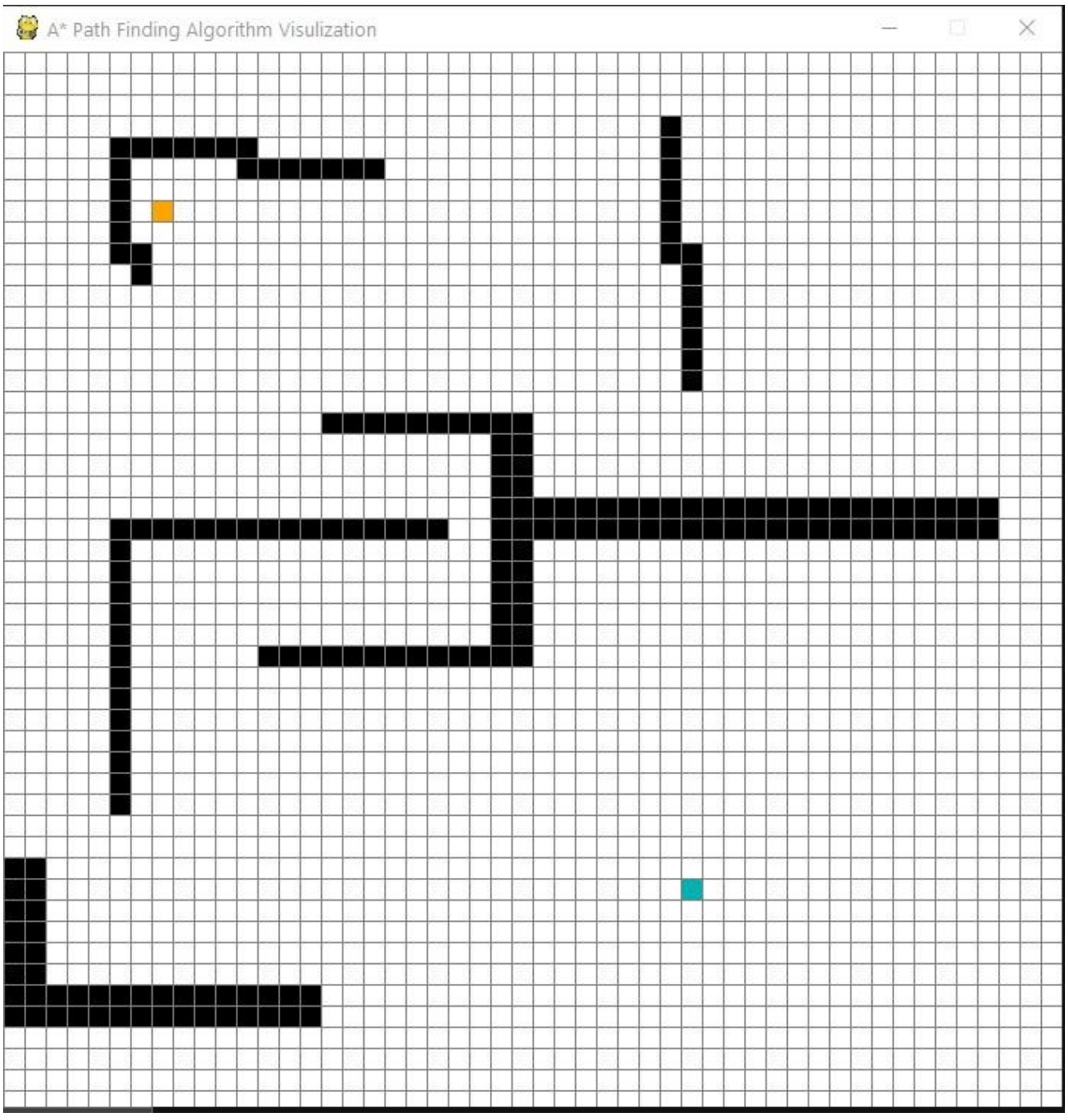
2. Clicking left mouse button on any cell will mark it as starting cell. Right clicking on the same cell will remove it as start node. (It is represented by orange color)



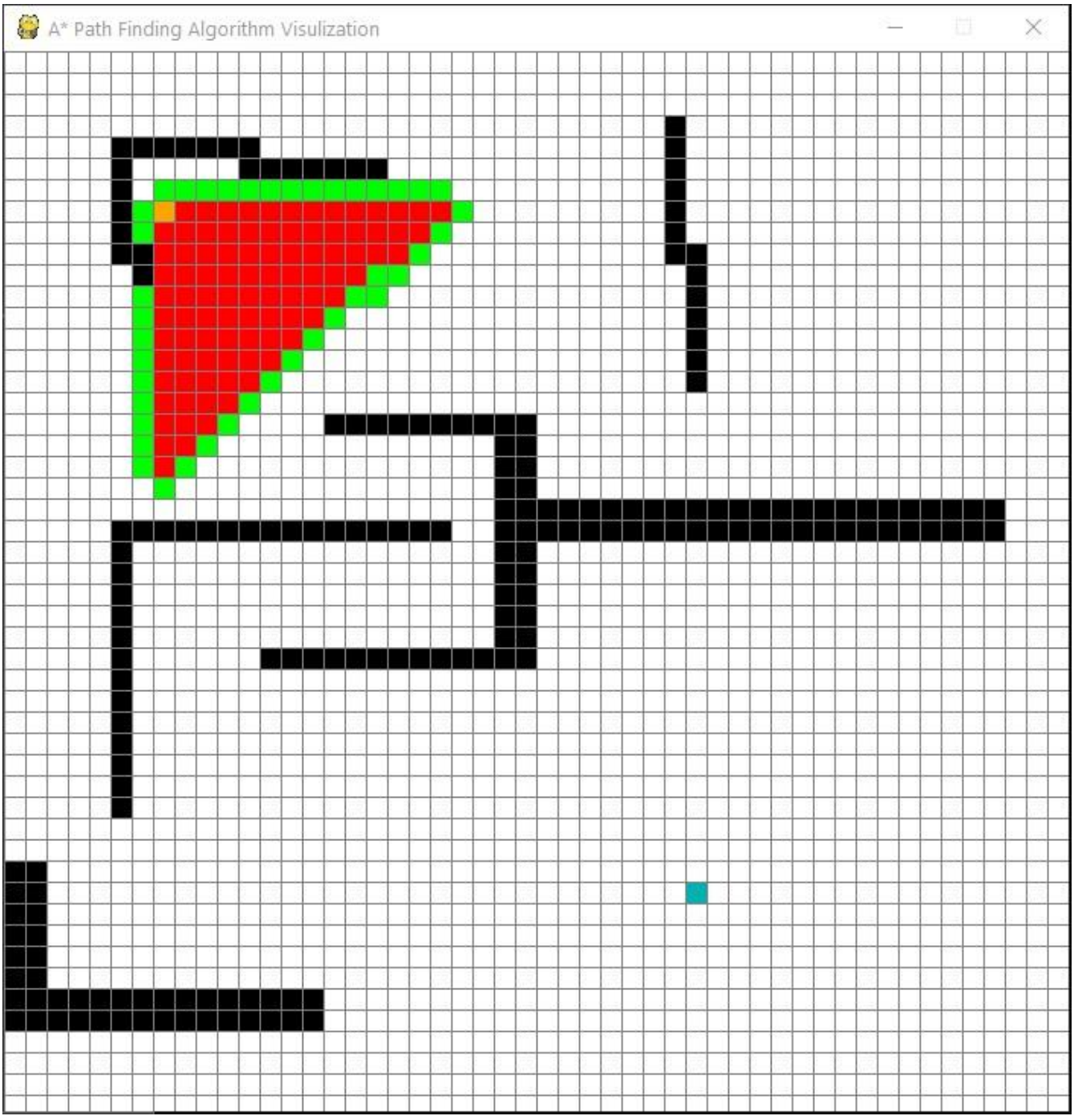
3. Clicking left mouse button on any cell other than start node will mark it as destination cell. Right clicking on the same cell will remove it as start node. (It is represented by turquoise color)



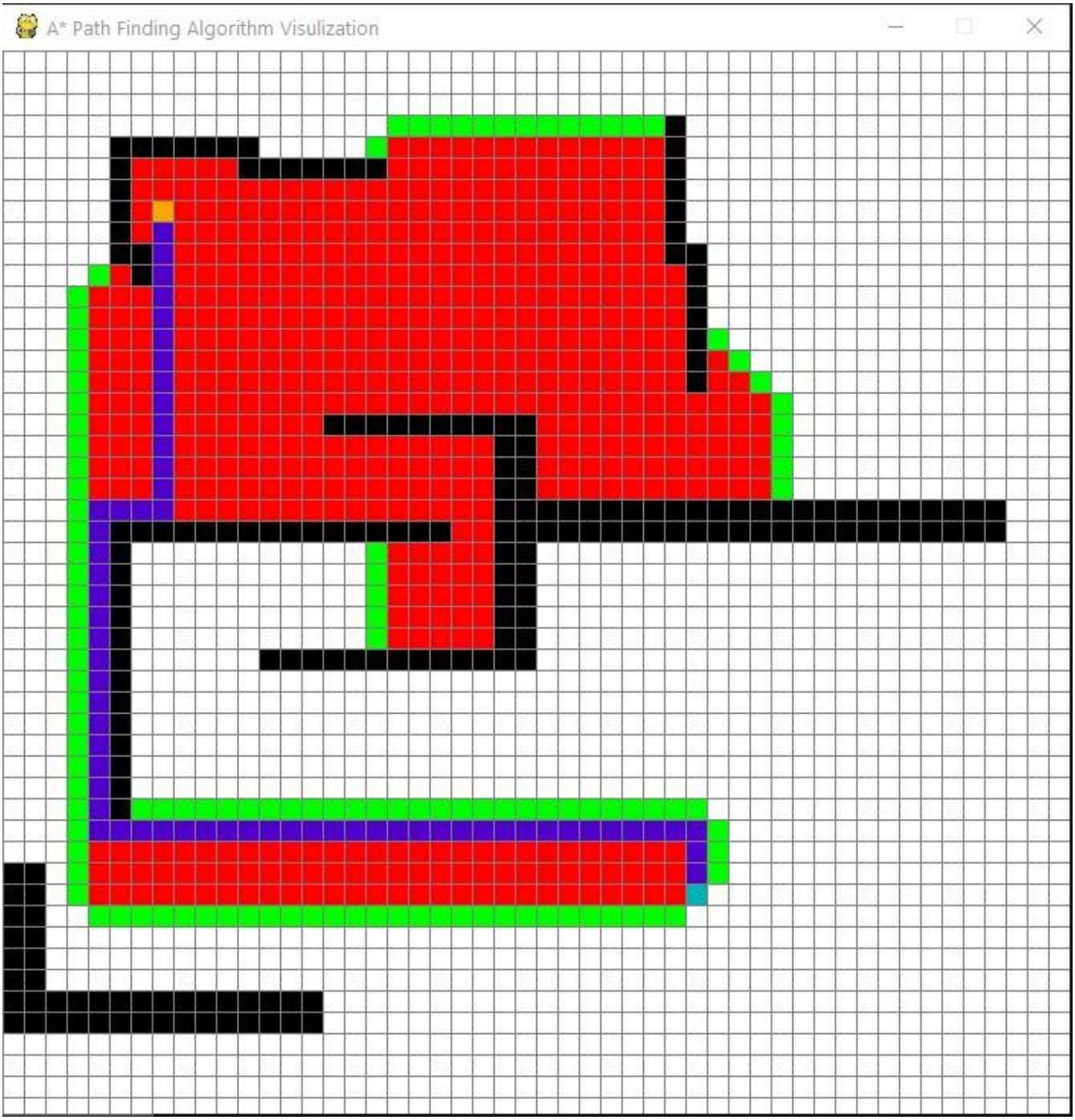
4. Now clicking left mouse button on any cell will mark it as an obstacle. Clicking right button on any obstacle will remove it from obstacle. (It is represented by black color)



5. Pressing spacebar will start finding the shortest path between the start and destination node. (Red color cells are closed nodes, green color cells are open nodes)



6. After the shortest path has been found, it will be represented by purple color path.



Conclusion:

A-star (A*) is a mighty algorithm in Artificial Intelligence with a wide range of usage. However, it is only as good as its heuristic function (which can be highly variable considering the nature of a problem). A* is the most popular choice for pathfinding because it's reasonably flexible.

It has found applications in many software systems, from Machine Learning and search Optimization to game development where characters navigate through complex terrain and obstacles to reach the player.

By doing this project we learned about the algorithm and how to make path finding visualization with python.