Shortest Path Visualizer: Greedy Approach

Presented By :

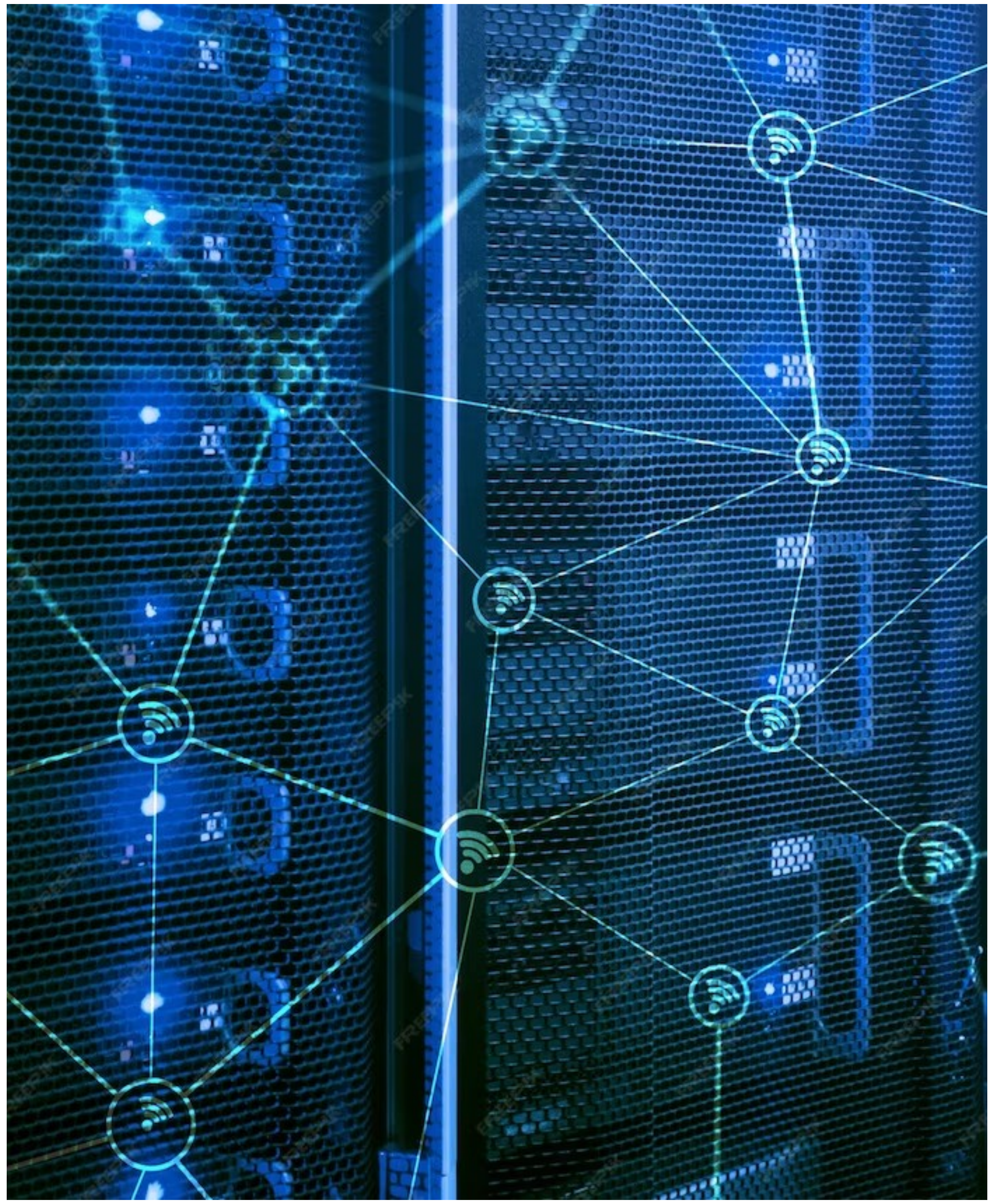
Sai Keshav (RA2211003011036)

Anwar Babu (RA2211003011042)

Rohith Kumar (RA2211003011044)

ABSTRACT

The ” Shortest Path Visualizer Using Greedy Approach " project is a Python-based application that demonstrates the Dijkstra's algorithm for finding the shortest path in a grid-based environment. Using the **Pygame** library for graphical user interface, the project allows users to interact with a grid, designating start and end points, as well as barriers. The application visually illustrates the process of finding the shortest path from the start point to the end point, highlighting the path in real-time. Users can clear the grid and create new scenarios for pathfinding. This project provides an educational and interactive tool for understanding the inner workings of the Dijkstra's algorithm, which has applications in fields such as computer science and robotics.

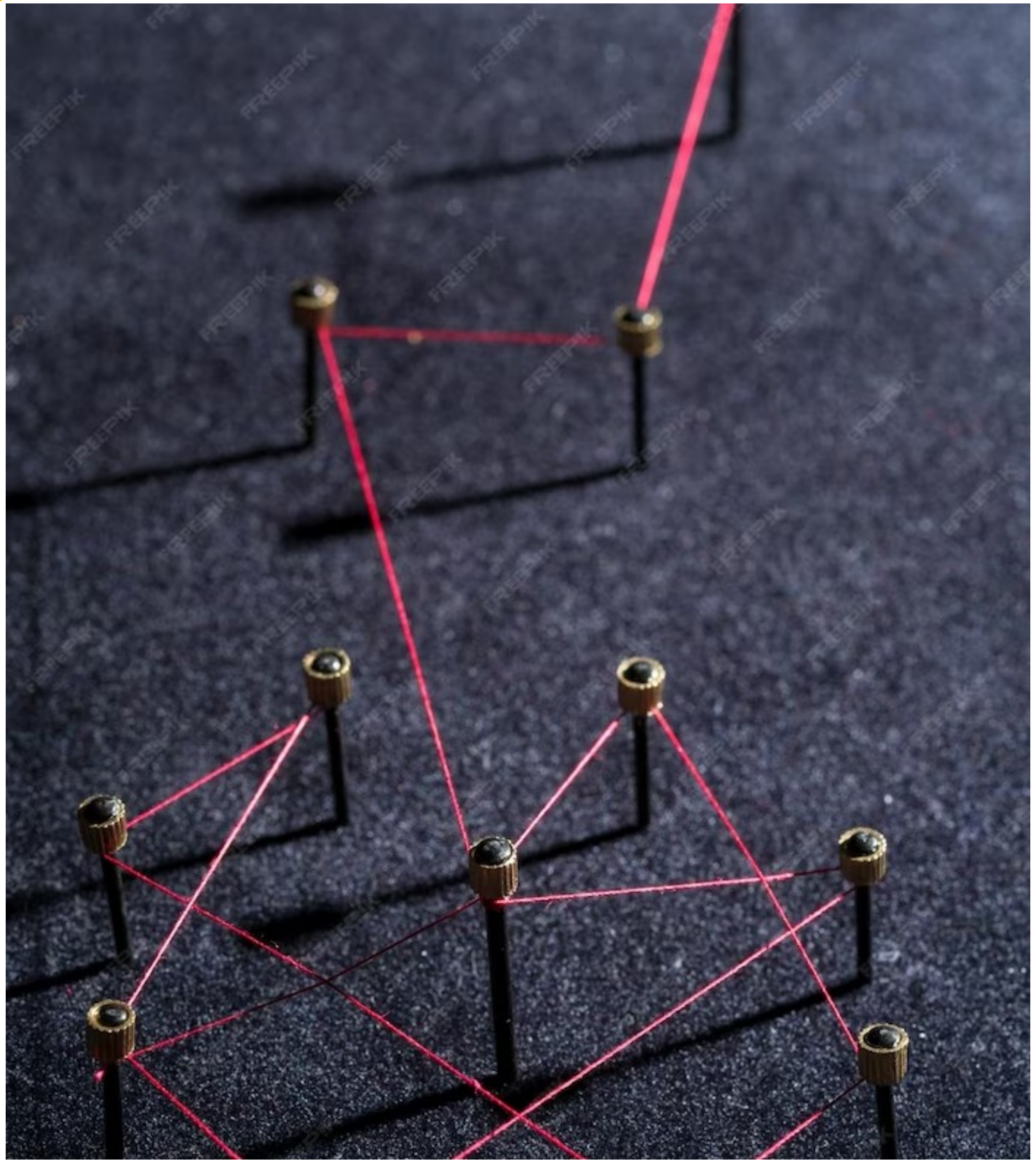


OBJECTVE

1. **Visualization of Dijkstra's Algorithm:** Provide a visual representation of Dijkstra's algorithm for finding the shortest path between two points on a grid. This helps users understand how the algorithm works.
2. **User Interaction:** Allow users to interact with the grid, designating the start and end points and adding barriers to customize the scenario. This hands-on interaction enhances the learning experience.
3. **Real-time Pathfinding:** Demonstrate the pathfinding process in real-time, highlighting the explored nodes and the shortest path as it unfolds. This dynamic visualization aids in comprehending the algorithm's step-by-step execution.
4. **Clear Grid Functionality:** Enable users to reset the grid, allowing them to experiment with different scenarios and learn how the algorithm adapts to changing conditions.
5. **Educational Tool:** Serve as an educational resource for teaching and learning about pathfinding algorithms, their applications in computer science and robotics, and their significance in solving real-world problems.

PROBLEM STATEMENT

In the realm of computer science education, comprehending complex algorithms like Dijkstra's pathfinding method can be challenging. There is a pressing need for an interactive, real-time visualization tool that simplifies the learning process, addressing issues of algorithm complexity. Educators and students often struggle to grasp these intricate concepts without practical, visual aids. To bridge this gap, we aim to develop an intuitive, tool that makes learning Dijkstra's algorithm engaging and informative. This educational resource will allow users to observe the algorithm's operation in real-time, fostering a deeper understanding of pathfinding processes, which are essential in various fields like robotics, game development, and network routing.



KEY PARTS IN THE CODE

1) Importing Libraries :

The code begins by importing essential libraries, including **pygame** for rendering graphics and queue. PriorityQueue for the priority queue implementation.

2) Creating a Grid :

The grid is represented as a 2D list of "spots." Each spot represents a cell on the grid with attributes like its position, color, and neighbors.

Spot class defines a grid cell. It has methods to check if a cell is open, closed, a barrier, or the start/end, and to change the cell's state.

3) Greedy Dijkstra's Algorithm:

The algorithm function implements Dijkstra’s greedy algorithm for path finding. It uses a priority queue to keep track of open set nodes.

The algorithm processes neighbors of the current node and calculates tentative scores to find the optimal path.

4) User Interaction:

Users interact with the grid by clicking on cells. Left-click sets the start, end, and barriers, and right-click clears cells.

The space bar triggers the visualization of the algorithm.

The 'c' key clears the grid for a new pathfinding attempt.

5. Real-Time Visualization:

Real-time visualization of the algorithm's execution is achieved through the draw function, which continually updates the grid.

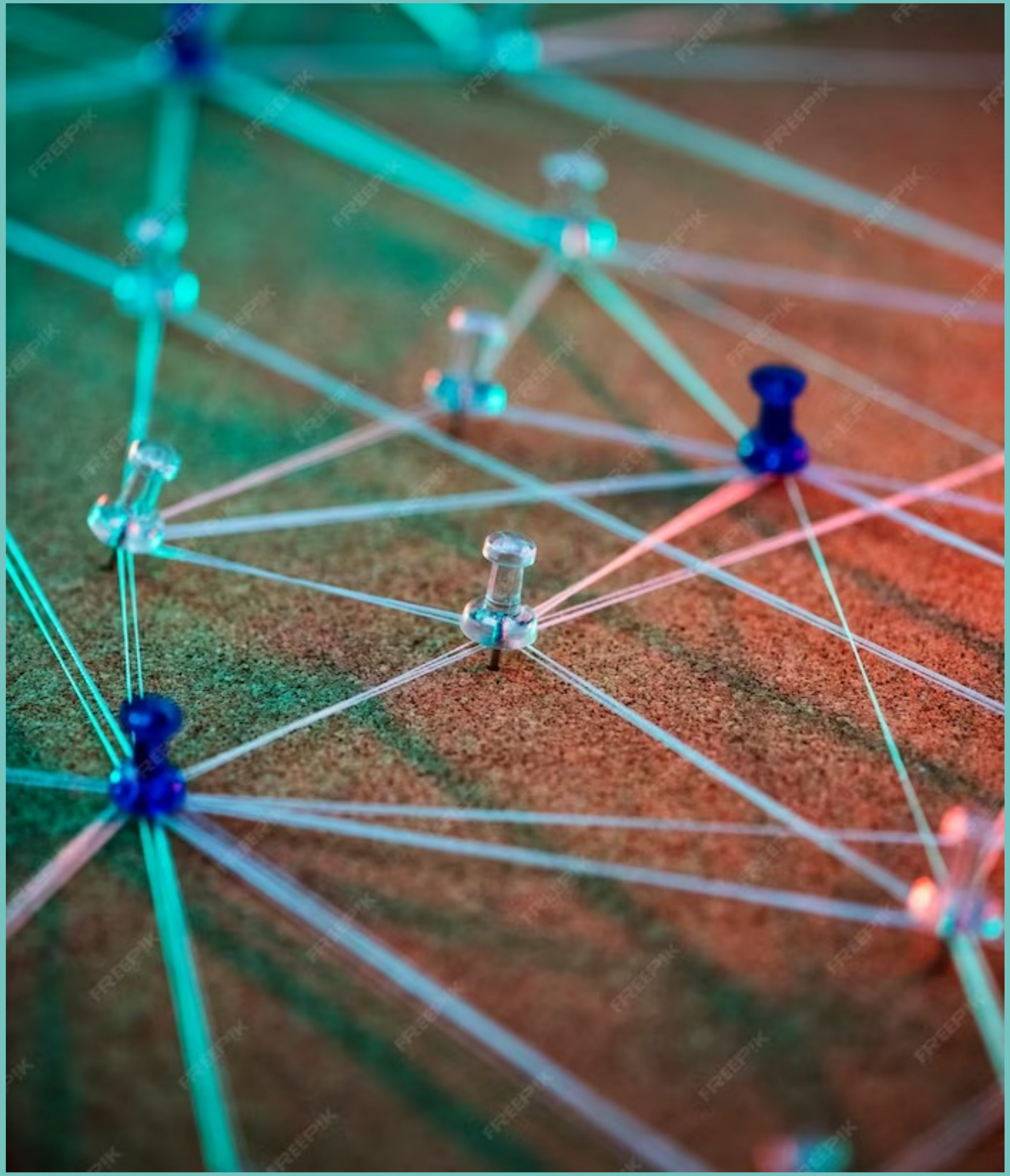
The pygame library is used for rendering and graphics display. Grid lines and cell colors change as the algorithm progresses.

6. Main Function:

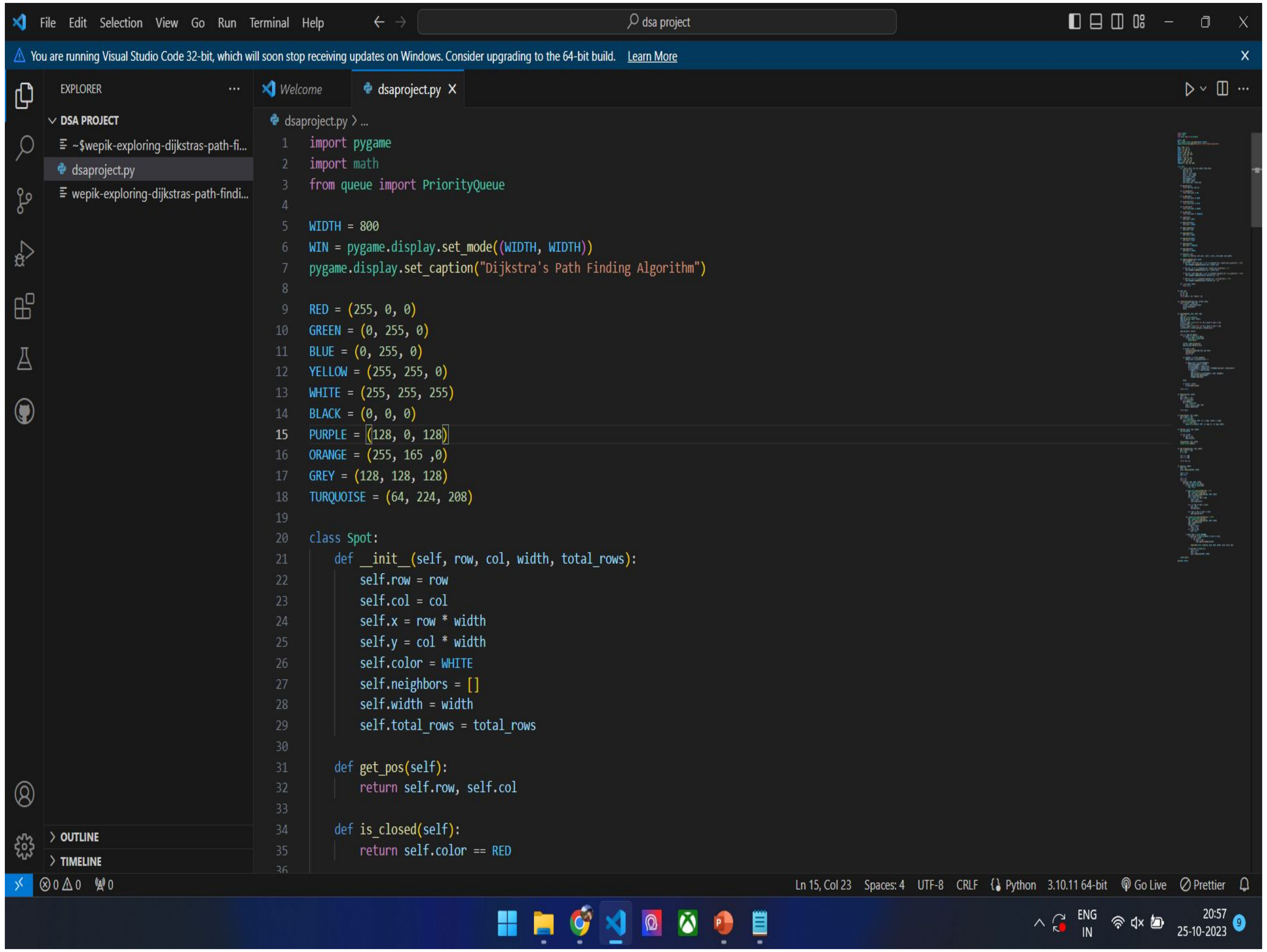
The main function initializes the grid, waits for user input, and interacts with the user to set up the pathfinding problem.

7. GUI and User Interface:

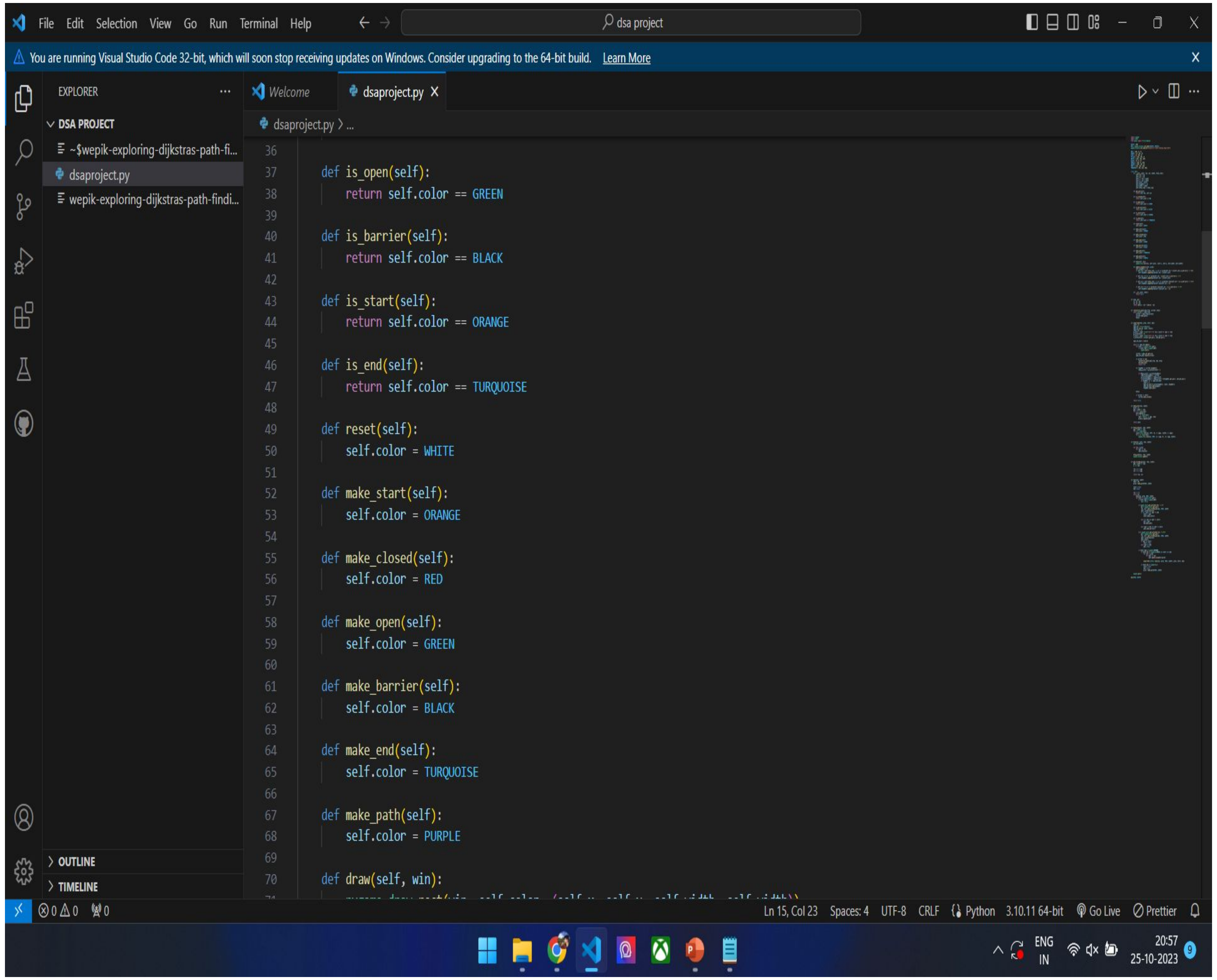
The code provides a graphical user interface (GUI) for users to interact with the grid and visualize the algorithm's execution in real-time.



IMPLEMENTATION

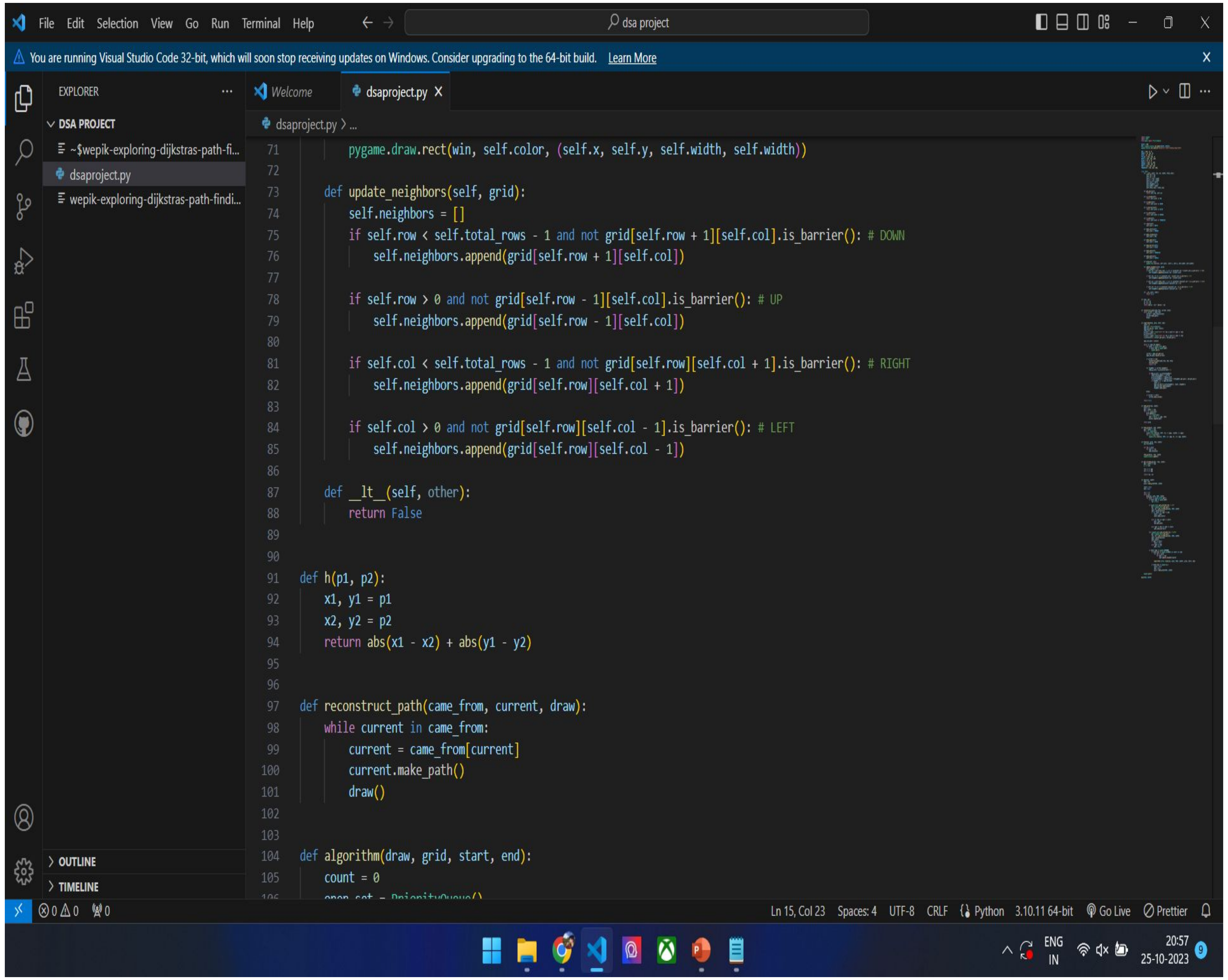


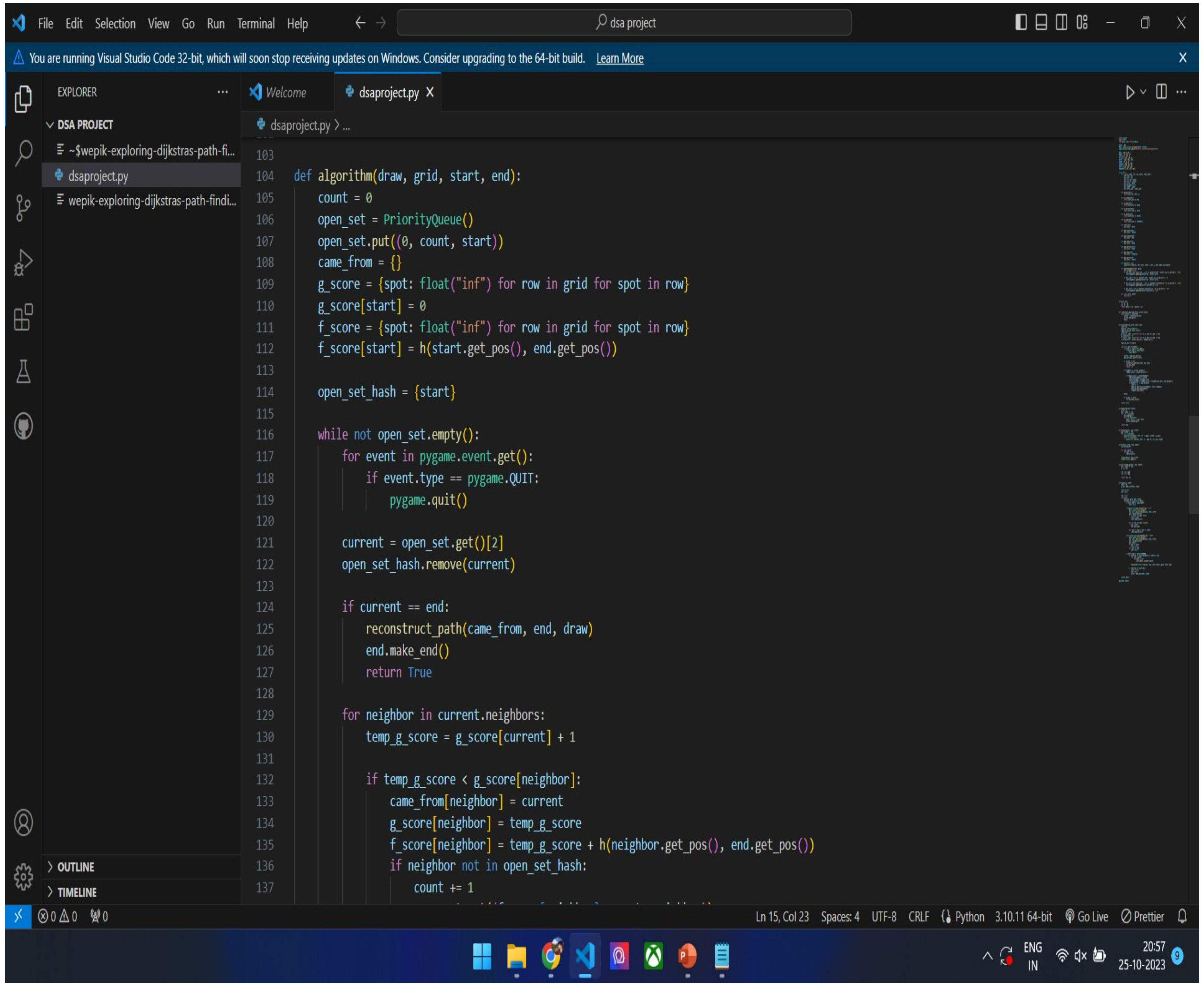
```python
import pygame
import math
from queue import PriorityQueue

WIDTH = 800
WIN = pygame.display.set_mode((WIDTH, WIDTH))
pygame.display.set_caption("Dijkstra's Path Finding Algorithm")

RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 255, 0)
YELLOW = (255, 255, 0)
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
PURPLE = (128, 0, 128)
ORANGE = (255, 165 ,0)
GREY = (128, 128, 128)
TURQUOISE = (64, 224, 208)

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
```



```python
    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

    def make_end(self):
        self.color = TURQUOISE

    def make_path(self):
        self.color = PURPLE

    def draw(self, win):
```

IMPLEMENTATION





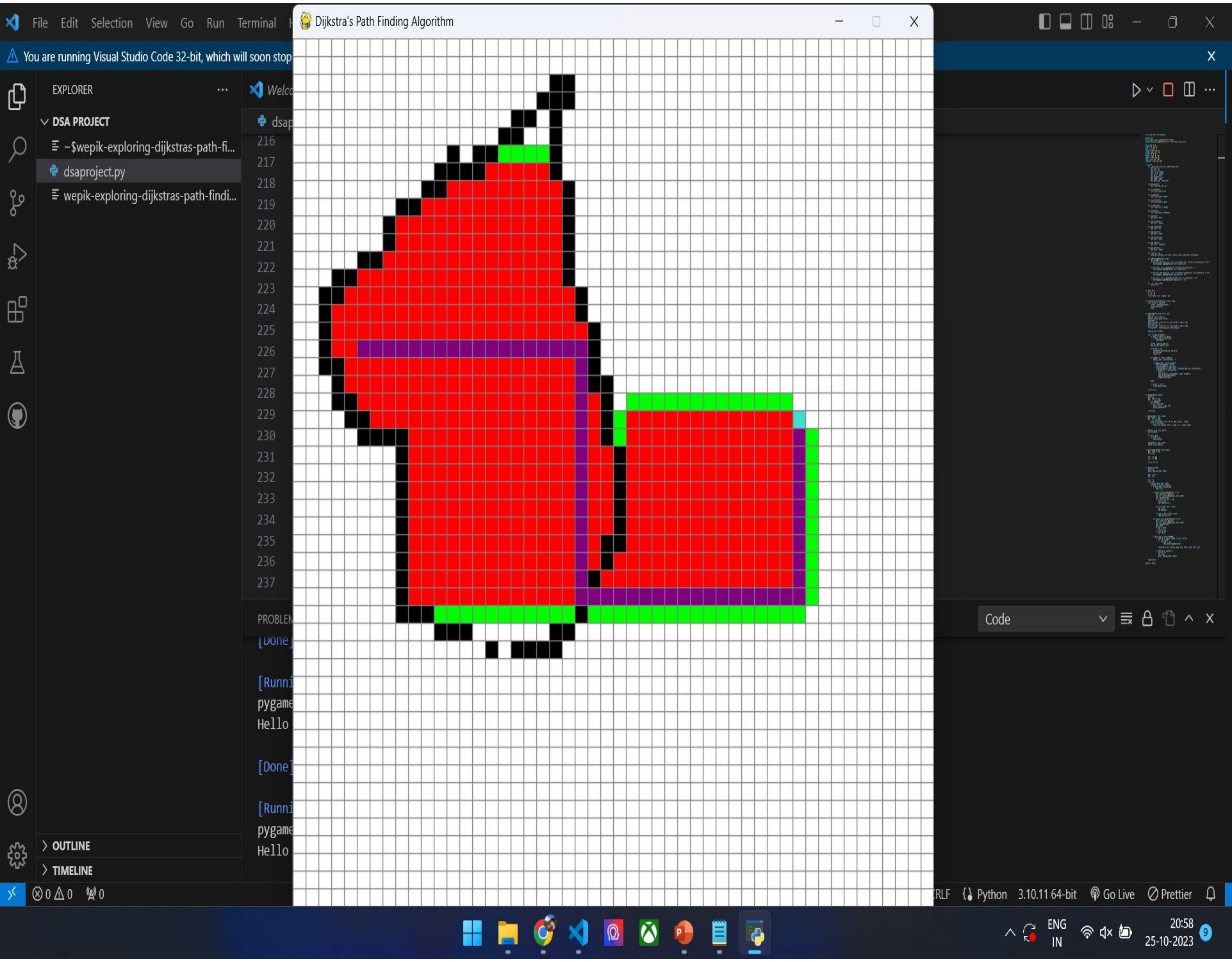
```python
                g_score[neighbor] = temp_g_score
                f_score[neighbor] = temp_g_score + h(neighbor.get_pos(), end.get_pos())
                if neighbor not in open_set_hash:
                    count += 1
                    open_set.put((f_score[neighbor], count, neighbor))
                    open_set_hash.add(neighbor)
                    neighbor.make_open()

        draw()

        if current != start:
            current.make_closed()

    return False


def make_grid(rows, width):
    grid = []
    gap = width // rows
    for i in range(rows):
        grid.append([])
        for j in range(rows):
            spot = Spot(i, j, gap, rows)
            grid[i].append(spot)

    return grid


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))

```

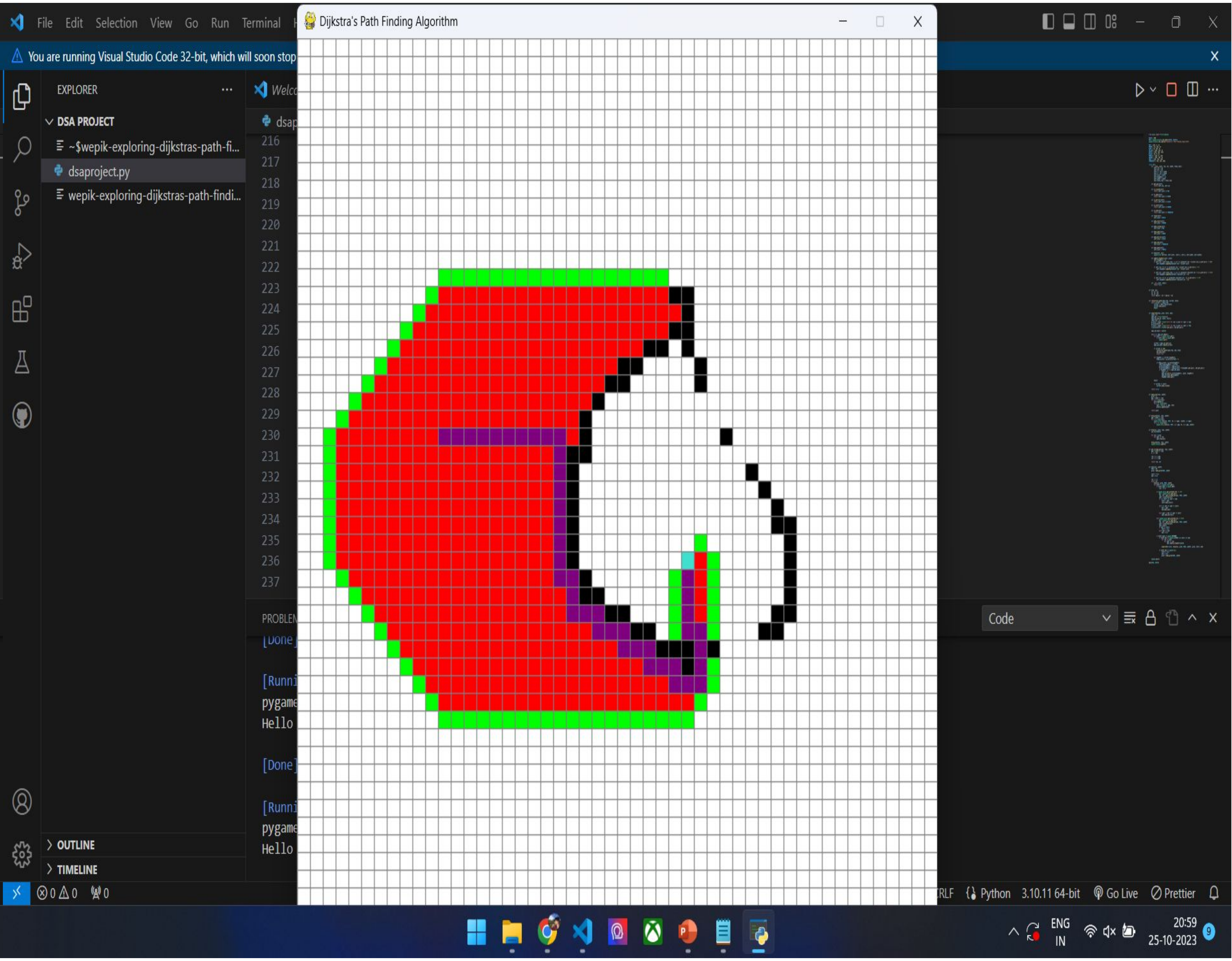
```python


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:
```

```python
        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()

                elif not end and spot != start:
                    end = spot
                    end.make_end()

                elif spot != end and spot != start:
                    spot.make_barrier()

            elif pygame.mouse.get_pressed()[2]: # RIGHT
                pos = pygame.mouse.get_pos()
                row, col = get_clicked_pos(pos, ROWS, width)
                spot = grid[row][col]
                spot.reset()
                if spot == start:
                    start = None
                elif spot == end:
                    end = None

            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_SPACE and start and end:
                    for row in grid:
                        for spot in row:
                            spot.update_neighbors(grid)

                    algorithm(lambda: draw(win, grid, ROWS, width), grid, start, end)
```

```python

                elif spot != end and spot != start:
                    spot.make_barrier()

            elif pygame.mouse.get_pressed()[2]: # RIGHT
                pos = pygame.mouse.get_pos()
                row, col = get_clicked_pos(pos, ROWS, width)
                spot = grid[row][col]
                spot.reset()
                if spot == start:
                    start = None
                elif spot == end:
                    end = None

            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_SPACE and start and end:
                    for row in grid:
                        for spot in row:
                            spot.update_neighbors(grid)

                    algorithm(lambda: draw(win, grid, ROWS, width), grid, start, end)

                if event.key == pygame.K_c:
                    start = None
                    end = None
                    grid = make_grid(ROWS, width)

    pygame.quit()

main(WIN, WIDTH)
```

OUTPUT


File Edit Selection View Go Run Terminal
Dijkstra's Path Finding Algorithm
You are running Visual Studio Code 32-bit, which will soon stop
EXPLORER
DSA PROJECT
~$wepik-exploring-dijkstras-path-fi...
dsaproject.py
wepik-exploring-dijkstras-path-findi...
OUTLINE
TIMELINE
Python 3.10.11 64-bit
Go Live
Prettier
ENG
IN
20:58
25-10-2023


File Edit Selection View Go Run Terminal
Dijkstra's Path Finding Algorithm
You are running Visual Studio Code 32-bit, which will soon stop
EXPLORER
DSA PROJECT
~$wepik-exploring-dijkstras-path-fi...
dsaproject.py
wepik-exploring-dijkstras-path-findi...
OUTLINE
TIMELINE
Python 3.10.11 64-bit
Go Live
Prettier
ENG
IN
20:59
25-10-2023

CONCLUSION

In this project, we have successfully implemented Dijkstra's Pathfinding Algorithm using Python and the Pygame library. This algorithm provides an efficient way to find the shortest path from a start point to an end point on a grid while avoiding obstacles. The project combines data structures, visualization, and interactive user input to create an engaging and educational tool for understanding pathfinding algorithms.

The user can interact with the application by placing the start and end points, as well as barriers, on the grid. The algorithm then efficiently calculates the shortest path, and the visualization illustrates the process. This project serves as an excellent educational resource for students, teachers, or anyone interested in learning about pathfinding algorithms.

Furthermore, it can be a valuable tool for understanding the concepts of graph theory and algorithm optimization. The project's implementation and visualization make it a versatile and engaging educational resource.