Path Finding Algorithms Visualizer (Dijkstra, BFS, A*)

A PROJECT REPORT

Submitted by

A. LALITH RAHUL [Reg No: RA2211003011009]

AIHANT DEBNATH [Reg No: RA221103011033]

ARYA RAY [Reg No: RA221103011059]

for the course 21CSC201J Data Structures and Algorithms

Under the Guidance of

DR. ANITHA RAMESH

Associate Professor, Department of COMPUTER SCIENCE ENGINEERING

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR – 603 203

NOVEMBER 2023

SRM INSTITUTION OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Certified that the 21CSC201J Data Structures and Algorithms course project report titled **"Path Finding Algorithms Visualizer (Dijkstra, BFS, A*)"** is the bonafide work done by **A. LALITH RAHUL [Reg No: RA2211003011009], AIHANT DEBNATH [Reg No: RA221103011033] & ARYA RAY [Reg No: RA221103011059]** who carried out under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other work.

DR. ANITHA RAMESH,

Faculty In-Charge,

Assistant Professor,

Department of Computing Technologies,

SRMIST.

Dr. M. Pushpalatha,

Professor and Head,

Department of Computing Technologies,

SRMIST.

ACKNOWLEDGEMENT

We express our heartfelt thanks to our honorable **Vice Chancellor Dr. C. Muthamizhchelvan**, for being the beacon in all our endeavors.

We would like to express my warmth of gratitude to our **Registrar Dr. S. Ponnusamy,** for his encouragement.

We express our profound gratitude to our **Dean (College of Engineering and Technology) Dr. T. V.Gopal,** for bringing out novelty in all executions.

We would like to express my heartfelt thanks to **Chairperson**, School of Computing **Dr. Revathi Venkataraman** and **Dr. M. Pushpalatha, Head of The Department,** Department of Computing Technologies for imparting confidence to complete my course project.

We wish to express my sincere thanks to **Course Audit Professors Dr. Vadivu. G, Professor,** Department of Data Science and Business Systems and **Dr. Sasikala. E Professor,** Department of Computing Technologies and **Course Coordinator Mrs. G. Malarselvi,** Department of Computing Technologies for their constant encouragement and support.

We are thankful to our Course project Faculty **DR. ANITHA RAMESH,, Assistant Professor,** Department of Computing Technologies, for his assistance, timely suggestion and guidance throughout the duration of this course project.

Finally, we thank our parents and friends near and dear ones who directly and indirectly contributed to the successful completion of our project. Above all, I thank the almighty for showering his blessings on me to complete my Course project.

TABLE OF CONTENTS

PROBLEM STATEMENT:

The project set out to create an interactive visualizer using Pygame to demonstrate and compare the functionalities of Dijkstra's algorithm, Breadth-First Search (BFS), and A* (A-star) algorithm for pathfinding. The primary objective was to provide a platform where users could dynamically create custom grids, designate starting and target points, and introduce obstacles, allowing for the real-time visualization of these algorithms in action. The goal was to vividly illustrate and contrast the step-by-step processes of these algorithms as they navigate through the provided grid, emphasizing their individual approaches in determining the shortest path from the starting node to the target node. Dijkstra's algorithm was expected to reveal its exhaustive exploration of potential paths, BFS its systematic traversal of neighbouring nodes, and A* algorithm its ability to efficiently balance actual path cost and heuristic estimated cost. The intention behind this interactive tool was to enable users to observe and analyse how these algorithms tackle diverse scenarios, fostering a comprehensive understanding of their strengths, weaknesses, and distinct pathfinding strategies. The overall aim was to create an engaging and educational environment that aids users in comprehending the nuances and practical applications of these essential pathfinding algorithms.

DATA STRUCTURES:

For the pathfinding algorithm visualizer, a few data structures are primarily used:

1. **Grid Representation:** The grid itself is represented using a 2D list (list of lists). Each element in this 2D list corresponds to a 'Box' object. The 'Box' class represents individual nodes in the grid, holding information about their position, color, neighbors, and various methods for handling their state, such as setting start or end points, marking barriers, etc.
2. **Priority Queue (from queue module):** The Priority Queue is used specifically in the A* algorithm implementation to maintain and manage nodes (boxes) based on their priority. It helps in prioritizing nodes during traversal based on their respective final scores or heuristics, allowing the algorithm to efficiently explore potential paths.
3. **Deque (from collections module):** The deque is used in the Dijkstra's algorithm implementation to manage the queue-like structure required for traversing the grid nodes. It stores nodes in a first-in, first-out (FIFO) manner, aiding in the exploration of neighboring nodes.
4. **Other Python Data Types:** Throughout the code, other fundamental Python data types such as lists, tuples, integers, and dictionaries are employed for various purposes. Lists and tuples are used to manage coordinates and grid representations, integers for counting and indices, and dictionaries to store scores and relationships between nodes (like in the 'came_from' dictionary to reconstruct the path).

- **Elaboration:** Dijkstra's algorithm relies on a queue to explore nodes, ensuring that the nodes are examined in the order they are added. A deque provides efficient FIFO operations, making it well-suited for this task. It allows nodes to be added to the end of the queue and removed from the front, maintaining the correct order for exploration.

2. **Other Python Data Types (Lists, Tuples, Integers, Dictionaries):**

    - **Justification:** These fundamental data types are used throughout the code for various purposes The choice of data structures in the pathfinding algorithm visualizer code is crucial to efficiently manage the grid, track node states, and facilitate the operations necessary for the implementation of pathfinding algorithms. Here's a justification and elaboration for the use of these specific data structures:

3. **2D List (Grid Representation):**

    - **Justification:** A 2D list is a natural choice for representing the grid because it mirrors the grid's structure, with each element corresponding to a grid node (Box). It allows for straightforward access to specific nodes based on their row and column coordinates.

    - **Elaboration:** Using a 2D list simplifies grid creation and manipulation. It provides a direct mapping of the grid's layout, making it easy to access, modify, and iterate through grid nodes. It also allows for efficient neighbor exploration, as each node can reference its neighboring nodes within the list.

4. ***Priority Queue (from queue module) - Used in A Algorithm:****

    - **Justification:** A Priority Queue is essential for the A* algorithm, as it prioritizes nodes based on their final scores (a combination of path cost and heuristic). This allows A* to explore nodes efficiently in order of their estimated cost to reach the target.

    - **Elaboration:** Without a priority queue, A* would need to search through a large list of nodes to find the one with the lowest final score, which would be inefficient. A priority queue ensures that the nodes with the lowest final scores are always explored first, optimizing the search for the shortest path.

5. **Deque (from collections module) - Used in Dijkstra's Algorithm:**

    - **Justification:** Dijkstra's algorithm employs a queue-like structure to explore neighboring nodes in the order they were discovered, ensuring that the shortest path is found. A deque is a suitable choice for this purpose.

    - ex:- Lists and tuples are employed to manage coordinates, integers for counting and indexing, and dictionaries for storing scores and relationships between nodes.

    - **Elaboration:** Lists and tuples are ideal for managing coordinates and grid representations, as they provide a structured way to store and access data. Integers are used for counting iterations and indexing elements. Dictionaries are crucial for storing relationships, such as the 'came_from' dictionary, which helps reconstruct the path by associating nodes with their predecessors.

TOOLS USED FOR IMPLEMENTATION

The implementation of the pathfinding algorithm visualizer involves a comprehensive set of tools and resources utilized for different aspects of development and execution:

1. **Pygame Framework:** Pygame, a prominent Python library, serves as the fundamental framework for creating the graphical user interface (GUI), managing graphics, handling user interactions, and rendering visual elements. It provides an array of functionalities for building interactive applications, making it an ideal choice for crafting the visualizer's graphical components and animations.

2. **Python Standard Library Modules:** The code leverages various modules from the Python Standard Library to implement essential functionalities. The **queue** module is used to facilitate priority queue operations crucial for the A* algorithm, while the **collections** module is employed for the deque data structure in Dijkstra's algorithm. Additionally, the **tkinter** module is utilized for displaying message box pop-ups, providing user feedback within the application.

3. **External Image and Font Resources:** External resources such as images and fonts are integrated into the visualizer to enhance its visual appeal and user experience. Image resources, like menu graphics, are employed to create a visually appealing user interface. Font resources are used to render algorithm names and information within the interface, contributing to a polished and informative display.

4. **Development Environment:** The code is crafted and developed within a Python-based development environment, including popular integrated development environments (IDEs) like PyCharm, Visual Studio Code, or other preferred text editors. These environments facilitate code writing, testing, debugging, and overall project management.

5. **Cross-Platform Capabilities:** Python, being a cross-platform language, ensures the visualizer's compatibility across multiple operating systems, including Windows, macOS, and Linux. This capability enables users to run the application seamlessly on their preferred operating systems without significant modifications or issues.

SOURCE CODE:

```
import pygame
import math
from queue import PriorityQueue
from collections import deque
from tkinter import messagebox, Tk

WIDTH = 800
WIN = pygame.display.set_mode((1200, 800))
pygame.init()
pygame.display.set_caption("Path Finding Visualiser")

lightblue = (64, 206, 227)
purple = (197, 114, 255)
white = (255, 255, 255)
black = (50, 50, 50)
yellow = (255, 254, 106)
green = (126, 217, 87)
gridcolor = (175, 216, 248)
orange = (255, 165, 0)
font = pygame.font.Font("elements/myfont.ttf", 40)
algos=['  A-Star','Dijkstras','   Greedy']
menu = pygame.image.load('elements/Visualizer-UI.png')

# ------------------------------------------class for nodes in grid ----------------------------------------------------
class Box:
    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 getPos(self):
    return self.row, self.col

def isStart(self):
    return self.color == green

def isEnd(self):
    return self.color == orange

def isBarrier(self):
    return self.color == black

def reset(self):
    self.color = white

def makeStart(self):
    self.color = green

def makeClosed(self):
    self.color = lightblue

def makeOpen(self):
    self.color = purple

def makeBarrier(self):
    self.color = black

def makeEnd(self):
    self.color = orange

def makePath(self):
    self.color = yellow


def draw(self, win):
    pygame.draw.rect(
```

```
            win, self.color, (self.x, self.y, self.width, self.width))

    def update_neighbors(self, grid):
        self.neighbors = []
        # DOWN
        if self.row < self.total_rows - 1 and not grid[self.row + 1][self.col].isBarrier():
            self.neighbors.append(grid[self.row + 1][self.col])

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

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

        if self.col > 0 and not grid[self.row][self.col - 1].isBarrier():  # LEFT
            self.neighbors.append(grid[self.row][self.col - 1])

    def __lt__(self, other):
        return False



def heu_func(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return abs(x1 - x2) + abs(y1 - y2)



def reconstruct_path(came_from, current, draw):
    while current in came_from:
        current = came_from[current]
        current.makePath()
        draw()

# ---------------------------------------------------------ALGORITHMS---------------------------------------------------------
```

```
def astar(draw, grid, start, end):
    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    came_from = {}
    g_score = {box: float("inf") for row in grid for box in row}
    g_score[start] = 0
    final_score = {box: float("inf") for row in grid for box in row}
    final_score[start] = heu_func(start.getPos(), end.getPos())

    open_set_hash = {start}

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

        current = open_set.get()[2]
        open_set_hash.remove(current)

        if current == end:
            reconstruct_path(came_from, end, draw)
            end.makeEnd()
            return True

        for neighbor in current.neighbors:
            temp_g_score = g_score[current] + 1

            if temp_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = temp_g_score
                final_score[neighbor] = temp_g_score + \
                    heu_func(neighbor.getPos(), end.getPos())
                if neighbor not in open_set_hash:
                    count += 1
                    open_set.put((final_score[neighbor], count, neighbor))
```

```
                open_set_hash.add(neighbor)
                neighbor.makeOpen()

        draw()

        if current != start:
            current.makeClosed()

    return False


def dj_algorithm(draw, grid, start, end):
    came_from = {}
    queue, visited = deque(), []

    queue.append(start)
    visited.append(start)

    while len(queue) > 0:
        current = queue.popleft()
        if current == end:
            reconstruct_path(came_from, end, draw)
            end.makeEnd()
            return True

        for neighbor in current.neighbors:
            if neighbor not in visited:
                visited.append(neighbor)
                came_from[neighbor] = current
                queue.append(neighbor)
                neighbor.makeOpen()
        draw()
        if current != start:
            current.makeClosed()

    return False
```

```
def greedy(draw, grid, start, end):
    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    came_from = {}
    final_score = {box: float("inf") for row in grid for box in row}
    final_score[start] = heu_func(start.getPos(), end.getPos())

    open_set_hash = {start}

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

        current = open_set.get()[2]
        open_set_hash.remove(current)

        if current == end:
            reconstruct_path(came_from, end, draw)
            end.makeEnd()
            return True

        for neighbor in current.neighbors:
            temp_f_score = heu_func(neighbor.getPos(), end.getPos())

            if temp_f_score < final_score[neighbor]:
                came_from[neighbor] = current
                final_score[neighbor] = temp_f_score
                if neighbor not in open_set_hash:
                    count += 1
                    open_set.put((final_score[neighbor], count, neighbor))
                    open_set_hash.add(neighbor)
                    neighbor.makeOpen()
```

```
        draw()

        if current != start:
            current.makeClosed()

    return False

# ------------------------------------------------------------------GRID FUNCTIONS----------------------------------------

def clear(grid):
    for row in grid:
        for box in row:
            if not (box.isStart() or box.isEnd() or box.isBarrier()):
                box.reset()

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

    return grid



def draw_grid(win, rows, width):
    gap = width // rows
    for i in range(rows):
        pygame.draw.line(win, gridcolor, (1, i * gap), (width, i * gap))
        for j in range(rows+1):
            pygame.draw.line(win, gridcolor, (j * gap, 0), (j * gap, width))
```

```
#----------------------------------------USED FOR DRAWING MENU AND GRID IN PYGAME ----------------------------------

def draw(win, grid, rows, width,selector):
    win.fill(white)
    win.blit(menu, (800, 0))
    for row in grid:
        for box in row:
            box.draw(win)

    draw_grid(win, rows, width)
    algoname = font.render(algos[selector], True, (255, 255, 255))
    win.blit(algoname, (910, 430))
    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 = 25
    grid = make_grid(ROWS, width)

    start = None
    end = None
    selector = 0
    run = True
    while run:
        draw(win, grid, ROWS, width,selector)
        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()
    #print(pos)
    if (800 <= pos[0] <= 1200):
        if(895 <= pos[0] <= 1105 and 535 <= pos[1] <= 610 and start and end):
            clear(grid)
            if selector == 0:
                for row in grid:
                    for box in row:
                        box.update_neighbors(grid)
                flag=astar(lambda: draw(win, grid, ROWS, width,selector),
                    grid, start, end)
                start.makeStart()
                if not flag:
                    Tk().wm_withdraw()
                    messagebox.showinfo("No Solution", "There was no solution")

            if selector == 1:
                for row in grid:
                    for box in row:
                        box.update_neighbors(grid)

                flag=dj_algorithm(lambda: draw(win, grid, ROWS, width,selector),
                     grid, start, end)
                start.makeStart()

                if not flag:
                    Tk().wm_withdraw()
                    messagebox.showinfo("No Solution", "There was no solution")

            if selector == 2:
                for row in grid:
                    for box in row:
```

```
                box.update_neighbors(grid)

            flag=greedy(lambda: draw(win, grid, ROWS, width,selector),
                grid, start, end)
            start.makeStart()

            if not flag:
                Tk().wm_withdraw()
                messagebox.showinfo("No Solution", "There was no solution")

    if(895 <= pos[0] <= 1105 and 620 <= pos[1] <= 695):
        clear(grid)

    if(895 <= pos[0] <= 1105 and 710 <= pos[1] <= 785):
        start = None
        end = None
        grid = make_grid(ROWS, width)



    if (801<= pos[0] <= 878 and 426 <= pos[1] <=501):
        #print('left')
        if selector == 0:
            selector = 3
        selector -= 1
        #selector = (selector-1)%3

    if (1123<= pos[0] <= 1200 and 426 <= pos[1] <= 501):
        #print('right')
        selector = (abs(selector+1))%3
        #76 x 76

else:
    row, col = get_clicked_pos(pos, ROWS, width)
    box = grid[row][col]
    if not start and box != end:
```

```
                start = box
                start.makeStart()

            elif not end and box != start:
                end = box
                end.makeEnd()

            elif box != end and box != start:
                box.makeBarrier()

        elif pygame.mouse.get_pressed()[2]:  # RIGHT
            pos = pygame.mouse.get_pos()
            if(800 < pos[0] < 1200):
                #print("MENU")
                pass
            else:
                row, col = get_clicked_pos(pos, ROWS, width)
                box = grid[row][col]
                box.reset()
                if box == start:
                    start = None
                elif box == end:
                    end = None


    pygame.quit()


main(WIN, WIDTH)
```

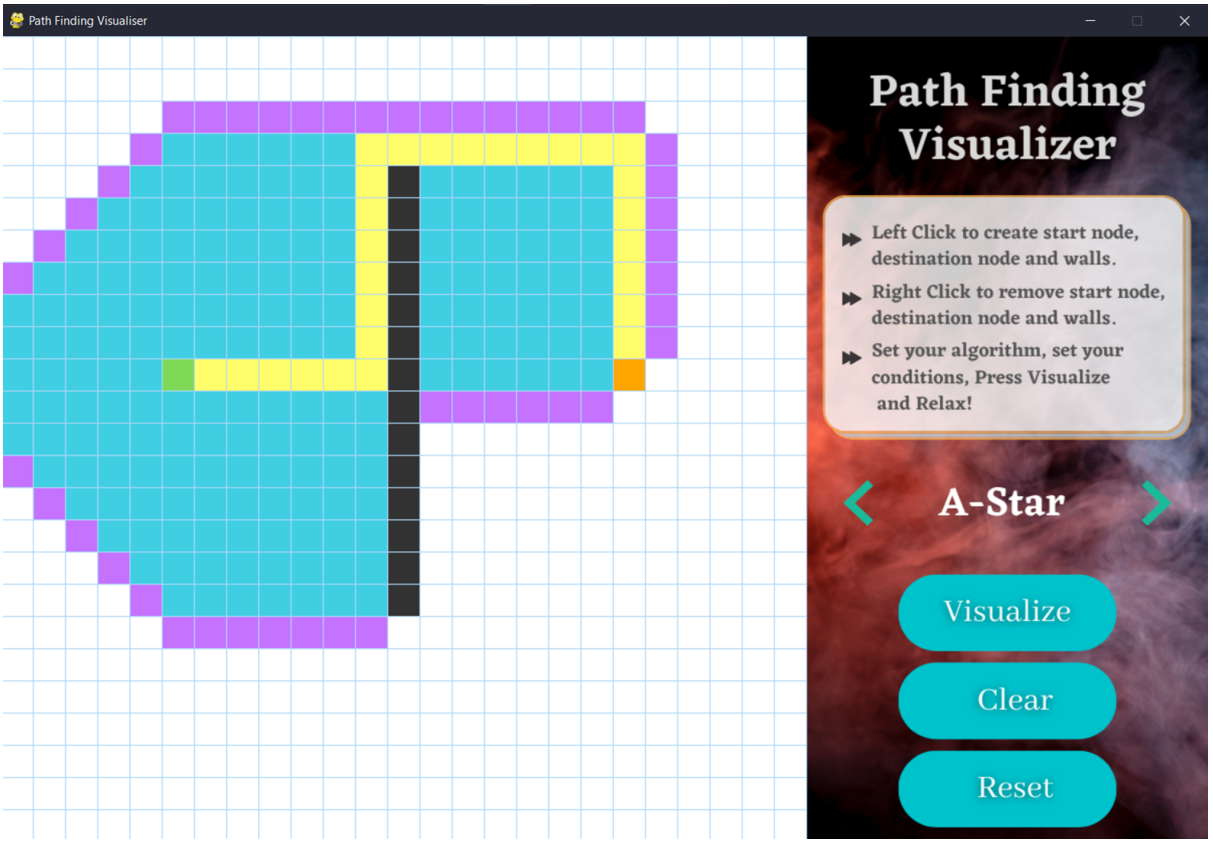
Components and Functions:

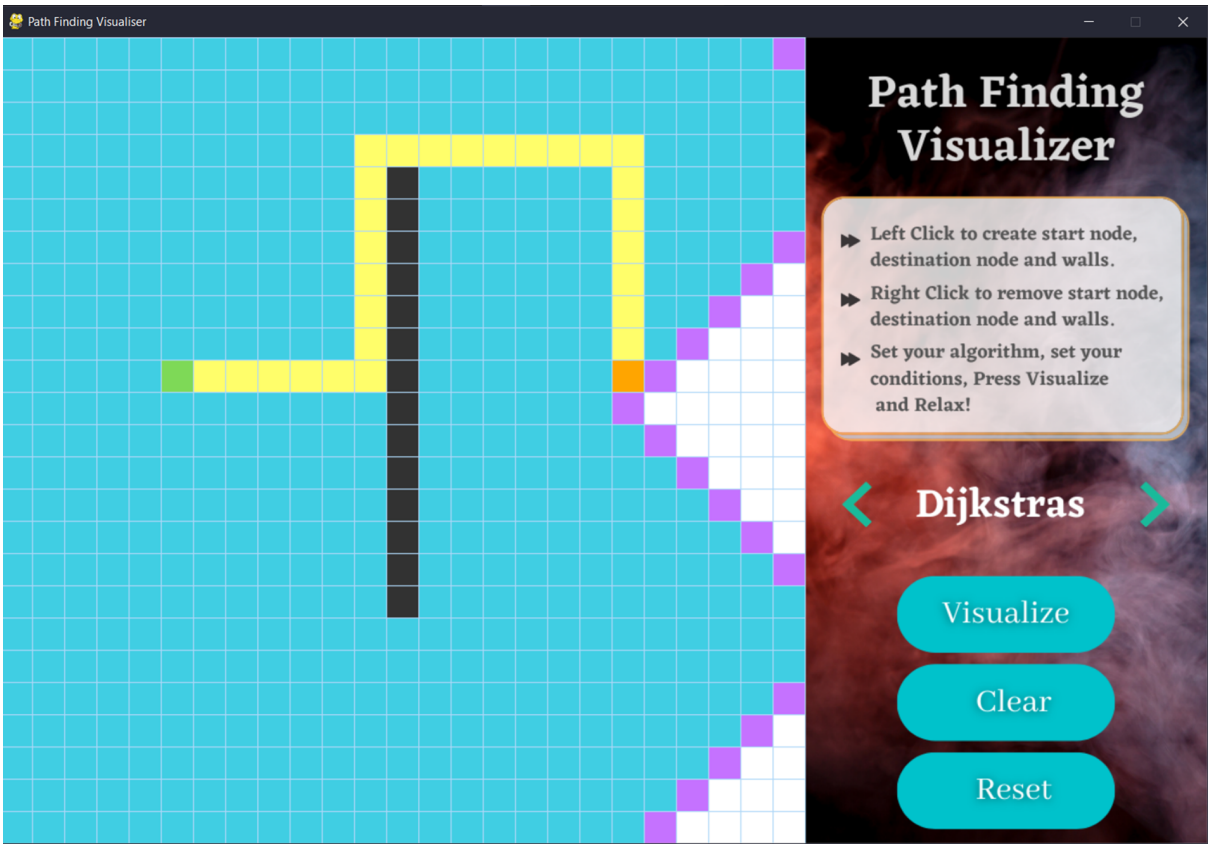
1. **Box Class:**
    - Represents nodes in the grid.
    - Stores information about node position, color, neighbors, and methods to manipulate the node state (start, end, barrier, path, etc.).
    - **getPos**, **isStart**, **isEnd**, **isBarrier**, **reset**, and other methods manage node state.
2. ***Algorithms (A, Dijkstra's, Greedy):****
    - **astar**, **dj_algorithm**, and **greedy** functions represent the implementations of the respective pathfinding algorithms.
    - Each algorithm explores nodes, manipulates their states, and calculates paths from the start to the end node while visualizing the process in real-time.
3. **Grid Generation and Manipulation:**
    - **make_grid**, **draw_grid**, **draw**, and other functions handle grid creation, drawing elements on the interface, and user interactions.
    - Allow users to place start and target points, introduce barriers, and clear the grid.
4. **UI and User Interactions:**
    - **main** function manages the main loop of the application.
    - Handles user inputs, like mouse clicks to designate start and end points, place barriers, select algorithms, and execute pathfinding.

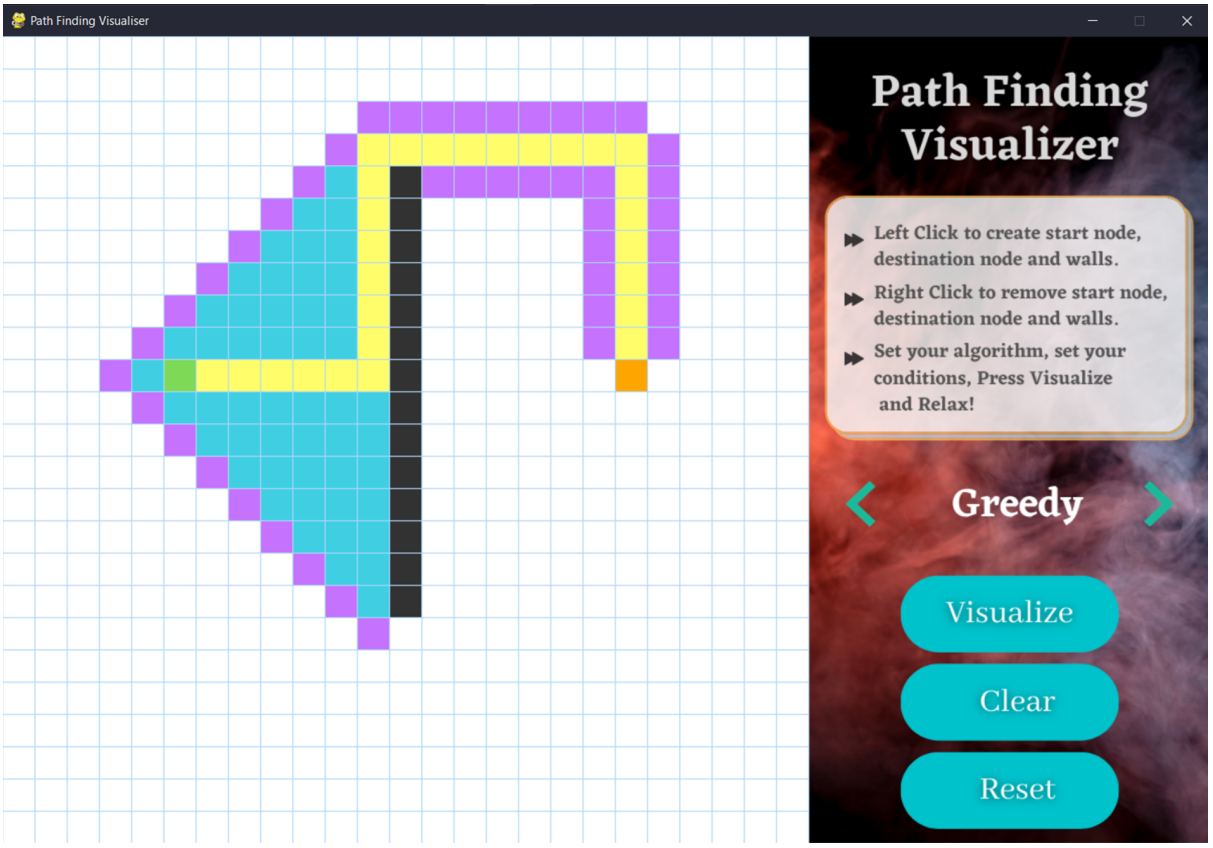
Core Functionalities:

- **Real-Time Visualization:** The code provides an interactive environment where users can observe the step-by-step process of pathfinding algorithms on a grid.
- **Algorithmic Differences:** Demonstrates the distinct approaches of Dijkstra's, A*, and Greedy algorithms in finding the shortest path.
- **User-Friendly Interface:** Offers a menu for selecting algorithms and functionalities to create, modify, and visualize grid-based pathfinding scenarios.

CODE OUTPUT


Path Finding Visualiser
Path Finding Visualizer
Left Click to create start node, destination node and walls.
Right Click to remove start node, destination node and walls.
Set your algorithm, set your conditions, Press Visualize and Relax!
A-Star
Visualize
Clear
Reset


Path Finding Visualiser
Path Finding Visualizer
Left Click to create start node, destination node and walls.
Right Click to remove start node, destination node and walls.
Set your algorithm, set your conditions, Press Visualize and Relax!
Dijkstras
Visualize
Clear
Reset



DISCUSSIONS AND INSIGHTS:

The time complexity of pathfinding algorithms provides insights into their efficiency, especially when considering different grid sizes, structures, and obstacles. Understanding how these complexities scale with grid size can offer valuable insights into algorithm performance. Let's discuss the output insights considering time complexities for different algorithms and grid sizes:

1. **Dijkstra's Algorithm:**
    - **Insights:**
        - Dijkstra's algorithm explores all nodes in the grid, giving the shortest path from a source node to all other nodes.
        - Time complexity: O((V + E) * log(V)), where V is the number of nodes and E is the number of edges.
        - In larger grids, the time complexity tends to increase substantially, particularly when the number of obstacles or edges grows.

- **Output Insights:**
    - For larger grid sizes, the algorithm might take significantly longer to compute the shortest path due to exploring a greater number of nodes and edges.

2. ***A (A-star) Algorithm:****
    - **Insights:**
        - A* algorithm combines Dijkstra's breadth-first search with a heuristic to improve efficiency.
        - Time complexity varies based on the heuristic's accuracy and admissibility.
        - With a good heuristic, it usually outperforms Dijkstra's algorithm in terms of pathfinding efficiency.
    - **Output Insights:**
        - A* algorithm might demonstrate quicker performance in finding the shortest path, especially in scenarios where the heuristic effectively guides the search towards the goal.

3. **Greedy Search Algorithm:**
    - **Insights:**
        - Greedy search focuses solely on the heuristic and may not find the shortest path.
        - It tends to explore nodes based solely on heuristic estimation, potentially leading to suboptimal paths.
    - **Output Insights:**
        - The Greedy algorithm might find paths faster but not necessarily optimal. It can demonstrate faster computation times but potentially sacrifice the shortest path in some scenarios.

Overall Insights:

- Larger grid sizes with increased nodes and edges significantly impact computation time for Dijkstra's and Greedy algorithms due to their exhaustive exploration.
- A* algorithm, if equipped with an effective heuristic, could significantly reduce the exploration time compared to Dijkstra's algorithm and might find shorter paths more efficiently in various scenarios.
- The trade-off exists between efficiency and path optimality, where A* aims to strike a balance between finding paths efficiently and maintaining optimality.

REFERENCES:

1. A.V.Aho, J.E Hopcroft , J.D.Ullman, Data structures and Algorithms, Pearson Education, 2003
2. A note on two problems in connection with graphs by E. W. Dijkstra. Numerische Mathematik, Vol. 1, pp. 269-271 (1959).

   Dijkstra's shortest paths algorithm.
3. About Pygame: 1. https://www.pygame.org/wiki/about
4. Working directions 2. https://clementmihailescu.github.io /Pathfinding-Visualizer