The Superior University

Project Title

A* pathfinding algorithm visualizer

Project Details

1. Course: Data Structures and Algorithm
2. Instructor: Muhammad Muneeb Saleem
3. Semester: Third
4. Section: 3B
5. Submission Date: (02/12/2024)
6. Group Members:

Name	Roll No	Email	Contact
Masooma Zahra	SU92-BSAIM-F23-088	Su92-bsaim-f23-088@superior.edu.pk	03450043022

Abstract

This project is a Python-based visualizer for the A* pathfinding algorithm, designed to help users understand how the algorithm explores and finds the shortest path between two points on a grid. Built using the Pygame library,

the program provides an interactive interface where users can define start and end points, draw barriers to create obstacles, and watch the algorithm dynamically compute the optimal path.
The application showcases the functionality of the A* algorithm, which uses a heuristic to evaluate paths and efficiently navigate complex grids. Users can modify the grid layout and observe how the algorithm adapts to different scenarios in real time.
The significance of this project lies in its educational value. It demystifies a key concept in computer science and artificial intelligence, making it accessible for students, enthusiasts, and professionals. The visual representation helps in understanding the interplay of heuristics, cost evaluations, and pathfinding strategies, making it a powerful tool for learning and teaching.

Table of Contents

1. Introduction

- **Overview of the Project:**

  This project focuses on creating a visual representation of the A* pathfinding algorithm, implemented using Python and the Pygame library. The tool allows users to interactively define start and end points on a grid, place obstacles, and observe how the A* algorithm determines the shortest path in real time. By combining an intuitive interface with an efficient pathfinding mechanism, the project provides a hands-on learning experience for understanding algorithmic concepts.

- **Explanation of the Selected Topic:**

  The A* algorithm was chosen due to its widespread use in fields like robotics, gaming, and AI for solving pathfinding and graph traversal problems. It combines the strengths of Dijkstra's algorithm and greedy best-first search by using both actual cost and heuristic estimation. The algorithm's efficiency and adaptability make it an ideal topic for demonstrating real-world applications of search and optimization techniques.

- **Relevance to Operating Systems Concepts:**

  The project demonstrates several core operating systems concepts:

  - **Resource Management**: The grid and nodes simulate memory allocation, where resources (nodes) are managed dynamically as the algorithm explores paths.
  - **Concurrency**: The visualization relies on real-time updates, similar to how operating systems manage concurrent processes.

- **Priority Queues**: The algorithm's use of a priority queue mirrors task scheduling mechanisms in operating systems, highlighting optimization techniques.

2. Objectives of the Project:

Implementation and Demonstration of A*:

- Design and implement the A* algorithm to illustrate its functionality and efficiency in solving pathfinding problems.
- Simulate the pathfinding process with visualizations that depict how nodes are explored and paths are determined.

Understanding Pathfinding and Optimization:

- Explore the use of heuristics in pathfinding and analyze their impact on performance metrics such as computation time and path optimality.

Educational and Practical Learning Outcomes:

- Develop a deeper understanding of the theoretical and practical aspects of the A* algorithm.
- Gain hands-on experience in implementing algorithms using programming constructs such as priority queues and grid-based systems.

3. System Requirements:

- **Hardware Requirements:**

  Processor: 12th Gen Intel(R) Core (TM) i3-1215U 1.20 GHz

  RAM: 8 GB

- **Software Requirements:**

  Operating System: Windows 10 Pro N

  Programming Language: python 3.11.9

  Libraries and Frameworks:

  Pygame: For grid visualization and user interaction.

  Queue Module: For implementing priority queues in the A* algorithm.

  Development Environment: Visual Studio Code

- **Additional Tools**:
  - Python package manager (`pip`) for library installation.

4. Methodology

1. Problem Definition and Planning

- **Objective:** The goal was to implement and visualize the working of A* algorithm.

2. Algorithm Selection and Design:

- **Algorithm:** Depending on the project, the key algorithms implemented include:
  - A* Algorithm:
    - Combines actual cost (g(n)) and heuristic estimation (h(n)).
    - Implements a priority queue to evaluate nodes efficiently.

3. Implementation and Development

- Frameworks and Libraries: Utilized Python for coding and Pygame for graphical visualizations.
- **Key Steps:**
  - **Grid and Node Representation (for A *)**: Represented the workspace as a grid of nodes, where each node maintains its state (start, end, obstacle, or path).
  - **Interactive Design:** Allowed users to define inputs dynamically, such as obstacles, start and end points, or process queues.

4. Testing and Debugging

- Conducted test cases to evaluate the algorithm's accuracy and efficiency:
  - Pathfinding tested against grids with varying obstacle patterns.
- Debugged issues related to performance, infinite loops, and incorrect visual updates.

5. Visualization and User Interface

- Designed an intuitive interface using Pygame for real-time visualization of algorithm behavior.
- Provided interactive controls for adding/removing elements like obstacles or processes.

6. Evaluation and Optimization

- Measured performance metrics such as execution time and resource utilization.
- Optimized code for better visualization and faster execution.

-

5. Implementation

The project implementation followed a structured approach to ensure clarity, functionality, and performance. Below are the detailed steps:

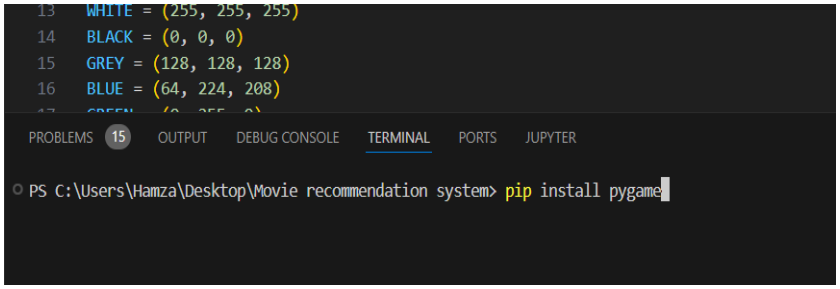
1. Setting Up the Environment:

Language and Tools:

- Used **Python** for implementation due to its simplicity and powerful libraries.
- Leveraged **Pygame** for creating an interactive graphical interface and real-time visualization.

Project Initialization:

- Installed the required libraries using `pip install pygame`.



- Created a structured file for organizing the code.

2. Core Components

a) Grid and Node System

- **Grid Representation**:
  - Represented the workspace as a grid of nodes (2D array). Each node can be a start, end, barrier, or path node.
  -

```python
def make_grid(rows, width):
    grid = []
    gap = width // rows
    for i in range(rows):
        grid.append([])
        for j in range(rows):
            node = Node(i, j, gap, rows)
            grid[i].append(node)
    return grid
```

- **Node Class**:
  - Each node holds attributes like position, color, neighbors, and status (open, closed, barrier, etc.).
  - Included methods for state changes (`make_start()`, `make_end()`, `make_barrier()`)

```python
class Node:
    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 == PURPLE

    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 = PURPLE

def make_path(self):
    self.color = BLUE
```

- **Neighbor Management**:
    - Each node dynamically updates its neighbors based on its position and the presence of barrier.

```
def update_neighbors(self, grid):
    self.neighbors = []
    # Check Down
    if self.row < self.total_rows - 1 and not grid[self.row + 1][self.col].is_barrier():
        self.neighbors.append(grid[self.row + 1][self.col])
    # Check Up
    if self.row > 0 and not grid[self.row - 1][self.col].is_barrier():
        self.neighbors.append(grid[self.row - 1][self.col])
    # Check Right
    if self.col < self.total_rows - 1 and not grid[self.row][self.col + 1].is_barrier():
        self.neighbors.append(grid[self.row][self.col + 1])
    # Check Left
    if self.col > 0 and not grid[self.row][self.col - 1].is_barrier():
        self.neighbors.append(grid[self.row][self.col - 1])
```

B) Algorithm logic:

- *A** Algorithm:
    - Used a **priority queue** to explore nodes based on their cost (`f_score = g_score + h_score`).

```
open_set = PriorityQueue()
open_set.put((0, count, start))
came_from = {}
g_score = {node: float("inf") for row in grid for node in row}
g_score[start] = 0
f_score = {node: float("inf") for row in grid for node in row}
f_score[start] = h(start.get_pos(), end.get_pos())

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.make_end()
        return True
```

- Implemented the **Manhattan heuristic** for estimating the distance to the target node.

```
# Heuristic Function (Manhattan Distance)
def h(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return abs(x1 - x2) + abs(y1 - y2)
```

- Process:
  - Start at the initial node and expand to neighbors.

```
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
        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)
```

- ♣ Track the path using a dictionary (`came_from`) and reconstruct the shortest path upon reaching the end node.

```
# Reconstruct the path
def reconstruct_path(came_from, current, draw):
    while current in came_from:
        current = came_from[current]
        current.make_path()
        draw()
```

3. Visualization Using Pygame

- **Interactive Interface**:
  - o Allowed users to click on the grid to define start/end nodes and barriers.
  - o Used `pygame.mouse.get_pressed()` to capture user input dynamically.

```
    if pygame.mouse.get_pressed()[0]:  # Left Click
        pos = pygame.mouse.get_pos()
        row, col = get_clicked_pos(pos, ROWS, width)
        node = grid[row][col]
        if not start and node != end:
            start = node
            start.make_start()
        elif not end and node != start:
            end = node
            end.make_end()
        elif node != end and node != start:
            node.make_barrier()

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

- **Real-Time Updates**:
  - Highlighted open, closed, and path nodes during the algorithm's execution.
  - Used colors to visually distinguish node types (e.g., orange for start, purple for end, red for closed nodes).
- **Dynamic Grid Drawing**:
  - Calculated grid dimensions and updated the display after every node modification.

```python
def draw(win, grid, rows, width):
    win.fill(WHITE)
    for row in grid:
        for node in row:
            node.draw(win)
    draw_grid(win, rows, width)
    pygame.display.update()
```

4. User Interaction

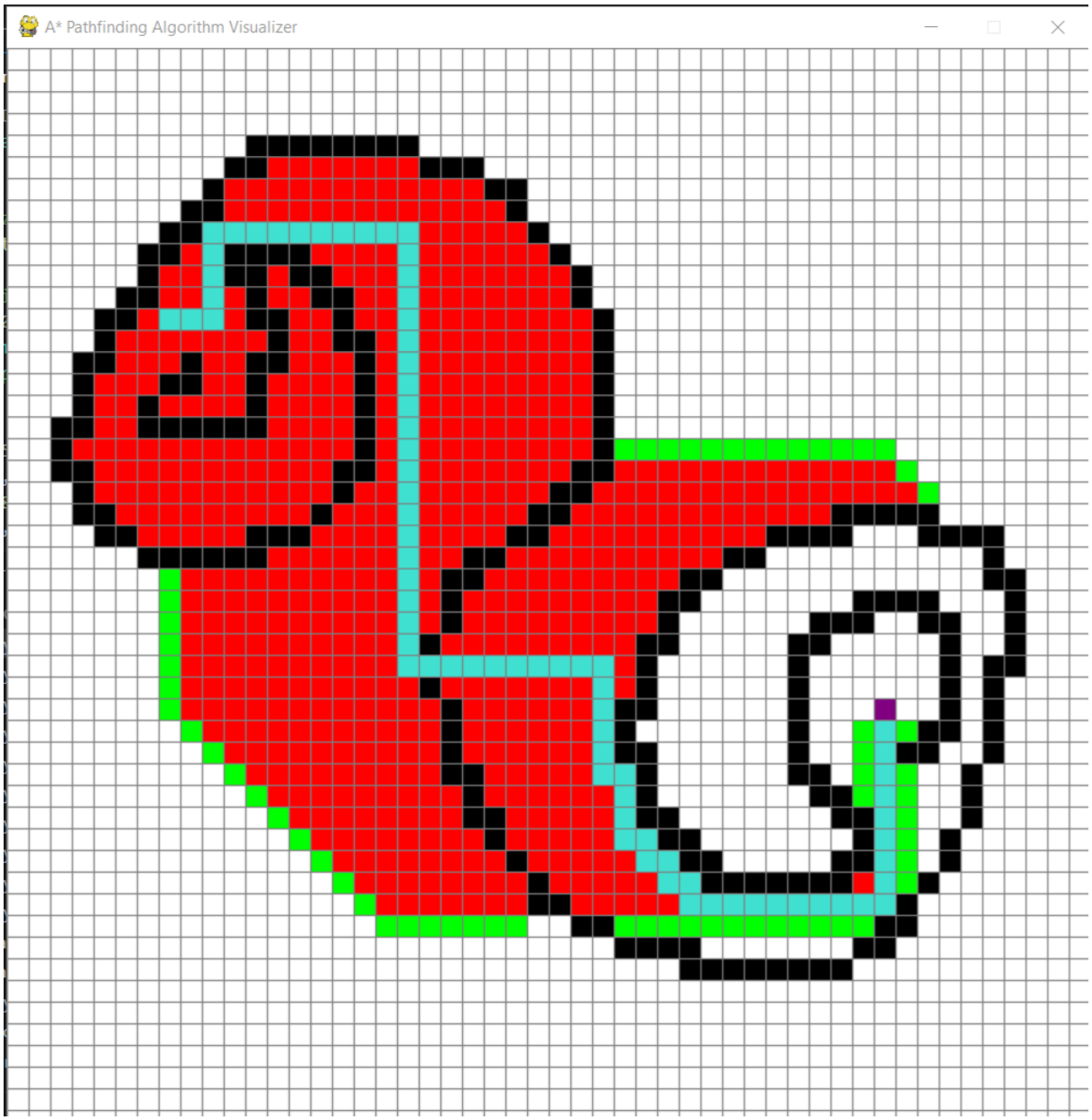
- **Input Controls**:
  - Left-click to set start, end, or barriers.
  - Right-click to reset nodes.
  - Press **Space** to run the algorithm.
  - Press **C** to clear the grid.

5. Testing and Debugging

- Tested the program using various grid configurations to ensure accuracy in pathfinding and process scheduling.
- Debugged issues like:
  - Infinite loops caused by poorly defined neighbor relationships.
  - Incorrect priority queue updates leading to suboptimal paths.

6. Optimization

- **Performance**:
  - Minimized redundant calculations in the heuristic function.
  - Optimized neighbor updates to reduce execution time.
- **Scalability**:
  - Ensured the program could handle larger grids without significant lag.



Here, the yellow square is the start node, purple square is the end node, the red squares are the closed nodes, green squares are open nodes, black ones are the barriers, and the turquoise path is the path, found using the A* algorithm, from starting point to the ending point.

6. Challenges and Solutions

- **Challenges Faced in the Project**

- **Algorithm Complexity and Debugging**: Implementing the A* algorithm required careful handling of data structures such as priority queues and dictionaries. Debugging was challenging when handling edge cases like unreachable nodes or infinite loops.

- **Visual Representation**: Integrating the algorithm with Pygame for a visual representation required additional effort to ensure smooth animations and interactions.
- **Efficient Neighbor Updates**: Efficiently identifying and updating neighboring nodes for each grid cell was a bottleneck, especially for larger grids.
- **User Input and Interaction**: Handling user inputs for setting start, end points, and barriers dynamically added complexity to the system.
- **Grid Size vs. Performance**: Larger grids caused noticeable slowdowns due to the computational overhead of updating and drawing each node.
-

- **Solutions to Overcome Challenges**

- **Thorough Testing and Debugging**:
  - Rigorous testing was done for edge cases.
  - Intermediate print statements and visual cues were used to track the algorithm's progression and identify bottlenecks.
- **Optimized Drawing Mechanism**:
  - Implemented a `draw` function to only redraw modified parts of the grid, reducing redundant computations and improving frame rates.
- **Efficient Neighbor Management**:
  - The `update_neighbors` function was optimized to only include non-barrier nodes and avoid unnecessary checks.
- **Interactive Features with Pygame**:
  - Utilized Pygame's event handling system to enable intuitive mouse and keyboard interactions for dynamic grid updates.
- **Balancing Grid Size and Speed**:
  - Limited grid size to 50x50 for manageable performance and visual clarity.
  - Considered implementing threading or optimizing the algorithm further for scalability in larger grids.
- These strategies ensured the project was functional, responsive, and user-friendly while maintaining computational efficiency.

-

7. Conclusion

- **Key Findings and Outcomes**

- **Effective Implementation of A***: The A* algorithm was successfully implemented to find the shortest path between two points on a grid, with visual feedback to demonstrate its step-by-step progression.
- **Interactive and User-Friendly Design**: Users could dynamically set start and end points, create barriers, and observe the algorithm's behavior in real-time using mouse and keyboard inputs.

- **Visualization of Pathfinding**: The project provided a clear visualization of pathfinding concepts, making it an educational tool for understanding how the A* algorithm operates.
- **Performance Optimization**: While maintaining accuracy, optimizations were implemented to balance grid size and computational efficiency for smooth real-time updates.
-

Potential Future Enhancements

- **Support for Additional Algorithms**:
  - Include other pathfinding algorithms like Dijkstra's, Breadth-First Search (BFS), or Depth-First Search (DFS) for comparison.
- **Scalability Improvements**:
  - Implement threading or parallel processing to handle larger grids without noticeable slowdowns.
- **Improved User Experience**:
  - Provide customizable grid sizes and themes.
- **Heuristic Customization**:
  - Allow users to choose different heuristic functions (e.g., Euclidean distance) to observe their effects on pathfinding.
- **3D Visualization**:
  - Expand the visualization to three-dimensional grids for more complex scenarios.
- **Dynamic Obstacles**:
  - Introduce moving barriers or weighted paths to simulate real-world conditions.