TABLE OF CONTENT

1.0 INTRODUCTION

In computer science, path-finding algorithms are very important when solving problems involving navigation resources and optimization. Of these algorithms, Dijkstra's algorithm forms a basic approach for searching the shortest path in a graph with negotiated weights. However, as the number of computing operations increases and as the applications become more complex and dynamic, there is a growing interest in developing algorithms that can learn to adjust to changes in the environment across time and even in real-time settings.

This project is named 'A Comparative Analysis of Adaptive Heuristic Dijkstra Algorithm for Dynamic Pathfinding', which is a proposed project to investigate and compare a dynamic pathfinding algorithm that is an improvement on the already existing Dijkstra's Algorithm using adaptive heuristics. Though very accurate and proven, the standard Dijkstra Algorithm takes longer in complicated and broad-grided structure Trent due to its exhaustive search technique. The Adaptive Heuristic Dijkstra Algorithm was designed to incorporate a heuristic component that uses the Manhattan distance when the surrounding area is densely obstructed while using the Euclidean distance where there are few obstacles.

The main aim of this study is to compare the Basic Dijkstra Algorithm with the Adaptive Heuristic Dijkstra Algorithm. In this way, we will compare the adaptive approach with the others concerning such criteria as computational complexity, scalability, and ability to operate in non-stationary conditions. Another part of the project is improving the. Storage of the most frequently used data, such as creating GUI for visualization of the pathfinding, which would make it easier to examine how each algorithm moves through the grid given different distributions of the obstacles.

In this paper, the author examines and analyzes a relatively recent type of algorithm, the Adaptive Heuristic Dijkstra Algorithm, that seeks to improve on conventional search algorithms. Thus, by incorporating adaptive heuristics, this algorithm adapts to change its search strategy depending on the density of the obstacles and can be applied to larger maps due to their dynamic nature. This study compares it with the Basic Dijkstra Algorithm, which has advantages in

computation, scalability, and real-time application. By flowing through this analysis, the report wishes to contribute to understanding the potentials of adaptive heuristic approaches in approaching various pathfinding problems to open broader ground for research in the future.

Recent research, such as the heuristic integrated scheduling algorithm (HIS-IDA) based on the improved Dijkstra algorithm proposed by Zhou et al. (2023), demonstrates the effectiveness of enhancing Dijkstra's algorithm with heuristic strategies. Their work incorporates layer priority and leaf node priority strategies to optimize scheduling in complex environments, highlighting the potential of heuristic-based improvements to traditional algorithms. This aligns with our project's focus on adaptive heuristics, where the Adaptive Heuristic Dijkstra Algorithm dynamically adjusts its search strategy based on obstacle density, using Manhattan and Euclidean distances. Such approaches not only improve computational efficiency but also enhance scalability and adaptability in dynamic settings.

By the end of this project, we hope to provide evidence to show the superiority of the Adaptive Heuristic Dijkstra Algorithm for applications in dynamic pathfinding problems in robotics, gaming, and logistics.

2.0 LITERATURE REVIEW

2.1 Overview of Dijkstra Algorithm and its Applications

Dijkstra's Algorithm is a fundamental technique introduced by the Dutch computer scientist Edsger W. Dijkstra in 1956, used for solving the shortest path problem in a graph with weighted edges. By systematically determining the shortest distance from a source node to all other nodes in a graph, the algorithm provides a reliable method for route optimization. It achieves this by initially setting the distance to all nodes as infinity, except for the source node, which starts at zero. Iteratively, it explores the unvisited node with the smallest known distance, updates neighboring nodes if a shorter path is found, and continues until all reachable nodes are visited, ensuring optimal solutions.

The ability of Dijkstra's Algorithm to guarantee the shortest path makes it invaluable for practical applications. It is widely used in GPS navigation systems to determine the fastest routes and in computer networks, where protocols like OSPF (Open Shortest Path First) depend on its efficiency. Beyond navigation, the algorithm is employed in resource allocation, logistics, and game development for NPC pathfinding. Despite its computational complexity in large or dense graphs and its inability to handle negative edge weights, the algorithm's strengths in precision and adaptability ensure its continued importance in solving complex graph-related problems.

2.2 Overview of Heuristic-Based Algorithms

Heuristic-based algorithms are problem-solving techniques designed to find efficient solutions by leveraging heuristic functions or rules of thumb to guide the search process. These algorithms aim to reduce computational complexity and improve efficiency by focusing on promising paths or solutions rather than exhaustively exploring all possibilities. While they may not always guarantee optimal solutions, they are highly effective in scenarios where time or computational resources are limited.

I) A Algorithm*: Combines actual cost (g(n)) with a heuristic estimate (h(n)) to find the shortest path efficiently. Widely used in navigation and game AI.

II) Greedy Best-First Search: Focuses purely on h(n), exploring paths that seem closest to the goal but may lead to suboptimal solutions.

2.3 Need for Adaptive Heuristic Dijkstra

While Dijkstra's algorithm and heuristic-based algorithms like A* are widely used for pathfinding, they each have limitations that can affect their performance in specific environments. Dijkstra's algorithm is exhaustive and computationally expensive, especially in large or dynamic graphs, as it does not prioritize paths toward the goal. On the other hand, heuristic algorithms like A* depend on a well-designed heuristic, which may lead to inefficiencies or suboptimal solutions if the heuristic is not suitable for the environment. These limitations highlight the need for a more adaptive and efficient approach.

The Adaptive Heuristic Dijkstra algorithm fills these gaps by introducing a heuristic component that adapts based on the environment's characteristics. It dynamically switches between Manhattan Distance and Euclidean Distance heuristics depending on the density of obstacles in the graph.

- When obstacles are sparse, Manhattan Distance is used for efficient grid-based search.
- When obstacles are dense, Euclidean Distance provides smoother pathfinding around complex environments.

This adaptive approach helps to focus exploration in more promising regions of the graph, improving efficiency without sacrificing the accuracy of Dijkstra's method. The result is a more flexible, efficient, and scalable solution for dynamic and complex pathfinding tasks.

3.0 METHODOLOGY

3.1 Algorithm Implementation

1. Basic Dijkstra's Algorithm

Description:

- Basic Dijkstra analyzes all possible paths from the start node to the goal node by extending nodes with the lowest total cost.
- Guarantees the quickest path, but may traverse redundant nodes in complex grids.

Equation (Cumulative Cost Function):

$$g(n) = Cost\ from\ the\ start\ node\ until\ the\ current\ node$$

- $g(n)$: Cumulative cost to reach node n.

Equation (Edge Relaxation):

$$g(v) = min(g(v), g(u) + cost(u, v))$$

- $g(u)$: The cost of reaching the present node u.
- $cost(u, v)$: The cost or weight of the edge connecting u and v.
- $g(v)$: The updated minimal cost to reach node v.

Equation (Priority Queue Selection):

$$n = arg\ min(g(n)),\ \forall n \in Unvisited\ Nodes$$

- Always expand the node (n) with the smallest cumulative cost (g) first.

Pseudocode:

1. Set the costs of all nodes to infinite, and the start node cost to zero (0).
2. Use a priority queue to extend nodes at the lowest cost.
3. For the current node:
    a. Check all neighbors.
    b. Measure the cost of reaching each neighbor g(n).
    c. Update the neighbor cost if a shorter path is discovered.
4. Stop when the goal node is reached or all nodes have been processed.
5. Backtrack from the goal to reconstruct the path.

2. Adaptive Heuristic Dijkstra Algorithm

Description:

- Introduces a heuristic component to help guide the search and avoid insufficient exploration.
- Dynamically switches heuristic method based on obstacle density:
    - Manhattan Distance: When obstacles are sparse
    - Euclidean Distance: When obstacles are dense

Equation (Evaluation Function):

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

- $g(n)$: The actual cost of reaching the node n
- $h(n)$: The heuristic estimate of the cost from n the goal

Equation (Heuristic):

- Manhattan Distance:

$$h(n) = |x_1 - x_2| + |y_1 - y_2|$$

- Euclidean Distance:

$$h(n) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Dynamic Heuristic Switching:

- The heuristic is determined by the density of the surrounding obstacles:
    - If there are less than two barriers surrounding the current node, utilize the Manhattan heuristic.
    - Otherwise, apply the Euclidean heuristic.

Pseudocode:

1. Set the costs of all nodes to infinite, and the start node cost to zero (0).
2. Using a priority queue, extend nodes with the smallest f(n) = g(n) + h(n).
3. For the current node:
    a. Check all neighbors.
    b. Measure the cost of reaching each neighbor g(n).
    c. Dynamically select heuristic h(n):
        i. Manhattan if no more than two obstacles surround the node.
        ii. Otherwise, select Euclidean.
    d. Measure f(n) and update the cost if the shortest path is discovered.
4. Stop when the goal node is reached or all nodes have been processed.
5. Backtrack from the goal to reconstruct the path.

Comparison with Basic Dijkstra:

- Basic Dijkstra minimizes g(n), whereas Adaptive Heuristic Dijkstra minimizes f(n), using an estimated cost component h(n) to lead the search.

3.2 Dynamic Maze Generation

Step:

1. Initialize Grid Size:
    - Create a 20x20 grid with all cells accessible.
2. Place Obstacles:
    - Randomly assign around 30% of cells as obstacles.
3. Guarantee Path:
    - weenUse a backtracking algorithm to ensure at least one valid path exists bet the start and goal nodes

3.3 Graphical User Interface (GUI)

Canvas:

- Displays the maze, including the starting point, goal, obstacles, and pathways.
- Color-coded:
    - Green: Indicates as the start node.
    - Gold: Indicates as the goal node.
    - Black: Indicates the obstacles.
    - Orange: Path discovered using Basic Dijkstra.
    - Blue: Path discovered using AdaptiveHeuristic Dijkstra.

Dynamic Goal Setting:

- Users can create new goals by clicking on any open cell in the grid.
- The goal position changes dynamically, and the grid redraws correspondingly.

Button:

- Run Basic Dijkstra: Executes the Basic Dijkstra algorithm and displays the path.
- Run AdaptiveHeuristic: Executes the Adaptive Heuristic Dijkstra algorithm and displays the path.
- Compare result: Display comparison metrics (nodes explored, path cost, and execution time).
- Reset: This option resets the grid layout and clears all results.

3.4 Tools and Techniques

Programming Language:

- Python

Libraries Used:

- Tkinter: To visualize and interact with GUIs.
- PriorityQueue: To manage nodes during algorithm execution.
- Time: To measure execution time.

3.5 Comparison Metrics

Path Cost:

- Total cost of the shortest path from start to goal node.

Nodes Explored:

- Total number of nodes visited during the search.
- A lower number suggests a more efficient search method.

Execution Time:

- Time takes for each algorithm to compute the shortest path.
- Measured in seconds with Python's time package.

4.0 RESULTS AND DISCUSSION

4.1 Output of Basic Dijkstra

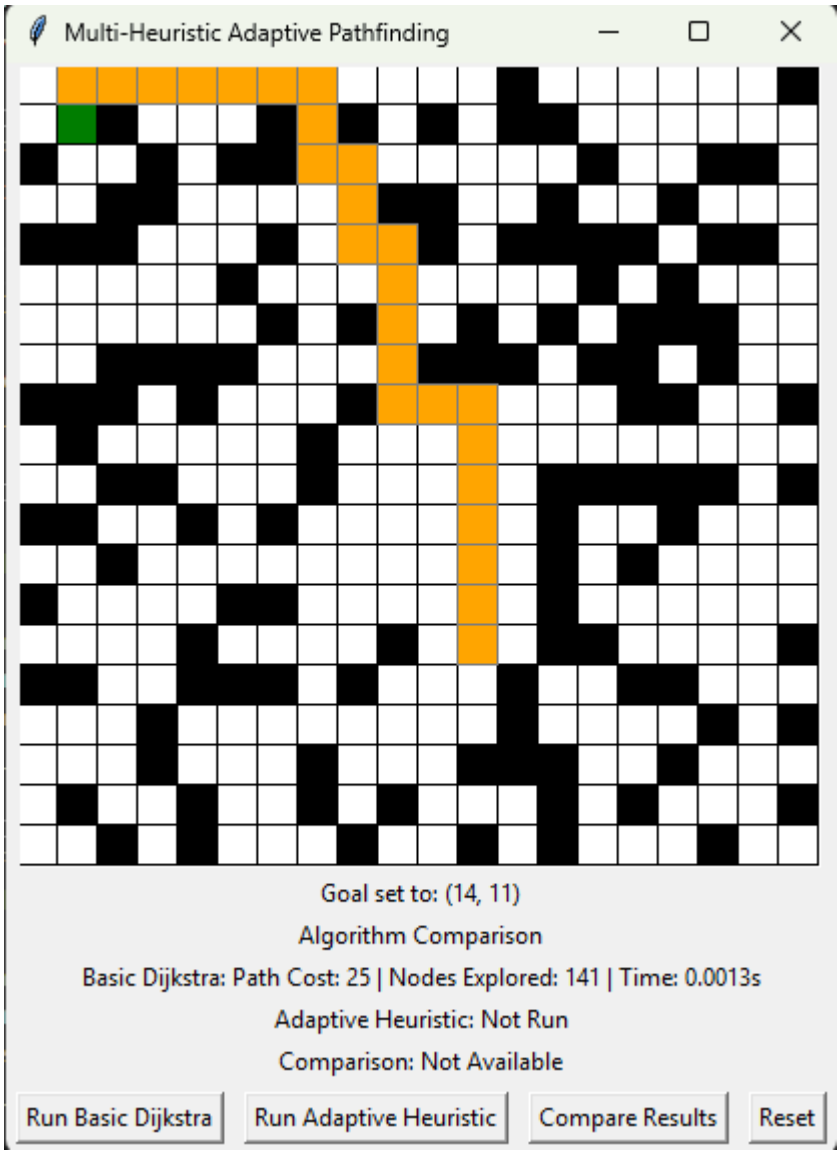

Figure 1: Output path of Basic Dijkstra Algorithm

4.2 Output of Adaptive Heuristic

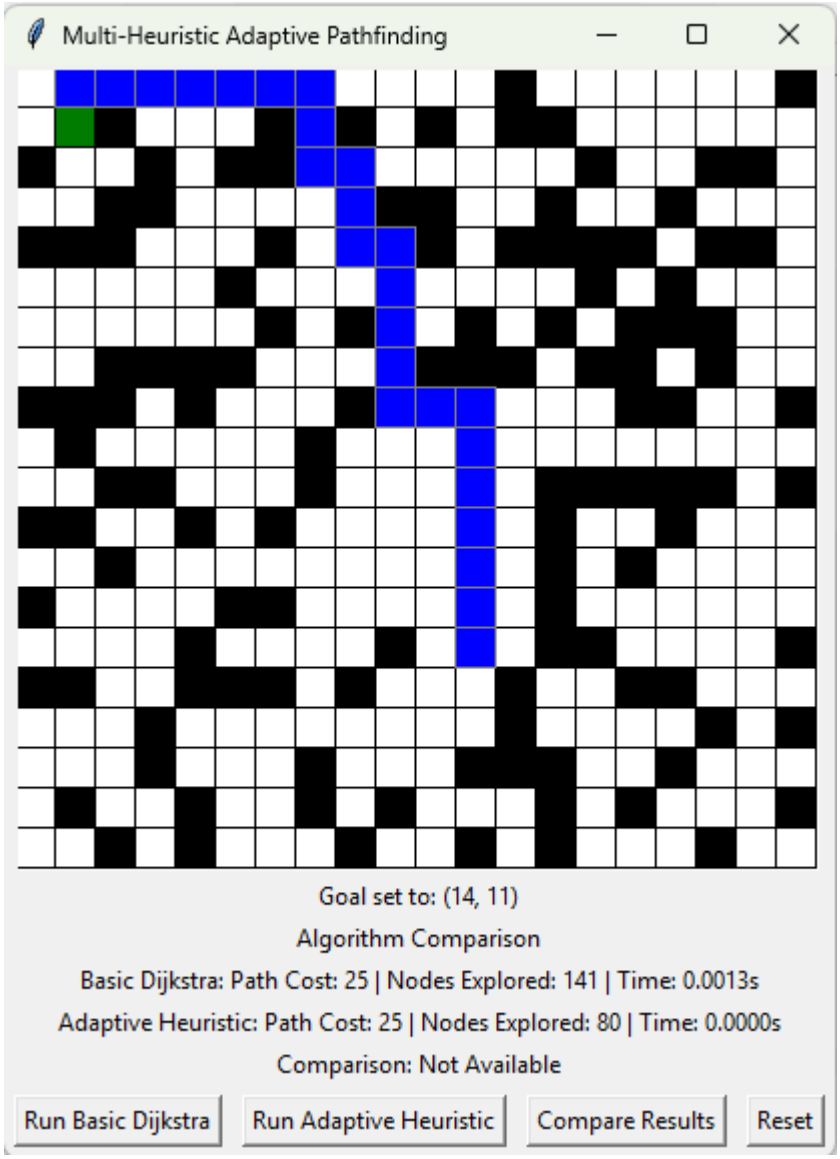

Figure 2: Output path of Adaptive Heuristic Algorithm

4.3 Comparison of Basic Dijkstra and Adaptive Heuristic

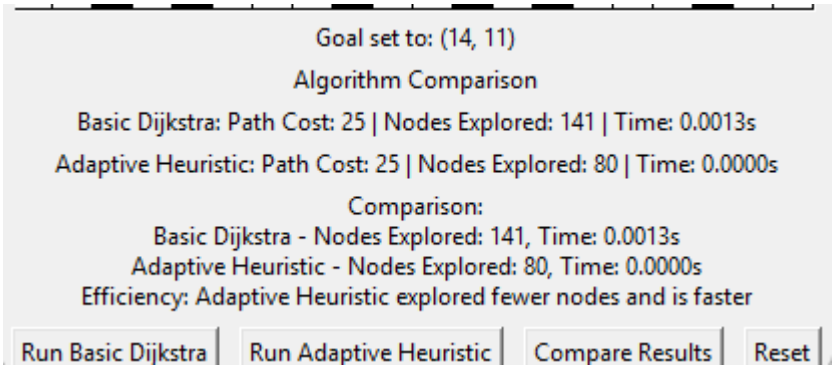

Figure 3: Compare Results of Basic Dijkstra and Adaptive Heuristic

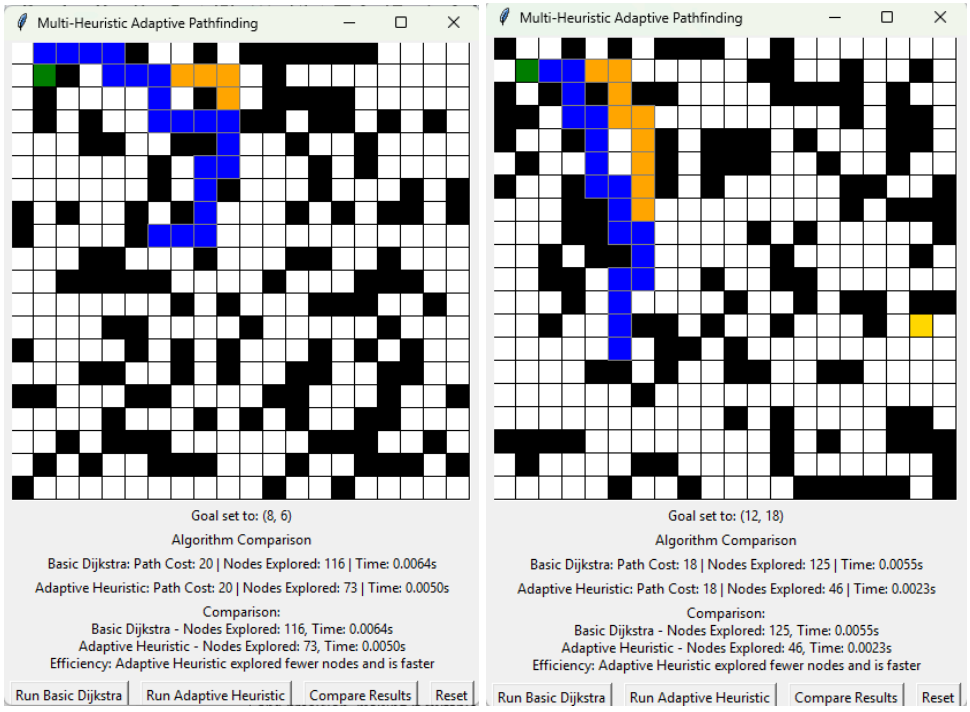

Figure 4: Compare Path & Results of Basic Dijkstra and Adaptive Heuristic

4.4 Strengths of the Adaptive Heuristic Algorithm

Efficiency	By dynamically switching between Manhattan and Euclidean heuristics, the algorithm optimizes node exploration based on the grid's characteristics. This approach balances computational overhead and precision, making it suitable for real-time applications.
Scalability	Adaptive Heuristic is better suited for larger grids or complex obstacle distributions due to its selective exploration strategy.
Flexibility	The dynamic nature of the heuristic allows it to adapt to various scenarios, such as clustered or sparse obstacle

4.5 Limitations of the Adaptive Heuristic Algorithm

Complexity	The added complexity of dynamically selecting heuristics may not be justified in grids with uniformly distributed obstacles. The implementation involves more overhead compared to the simplicity of Basic Dijkstra.

Grid Assumptions:	The heuristic assumes a uniform cost for grid traversal. It may perform poorly in weighted grids or environments with varying traversal costs.
Non-Optimal Cases:	In certain configurations, the adaptive heuristic may perform only marginally better or even equivalently to Basic Dijkstra, particularly in small grids

4.6 Proposed Areas for Improvement

Improvement	**Details**
Incorporating Weighted Grids	Extend the adaptive heuristic to handle weighted grids where movement costs vary across cells.
Enhanced Heuristic Selection	Develop a more sophisticated heuristic-switching mechanism that considers additional factors, such as the density of obstacles or path curvature.
Parallel Processing	Implement parallelized exploration to further reduce runtime, particularly for larger grids.
Dynamic Goal Adjustment	Adapt the algorithm to handle dynamic environments where the goal position or obstacles change during execution.
Hybrid Approaches	Combine the adaptive heuristic with other pathfinding techniques like A* or bidirectional search to leverage the strengths of multiple algorithms.

5.0 CONCLUSION

Thus, this paper compared and contrasted the Basic Dijkstra Algorithm and the Adaptive Heuristic Dijkstra Algorithm towards dynamic pathfinding for this project. A choice of the specific algorithms and the data we highlighted proved that the Adaptive heuristic Dijkstra Algorithm, with the dynamic change of a heuristic function depending on the obstacles' density, outperforms the Basic Dijkstra Algorithm in terms of efficiency, scalability, and adaptability. In particular, it limits the number of exploring nodes and provides better time performance, making it more appropriate for real-life applications with high dynamics, such as robot control, gaming, and logistics. This is in line with Zhou et al. (2023) HIS-IDA framework, which was based on an advanced Dijkstra algorithm, and focused on heuristics as key in increasing efficiency in complex settings.

Given The weakness of Dijkstra's Algorithm adaptive heuristics play an important role. They use optimization for search in the grid with high levels es. It also improves the computational rate besides allowing for the mapping of any changing grid. Specifically for this project, the GUI helps enhance visualization of the performance of both the algorithms and provides insights into the actual functionality and operations of each.

For future work it is advisable to apply the Adaptive Heuristic Dijkstra Algorithm for more complex grids and for the grids where the traversal costs are not equal. Furthermore, future work should also focus on heuristic selection mechanism, parallel processing, investigations in A* or bidirectional search based solution. Extending the research on dynamic goal adjustment and the changes in obstacles in real time would also add further credibility in dynamic settings. Such improvement would also help to enhance adaptive heuristic-based pathfinding algorithms and real-world applications of the former.

REFERENCES

Zhou, P., Xie, Z., Zhou, W., & Tan, Z. (2023). A Heuristic Integrated Scheduling Algorithm Based on Improved Dijkstra Algorithm. *Electronics*, *12*(20), 4189–4189. https://doi.org/10.3390/electronics12204189

Liu, J., Fu, M., Zhang, W., Chen, B., Prakapovich, R., & Sychou, U. (2023). *CDT-Dijkstra: Fast Planning of Globally Optimal Paths for All Points in 2D Continuous Space*. ArXiv.org. https://doi.org/10.48550/arXiv.2308.03026

Okengwu, Ugochi A, Nwachukwu, E. O., & Osegi, E. N. (2015). *Modified Dijkstra Algorithm with Invention Hierarchies Applied to a Conic Graph*. ArXiv.org. https://doi.org/10.48550/arXiv.1503.02517

Fan, D., & Shi, P. (2010). Improvement of Dijkstra's algorithm and its application in route planning. *2010 Seventh International Conference on Fuzzy Systems and Knowledge Discovery*, 1901–1904. https://doi.org/10.1109/fskd.2010.5569452

D. Verma, D. Messon, M. Rastogi and A. Singh, "Comparative Study Of Various Approaches Of Dijkstra Algorithm," 2021 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS), Greater Noida, India, 2021, pp. 328-336, https://ieeexplore.ieee.org/document/9397200

Javaid, A. (2013). Understanding Dijkstra Algorithm. SSRN Electronic Journal. https://doi.org/10.2139/ssrn.2340905

Kokash, N. (n.d.). An introduction to heuristic algorithms. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=8314bf307808718680767 75ba62759f1faf8c9f0

Eiselt, H. A., & Sandblom, C.-L. (2000). Heuristic Algorithms. Springer EBooks, 229–258. https://doi.org/10.1007/978-3-662-04197-0_11

Fu, L., Sun, D., & L.R. Rilett. (2005). Heuristic shortest path algorithms for transportation applications: State of the art. Computers & Operations Research, 33(11), 3324–3343. https://doi.org/10.1016/j.cor.2005.03.027

APPENDICES

Code Implementation

```python
# Define the Manhattan heuristic
def manhattan_heuristic(node, goal):
    """Calculate Manhattan distance as the heuristic.
    Used for direct paths in sparse grids."""
    return abs(node[0] - goal[0]) + abs(node[1] - goal[1])

# Define the Euclidean heuristic
def euclidean_heuristic(node, goal):
    """Calculate Euclidean distance as the heuristic.
    Used for diagonal paths in dense grids."""
    return ((node[0] - goal[0]) ** 2 + (node[1] - goal[1]) ** 2) ** 0.5

# Define the dynamic heuristic function
def dynamic_heuristic(node, goal, grid):
    """Switch heuristics dynamically based on grid conditions.
    Manhattan is used for fewer obstacles; Euclidean is used for dense obstacles."""
    obstacle_count = sum(
        1
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]
        if 0 <= node[0] + dx < GRID_SIZE and 0 <= node[1] + dy < GRID_SIZE and grid[node[0] + dx][node[1] + dy] == 1
    )
    if obstacle_count < 2:
        return manhattan_heuristic(node, goal)
    else:
        return euclidean_heuristic(node, goal)
```

Heuristic Function (Manhattan, Euclidean, Dynamic Heuristic)

```python
# Adaptive Heuristic Dijkstra Algorithm
def adaptive_dijkstra(grid, start, goal):
    """Modified Dijkstra's algorithm with dynamic heuristic adaptation."""
    rows, cols = len(grid), len(grid[0])
    visited = set()  # Track visited nodes
    pq = PriorityQueue()  # Priority queue to prioritize nodes
    pq.put((0, start))  # Add start node with cost 0
    costs = {start: 0}  # Track cumulative costs
    came_from = {}  # Track path information

    while not pq.empty():
        current_cost, current = pq.get()

        if current in visited:
            continue

        visited.add(current)

        if current == goal:  # Goal found; reconstruct path
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.reverse()
            return path, costs[goal], len(visited)

        # Explore neighbors
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            neighbor = (current[0] + dx, current[1] + dy)
            if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols and grid[neighbor[0]][neighbor[1]] == 0:
                new_cost = costs[current] + 1  # Cost to reach neighbor
                if neighbor not in costs or new_cost < costs[neighbor]:
                    costs[neighbor] = new_cost
                    priority = new_cost + dynamic_heuristic(neighbor, goal, grid)
                    pq.put((priority, neighbor))
                    came_from[neighbor] = current

    return [], float('inf'), len(visited)  # No path found

```

Adaptive Heuristic Dijkstra Algorithm

```python
# Basic Dijkstra's Algorithm
def dijkstra(grid, start, goal):
    """Basic Dijkstra's algorithm."""
    rows, cols = len(grid), len(grid[0])
    visited = set()  # Track visited nodes
    pq = PriorityQueue()  # Priority queue to prioritize nodes
    pq.put((0, start))  # Add start node with cost 0
    costs = {start: 0}  # Track cumulative costs
    came_from = {}  # Track path information

    while not pq.empty():
        current_cost, current = pq.get()

        if current in visited:
            continue

        visited.add(current)

        if current == goal:  # Goal found; reconstruct path
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.reverse()
            return path, costs[goal], len(visited)

        # Explore neighbors
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            neighbor = (current[0] + dx, current[1] + dy)
            if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols and grid[neighbor[0]][neighbor[1]] == 0:
                new_cost = costs[current] + 1  # Cost to reach neighbor
                if neighbor not in costs or new_cost < costs[neighbor]:
                    costs[neighbor] = new_cost
                    pq.put((new_cost, neighbor))
                    came_from[neighbor] = current

    return [], float('inf'), len(visited)  # No path found
```

Basic Dijkstra Algorithm

```python
    def generate_maze(self):
        """Generate a random maze with a guaranteed path."""
        self.grid = [[0 for _ in range(GRID_SIZE)] for _ in range(GRID_SIZE)]
        for i in range(GRID_SIZE):
            for j in range(GRID_SIZE):
                if random.random() < 0.3 and (i, j) not in [self.start, self.goal]:
                    self.grid[i][j] = 1  # Mark as obstacle

```

Dynamic Maze Genereation

```python
    def display_results(self, path, cost, nodes_explored, execution_time, algorithm_name, result_label, path_color):
        """Display the results of the algorithm and highlight the path."""
        for i, j in path:
            self.canvas.create_rectangle(j * CELL_SIZE, i * CELL_SIZE, (j + 1) * CELL_SIZE, (i + 1) * CELL_SIZE, fill=path_color, outline="gray")
        result_label.config(text=f"{algorithm_name}: Path Cost: {cost} | Nodes Explored: {nodes_explored} | Time: {execution_time:.4f}s")

    def compare_results(self):
        """Compare the results of both algorithms."""
        if self.basic_data and self.adaptive_data:
            basic_cost, basic_nodes, basic_time, basic_path = self.basic_data
            adaptive_cost, adaptive_nodes, adaptive_time, adaptive_path = self.adaptive_data

            # Highlight both paths
            self.draw_grid()
            for i, j in basic_path:
                self.canvas.create_rectangle(j * CELL_SIZE, i * CELL_SIZE, (j + 1) * CELL_SIZE, (i + 1) * CELL_SIZE, fill="orange", outline="gray")
            for i, j in adaptive_path:
                self.canvas.create_rectangle(j * CELL_SIZE, i * CELL_SIZE, (j + 1) * CELL_SIZE, (i + 1) * CELL_SIZE, fill="blue", outline="gray")

            # Display comparison results
            comparison = (
                f"Comparison:\n"
                f"Basic Dijkstra - Nodes Explored: {basic_nodes}, Time: {basic_time:.4f}s\n"
                f"Adaptive Heuristic - Nodes Explored: {adaptive_nodes}, Time: {adaptive_time:.4f}s\n"
                f"Efficiency: {'Adaptive Heuristic is more efficient' if adaptive_nodes < basic_nodes else 'Basic Dijkstra is more efficient'}"
            )
            self.comparison_results.config(text=comparison)
```

Path Visualization & Result Display

```python
class PathfindingApp:
    def __init__(self, master):
        """Initialize the GUI application."""
        self.master = master
        self.master.title("Multi-Heuristic Adaptive Pathfinding")

        # Initialize grid, start, and goal positions
        self.grid = [[0 for _ in range(GRID_SIZE)] for _ in range(GRID_SIZE)]
        self.start = (1, 1)  # Start node
        self.goal = (18, 18)  # Goal node

        # Generate the maze
        self.generate_maze()

        # Create canvas for visualization
        self.canvas = tk.Canvas(self.master, width=GRID_SIZE * CELL_SIZE, height=GRID_SIZE * CELL_SIZE, bg="white")
        self.canvas.pack()
        self.canvas.bind("<Button-1>", self.handle_click)  # Set goal on mouse click

        # Create information and control frames
        self.info_frame = tk.Frame(self.master)
        self.info_frame.pack()

        # Info and results labels
        self.info_label = tk.Label(self.info_frame, text="Click on the grid to set the goal location.")
        self.info_label.pack()

        self.basic_results = tk.Label(self.info_frame, text="Basic Dijkstra: Not Run")
        self.basic_results.pack()
```

GUI Setup

```
        self.adaptive_results = tk.Label(self.info_frame, text="Adaptive Heuristic: Not Run")
        self.adaptive_results.pack()

        self.comparison_results = tk.Label(self.info_frame, text="Comparison: Not Available")
        self.comparison_results.pack()

        # Create control buttons
        self.control_frame = tk.Frame(self.master)
        self.control_frame.pack()

        self.run_basic_btn = tk.Button(self.control_frame, text="Run Basic Dijkstra", command=self.run_basic)
        self.run_basic_btn.grid(row=0, column=0, padx=5, pady=5)

        self.run_adaptive_btn = tk.Button(self.control_frame, text="Run Adaptive Heuristic", command=self.run_adaptive)
        self.run_adaptive_btn.grid(row=0, column=1, padx=5, pady=5)

        self.compare_btn = tk.Button(self.control_frame, text="Compare Results", command=self.compare_results)
        self.compare_btn.grid(row=0, column=2, padx=5, pady=5)

        self.reset_btn = tk.Button(self.control_frame, text="Reset", command=self.reset)
        self.reset_btn.grid(row=0, column=3, padx=5, pady=5)

        # Data storage for results
        self.basic_data = None
        self.adaptive_data = None

        # Draw initial grid
        self.draw_grid()
```

GUI Setup