Optimal Drone Navigation in a MATLAB Simulator using A* and Bellman-Ford Algorithms

Naysah Sheikh and Srevatshen S G

Vellore Institute of Technology, Chennai

naysah.sheikh2022@vitstudent.ac.in, srevatshen.sg2022@vitstudent.ac.in

Abstract

This project aims to develop a web-based prototype for simulating optimal drone navigation using two prominent algorithms: A* and Bellman-Ford. The system is designed to allow users to toggle between these algorithms, facilitating a comparative analysis of their performance in various navigational scenarios. A key feature of this prototype is the implementation of an ensemble approach that leverages the strengths of the A* algorithm for real-time navigation while utilizing the Bellman-Ford algorithm to manage negative edge weights and detect potential hazards. The simulation will encompass various environmental conditions, including physical obstacles, terrain variations, signal interference, and adverse weather effects. A detailed performance analysis will focus on pathfinding efficiency, computational complexity, and response time across different environments, providing valuable insights into the strengths and weaknesses of each algorithm in practical applications.

Keywords: A* Algorithm, Bellman-Ford Algorithm, Ensemble Approaches, MATLAB Simulation, Performance Analysis

I. Introduction

1.1 Importance of Drone Navigation Systems

In recent years, unmanned aerial vehicles (UAVs), often referred to as drones, have received significant attention, and are extensively employed in various fields, such as traffic inspection, disaster rescue, cargo delivery, and target reconnaissance. It is worth noting that path planning plays a key role to realize the autonomous control of the UAV system, which facilitates the effective application of UAVs.

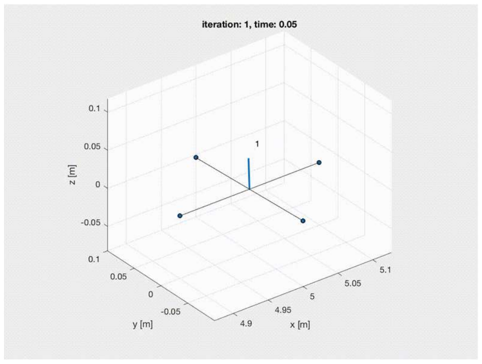

Figure-1:

Sample drone simulations in a 3-D area

1.2 Challenges faced in existing systems

Currently, it is still a significant challenge to realize efficient path planning for UAVs in complex environments with dense obstacles and uncertainties while considering several demands, such as obstacle avoidance, trajectory feasibility, real-time planning capability, and satisfactory path length. The key challenges in drone navigation systems include: physical obstacles, environmental conditions such as sudden changes in weather (strong winds, rain, fog) and temperature variations that can affect battery life and sensor performance, terrain variations, signal interference, and dynamic changes that can make certain paths temporarily less favourable due to factors like sudden weather changes, no-fly zones, or restricted airspaces. These problems especially arise when the goal point is completely surrounded by hazards.

1.3 Our proposed solution

The purpose of this experiment is to explore optimal navigation strategies for drones by implementing two well-established algorithms: A* and Bellman-Ford. The A* algorithm is widely recognized for its efficiency in pathfinding and graph traversal, particularly in scenarios requiring real-time navigation. It combines the benefits of Dijkstra's algorithm with heuristics to optimize the search process. Conversely, the Bellman-Ford algorithm excels in handling graphs with negative edge weights and detecting negative cycles, making it suitable for environments where hazards may temporarily alter the cost of paths.

This study aims to provide a comprehensive understanding of how these algorithms can be applied in drone navigation, particularly under varying environmental conditions such as sudden weather changes and terrain obstacles. By comparing the two algorithms through a MATLAB simulation, we aim to derive conclusions about their relative performance and applicability in real-world scenarios. The insights gained from this research could significantly contribute to enhancing drone navigation systems, making them more robust and adaptable to dynamic environments.

This study extends the ensemble approach by incorporating hazard-aware pathfinding. The system:

1. Uses Bellman-Ford to detect hazards and adapt to dynamic conditions.
2. Employs A* for efficient real-time navigation.
3. Introduces risk factor evaluation to determine the least risky path when all neighbours are blocked.

II. Materials and Methodology

2.1 Development Platform

The project utilizes Python and Streamlit as the primary development platform for simulating drone navigation. Key materials include Python's built-in functions for implementing the A* and Bellman-Ford algorithms, as well as optimized algorithm that combines the features of both A* as well as Bellman Ford Algorithm to uncover the shortest path possible. The simulation environment is based on Streamlit to replicate various conditions that drones may encounter, including physical obstacles (e.g., buildings or trees), signal interference (e.g., communication disruptions), and extreme weather effects on battery life and sensor performance.

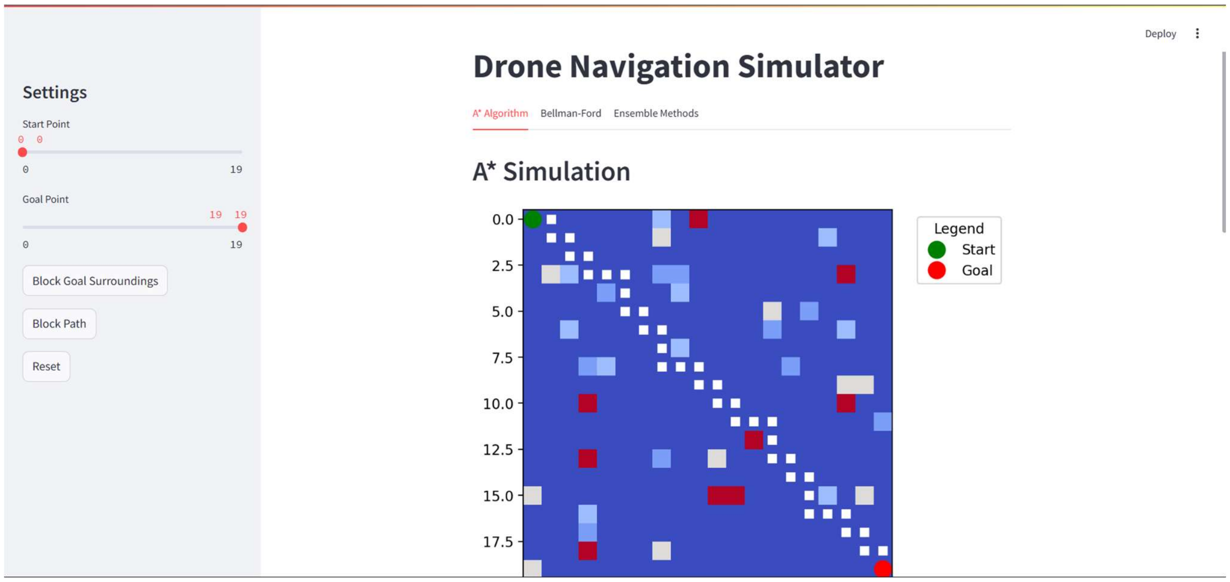

Figure-3: The app allows setting the start and goal points dynamically, along with the algorithm of choice to allow simulation of different circumstances.

2.2 Data

Data collection involved measuring several performance metrics: pathfinding efficiency (distance traveled), computational complexity (time taken to compute paths), and response time (latency in navigating between waypoints). The ensemble approach will be tested by integrating both algorithms into a single framework that dynamically selects the appropriate algorithm based on current environmental conditions. Observations will be recorded systematically to ensure reproducibility and reliability of results.

2.3 Key Components

i. **Main Function** (droneNavigationSimulator): This function takes inputs such as starting point (start), goal point (goal), grid size (gridSize), and the algorithm type (algorithm). It initializes the environment and runs either the A*, Bellman-Ford or combined algorithm based on user input.

Pseudocode:

```
function droneNavigationSimulator()
    Initialize environment
    Create grid
    Select start and goal positions
    Generate hazards using generate_hazards()
    Run A* algorithm
    Run Bellman-Ford algorithm
    Run Ensemble Approach
    Capture performance metrics (path, nodes_expanded, response_time)
end function
```

ii. **Performance Metrics**: The performance structure captures execution time (reponse_time), total distance travelled (path), and number of nodes expanded during the search (nodes_expanded).

Pseudocode:

```
function display_metrics(path, nodes_expanded, response_time):
    # Update metrics
    path_length = len(path) if path else 0
    path_lengths.append(path_length)
    nodes_expanded_list.append(nodes_expanded)
    response_times.append(response_time)
    # Print metrics
    st.subheader("Performance Metrics:")
    st.write(f"Pathfinding Efficiency (Path Length): {path_length}")
    st.write(f"Computational Efficiency (Nodes Expanded): {nodes_expanded}")
    st.write(f"Response Time: {response_time:.4f} seconds")
end function
```

iii. **Hazards Creation** (generate_hazards): This function initializes a grid with random obstacles set with cost values, representing impassable areas.

Pseudocode:

```
function generate_hazards(grid, start, goal)
    for hazard, cost in hazard_types.items():
        for _ in range(10):  # Number of hazards per type
            y, x = np.random.randint(0, grid.shape[0]), np.random.randint(0,
grid.shape[1])
            # Ensure hazards are not placed on start or goal nodes
            while (x, y) == start or (x, y) == goal:
                y, x = np.random.randint(0, grid.shape[0]),
np.random.randint(0, grid.shape[1])
            hazards[y, x] = cost
            hazard_map[y, x] = hazard  # Assign the hazard type
    return hazards, hazard_map
end function
```

iv. **Block Goal Surroundings:** This button enables users to simulate a scenario where all nodes surrounding the goal are blocked by hazards. This feature is crucial for testing the resilience of the pathfinding algorithms in extreme situations where direct paths to the goal are entirely obstructed.

Pseudocode:

```
function BlockGoalSurroundings(grid, goal):
    Get coordinates of goal node (y, x)
    For each neighbor of (y, x):
        If neighbor is within grid bounds and not the goal:
            Set hazard value to high (e.g., 100)
    Return modified grid
```

v. **Block Path:** This button enables users to simulate a scenario where all nodes in a row or a column between the start and goal points are blocked by hazards. This feature is crucial for testing the resilience of the pathfinding algorithms in extreme situations where direct paths to the goal are entirely obstructed.

Pseudocode:

```
function BlockPath(grid, hazard map, hazard types):
    Randomly decide whether to block a row or column
    Select a random row or column to block
    For col in range(cols):
        Assign a random hazard type and its cost
    Return updated grid, updated hazard map
```

2.4 Algorithms Implemented

- **A* Algorithm:** This heuristic-based algorithm efficiently finds the shortest path by evaluating nodes based on a cost function that combines actual travel cost and estimated cost to reach the goal.

  Pseudocode:

```
function AStar(start, goal, grid)
    Initialize open set and closed set
    Set cost from start to each node and heuristic to goal

    While open set is not empty do
        CurrentNode = node with lowest cost in open set
        If CurrentNode is goal then return path

        Remove CurrentNode from open set and add to closed set

        For each neighbour of CurrentNode do
            If neighbour is not in closed set and is valid then
                Calculate new cost and heuristic for neighbour

                If neighbour not in open set then
                    Add neighbour to open set with updated cost and heuristic

                Else if new cost is lower than previously recorded cost then
                    Update cost for neighbour

    Return failure (no path found)
end function
```

- **Bellman-Ford Algorithm:** This algorithm is capable of handling graphs with negative edge weights and can detect negative cycles, making it suitable for dynamic environments where paths may become less favourable due to sudden changes.

  Pseudocode:

```
function BellmanFord(start, graph)
    Initialize distances from start to all nodes as infinity, except start =
0

    For each node in graph do
        For each edge in graph do
            Relax edge if possible (update distance)
        End For

        If no updates occur in this pass then break loop (optimal found)
    End For

    Check for negative cycles

    Return distances or failure if negative cycle detected
end function
```

- **Ensemble Approach:** This method combines both algorithms by first using Bellman-Ford to assess potential hazards before employing A* for immediate navigation decisions.

  Pseudocode:

```
function EnsemblePathfinding(grid, start, goal, hazards, hazard_map):
    While current != goal:
        If current node is hazardous:
            Use Bellman-Ford for pathfinding
            If no valid path is found:
                Calculate risk for all neighbors
                Select the least risky node
                Return path and least risky node
        Else:
            Use A* for real-time pathfinding
    Return final path
```

2.4.1 Utility Functions:

- **getNeighbors**: Finds valid neighboring cells.

  Pseudocode:

```
function getNeighbors(node, rows, cols)
    y, x = node
    neighbors = [(y + dy, x + dx) for dy, dx in [(-1, 0), (1, 0), (0, -1), (0,
1)]]
```

```
    return [(ny, nx) for ny, nx in neighbors if 0 <= ny < rows and 0 <= nx <
cols]
end function
```

- **heuristic** and **distance**: Calculate heuristic values and distances between nodes.

  Pseudocode:

```
function heuristic(nodeA, nodeB)
    Return estimated cost from nodeA to nodeB (e.g., Euclidean or Manhattan
distance)
end function

function distance(nodeA, nodeB)
    Return actual travel cost between nodeA and nodeB (e.g., weight of edge)
end function
```

2.5 Risk Factor Considerations

The system evaluates risk factors to choose paths intelligently:

- **Battery Consumption:** Longer paths or delays increase risk.
- **Wind Factor:** Randomized values simulate environmental unpredictability.
- **Network Disruption:** Depends on the type of hazard (e.g., signal interference, terrain).
- **Wear and Tear:** Based on physical obstacle severity.

Pseudocode:

```
function CalculateRisk(hazard_type, path_length, wind_factor):
    Define risk_weights for hazards
    Calculate battery_risk, network_risk, and wear_tear_risk
    total_risk = battery_risk + network_risk + wear_tear_risk +
wind_factor
    Return total_risk
```

2.6 Design Flow Diagram

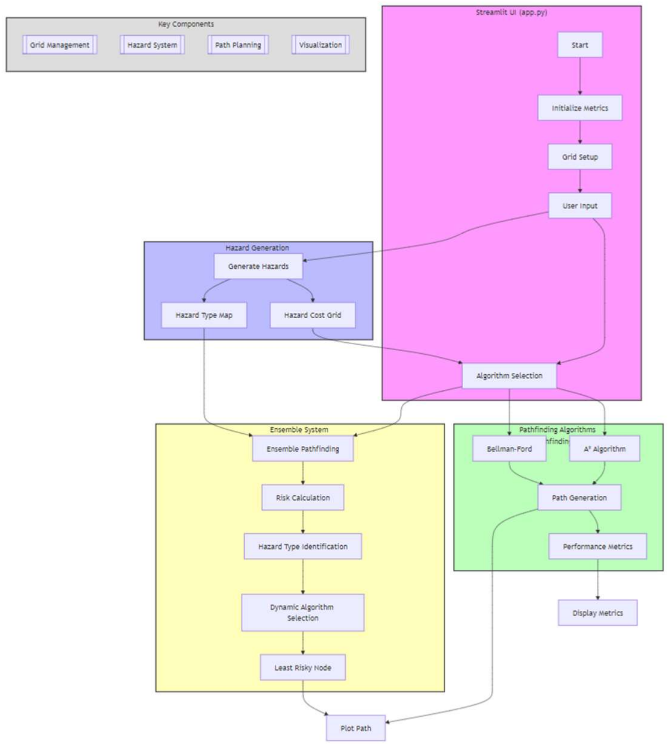

Figure-3
Design-Flow Diagram for the entire concept

2.7 Final implementation

The simulation environment is represented as a grid where obstacles are randomly placed to mimic real-world challenges faced by drones during flight. The start and goal points, a well as, hazards can be adjusted based on user input to test different scenarios.

An ensemble approach combining both algorithms is also explored to enhance pathfinding efficiency and adaptability to dynamic scenarios, along with the individual algorithms. The ensemble approach first applies the Bellman-Ford algorithm to identify hazards based on detected conditions and then utilizes the A* algorithm for real-time navigation adjustments. If the path to the goal is completely blocked, the ensemble algorithm evaluates all possible risk factors and chooses the path with the minimum risk. Data collection involves measuring execution times and distances across various test cases with different configurations.

III. Results

3.1 Tabulated Results

Algorithm Used	**Time Taken (in seconds)**		
	Case-1 (Goal not surrounded)	**Case-2 (Goal surrounded)**	**Case-3 (Path bloacked)**
Bellman Ford Algorithm	1.8302	2.7674	1.7993
A* Algorithm	0.0005	0.0015	0.0015
Ensemble Algorithm	0.0000 (very less time)	0.0000 (very less time)	0.0000 (very less time)

Table-1
Path-finding efficiency measured as time taken in seconds by each of the algorithms used

For now, we have factored the pathfinding efficiency and tabulated the same in the following table (refer Table-1) for the given graphs in Figure-5. The optimized algorithm shows the highest efficiency followed by A* Algorithm, and finally, Bellman Ford Algorithm in the given scenario.

3.2 Summary of results

This report presents a study on optimal drone navigation strategies using A* and Bellman-Ford algorithms, as well as, a novel Ensemble algorithm implemented in Python. The primary objective is to evaluate the performance of these algorithms in navigating environments with various potential hazards, including physical obstacles, environmental conditions, terrain variations, and signal interference. Key performance metrics such as pathfinding efficiency (distance travelled), computational complexity (time taken to compute paths), and response time (latency in navigating between waypoints) are analyzed.

We observe that while A* performs better in straightforward environments with minimal obstacles, Bellman-Ford may demonstrate superior adaptability when faced with negative edge weights or sudden changes in terrain conditions.

Performance metrics are collected during simulations, including:

- **Pathfinding Efficiency**: Measured by total distance traveled by the drone.
- **Computational Complexity**: Assessed by the time taken to compute paths.
- **Response Time**: Defined as latency in navigating between waypoints.

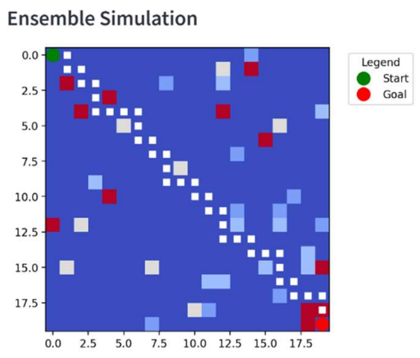

Figure-4
Optimized path using a mix of A and Bellman Ford Algorithms and risk factorization when Goal point is completely surrounded. Here, coloured larger squares indicate the obstacles.*

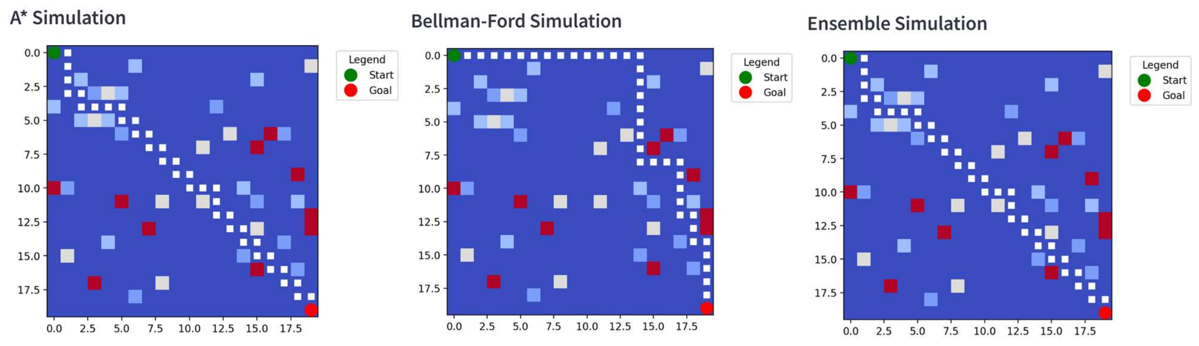

Figure-5
Path visualization by A, Bellman-Ford as well as Ensemble Algorithm*

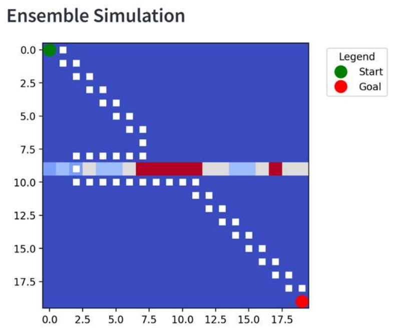

Figure-6
Path visualization by Ensemble Algorithm for blocked path

IV. Discussion

4.1 Conclusion

The study highlights the crucial role of advanced pathfinding algorithms in optimizing UAV operations within complex and dynamic environments like those found in Tamil Nadu. The implementation of the custom optimized algorithm, which blends the advantages of A* and Bellman-Ford algorithms while incorporating real-time adaptive heuristics, has the potential to demonstrate superior performance in dynamic settings. This approach allows for continuous adjustment to environmental changes, ensuring optimal path selection and resource usage.

The comparative analysis of the three algorithms revealed distinct strengths and weaknesses, highlighting the importance of selecting the appropriate algorithm based on the specific operational context.

4.2 Future Work

1. Algorithm Refinement and Optimization

2. Risk Evaluation and Path Finding
3. Scalability and Deployment in Larger Environments
4. Integration of Machine Learning Techniques
5. Real-World Testing and Validation
6. Regulatory and Ethical Considerations

By pursuing these research directions, the pathfinding capabilities of UAVs can be significantly enhanced, paving the way for their integration into a wide array of applications, from urban logistics to environmental conservation efforts.

Reference Citations

1. O'Connor, D., 2024. *The Bellman-Ford-Moore Shortest Path Algorithm. MATLAB Central File Exchange.*

2. Kanathasan, V., 2024. *Djikstra and bellman. MATLAB Central File Exchange.*

3. MathWorks, 2023. *Guidance, Navigation, and Control - MATLAB & Simulink.*

4. MathWorks, 2023. *Drone Navigation - MATLAB & Simulink.*

5. Lu, Y., Cai, M., Ling, W., & Zhou, X. (2017). *Quadrotor control, path planning and trajectory optimization*. Zenodo. https://doi.org/10.5281/zenodo.6796215