Project Report

Formation Based UAV Path Planning

Nirbhay Kachhatiya (240702)
Rohan R Bharadwaj (250911)

Goals of the Project

- Python Libraries
  - Matplotlib
  - Numpy
- Path-Planning Algorithms
  - BFS, DFS - For Unweighted graphs
  - Dijkstra's, A* - For Weighted graphs

- Trajectory Smoothing Techniques
- UAV Formation Simulation
- Comparing minimum-time and minimum-energy paths

Python Libraries

Matplotlib:

- We were introduced to Matplotlib as a fundamental library for data visualization in Python.
- We learnt how to create basic plots such as line graphs, scatter plots, and bar charts, and how to label axes, add titles, legends, and grids.
- We also went through customizing plots, plotting multiple datasets, and visualizing simulation results, which showed how Matplotlib helps interpret data effectively.

Numpy:

- We were introduced to NumPy as a foundation for numerical computing in Python.
- We learnt how to create arrays, understand shapes and data types, and apply indexing and slicing.
- We also went through broadcasting, basic mathematical and statistical functions, random number generation, and simple linear algebra which highlighted NumPy's efficiency over ordinary Python lists.

Graphs and Their Representation

A Graph consists of a set of vertices (V) and a set of edges (E). A graph can be stored as an adjacency matrix or an adjacency list.

Adjacency Matrix:

- The graph is stored as a VxV 2D array, with each cell representing whether an edge exists between a pair of vertices (its weight if it exists).
- It requires $O(V^2)$ space but the presence/weight of an edge can be checked in O(1) time.
- It's suitable for a dense graph with many edges.

Adjacency List:

- The graph is stored as an array of arrays. Each inner array is the list of neighbours of the vertex numbered with the index of that array.
- It requires O(V+E) space. Presence of an edge requires more time to check but the neighbours of a vertex can be accessed relatively easily.
- It's suitable for sparse graphs.

Graph Traversal Algorithms for Unweighted Graphs

Breadth-First Search (BFS):

- Starting from a source node, it first visits all immediate neighbors before moving to their neighbors.
- This approach ensures that nodes closer to the source are visited first and helps find the shortest path in unweighted graphs.
- Implementation using queue: The current node is visited, and all its unvisited adjacent nodes are enqueued. The node dequeued from the queue is then visited, and the process continues until the queue is empty.

Depth-First Search (DFS):

- Starting from a source node, DFS explores as deep as possible along one branch before backtracking to explore other branches.
- This approach does not guarantee the shortest path but is useful to explore all possible paths and analyzing graph structure.
- Implementation using stack: The current node is visited and pushed onto a stack. An unvisited node is pushed onto the stack and visited next. When no unvisited neighbors remain, nodes are popped from the stack to backtrack.

Dijkstra’s Algorithm & Implementation

1. Data Structures needed

- We create a “distances” array where we will track the shortest distance from our starting node to every other node
- We create a “visited” set/array to keep track of which nodes we've already processed. This ensures we don't waste time revisiting nodes we've already finalized the shortest path for.
- We create a priority queue (min-heap) that will help us always pick the next node with the smallest known distance
- We optionally create a previous array to reconstruct the actual shortest path later.

2. The Algorithm

Step 1: Initialization

- We set the distance to our starting node as 0 (because we're already there)
- For all other nodes, we set their distance to infinity (since we haven't found a path to them yet)
- We add the starting node to our priority queue with distance 0
- We mark all nodes as unvisited

Step 2: The Main Loop

We keep going while there are still unvisited nodes in our priority queue:

1. We extract the node with the smallest distance from our priority queue
2. We mark this node as visited so we don't process it again
3. For each neighbor of this current node:
   - We calculate a tentative distance = (current node's distance) + (weight of edge to neighbor)
   - If this tentative distance is smaller than what we have recorded for that neighbor:
     - We update the neighbor's distance in our distances array
     - We update the neighbor's "previous" node to point back to the current node (for path reconstruction)
     - We add this neighbor (with its new distance) to our priority queue

Step 3: Termination

- When our priority queue is empty (or we've visited our target node if we only care about one destination), we stop.
- Now our distances array contains the shortest distances from our starting node to every other reachable node
- If we want the actual path to a node, we backtrack through the previous array from destination to source

A* Algorithm and Implementation

1. Data Structures needed

- Open Set (Priority Queue/Min-Heap): Nodes to be evaluated, prioritized by their estimated total cost (f = g + h)
- Closed Set (Set/Array): Nodes already evaluated (visited)
- g Scores Array: Tracks the actual shortest distance from start node to each node (like Dijkstra's distances)
- f Scores Array: Tracks estimated total cost from start to goal through each node (g + heuristic)
- Came From Array: For path reconstruction, tracks the previous node in the optimal path

2. The Algorithm

Step 1: Initialization

- Set g(start) = 0 (actual cost from start to start)
- Set f(start) = h(start) (estimated total cost)
- Add start node to open set with priority f(start)
- Initialize all other g scores to infinity
- Initialize all other f scores to infinity
- Mark all nodes as unvisited (not in closed set)

Step 2: The Main Loop

While open set is not empty:

- Get current node: Extract node with lowest f-score from open set
- Check if goal reached: If current node is the goal, reconstruct and return path
- Move to closed set: Mark current node as visited by adding to closed set
- Explore neighbors: For each neighbor of current node:
  - Skip if neighbor is in closed set (already evaluated)
  - Calculate tentative_g = g(current) + cost(current, neighbor)
  - If tentative_g < g(neighbor):
    - Update path: Set came_from[neighbor] = current
    - Update g-score: g(neighbor) = tentative_g
    - Update f-score: f(neighbor) = g(neighbor) + h(neighbor, goal)
    - Add to open set: If neighbor not in open set, add it with priority f(neighbor)

Step 3: Termination & Path Reconstruction

- If goal found: Backtrack using came_from array from goal to start to get optimal path
- If open set empty: No path exists from start to goal
- Result: Returns either the optimal path or failure indication

4. Heuristic Requirements

- Admissible: $h(n) \leq$ actual cost from n to goal (never overestimates)
- Consistent (Monotonic): $h(n) \leq cost(n, n') + h(n')$ for all neighbors n'
- Common Heuristics:
  - Manhattan distance (grids with 4-direction movement)
  - Euclidean distance (direct line distance)
  - Chebyshev distance (grids with 8-direction movement)

THANK YOU