UNIVERSITI TEKNOLOGI PETRONAS

SEPTEMBER 2024 SEMESTER

TEB1113/TFB2023: Algorithm and Data Structure

Drone Simulation Millitary Drone

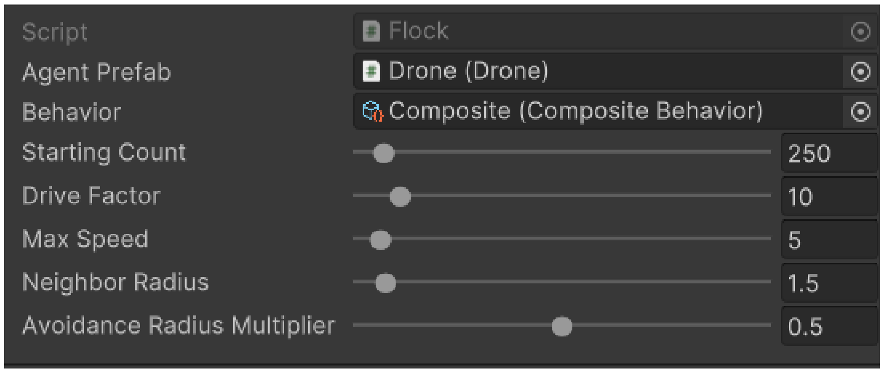
No.	Name	ID	Program
1	Azri Bin Pakrurosi	22006248	Computer Engineering
2	Mohd Hazeem Bin Mohd Hafiz	22005131	Computer Engineering
3	Adeeb Muhammad Bin Anuar	22005531	Computer Engineering
4	Muhammad Izzat Haikal Bin Ibrahim	22006122	Computer Engineering
5	Muhammad Irham Bin Mohd Isham	22002975	Computer Engineering

Contents

1. **PROJECT INTRODUCTIO**N

The project focuses on a military drone swarm simulation, where multiple drones operate as part of a network. The system includes functionalities for drone communication, pathfinding, and control. Drones can be manually controlled or autonomously behave based on flocking behavior. The network supports operations like searching for drones, controlling their movement, and calculating the shortest paths between them. The system is partitioned into networks based on drone attributes, allowing for different strategies in drone management and communication. The project aims to simulate a dynamic and responsive drone system for military applications.

2. **DETAIL OF THE SIMULATION**



Script	Flock
Agent Prefab	Drone (Drone)
Behavior	Composite (Composite Behavior)
Starting Count	250
Drive Factor	10
Max Speed	5
Neighbor Radius	1.5
Avoidance Radius Multiplier	0.5

Partition:

The drone partition is randomized

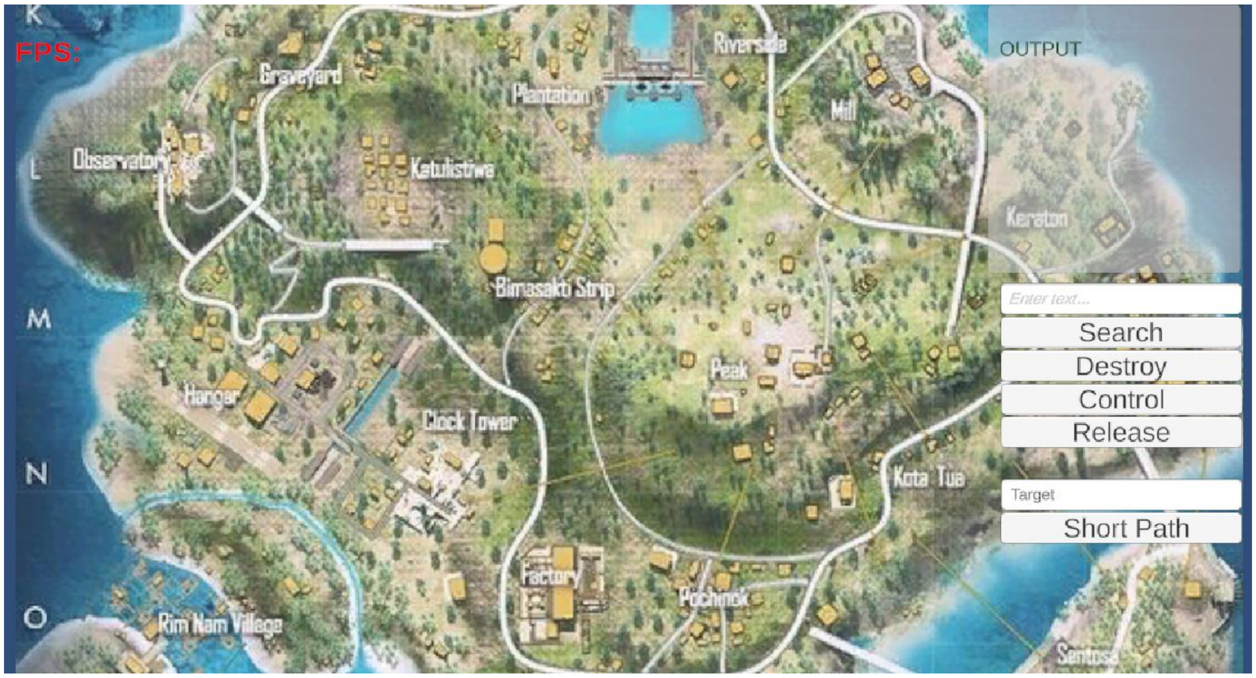
Partition 1: Droneammo<50 = green

Partition 2: Droneammo>50 = light brown

Specification

Device name	LAPTOP-GQ61BGNP
Processor	11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz 2.69 GHz
Installed RAM	16.0 GB (15.7 GB usable)
System type	64-bit operating system, x64-based processor

2.1 Simulation Display



Enter text: Input for Drone ID

Search button: To search for the drone location

Destroy button: To destroy the drone

Control button: To control the current drone

Release button: To release the current controlled drone

Target button: Input for target drone to find short path

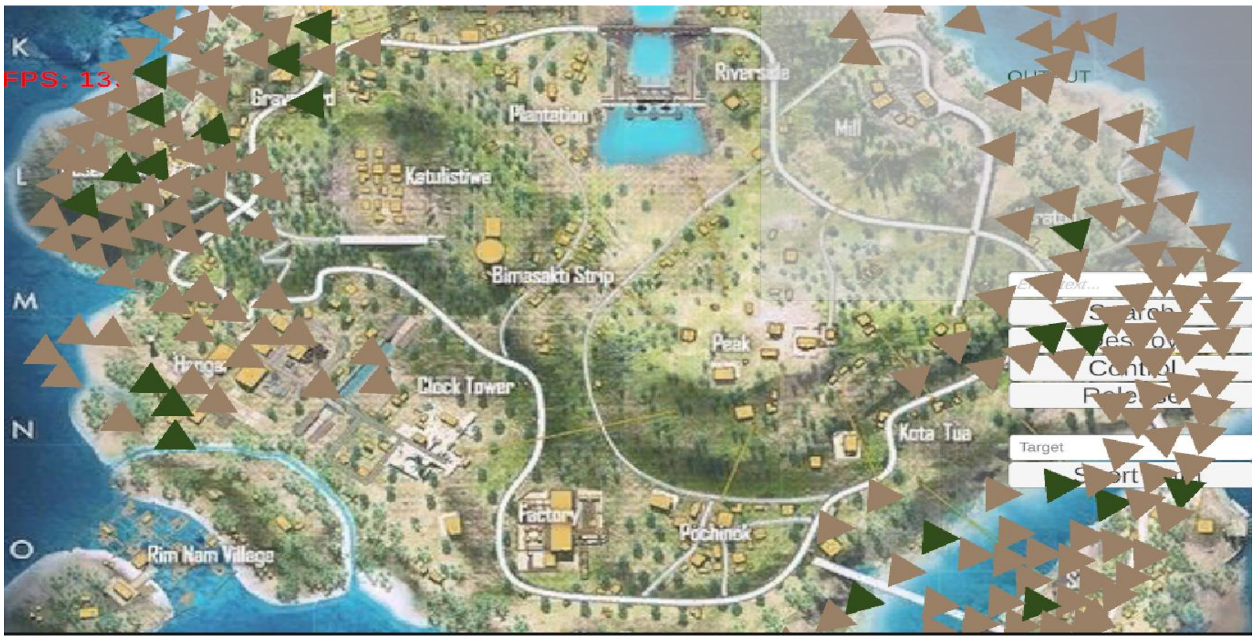
Short Path button: To get the short path from “enter text” drone ID and “Target” drone ID

FPS: Display Frame Rate per Second

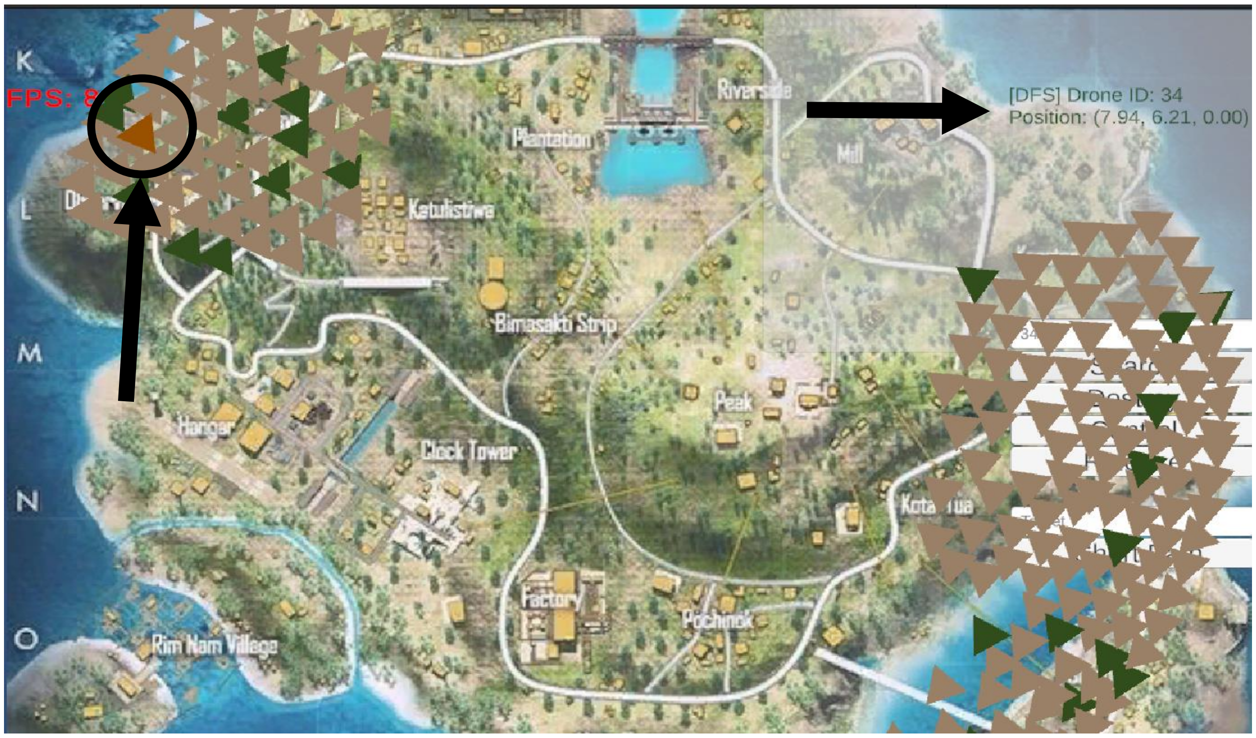
Output: Display all the output from the button

2.2 Simulation Tutorial

Simulation Scene



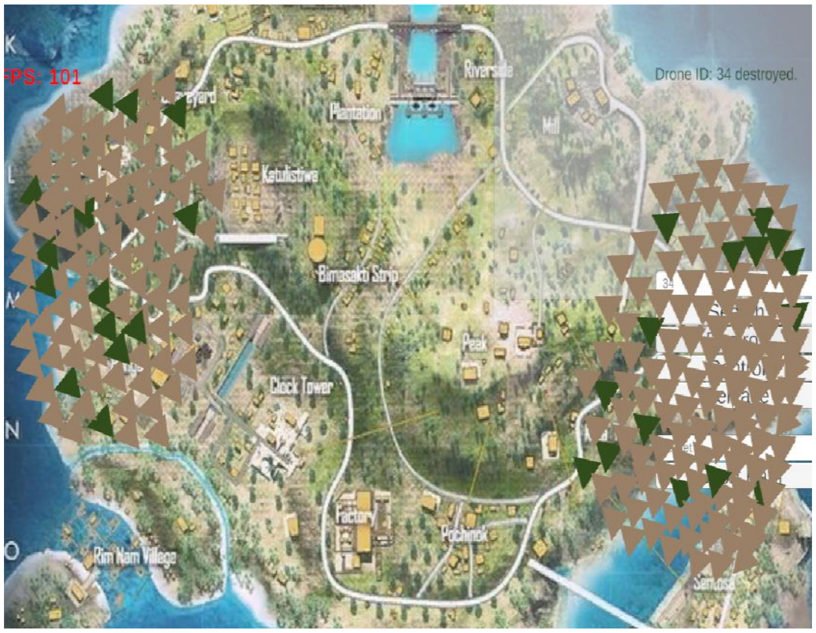
When click search:



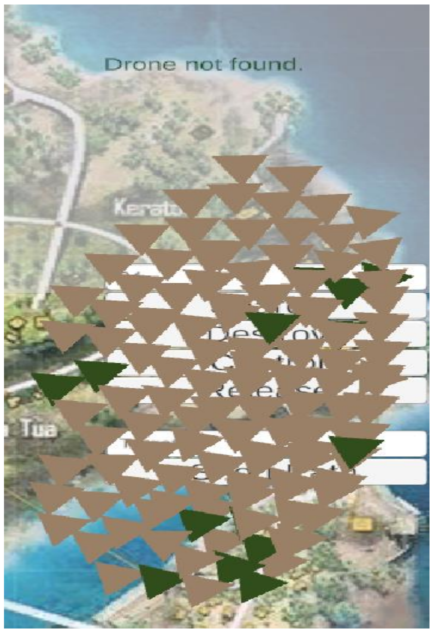
[DFS] Drone ID: 34
Position: (7.94, 6.21, 0.00)

Drone change color and report the position

When Click destroy:

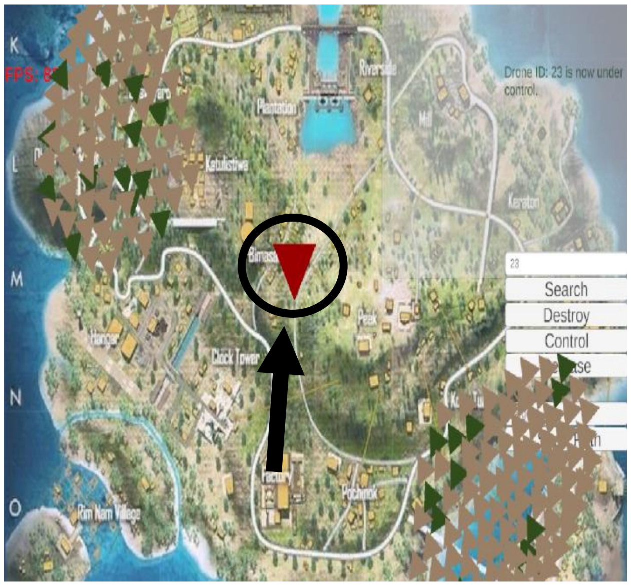

FPS: 101

Drone ID: 34 destroyed.


Drone not found.

Drone will permanently destroy and when click search the drone will not found

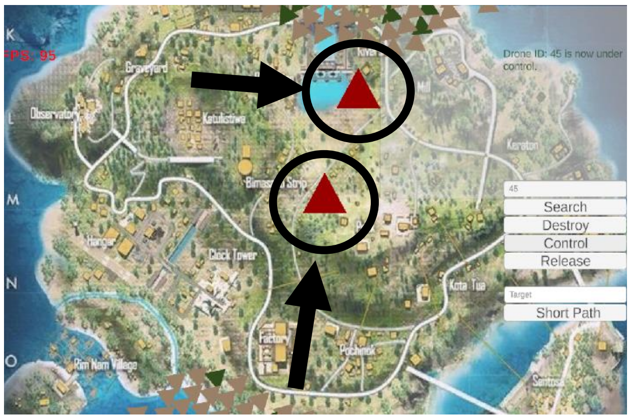
When click control and release:


Drone ID: 23 is now under control.

Drone can be controlled using UpArrow, DownArrow, LeftArrow and RightArrow.

When click the release button selected Drone will convert back to their original shape and get back to the original behavior.

Drones can be controlled simultaneously but need to put the id 1 by 1 and when the user want to release the drone they need to release by putting the ID back 1 by 1



Drones can be controlled simultaneously but need to put the id 1 by 1.

If the user want to release the drone they need to release by putting the ID back 1 by 1

3. PERFORMANCE

1. When the game start the FPS are 180. Then it will quickly drop to 90 – 110.
2. The FPS will drop to 85- 95 when the search button is clicked. This maybe because the system need time to track the drone location.
3. However, the FPS will continue drop depending on the number of drone that are being control currently.

4. EXPLANATION OF THE CODE

4.1 Graph

This class is designed to represent a communication network among drones using a graph data structure. Each drone is a node, and a communication link between two drones is an edge. The graph is managed using an adjacency list.

The communication network is modeled as an undirected graph, where:

- **Nodes**: Represent drones.
- **Edges**: Represent connections (communication links) between drones.

*4.*1.1 Adjacency List

The **adjacency list** is a data structure used to store the graph efficiently:

- **Key**: The DroneId of a node (drone).
- **Value**: A list of integers representing the IDs of neighboring drones connected to the key drone.

4.1.2 Searching Using BFS

The Breadth-First Search (BFS) algorithm is used to traverse the graph and search for a drone starting from a given node.

Key Functionality

- **Input**: Starting drone ID (droneId).
- **Output**: The drone object or null if not found.
- **Algorithm**:
    - Use a queue to explore nodes level by level.
    - Maintain a set of visited nodes to avoid cycles.
    - Traverse neighbors of each node and enqueue them.

4.1.3 Finding the Shortest Path

The shortest path between two drones is calculated using **BFS**. BFS naturally finds the shortest path in an unweighted graph by exploring all neighbors level by level.

Key Functionality

- **Input**: Starting (startId) and ending (endId) drone IDs.
- **Output**: A list of drone IDs representing the shortest path, or null if no path exists.
- **Algorithm**:
  - Use BFS to traverse the graph from startId.
  - Maintain a previous dictionary to reconstruct the path.
  - Stop traversal when the endId is reached.
  - Reconstruct the path using the previous dictionary.

However, there is a problem we encounter here...

- **Challenge**: The FindShortestPath function assumes there is always a valid path. Our group faced issues in cases where:
  - Some drones were not connected.
  - The adjacencyList did not correctly initialize or update when drones were added or removed.
- **Impact**: The function returned null unexpectedly or generated incomplete paths due to missing connections.

4.2 UIManager

This Unity C# script, UIManager, is designed to interact with a flock of drones in a simulation. Here's a breakdown of the key parts of the code and their functionality:

1. **UI Elements:**
   a. **TMP_InputField inputText and TMP_InputField targetText:** Used to input drone IDs for searching and pathfinding.
   b. **Button findDrone, destroyDrone, controlDrone, etc.:** Buttons trigger actions like finding, destroying, or controlling drones.

    c. **TMP_Text position:** Displays information about the drones, such as position, status, or errors.

2. **Flock and Drone Interaction:**
    a. **Flock flock:** A reference to the Flock class that manages the drones and their networks.
    b. **DroneNetworkCommunication currentNetwork:** Holds the network in which the currently found drone resides.
    c. **Drone controlledDrone:** The drone currently under user control.

Key Functions:

1. **FindDroneButton()**
   Purpose: Searches for a drone by its ID using BFS or DFS within the drone networks.

2. **DestroyDroneButton()**
   Purpose: Destroys (removes) a drone from both networks and deactivates it in the simulation.

3. **ControlDroneButton()**
   Purpose: Grants user control over a specific drone.
4. **ReleaseControlButton()**
   Purpose: Releases control of a previously controlled drone.

5. **FindPathButton()**
   Purpose: Finds the shortest path between two drones using a pathfinding algorithm.

4.3 Flock

The Flock class is responsible for initializing, managing, and simulating the behavior of a group of drones. It provides a basis for drone coordination and interaction by combining movement logic, drone network partitioning, and flocking behavior.

Key Functions:

1. **InitializeParameters()**
   Purpose: Sets up essential parameters for the flock's behavior, ensuring efficient calculations during simulation.

2. **InitializeAgents()**
   Purpose: Spawns and initializes all drones in the simulation.

3. **PartitionAndInitializeNetworks()**
   Purpose: Divides drones into two networks based on a property (e.g., ammo count) and initializes them.

4. **PartitionDronesByAmmo()**
   Purpose: Splits the array of drones based on their Ammo property relative to a pivot value.

5. **GetNearbyObjects(Drone agent)**
   Purpose: Identifies objects (drones) within a defined radius around a specific drone.

6. **FindDroneById(int droneId)**
   Purpose: Locates a specific drone by its unique ID.

7. **Update()**
   Purpose: Executes the flocking behavior for all drones in every frame.

8. **AddDronesToNetwork(Drone[] drones, DroneNetworkCommunication network, Color color)**
   Purpose: Adds a set of drones to a specified network and assigns them a visual color.

9. **CalculateMove(Drone agent, List<Transform> context, Flock flock) (Behavior Script)**
   Purpose: Although not part of this script, this function is invoked in Update() to compute the movement of each drone.

4.4 Drone

The Drone class defines the behavior and properties of individual drone objects in a simulation. Each drone has attributes like **DroneId**, **Ammo**, **WeaponCapacity**, and **Temperature**, along with behaviors for randomization, movement, and manual control.

Key Functions:

1. Initialize(Flock flock)

- **Purpose**: Sets up the drone with a reference to the Flock object and assigns it a unique ID.

2. ChangeColor(Color newColor)

- **Purpose**: Changes the visual appearance of the drone by modifying its material color.

3. Move(Vector2 velocity)

- **Purpose**: Handles movement of the drone.

4. HandleManualControl()

- **Purpose**: Allows manual control of the drone using keyboard input (arrow keys).

5. SelfDestruct()

- **Purpose**: Deactivates the drone object.

6. RandomizeAttributes()

- **Purpose**: Assigns random values to the drone's attributes (Ammo and WeaponCapacity).

7. Start()

- **Purpose**: Initializes key components and starts periodic randomization of attributes.

8. RandomizeAttributesPeriodically()

- **Purpose**: Updates the drone’s attributes periodically during the simulation.

9. Update()

- **Purpose**: Continuously updates the drone's state each frame.

Key Properties:

1. DroneId

- A unique identifier for the drone.

2. Ammo and WeaponCapacity

- Represent the drone's ammunition and weapon system capacity, updated randomly.

3. Temperature

- Simulates dynamic temperature changes for the drone.

4. IsUnderControl

- Tracks whether the drone is under manual control.

5. AgentFlock and AgentCollider

- References to the flock the drone belongs to and its collider for interactions.

5. TIME COMPLEXITY

1. AddDrone(Drone drone)

- **Operation**: Adds a drone to the drones dictionary and initializes an empty adjacency list for the drone.
- **Time Complexity**: O(1)
  - **Reason**: Both the insertion into the dictionary and the creation of a new list are constant-time operations.

2. ConnectDrones(int droneId1, int droneId2)

- **Operation**: Adds an undirected connection between two drones in the adjacency list.
- **Time Complexity**: O(1)
  - **Reason**: Dictionary lookups and list additions are constant-time operations.

3. GetDroneById(int droneId)

- **Operation**: Retrieves a drone by its ID from the drones dictionary.
- **Time Complexity**: O(1)
  - **Reason**: Dictionary lookup is a constant-time operation.

4. SearchDroneWithPositionBFS(int droneId)

- **Operation**: Performs a breadth-first search (BFS) starting from the given drone ID to search for a drone.
- **Time Complexity**: O(V + E)
  - **Reason**: In BFS, each vertex (drone) and each edge (connection between drones) is visited at most once. V represents the number of drones (vertices), and E represents the number of connections (edges).

5. FindShortestPath(int startId, int endId)

- **Operation**: Finds the shortest path between two drones using BFS.
- **Time Complexity**: O(V + E)

- **Reason**: Like the BFS search, this algorithm explores each vertex and edge once to find the shortest path. The additional work of reconstructing the path is linear in terms of the number of vertices in the path.

6. ReconstructPath(Dictionary<int, int> previous, int startId, int endId)

- **Operation**: Reconstructs the path from startId to endId by following the previous dictionary.
- **Time Complexity**: O(V)
    - **Reason**: In the worst case, the entire path could involve visiting all vertices, resulting in a linear time complexity with respect to the number of vertices in the path.

7. RemoveDrone(int droneId)

- **Operation**: Removes a drone from the drones dictionary and the adjacency list, and also removes all references to the drone from other adjacency lists.
- **Time Complexity**: O(V + E)
    - **Reason**:
        - Removing the drone from the drones dictionary and the adjacency list is O(1).
        - Removing all references to the drone from other adjacency lists requires iterating through all the drone's connections, which could involve iterating over all edges (in the worst case). Therefore, the overall complexity is O(V + E).

Summary of Time Complexities:

- AddDrone: O(1)
- ConnectDrones: O(1)
- GetDroneById: O(1)
- SearchDroneWithPositionBFS: O(V + E)
- FindShortestPath: O(V + E)
- ReconstructPath: O(V)
- RemoveDrone: O(V + E)