Drone Operating System Documentation

Overview

This project involves the creation of a simulated drone operating system with basic navigation commands (takeoff, land, return to home, survey, and failure) and mission planning capabilities. The mission planning uses Dijkstra's algorithm to find the shortest path in a graph representing various waypoints. The system is designed using object-oriented principles and includes multithreading for concurrent operations.

Classes and Responsibilities

1. DroneOperation Class

This is the base class for all drone operations, which are executed using polymorphism.

Derived Classes

-Takeoff: Handles the drone's takeoff process.

-Land: Handles the drone's landing process.

-ReturnToHome: Manages the drone's return to the home location.

-Survey: Conducts a survey operation over a specified area.

-Failure: Manages failure scenarios for the drone.

Time Complexity: O(1) for each operation execution.

Pseudocode:

```
pseudo
class DroneOperation
    method execute()


class Takeoff extends DroneOperation
    method execute()
        print "Drone taking off..."


class Land extends DroneOperation
    method execute()
        print "Drone landing..."


class ReturnToHome extends DroneOperation
    method execute()
```

```
        print "Drone returning to home..."


class Survey extends DroneOperation
    method execute()
        print "Drone surveying area..."


class Failure extends DroneOperation
    method execute()
        print "Drone operation failure!"
```

2. MissionPlanning Class

This class is responsible for planning the mission using a graph where nodes represent waypoints and edges represent paths between them. It uses Dijkstra's algorithm to find the shortest path.

Time Complexity:

- Initialization: O(V), where V is the number of nodes.

- Edge Addition: O(1) per edge.

- Dijkstra's Algorithm: O(E + V log V), where E is the number of edges and V is the number of nodes.

Pseudocode:

```
pseudo
class MissionPlanning
    method addEdge(u, v, weight)
        add edge (v, weight) to adjacencyList[u]
        add edge (u, weight) to adjacencyList[v]


    method findShortestPath(start, end):
        create priorityQueue
        distances[startNode] = 0
        add startNode to priorityQueue with priority 0


        while priorityQueue is not empty:
            currentNode = priorityQueue.removeMin()
            for each neighbor of currentNode:
                newDistance = distances[currentNode] + edgeWeight(currentNode, neighbor)

                if newDistance < distances[neighbor]:
                    distances[neighbor] = newDistance
                    add neighbor to priorityQueue with priority newDistance

        return distances
```

3. Survey Class

Handles the survey operation that the drone performs once it reaches the destination.

Time Complexity: O(1) for executing the survey.

4. DroneController Class

Manages the execution of various drone operations and mission planning, and coordinates multithreading for these tasks.

Time Complexity: Dependent on the specific operations being executed (as described above).

Multithreading Logic:

1. Create a thread for each operation.
2. Start each thread.
3. Wait for all threads to complete using `join`.

Pseudocode:

```
pseudo
class DroneController
    method executeOperation(operation)
        operation.execute()

    method performMissionPlanning(mp, start, end)
```

```
        path = mp.findShortestPath(start, end)
        print "Path: ", path

    method performSurvey()
        survey.execute()
```

Multithreading Pseudocode

```
pseudo
method main()
    create DroneController instance

    create thread for takeoff operation
    create thread for mission planning
    create thread for survey operation
    create thread for return to home operation

    start all threads
    join all threads
```

Algorithms and Logic

1. Polymorphism for Drone Operations:

- The `DroneOperation` base class uses polymorphism to allow different operations to be executed using a common interface.
- Each derived class overrides the `execute` method to provide specific functionality for the drone operation.

2. Dijkstra's Algorithm for Mission Planning:

- Dijkstra's algorithm is used to find the shortest path in a graph.
- The algorithm maintains a priority queue to explore the shortest paths efficiently.
- It updates distances to each vertex based on the shortest known path and iteratively improves these distances until the shortest path to the destination is found.

Selection of Dijkstra's Algorithm:

- Reason: Dijkstra's algorithm is well-suited for finding the shortest path in graphs with non-negative weights. It is efficient with a time complexity of O(E + V log V) when implemented with a priority queue (min-heap), making it optimal for the mission planning task.

Simulation Scenarios

Best Case:

- The best case scenario occurs when the start and end nodes are directly connected with minimal edges, leading to the shortest path calculation in O(1) time for edge retrieval and priority queue operations.

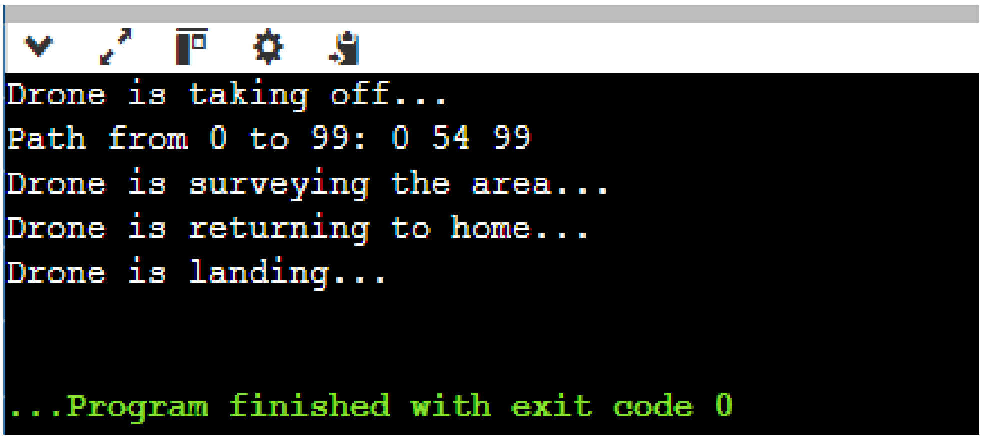
Worst Case:

- The worst case scenario involves a highly connected graph where each node is connected to multiple other nodes. This results in O(E + V log V) complexity due to extensive edge relaxation and priority queue operations.

Summary

This documentation provides a comprehensive overview of the drone operating system project, detailing the classes, their responsibilities, the algorithms used, and the time complexity of various operations. The implementation is designed using object-oriented principles and incorporates multithreading for concurrent execution of drone operations.

Results



```
Drone is taking off...
Path from 0 to 99: 0 54 99
Drone is surveying the area...
Drone is returning to home...
Drone is landing...


...Program finished with exit code 0
```