TASK-3

Brief Description of Problem Statement :

We are given a 2D plane represented as an M x N grid. We need to design an algorithm that guides multiple drones flying autonomously to their respective destinations without any collisions. The input to the algorithm includes the number of drones and their starting and ending positions along with the time at which they will start.

Algorithm Used : A* search algorithm.

Approach

- To solve the problem of multiple drones flying in a 2-D space autonomously, we can use the A* search algorithm, which is a widely used pathfinding algorithm. The A* algorithm works by finding the shortest path between a start node and an end node in a graph. In our case, each drone can be considered as a node, and the edges between them represent the path that the drone can take to reach its destination.

Algorithm

- The first step in the algorithm would be to create a graph representation of the problem.
- Once the graph is created, we can apply the A* algorithm to each drone node to find the shortest path to its destination.
- To ensure that there are no collisions between the drones, we can use a simple collision detection mechanism.

  In addition to the A* algorithm, we can also use other optimization techniques like genetic algorithms or simulated annealing to further optimize the pathfinding algorithm.

  Overall, the algorithm we have described above should be able to solve the problem of multiple drones flying in a 2-D space autonomously while avoiding collisions and minimizing the time taken to reach the destination.

Explanation of code

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <cmath>
using namespace std;
```

This section includes the necessary headers for the program, such as iostream for input/output operations, vector for dynamically-sized arrays, queue for implementing priority queues, algorithm for functions such as reverse, and cmath for mathematical operations

```
const int GRID_SIZE = 20;
const int DIRECTION = 8;
const int dx[DIRECTION] = { -1, -1, 0, 1,
1, 1, 0, -1 };
const int dy[DIRECTION] = { 0, 1, 1, 1, 0, -
1, -1, -1 };
```

These constants define the size of the grid, the number of directions a node can move, and the relative x/y coordinates of each direction.

```
struct Drone {
   int x1, y1, x2, y2, t;
};
```

This struct defines a drone with two x/y coordinates and a time variable.

```
struct Node {
   int x, y, f, g, h;
   Node* parent;
   Node(int x_, int y_) : x(x_),
y(y_), f(0), g(0), h(0),
parent(nullptr) {}
};
```

This struct defines a node in the A* algorithm with x/y coordinates, and f/g/h values for the node's cost, as well as a pointer to the node's parent.

```
struct CompareNode {
   bool operator()(const
Node* n1, const Node* n2)
const {
      return n1->f > n2->f;
   }
};
```

This struct defines a comparison operator for nodes in the priority queue, sorting them by their f value.

```
class Grid {
public:
   Grid(int size_) : size(size_) {
      data.resize(size_, vector<int>(size_));
   }

   bool is_valid(int x, int y) const {
      return x >= 0 && x < size && y >= 0
&& y < size && data[x][y] == 0;
   }

   void set_obstacle(int x, int y) {
      data[x][y] = 1;
   }

   void reset() {
      data.assign(size, vector<int>(size));
   }

private:
   int size;
   vector<vector<int>> data;
};
```

This class defines a grid object, which has a size and a two-dimensional vector of integers to represent obstacles. The class also has functions to check if a coordinate is valid, set obstacles, and reset the grid.

```
class AStar {
public:
  AStar(Grid& grid_, int sx_, int sy_, int tx_, int ty_) :
    grid(grid_), sx(sx_), sy(sy_), tx(tx_), ty(ty_) {}

  bool search(vector<pair<int, int>>& path) {
    priority_queue<Node*, vector<Node*>, CompareNode> open_list;
    vector<vector<bool>> closed_list(grid.size, vector<bool>(grid.size, false));
    Node* start = new Node(sx, sy);
    start->g = 0;
    start->h = heuristic(start);
    start->f = start->g + start->h;
    open_list.push(start);

    while (!open_list.empty()) {
      Node* curr = open_list.top();
      open_list.pop();
      closed_list[curr->x][curr->y] = true;

      if (curr->x == tx && curr->y == ty) {
        path.clear();
        while (curr) {
          path.emplace_back(curr->x, curr->y);
          curr = curr->parent;
        }
        reverse(path.begin(), path.end());
        return true;
      }

      for (int i = 0; i < DIRECTION; i++) {
        int nx = curr->x + dx[i];
        int ny = curr->y + dy[i];
        if (!grid.is_valid(nx, ny) || closed_list[nx][ny]) continue;
        Node* neighbor = new Node(nx, ny);
        neighbor->g = curr->g + 1;
        neighbor->h = heuristic(neighbor);
        neighbor->f = neighbor->g + neighbor->h;
        neighbor->parent = curr;
        open_list.push(neighbor);
      }
    }
    return false;
  }

private:
  Grid& grid;
  int sx, sy, tx, ty;

  int heuristic(Node* node) const {
    int dx = abs(node->x - tx);
    int dy = abs(node->y - ty);
                              return
```

- This code defines a class called AStar which implements the A* search algorithm for pathfinding on a grid.
- The A* algorithm is implemented using a priority queue to store the open list of nodes to be explored, and a closed list to keep track of nodes that have already been explored.
- Overall, this code provides a basic implementation of the A* algorithm for grid-based pathfinding.