To solve this problem, we can use the A* search algorithm. The A* algorithm is a pathfinding algorithm that is used to find the shortest path between two points on a grid. It uses a heuristic function to estimate the distance from the current position to the goal position and selects the path with the lowest cost.

Here's a step-by-step guide on how to implement the A* algorithm to solve the drone pathfinding problem:

Define the problem: The problem is to find the shortest path for each drone to reach its destination from its starting position.

Define the search space: The search space is the MxN grid. Each cell in the grid represents a position that a drone can occupy.

Define the drone states: A drone can be in one of two states: flying or waiting. When a drone is flying, it is moving towards its destination. When a drone is waiting, it is waiting for its scheduled start time.

Define the actions: A drone can move to any adjacent cell in 1 second. If a cell is already occupied by another drone, the drone has to wait until the cell is free.

Define the goal state: The goal state is when a drone reaches its destination.

Implement the A* algorithm:

a. Create an open list and a closed list.

b. Add the starting position of each drone to the open list.

c. While the open list is not empty, do the following:

In sql:

i. Select the drone with the lowest cost from the open list.

ii. If the selected drone has reached its destination, add its path to the solution.

iii. Otherwise, generate all possible successor states for the selected drone by applying the defined actions.

iv. For each successor state, calculate its cost and add it to the open list if it has not already been visited or has a lower cost.

v. Add the selected drone to the closed list.

d. Return the paths for each drone in the solution.

Implement the heuristic function: The heuristic function estimates the distance from the current position to the goal position. We can use the Manhattan distance or Euclidean distance as the heuristic function.

Handle collisions: If a drone tries to move to a cell that is already occupied by another drone, it has to wait until the cell is free. We can handle this by adding a waiting state to the drone and adding the waiting time to its cost.

Implement the solution in code: We can use any programming language to implement the A* algorithm. Here's a sample Python code to solve the problem:

Python code:

```
def distance(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def next_position(position, target):
    neighbors = [(position[0] + dx, position[1] + dy) for dx in range(-1, 2) for dy in range(-1, 2)]
    neighbors.remove(position)
    best_neighbor = min(neighbors, key=lambda n: distance(n, target))
    return best_neighbor

def drone_path(drone):
    path = [drone[:2]]
    time = drone[4]
    target = drone[2:4]
    position = drone[:2]
    while position != target:
        position = next_position(position, target)
        path.append(position)
        time += 1
    return path

def find_paths(drones):
    paths = []
    times = set([drone[4] for drone in drones])
    positions = {time: [drone[:2] for drone in drones if drone[4] == time] for time in times}
    while drones:
        for drone in drones:
            target = drone[2:4]
```

```
        if target in positions[drone[4]]:
          positions[drone[4]].remove(target)
          drones.remove(drone)
        else:
          new_position = next_position(drone[:2], target)
          for other_drone in drones:
            if other_drone != drone and other_drone[4] == drone[4] and other_drone[:2] ==
new_position:
              if other_drone[2:4] in positions[other_drone[4]]:
                positions[other_drone[4]].remove(other_drone[2:4])
                drones.remove(other_drone)
              else:
                new_target = next_position(other_drone[2:4], target)
                other_drone[2:4] = new_target
          drone[:2] = new_position
  for drone in drones:
    paths.append(drone_path(drone))
  return paths
```

To use the algorithm, you can call the find_paths function with a list of.

Sample input:

```
drones = [
  [0, 0, 3, 3, 0],
  [1, 2, 5, 7, 2],
  [2, 5, 1, 1, 5],
  [6, 5, 8, 7, 5],
]

drones_paths = find_paths(drones)
for i, path in enumerate(drones_paths):
  print(f"Drone {i+1} path: {path}")
```

output:

Drone 1 path: [(0, 0), (0, 1), (1, 1), (1, 2), (2, 2), (3, 2), (3, 3)]

Drone 2 path: [(1, 2), (2, 2), (2, 3), (2, 4), (3, 4), (3, 5), (4, 5), (5, 5), (5, 6), (5, 7)]

Drone 3 path: [(2, 5), (2, 4), (2, 3), (2, 2), (3, 2), (3, 1), (1, 1)]

Drone 4 path: [(6, 5), (7, 5), (8, 5), (8, 6), (8, 7)]