Algorithm to solve this problem:

Step 1: Read input from the user, which includes the number of drones, their starting positions, end positions, and start times.

Step 2: Initialize a priority queue to maintain the order in which the drones will move.

Step 3: For each drone, calculate the shortest path from its starting position to its end position using a pathfinding algorithm like A\* or Dijkstra's Algorithm.

Step 4: Add the drone to the priority queue with its start time as the priority.

Step 5: While there are drones in the priority queue:

a. Pop the drone with the earliest start time from the priority queue.

b. Move the drone to its next position along its shortest path.

c. Check if there is a collision with any other drone at that position and time. If there is a collision, re-route one or both drones to avoid the collision.

d. If the drone has reached its destination, mark it as completed.

e. If the drone has not reached its destination, add it back to the priority queue with its new start time (the current time plus the time it takes to move to its next position).

Step 6: Repeat step 5 until all drones have reached their destinations.

Python code implementation for above algorithm:

import heapq

# Define the size of the grid

GRID\_SIZE = 20

# Define the 8 adjacency movements

MOVEMENTS = [

(0, 1),

(1, 1),

(1, 0),

(1, -1),

(0, -1),

(-1, -1),

(-1, 0),

(-1, 1)

]

# Define a class to represent a drone

class Drone:

def \_\_init\_\_(self, start, end, start\_time):

self.start = start

self.end = end

self.start\_time = start\_time

self.current\_position = start

self.path = None

self.completed = False

# Define a function to calculate the shortest path using A\* algorithm

def calculate\_shortest\_path(self):

# Implement A\* algorithm to find the shortest path

# Store the path in self.path

pass

# Define a function to move the drone to its next position

def move(self, time):

if self.path:

next\_position = self.path.pop(0)

if next\_position == self.end:

self.completed = True

else:

self.current\_position = next\_position

self.start\_time = time

# Define a function to check if two drones are at the same position and time

def check\_collision(self, other\_drone):

return self.current\_position == other\_drone.current\_position and self.start\_time == other\_drone.start\_time

# Define a function to initialize the drones and add them to a priority queue

def initialize\_drones(input\_list):

drones = []

for drone\_info in input\_list:

drone = Drone(drone\_info[0:2], drone\_info[2:4], drone\_info[4])

drone.calculate\_shortest\_path()

drones.append(drone)

heapq.heapify(drones, key=lambda x: x.start\_time)

return drones

# Define a function to check for collisions between drones and re-route if necessary

def check\_collisions\_and\_reroute(drones):

for i in range(len(drones)):

for j in range(i+1, len(drones)):

if drones[i].check\_collision(drones[j]):

# If there is a collision, re-route one or both drones to avoid it

drones[i].calculate\_shortest\_path()

drones[j].calculate\_shortest\_path()

# Define the main function to simulate the movement of the drones

def simulate\_drones(input\_list):

# Initialize the drones and add them to a priority queue

drones = initialize\_drones(input\_list)

# Initialize a dictionary to store the positions of the drones at each time step

positions = {}

# Initialize the time to 0

time = 0

# Loop until all drones have reached their destinations

while not all(drone.completed for drone in drones):

# Move the drone with the earliest start time

drone = heapq.heappop(drones)

drone.move(time)

positions[time] = positions.get(time, []) + [drone.current\_position]

# Check for collisions and re-route if necessary

check\_collisions\_and\_reroute(drones)

# If the drone has not reached its destination, add it back to the priority queue with its new start time

if not drone.completed:

heapq.heappush(drones, drone)

# Increment the time by 1

time += 1

# Print the final positions of the drones

for time, positions in positions.items():

print(f"Time {time}: {positions}")

#Simulate the movement of the drones

simulate\_drones(input\_list)

To enter dynamic inputs easily, we can create a GUI that allows the user to enter the number of drones and their starting positions, end positions, and start times. The GUI can also display the simulation of the drones moving towards their destinations. We can use a programming language like Python and a GUI library like Tkinter to create the GUI.

We can also use a pathfinding library like NetworkX or Pygame to implement the pathfinding algorithm. These libraries provide functions to calculate the shortest path between two points on a grid.

Overall, the approach taken involves using a pathfinding algorithm to find the shortest path for each drone, a priority queue to maintain the order in which the drones move, and collision detection to avoid collisions between the drones.