▪ 2024학년도 STEAM MAKER

2024. 창의융합(STEAM MAKER)탐구대회 프로젝트 결과보고서

드론 네비게이션 알고리즘 개발 및 연구

	학년/반/번호	연락처
대표학생(1)	[redacted]	
참가학생(2)		
참가학생(3)		
참가학생(4)		
참가학생(5)		
팀명	Excelsior	
컨설턴트 교사	(인)	

[redacted]

1. STEAM MAKER 프로젝트 개요

이 연구의 목적은 미래 모빌리티 산업이 지속적으로 성장하는 가운데, 많은 주목받고 있는 분야인 드론 산업에서 가장 효율적이고 안전한 알고리즘을 고안 해내기 위함이다. 이를 위해, 여러 가지 드론에 적용될 수 있는 알고리즘들을 살펴보고 각각 시뮬레이션을 통해 여러 값을 기준으로 비교하며 검증한다. 이를 통해, 찾아낸 알고리즘은 실제로 드론에 적용되면 높은 효율로 안정적인 드론 운영이 가능해질 것으로 전망된다.

2. 이론적 배경

이 연구에서 비교되는 각 알고리즘은 총 5가지인데, 각각 축 이동 알고리즘, 직선 이동 알고리즘, A* 알고리즘, D* 알고리즘, BFS 알고리즘이다.

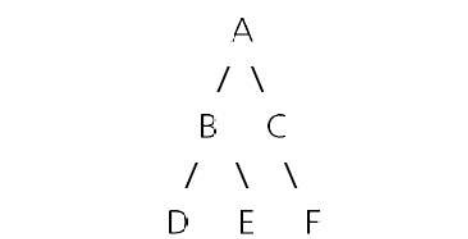
A* 알고리즘은 최단 경로를 찾는 알고리즘으로, 지금까지 걸린 비용 g(n)과 목표까지의 예상 비용 h(n)을 더해서 가장 비용이 적은 경로를 찾는 방식이다.

g(n): 시작 지점에서 현재 노드까지의 실제 거리 (이미 걸린 비용)
h(n): 현재 노드에서 목표 지점까지의 예상 거리 (추정 비용, 휴리스틱)
f(n) = g(n) + h(n): 전체 비용 (우선 순위를 결정하는 값)

f(n)이 가장 적은 경로가 최적의 경로이다.

D* 알고리즘은 A* 알고리즘은 기존에 변하지 않는 고정된 환경에서 적용되는 것과는 달리, 실시간으로 환경이 변하는 동적 환경에 적용되는 알고리즘이다. 이 알고리즘은 A* 알고리즘의 계산법은 그대로 적용되지만 이동 도중 새로운 장애물이 나타나거나 기존의 경로가 막히게 된다면 그 부분만 따로 업데이트하여, 계산하기 때문에 동적인 환경에 놓인 상황에서는 A* 알고리즘 보다 D* 알고리즘이 더 효율적일 것으로 예측된다.

마지막으로 BFS 알고리즘은 말을 풀어쓰면 너비 우선 탐색 알고리즘으로 불린다. 이 알고리즘은 트리에서 가장 가까운 노드부터 탐색하는 알고리즘이다. 이 알고리즘은 큐(Queue)라는 자료구조를 사용하며 선입선출(FIFO) 구조로 작동한다.



왼쪽의 그래프를 BFS로 탐색하면 아래의 표와 같은 결과가 나오게 된다. BFS 알고리즘은 전체 노드를 모두 탐색할 수 있다는 장점이 있지만, 큐에 매우 많은 노드가 저장되므로 메모리를 많이 사용하며 경로 계산 시간이 오래 걸린다는 단점 또한 존재한다.

단계	큐 상태	방문 순서
1	A	A
2	B, C	A, B
3	C, D, E	A, B, C
4	D, E, F	A, B, C, D
5	E, F	A, B, C, D, E
6	F	A, B, C, D, E, F

3. 연구 동기(및 목적)

https://www.hankookilbo.com/News/Read/A2020111616550000873

4년전 수성못에서 드론 택시 시현을 한 적이 있었다. 드론 택시 시연을 통해, 미래에 드론이 점점 일상화된다면 공중에서 발생할 수 있는 충돌 사고가 문제가 될 수 있겠다는 생각이 들었다. 여러 대의 드론이 동시에 비행하는 환경에서 어떻게 안전을 유지할 수 있을지에 대한 궁금증이 생겨, 여러 드론들을 안전하게 컨트롤할 수 있는 네비게이션 알고리즘을 연구하고자 이 프로젝트를 기획하였다.

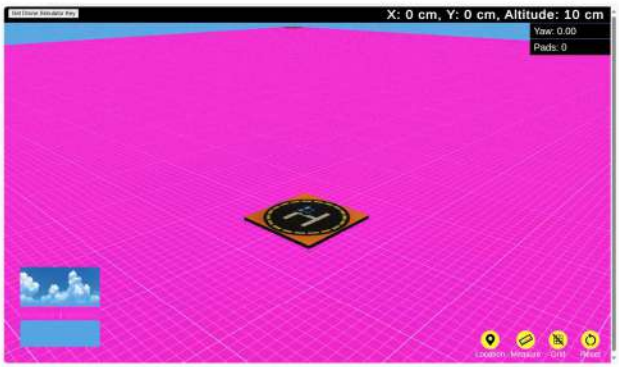
4. 연구일정

월	내 용	비 고
10/14	● 드론 비행 원리 및 기본 구조 이해 ● 드론 시뮬레이션 제작	
10/16	● 드론 시뮬레이션 제작 ● 길찾기 알고리즘 시뮬레이션 및 평가	
10/21	● 드론 세트장 제작 ● 드론 본체 제작	
10/23	● 드론 본체 제작 ● 드론 구동 코드 작성	
10/30	● 드론 알고리즘 시현 및 평가 ● 결과 보고서 작성	

5. 연구 과정

전체적인 연구 과정은 각 알고리즘들을 토대로 드론 운행 프로그래밍을 고딩하여 구현한 뒤, 3D 시뮬레이션과 이에 따라 나오는 이동 시간 및 경로 계산 시간과 장애물 회피 여부와 같은 데이터를 비교하여 가장 최적의 알고리즘을 찾는다. 이후 실제 드론에 코드를 적용시켜 본다.

이러한 드론의 시뮬레이션을 구현하기 위해 다양한 라이브러리를 찾아 보았을 때, 가장 적합하다고 생각하는 라이브러리는 DroneBlocksTelloSimulator 를 사용하였다. 이 라이브러리를 사용한 이유는 드론의 시뮬레이션 코드에서 드론을 움직이는 코드가 하단에 시술할 실물 드론 라이브러리에서 드론을 조종하는 부분과 가장 유사하였기 때문이며, 또한 드론의 이동 경로등과 같이 여러 가지 요소를 고려할 수 있었기 때문이다.



[그림1] 드론 시뮬레이션 실행 화면

이후 실물 드론을 조종하기 위해 프로그래밍이 가능한 CoDrone Mini 드론을 구매하였으며, 이 드론을 프로그래밍하여 조종하기 위해 e_drone 라이브러리를 사용하였다.

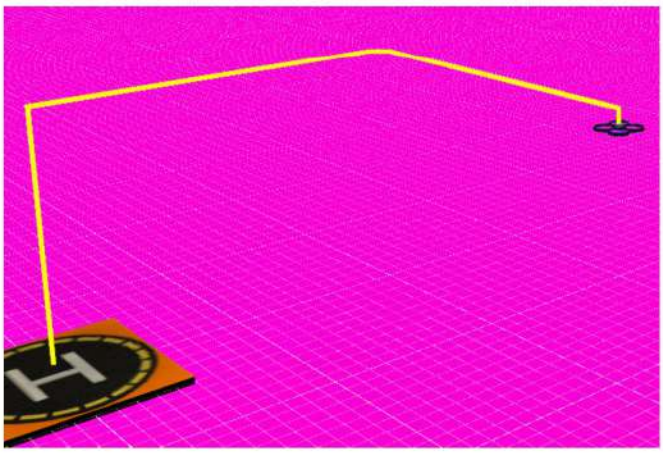
[그림2] CoDrone Mini

먼저 축 이동 알고리즘부터 구현해 보았다.

```
import time
```

```
From DroneBlocksTelloSimulator.DroneBlocksSimulatorContextManager     import DroneBlocksSimulatorContextManager,
SimulatedDrone
# https://coding-sim.droneblocks.io/
sim_key  = input("SIM 연결 키? ")
start_pos  = (0, 0, 10)      # X, Y, Altitude [cm]
goal_pos  = (200, 200, 75)   # X, Y, Z [cm]
lift_distance  = 10
distance  = 40
With DroneBlocksSimulatorContextManager(  simulator_key =sim_key ) as drone :
    drone : SimulatedDrone  = drone    # Typing
    # 시작 시간
    start = time.time()
    # 드론 상승
    drone.takeoff()
    drone.fly_to_xyz( 0, 0, goal_pos [2] + lift_distance , 'cm')
    drone.fly_to_xyz( 0, goal_pos [1], 0, 'cm')
    drone.fly_to_xyz( goal_pos [0], 0, 0, 'cm')
    drone.fly_to_xyz( 0, 0, lift_distance  * -1, 'cm')
    # 종료 시간
    end =time.time()
    # 경로 이동 시간
    print(f"경로 이동 시간: {round(end - start, 3)}seconds")
    # 착륙
    # 다만 경로 이동 시간 측정이므로 미필요
    # drone.land()
```

위의 코드가 축 이동 알고리즘을 바탕으로 구현된 코드이다. 이 알고리즘의 이름에서 알 수 있듯이 x, y, z축의 위아래로 이동한다. 이러한 축 이동 알고리즘을 사용하여 3D 시뮬레이션을 돌렸다.



[그림3] 축 이동 알고리즘 실행 결과

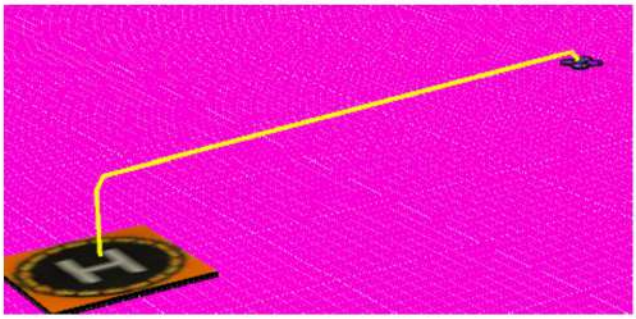
위의 사진이 시뮬레이션을 돌린 결과다. 이동 시간은 25.923초가 나왔다. 다만, 장애물을 회피할 수 없다는 점이 확인되었다. 또한 알고리즘이 매우 간단하기 때문에 경로를 계산하는 시간이 0초로 계산되었다.

다음으로 직선이동 알고리즘이다. 이 알고리즘은 5가지 알고리즘 중, 가장 간단한 알고리즘이다.

```
import time
from DroneBlocksTelloSimulator.DroneBlocksSimulatorContextManager     import DroneBlocksSimulatorContextManager,
SimulatedDrone
# https://coding-sim.droneblocks.io/
sim_key  = input("SIM 연결 키? ")
```

```
start_pos = (0, 0, 10)      # X, Y, Altitude [cm]
goal_pos = (200, 200, 75)   # X, Y, Z [cm]
lift_distance = 10
distance = 40
With DroneBlocksSimulatorContextManager(  simulator_key =sim_key ) as drone :
    drone : SimulatedDrone  = drone    # Typing
    # 시작 시간
    start = time.time()
    # 드론 상승
    drone.takeoff()
    drone.fly_to_xyz( goal_pos [0], goal_pos [1], goal_pos [2] + lift_distance , 'cm')
    drone.fly_to_xyz( 0, 0, lift_distance  * -1, 'cm')
    # 종료 시간
    end = time.time()
    # 경로 이동 시간
    print(f"경로 이동 시간: {round(end - start, 3)}seconds")
    # 착륙
    # 나만 경로 이동 시간 측정이므로 비필요
    # drone.land()
```

이 코드가 직선이동 알고리즘은 코드이다. 시작점부터 도착점까지 직진하는 알고리즘이다. 이러한 직선이동 알고리즘을 시뮬레이션 돌린 결과는 다음과 같다.



[그림4] 직선이동 알고리즘 실행 결과

이 알고리즘의 이름에서 알 수 있듯이 시작점부터 도착점까지 직진하므로 이동 시간은 5가지 알고리즘 중 가장 적은 수치인 17.024초가 나왔다.
하지만 직선이동 알고리즘은 축 이동 알고리즘과 같이 장애물 회피가 불가능하다. 또한 축 이동 알고리즘과 같이 알고리즘의 로직이 단순하기 때문에 경로 계산 시간이 0초가 나온 것 또한 확인할 수 있었다.

세 빈째 알고리즘은 A* 알고리즘이다. A* 알고리즘은 최단 경로를 찾는 알고리즘으로, 지금까지 걸린 비용 g(n)과 목표까지의 예상 비용 h(n)을 더해서 가장 비용이 적은 경로를 찾는 방식이다.

```
import time
import heapq
import math
from DroneBlocksTelloSimulator.DroneBlocksSimulatorContextManager     import DroneBlocksSimulatorContextManager,
SimulatedDrone

class AStar:
    def __init__(self, start, goal, buildings ):
        self.start = start
```

```
        self.goal = goal
        self.buildings = buildings
        self.open_list = []
        self.closed_list = set()
        self.came_from = {}
        self.g_score = {start: 0}
        self.f_score = {start: self.heuristic(start)}
        heapq.heappush(self.open_list, (self.f_score[start], start))
    def heuristic(self, current):
        # 유클리드 거리 (직선거리)를 휴리스틱으로 사용
        return math.dist(current, self.goal)
    def is_collision(self, x, y):
        # 건물의 크기를 기준으로 충돌 여부 판단
        for (bx, by, size) in self.buildings:
            # 건물은 (bx, by)를 중심으로 size 크기를 가지므로, 불가능 영역 계산
            half_size = size / 2
            if bx - half_size <= x <= bx + half_size and by - half_size <= y <= by + half_size:
                return True
        return False
    def neighbors(self, current):
        # 8방향으로 이동 (상, 하, 좌, 우, 대각선 포함)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1),   # 상, 하, 좌, 우
                      (-1, -1), (1, -1), (-1, 1), (1, 1)] # 대각선
        neighbors =[]
        for dx, dy in directions:
            nx, ny = current[0] + dx, current[1] + dy
            if not self.is_collision(nx, ny):
                neighbors.append((nx, ny))
        return neighbors
    def reconstruct_path(self):
        # 경로 복원
        path = []
        current = self.goal
        while current != self.start:
            path.append(current)
            current = self.came_from[current]
        path.append(self.start)
        path.reverse()
        return path
    def optimize_path(self, path):
        # 최적화: 경로 중간에 불필요한 점을 제거하여 직선 경로로 만듦
        optimized_path = [path[0]]
        for i in range(2, len(path)):
            prev, curr, next_point =path[i - 2], path[i - 1], path[i]
            # 직선 여부 확인 (세 점이 일직선 상에 있으면 현재 점을 건너뛰기)
            if(next_point[1] - prev[1]) * (curr[0] - prev[0]) == (curr[1] - prev[1]) * (next_point[0] - prev[0]):
                continue
            optimized_path.append(curr)
        optimized_path.append(path[-1])
        return optimized_path
    def find_path(self):
        while self.open_list:
            _, current = heapq.heappop(self.open_list)
            if current == self.goal:
                # 목표에 도달하면 경로 복원
                path = self.reconstruct_path()
                optimized_path = self.optimize_path(path)
                return optimized_path
            self.closed_list.add(current)
            for neighbor in self.neighbors(current):
                if neighbor in self.closed_list:
                    continue
                tentative_g_score = self.g_score[current] + 1  # 이동 비용 (상하좌우, 대각선은 1)
                if neighbor not in self.g_score or tentative_g_score < self.g_score[neighbor]:
```

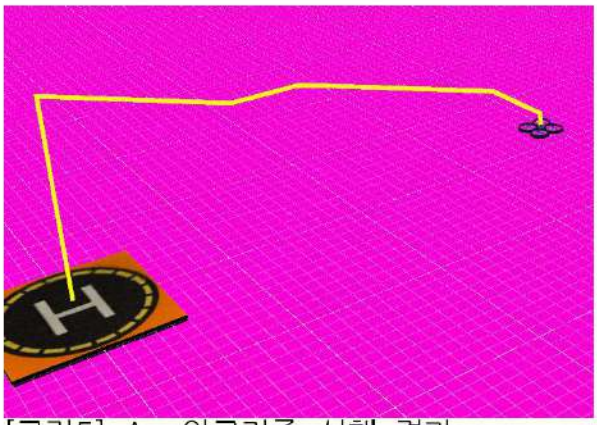
```
                    self.came_from[neighbor] = current
                    self.g_score[neighbor] = tentative_g_score
                    self.f_score[neighbor] = tentative_g_score  +self.heuristic(neighbor)
                    heapq.heappush(self.open_list, (self.f_score[neighbor], neighbor))
        return None  # 경로가 없으면 None 반환

# https://coding-sim.droneblocks.io/
sim_key = input("SIM 연결 키? ")
start_pos = (0, 0, 10)      # X, Y, Altitude [cm]
goal_pos = (200, 200, 75)   # X, Y, Z [cm]]
buildings = [
    (100, 100, 40)
]
lift_distance = 10

astar = AStar(start_pos[0:2], goal_pos[0:2], buildings)
paths = astar.find_path()

With DroneBlocksSimulatorContextManager(  simulator_key=sim_key) as drone:
    drone: SimulatedDrone =drone   # Typing
    # 시작 시간
    start = time.time()
    # 드론 상승
    drone.takeoff()
    drone.fly_to_xyz(0, 0, goal_pos[2] + lift_distance, 'cm')
    previous_path = start_pos[0:2]
    for path in paths:
        drone.fly_to_xyz(path[0] - previous_path[0], path[1] - previous_path[1], 0, 'cm')
        previous_path = path
    drone.fly_to_xyz(0, 0, lift_distance * -1, 'cm')
    # 종료 시간
    end =time.time()
    # 경로 이동 시간
    print(f"경로 이동 시간: {round(end - start, 3)}seconds")
    # 착륙
    # 다만 경로 이동 시간 측정이므로 미필요
    # drone.land()
```

이 긴 코드들이 A* 알고리즘의 코드이다. 지금까지 걸린 비용 g(n)과 목표까지의 예상 비용 h(n)을 더해야 하는데 이때, h(n) 즉, 휴리스틱 함수에 따라서 알고리즘의 효율이 정해진다. 이러한 휴리스틱 함수의 기준을 직선거리인 유클리드 거리도 정하여서 고드를 짰다. 다음이 이러한 A* 알고리즘을 시뮬레이션 돌린 결과이다.



[그림5] A* 알고리즘 실행 결과

이 알고리즘의 이동 시간은 29.630초로 지금까지 측정한 값들 중 가장 긴 값을 기

록했다. 하지만 앞은 축이동 알고리즘과 직선이동 알고리즘과는 달리 장애물 회피가 가능하고, 앞에서의 알고리즘 보다 더 복잡한 시간 복잡도를 가지기 때문에 경로 계산 시간 도 0.012초로 더 높게 측정 됐다.

그 다음으로는 D* 알고리즘을 구현하였는데, 이 알고리즘은 A*의 상위호환 버전으로 정적인 환경에서 가장 효율을 보이는 A* 알고리즘과는 달리 동적인 상황에서 높은 효율을 보여주기 때문에 다양한 환경에 적응할 수 있는 알고리즘이다.

```python
import time
import heapq
import math
from DroneBlocksTelloSimulator.DroneBlocksSimulatorContextManager    import DroneBlocksSimulatorContextManager,   SimulatedDrone

class DStar:
    def __init__(self, start, goal, buildings):
        self.start = start
        self.goal = goal
        self.buildings = buildings
        self.open_list = []
        self.closed_list = set()
        self.came_from = {}
        self.g_score = {start: 0}
        self.f_score = {start: self.heuristic(start)}
        self.km = 0  # 변경된 환경에 따른 인덱스 (k-parameter)
        heapq.heappush(self.open_list, (self.f_score[start], start))
    def heuristic(self, current):
        # 유클리드 거리 (직선거리)를 휴리스틱으로 사용
        return math.dist(current, self.goal)
    def is_collision(self, x, y):
        # 건물의 크기를 기준으로 충돌 여부 판단
        for (bx, by, size) in self.buildings:
            # 건물은 (bx, by)를 중심으로 size 크기를 가지므로, 불가능 영역 계산
            half_size = size / 2
            if bx - half_size <= x <= bx + half_size and by - half_size <= y <= by + half_size:
                return True
        return False
    def neighbors(self, current):
        # 8방향으로 이동 (상, 하, 좌, 우, 대각선 포함)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1),   # 상, 하, 좌, 우
                      (-1, -1), (1, -1), (-1, 1), (1, 1)]  # 대각선
        neighbors = []
        for dx, dy in directions:
            nx, ny = current[0] + dx, current[1] + dy
            if not self.is_collision(nx, ny):
                neighbors.append((nx, ny))
        return neighbors
    def reconstruct_path(self):
        # 경로 복원
        path = []
        current = self.goal
        while current != self.start:
            path.append(current)
            current = self.came_from[current]
        path.append(self.start)
        path.reverse()
        return path
    def optimize_path(self, path):
        # 최적화: 경로 중간에 불필요한 점을 제거하여 직선 경로로 만듦
        optimized_path = [path[0]]
        for i in range(2, len(path)):
            prev, curr, next_point = path[i - 2], path[i - 1], path[i]
            # 직선 여부 확인 (세 점이 일직선 상에 있으면 현재 점을 건너뛰기)
            if(next_point[1] - prev[1]) * (curr[0] - prev[0]) == (curr[1] - prev[1]) * (next_point[0] - prev[0]):
                continue
            optimized_path.append(curr)
        optimized_path.append(path[-1])
        return optimized_path
    def update(self, start, goal, buildings):
```

```
        # 장애물 변화가 있을 때, 새로운 환경으로 경로를 재계산
        self.start = start
        self.goal = goal
        self.buildings = buildings
        self.open_list = []
        self.closed_list = set()
        self.came_from = {}
        self.g_score = {start: 0}
        self.f_score = {start: self.heuristic(start)}
        self.km += 1
        heapq.heappush(self.open_list, (self.f_score[start], start))
    def find_path(self):
        while self.open_list:
            _, current = heapq.heappop(self.open_list)
            if current == self.goal:
                # 목표에 도달하면 경로 복원
                path = self.reconstruct_path()
                optimized_path = self.optimize_path(path)
                return optimized_path
            self.closed_list.add(current)
            for neighbor in self.neighbors(current):
                if neighbor in self.closed_list:
                    continue
                tentative_g_score = self.g_score[current] + 1  # 이동 비용 (상하좌우, 대각선은 1)
                if neighbor not in self.g_score or tentative_g_score < self.g_score[neighbor]:
                    self.came_from[neighbor] = current
                    self.g_score[neighbor] = tentative_g_score
                    self.f_score[neighbor] = tentative_g_score + self.heuristic(neighbor) + self.km
                    heapq.heappush(self.open_list, (self.f_score[neighbor], neighbor))
        return None  # 경로가 없으면 None 반환

# https://coding-sim.droneblocks.io/
sim_key = input("SIM 연결 키? ")
start_pos = (0, 0, 10)      # X, Y, Altitude [cm]
goal_pos = (200, 200, 75)   # X, Y, Z [cm]]
buildings = [
    (100, 100, 40)
]
lift_distance = 10
dstar = DStar(start_pos[0:2], goal_pos[0:2], buildings)
paths = dstar.find_path()
With DroneBlocksSimulatorContextManager(  simulator_key=sim_key) as drone:
    drone: SimulatedDrone = drone   # Typing
    # 시작 시간
    start = time.time()
    # 드론 상승
    drone.takeoff()
    drone.fly_to_xyz(0, 0, goal_pos[2] + lift_distance, 'cm')
    previous_path = start_pos[0:2]
    for path in paths:
        drone.fly_to_xyz(path[0] - previous_path[0], path[1] - previous_path[1], 0, 'cm')
        previous_path =path
    drone.fly_to_xyz(0, 0, lift_distance * -1, 'cm')
    # 종료 시간
    end =time.time()
    # 경로 이동 시간
    print(f"경로 이동 시간: {round(end - start, 3)}seconds")
    # 착륙
    # 다만 경로 이동 시간 측정이므로 미필요
    # drone.land()
```

위의 코드가 D* 알고리즘의 코드이다. D* 알고리즘은 A* 알고리즘의 최단 경로를 계산하는 방식은 그대로 가져왔지만, 동적인 상황에 대처하는 코드가 추가되었다. 그렇기 때문에 시뮬레이션을 돌린 결과는 A* 알고리즘과 같으며 이동 시간의 미묘한 차이만 관찰되었다. D* 알고리즘의 이동 시간은 29.629초로 A* 알고리즘과 비교했을 때, 0.001초 차이 난다. 따라서 거의 차이가 없는 것으로 확인되었다. A* 알고리즘과 마찬가지로 장애물 회피가 가능하며, 경로 계산 시간 또한 같은 계산법을 사용하기 때문에 0.012초로 A* 알고리즘과 같다.

마지막으로 BFS 알고리즘이다. 이 알고리즘은 너비 우선 탐색이라고도 하는데, 가장 가까운 노드부터 탐색하여, 그래프 전체를 탐색할 수 있는 알고리즘이다.

```
import time
from collections import deque
import math
from DroneBlocksTelloSimulator.DroneBlocksSimulatorContextManager import DroneBlocksSimulatorContextManager,
SimulatedDrone

class BFS:
    def __init__(self, start, goal, buildings):
        self.start = start
        self.goal = goal
        self.buildings = buildings
        self.visited = set()  # 방문한 노드
        self.came_from = {}  # 경로 복원용
        self.queue = deque([start])  # 탐색할 위치 큐
    def is_collision(self, x, y):
        # 건물의 크기를 기준으로 충돌 여부 판단
        For (bx, by, size) in self.buildings:
            # 건물은 (bx, by)를 중심으로 size 크기를 가지므로, 불가능 영역 계산
            half_size = size / 2
            if bx - half_size <= x <= bx + half_size and by - half_size <= y <= by + half_size:
                return True
        return False
    def neighbors(self, current):
        # 8방향 이동
        directions =[(-1, 0), (1, 0), (0, -1), (0, 1),   # 상, 하, 죄, 우
                     (-1, -1), (1, -1), (-1, 1), (1, 1)] # 대각선 방향
        neighbors = []
        for dx, dy in directions:
            nx, ny = current[0] + dx, current[1] + dy
            if not self.is_collision(nx, ny):
                neighbors.append((nx, ny))
        return neighbors
    def reconstruct_path(self):
        # 경로 복원
        path = []
        current = self.goal
        while current != self.start:
            path.append(current)
            current = self.came_from[current]
        path.append(self.start)
        path.reverse()
        return path
    def find_path(self):
        # BFS 탐색
        while self.queue:
            current = self.queue.popleft()
            if current == self.goal:
                # 목표에 도달하면 경로 복원
                path = self.reconstruct_path()
                return path
            for neighbor in self.neighbors(current):
                if neighbor not in self.visited:
                    self.visited.add(neighbor)
                    self.came_from[neighbor] = current
                    self.queue.append(neighbor)
        return None  # 경로가 없으면 None 반환
```

```python
# https://coding-sim.droneblocks.io/
sim_key = input("SIM 연결 키? ")
start_pos = (0, 0, 10)      # X, Y, Altitude [cm]
goal_pos = (200, 200, 75)   # X, Y, Z [cm]]
buildings = [
    (100, 100, 40)
]
lift_distance = 10
bfs = BFS(start_pos[0:2], goal_pos[0:2], buildings)
paths = bfs.find_path()

With DroneBlocksSimulatorContextManager(  simulator_key=sim_key) as drone:
    drone: SimulatedDrone = drone   # Typing
    # 시작 시간
    start = time.time()
    # 드론 상승
    drone.takeoff()
    drone.fly_to_xyz( 0, 0, goal_pos[ 2] + lift_distance,  'cm')
    previous_path = start_pos[ 0:2]
    for path in paths:
        drone.fly_to_xyz(path[ 0] - previous_path[ 0], path[1] - previous_path[ 1], 0, 'cm')
        previous_path = path
        drone.fly_to_xyz( 0, 0, lift_distance * -1, 'cm')

    # 종료 시간
    end = time.time()
    # 경로 이동 시간
    print(f"경로 이동 시간: {round(end - start, 3)}seconds")
    # 착륙
    # 다만 경로 이동 시간 측정이므로 미필요
    # drone.land()
```

이 코드가 BFS 알고리즘의 코드이다. 이 알고리즘은 큐(Queue)라는 선입선출(FIFO)이라는 로직을 가진 자료구조를 사용한다. 이 알고리즘에 따른 시뮬레이션 결과는 앞의 A* 알고리즘과 D* 알고리즘의 결과와 동일하게 나왔다. 장애물 회피 또한 가능했다. 하지만 차이가 난 것은 이동 시간과 경로 계산 시간이다. 이동시간은 29.559초로 A*, D* 알고리즘에 비해 빠르게 나왔다. 하지만 경로 계산 시간은 2.189초로 눈에 띄도록 느린 결과가 나왔다. 이에 대한 원인은 BFS 알고리즘의 특성에서 확인하였다. 위에서도 말했듯이, 이 알고리즘은 큐라는 자료구조를 사용하는데, 탐색하는 과정에서 큐의 메모리를 매우 많이 사용하기 때문에 다른 알고리즘들과는 확연히 다른 결과가 나온 것이다.

6. 연구 결과

	축이동	직선이동	A*	D*	BFS
이동 시간	25.923	17.024	29.630	29.629	29.559
장애물 회피 여부	X	X	O	O	O
경로 계산 시간	0.000	0.000	0.012	0.012	2.189

[표1] 알고리즘 별 비교

이러한 알고리즘들의 비교를 통해 적합한 알고리즘을 찾아보았다.

효율성 측면에서는 직선 이동이 가장 빠른 방법으로, 장애물이 없는 환경에서 가장 적합하며, 축 이동은 상대적으로 빠르긴 하지만 비효율적인 경로로 인해 한계가 있는 것으로 보였다. 장애물 회피를 위해 A*, D*, BFS 알고리즘을 활용해본 결과, 복잡한 환경에서도 모두 적합하다고 판단되었다. 특히 D* 알고리즘은 A*와 동일한 성능을 유지하면서도 동적 환경에서도 유용하여 더욱 신뢰할만 했다. 계산 성능에 있어서는 A*와 D*가 경로를 빠르게 계산할 수 있었던 반면, BFS는 최단 경로를 보장하기는 하지만 계산 비용이 다소 높다는 점이 확인되었다.

이러한 알고리즘들을 비교하였을 때 단순한 환경에서는 직선 이동 알고리즘이 최적이며, 복잡한 환경에서는 장애물 회피와 효율성을 위해 A* 알고리즘 또는 D* 알고리즘이 적합하다고 판단하였으며, 동적 환경에서는 D* 알고리즘, 시간에 민감하지 않은 환경에서는 BFS 가장 안전한 선택이라고 판단하였다.

이후 실제 드론에서 코드를 실행할 수 있도록 수정한 후 드론을 구동하였을 때 드론이 알고리즘의 경로를 따라 이동하기는 하였으나 일부 오차가 있다는 것을 확인하였다.

7. 주요 성과

이 연구는 드론의 경로 탐색 및 장애물 회피를 위해 축 이동, 직선 이동, A*, D*, BFS 알고리즘을 비교·분석하며, 이를 시뮬레이션과 실물 드론에 적용하는 과정을 통해 각 알고리즘의 성능을 정량적으로 검증하였다. 주요 성과로는 첫째, 각 알고리즘의 강점과 한계를 명확히 파악하여 적용 가능한 환경을 정의한 점, 둘째, A* 알고리즘, D* 알고리즘, BFS 알고리즘들의 장애물 회피 능력을 활용해 동적 환경에서의 최적 경로 탐색 가능성을 입증한 점, 셋째, 축 이동 및 직선 이동 알고리즘의 단순성을 기반으로 단시간 내 경로 이동이 필요한 상황에 대한 대안을 제시한 점 등이 있다. 총합적으로 알고리즘들의 경로 계산 시간과 이동 시간 간의 균형, 실시간 환경 변화 대응 능력, 메모리 효율성 등을 비교 및 분석하여 알고리즘들의 장단점을 분석할 수 있었다.

8. 총평 및 제언

이 연구를 통해, 미래 드론 네비게이션 기술의 적용 될 수 있는 길찾기 알고리즘들의 실제 구현 가능성을 판단할 수 있던 의미 있는 연구였다고 평가하며, 향후 연구에서는 더욱 복잡한 환경과 변수에 대응할 수 있는 다차원 알고리즘 통합 혹은 AI 기반 길찾기 알고리즘을 통합하여 더 나은 결과를 도출하는 것이나 최적화를 목표로 삼는 것이 바람직하다고 생각한다.

9. 각 학생별 소감(자신의 역할, 힘들었던 점과 극복한 방법, 성장한 부분 등)

[redacted] 내가 맡은 역할은 시뮬레이션으로써 작동하는 드론을 보고 입력값대로 작동하는지 확인하는 역할이다. 하지만 드론이 시뮬레이션과 달리 정확히 움직이지 않다는 문제를 찾았고, 이 문제를 드론 미세 조정으로 해결하였다. 이 과정에서 친구들이 만든 알고리즘과 코딩으로 제대로 작동하지 않는 부분을 보고 그 부분을 미세 조정하여 문제를 해결하였다. 이로써 드론 비행 원리와 기본 구조를 이해하였고 프로그래밍에 대한 이해도 상승하게 되었다.

[redacted] 내가 맡은 역할은 네비게이션 알고리즘에 해당하는 여러 알고리즘들을 찾아보았고, 이러한 알고리즘의 로직에 따라 코드를 작성하였다. 이 과정에서 기존의 알고리즘을 드론에 적용시키는 데, 복잡하고 어려웠다. 하지만, 많은 시간 투자와 오랜 고민 끝에 차근차근 코드를 완성시켰다. 특히, A* 알고리즘과 D* 알고리즘을 구현할 때, 휴리스틱 함수를 이해하고, 맞는 기준을 칮는데에는 드론에 대한 이해가 필요했기 때문에 더 많은 시간이 소요되었다. 오랜 시간 끝에 유클리드 거리라는 기준을 찾게 되었다. 이번 연구를 통해 여러 알고리즘에 대한 깊은 이해를 얻게 되었고, 각 알고리듬 중 가장 효율적인 알고리즘을 여러 대이터들을 비교하면서 찾는 법을 알게되었다. 이와 동시에, 깔끔한 코드를 작성하기 위해 노력한 결과 프로그래밍 실력도 상승하게 되었다.

[redacted] 이번 활동에서 나의 역할은 드론의 알고리즘을 보내주면 그알고리즘을 분석하여 어떤 알고리즘이 더나은지 평가하는 역할을 맡았고 드론 알고리즘의 장단점이 명확히 들어나있지 않아서 드론 알고리즘의 장단점 구분에 어려움을 겪었음 이러한 과정을 통해 장단점을 구분하는 방법 및 결과를 도출해는 방법에대한 성장을 하였다

[redacted] 드론 시뮬레이션 및 실물 드론의 알고리즘 코드를 작성하는 역할 및 팀장 역할을 맡았다. 드론 알고리즘 코드를 구현하는 중에 각각의 드론 알고리즘을 이해하는 과정에서 어려움을 겪었으나, 알고리즘에 대한 원리와 구조를 차근차근 이해함으로써 알고리즘에 대한 지식을 마련할 수 있었으며, 이러한 지식을 바탕으로 알고리즘을 드론 시뮬레이션 및 실물 드론에서 작동할 수 있도록 구현하였다. 또한 실물 드론이 작동하는 과정에서 발생한 연결 문제, 드론 기울어짐 문제등의 어려움이 있었으나, 이를 해결하기 위한 방빕을 찾아보고 계속 시도하여 봄으로써 이러한 문제들을 해결하였다. 이번 스팀 메이커 활동을 통해 복잡한 알고리즘을 프로그래밍으로 구현하는 과정에서 알고리즘을 이해하여 프로그래밍으로 구현하는 실력이 늘어났다고 생각한다. 또한 1학기 수학 수행평가에서 살펴보았던 A* 알고리즘에 대한 지식이 이번 활동에서 A* 알고리즘을 구현할 때에 도움이 되었으며, A* 알고리즘을 다른 알고리즘들과 비교해 볼때 바탕 지식이 되어 도움이 되었다고 생각한다.