Start coding or generate with AI.

ASSIGNMENT AI 1

```
# PART 1 ENVIRONMENT SETUP


import random

class Environment:
    def __init__(self, N, num_static_obstacles=10, num_dynamic_obstacles=5):
        self.N = N
        self.grid = [['.' for _ in range(N)] for _ in range(N)]
# grid setup
        self.weights = [[random.randint(1, 10) for _ in range(N)] for _ in range(N)]
        self.static_obstacles = []
        self.dynamic_obstacles = []
        self.agent_positions = {}

        self.place_static_obstacles(num_static_obstacles)
        self.place_dynamic_obstacles(num_dynamic_obstacles)

    def place_static_obstacles(self, num):
        """Place static obstacles randomly in the grid."""
        while len(self.static_obstacles) < num:
            x, y = random.randint(0, self.N - 1), random.randint(0, self.N - 1)
            if self.grid[x][y] == '.':
                self.grid[x][y] = 'S'
# static obstacle
                self.static_obstacles.append((x, y))

    def place_dynamic_obstacles(self, num):
        """Place dynamic obstacles randomly in the grid."""
        while len(self.dynamic_obstacles) < num:
            x, y = random.randint(0, self.N - 1), random.randint(0, self.N - 1)
            if self.grid[x][y] == '.':
                self.grid[x][y] = 'D'
# dynamic obstacle
                self.dynamic_obstacles.append({'position': (x, y), 'movement_pattern': self.create_movement_pattern(x, y)})

def create_movement_pattern(self, x, y):
        """Generate a deterministic movement pattern for dynamic obstacles."""
# Oscillating movement between two fixed points
        dx, dy = random.choice([(1, 0), (-1, 0), (0, 1), (0, -1)])
# choose a movement direction
        return [(x, y), (x + dx, y + dy)]

    def move_dynamic_obstacles(self):
        """Move the dynamic obstacles according to their predefined patterns."""
        for obstacle in self.dynamic_obstacles:
            current_position = obstacle['position']
            movement_pattern = obstacle['movement_pattern']
# Switch between two points
            next_position = movement_pattern[0] if current_position == movement_pattern[1] else movement_pattern[1]
            if self.grid[next_position[0]][next_position[1]] == '.':
# Ensure the new position is traversable
                self.grid[current_position[0]][current_position[1]] = '.'
# Free the previous cell
                self.grid[next_position[0]][next_position[1]] = 'D'
# Mark the new cell as a dynamic obstacle
                obstacle['position'] = next_position

    def find_valid_position(self):
        """Find a valid unoccupied position for agents."""
        while True:
            x, y = random.randint(0, self.N - 1), random.randint(0, self.N - 1)
            if self.grid[x][y] == '.':
# check for  blocked
                return (x, y)

    def place_agent(self, agent_id, start=None, goal=None):
        """Assign starting and goal positions for agents."""
        if start is None:
            start = self.find_valid_position()
# find valid start if not provided
```

```
        if goal is None:
            goal = self.find_valid_position()
# find valid goal if not provided

# check start and goal positions are not blocked
        if self.grid[start[0]][start[1]] == '.' and self.grid[goal[0]][goal[1]] == '.':
            self.agent_positions[agent_id] = {'start': start, 'goal': goal}
            self.grid[start[0]][start[1]] = f'A{agent_id}'
# Mark the agent's starting position
        else:
            raise ValueError("Start or goal position is blocked by an obstacle.")

    def print_grid(self):
        """Print the current state of the grid."""
        for row in self.grid:
            print(' '.join(row))

# Create the environment
N = 10
env = Environment(N)

# Place agents in the environment
try:
    env.place_agent(1, (0, 0), (9, 9))
# Custom start and goal
    env.place_agent(2)
#  generate valid start and goal for agent 2
except ValueError as e:
    print(f"Error: {e}")

# Print the initial grid
print("Initial grid setup:")
env.print_grid()

for step in range(3):
# Simulate 3 time steps
    print(f"\nTime step {step + 1}:")
    env.move_dynamic_obstacles()
# Move dynamic obstacles
    env.print_grid()
# Print updated grid
```

```
Error: Start or goal position is blocked by an obstacle.
Initial grid setup:
S . . . . . . . . .
. . . . . . . . . .
. . D . . . . . S .
. . . . . . . . . .
. . . . . . . . . .
. S . . . . S . D S
. S . . . . . . S .
D . S . . S . . . .
. . S . . . . D . .
. . . . . . . . . D

Time step 1:
S . . . . . . . . .
. . . . . . . . . .
. D . . . . . . S .
. . . . . . . . . .
. . . . . . . . . .
. S . . . . S . D S
D S . . . . . . S .
. . S . . S . . . .
. . S . . . . . D D
. . . . . . . . . .

Time step 2:
S . . . . . . . . .
. . . . . . . . . .
. . D . . . . . S .
. . . . . . . . . .
. . . . . . . . . .
. S . . . . S . D S
. S . . . . . . S .
D . S . . S . . . .
. . S . . . . D . .
. . . . . . . . . D
```

```
Time step 3:
S . . . . . . . . .
. . . . . . . . . .
. D . . . . . . S .
. . . . . . . . . .
. . . . . . . . . .
. S . . . . S . D S
D S . . . . . . S .
. . S . . S . . . .
. . S . . . . . D D
. . . . . . . . . .
```

Start coding or generate with AI.

Part 2: Single-Agent Navigation

```
# DFS
def dfs(grid, start, goal):
    stack = [(start, [])]
# Stack holds the current position and the path taken so far
    visited = set()

    while stack:
        (x, y), path = stack.pop()

        if (x, y) == goal:
            return path + [(x, y)]

        if (x, y) not in visited:
            visited.add((x, y))
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
# Up, Down, Left, Right
                nx, ny = x + dx, y + dy
                if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]) and grid[nx][ny] != 'S' and grid[nx][ny] != 'D':
                    stack.append(((nx, ny), path + [(x, y)]))

    return None
```

BFS

```
from collections import deque

def bfs(grid, start, goal):
    queue = deque([(start, [])])
    visited = set()

    while queue:
        (x, y), path = queue.popleft()

        if (x, y) == goal:
            return path + [(x, y)]

        if (x, y) not in visited:
            visited.add((x, y))
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nx, ny = x + dx, y + dy
                if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]) and grid[nx][ny] != 'S' and grid[nx][ny] != 'D':
                    queue.append(((nx, ny), path + [(x, y)]))

    return None
```

UFS

```
import heapq

def ufs(grid, weights, start, goal):
    pq = [(0, start, [])]
# Priority queue (min-heap) stores (cost, current_position, path)
    visited = set()
    while pq:
        cost, (x, y), path = heapq.heappop(pq)

        if (x, y) == goal:
            return path + [(x, y)], cost

        if (x, y) not in visited:
            visited.add((x, y))
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nx, ny = x + dx, y + dy
                if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]) and grid[nx][ny] != 'S' and grid[nx][ny] != 'D':
                    heapq.heappush(pq, (cost + weights[nx][ny], (nx, ny), path + [(x, y)]))

    return None
```

A* Search Algorithm

```
def heuristic(a, b):

    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def a_star(grid, weights, start, goal):
    pq = [(0 + heuristic(start, goal), 0, start, [])]
    visited = set()

    while pq:
        f, g, (x, y), path = heapq.heappop(pq)

        if (x, y) == goal:
            return path + [(x, y)], g

        if (x, y) not in visited:
            visited.add((x, y))
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nx, ny = x + dx, y + dy
                if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]) and grid[nx][ny] != 'S' and grid[nx][ny] != 'D':
                    h = heuristic((nx, ny), goal)
                    heapq.heappush(pq, (g + weights[nx][ny] + h, g + weights[nx][ny], (nx, ny), path + [(x, y)]))

    return None
# No path found
```

Dynamic and Weighted Pathfinding

```
def replan_path(search_algorithm, grid, weights, start, goal):
# Check for changes in the environment
    if grid_has_changed(grid):
# Some custom condition to detect changes (moving obstacles)
        return search_algorithm(grid, weights, start, goal)
    return None

def grid_has_changed(grid):
# This function checks if dynamic obstacles have moved or if any weights have changed.
# For simplicity, assume this returns True if any dynamic obstacle has moved.
    return True
# This is where you'd put actual logic to detect changes
```

Comparison and Evaluation

```
import time

def compare_algorithms(grid, weights, start, goal):
    algorithms = {
        'DFS': dfs,
        'BFS': bfs,
        'UFS': ufs,
        'A*': a_star
    }
    results = {}
    for name, algorithm in algorithms.items():
        start_time = time.time()
        if name in ['UFS', 'A*']:
            path, cost = algorithm(grid, weights, start, goal)
        else:
            path = algorithm(grid, start, goal)
            cost = len(path)
# Path length can be used as a simple cost measure
        end_time = time.time()

        if path:
            results[name] = {
                'time': end_time - start_time,
                'cost': cost,
                'path': path,
            }
        else:
            results[name] = {'time': float('inf'), 'cost': float('inf'), 'path': None}

    return results

# Example usage
start = (0, 0)
goal = (9, 9)
results = compare_algorithms(env.grid, env.weights, start, goal)

# Print out the comparison results
for algo_name, result in results.items():
    print(f"Algorithm: {algo_name}, Time: {result['time']:.4f} sec, Cost: {result['cost']}, Path Length: {len(result['path']) if result['pat
```

```
Algorithm: DFS, Time: 0.0006 sec, Cost: 53, Path Length: 53
Algorithm: BFS, Time: 0.0003 sec, Cost: 19, Path Length: 19
Algorithm: UFS, Time: 0.0004 sec, Cost: 70, Path Length: 19
Algorithm: A*, Time: 0.0005 sec, Cost: 70, Path Length: 19
```

Part 3: Multi-Agent Pathfinding

Start coding or generate with AI.

```
#Step 1: Cooperative Agents


import heapq

class CooperativeAStar:
    def __init__(self, grid, weights, agents):
        self.grid = grid
        self.weights = weights
        self.agents = agents
# List of (start, goal) tuples for each agent

    def heuristic(self, a, b):
        return abs(a[0] - b[0]) + abs(a[1] - b[1])
# Manhattan distance

    def a_star(self, start, goal, other_agents_paths):
        pq = [(0 + self.heuristic(start, goal), 0, start, [])]
        visited = set()

        while pq:
            f, g, (x, y), path = heapq.heappop(pq)

            if (x, y) == goal:
                return path + [(x, y)]

            if (x, y) not in visited:
                visited.add((x, y))

                for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                    nx, ny = x + dx, y + dy

# Check if the cell is valid and not an obstacle
                    if 0 <= nx < len(self.grid) and 0 <= ny < len(self.grid[0]) and self.grid[nx][ny] != 'S' and self.grid[nx][ny] != 'D':
                        if self.conflict_free((nx, ny), g + 1, other_agents_paths):
                            heapq.heappush(pq, (g + self.weights[nx][ny] + self.heuristic((nx, ny), goal), g + self.weights[nx][ny], (nx, ny

        return None
# No path found

    def conflict_free(self, position, time, other_agents_paths):
        for agent_path in other_agents_paths:
            if time < len(agent_path) and agent_path[time] == position:
                return False
        return True

    def cooperative_plan(self):
        paths = []
        for i, (start, goal) in enumerate(self.agents):
            other_agents_paths = [paths[j] for j in range(len(paths))]
# Paths of agents before this one
            path = self.a_star(start, goal, other_agents_paths)
            if path:
                paths.append(path)
            else:
                print(f"Agent {i} could not find a path")
                return None
# Return None if any agent cannot find a path
        return paths
```

Step 2: Competitive Agents

```
class CompetitiveAStar(CooperativeAStar):
    def competitive_plan(self):
        paths = []
        priorities = list(range(len(self.agents)))
# Prioritize agents by their order
        for i in priorities:
            start, goal = self.agents[i]
            other_agents_paths = [paths[j] for j in range(len(paths))]
# Previous agents' paths
            path = self.a_star(start, goal, other_agents_paths)
            if path:
                paths.append(path)
            else:
                print(f"Agent {i} could not find a path")
                return None
# If any agent can't find a path, return None
        return paths
```

Conflict Resolution Example:

```
def resolve_conflicts(agent_paths):
# Resolve conflicts when agents want to move into the same space
    resolved_paths = []
    for t in range(max(len(path) for path in agent_paths)):
        positions_at_t = [path[t] if t < len(path) else path[-1] for path in agent_paths]
# Agents' positions at time t
        if len(set(positions_at_t)) != len(positions_at_t):
            for i in range(len(agent_paths)):
                if positions_at_t.count(positions_at_t[i]) > 1:
                    print(f"Conflict detected at time {t} for agent {i}, replanning...")
        resolved_paths.append(positions_at_t)
    return resolved_paths
```

Step 3: Multi-Agent Planning Algorithms

```
class Environment:
    def __init__(self, N, weights, obstacles):
        self.N = N
        self.grid = [['.' for _ in range(N)] for _ in range(N)]
        self.weights = weights
        self.static_obstacles = obstacles
        self.dynamic_obstacles = []
        self.agents = {}

# Place static obstacles
        self.place_static_obstacles()

    def place_static_obstacles(self):
        """Place predefined static obstacles in the grid."""
        for x, y in self.static_obstacles:
            if self.grid[x][y] == '.':
                self.grid[x][y] = 'X'
# Mark as obstacle



# Define grid size and other required parameters
grid_size = 10
# Example grid size 10x10
weights = [[1 for _ in range(grid_size)] for _ in range(grid_size)]
# Set all weights to 1
obstacles = [(2, 3), (4, 5), (7, 8)]

agents = [((0, 0), (9, 9)), ((1, 1), (8, 8)), ((2, 2), (7, 7))]
# Example agent start and goal positions

# Initialize the environment and agents
env = Environment(grid_size, weights, obstacles)
cooperative_agents = CooperativeAStar(env.grid, env.weights, agents)
competitive_agents = CompetitiveAStar(env.grid, env.weights, agents)

# Run the evaluation function
evaluate_multi_agent_systems(env, cooperative_agents, competitive_agents)
```

```
Evaluating Cooperative Agents...
Cooperative agents successfully planned paths!

Evaluating Competitive Agents...
Competitive agents successfully planned paths!
Cooperative agents time: 0.0077 seconds
Competitive agents time: 0.0064 seconds
```

Part 4: Advanced Requirements (Optional)

```
#1. Time-Dependent Heuristics


# f(n) = g(n) + h(n)
# Where g(n) is the cost to reach node n, and h(n) is the heuristic estimation

# Time-dependent version of the heuristic
def time_dependent_heuristic(node, goal, current_time):
    distance = abs(goal[0] - node[0]) + abs(goal[1] - node[1])

# Adjust based on time
    if current_time % 10 == 0:
# Every 10 time steps, for example
        return distance * 1.5
# Increase the heuristic value to simulate increased difficulty
    else:
        return distance
# Normal heuristic
```

2. Hierarchical Search

```
class HierarchicalSearch:
    def __init__(self, grid, sub_grid_size):
        self.grid = grid
        self.sub_grid_size = sub_grid_size
        self.num_regions = len(grid) // sub_grid_size

    def high_level_plan(self, start, goal):
        """Plan the route at a high-level across regions."""
        start_region = self.get_region(start)
        goal_region = self.get_region(goal)
# Use A* or BFS for high-level region-to-region search
        return self.a_star_region_search(start_region, goal_region)

    def get_region(self, position):
        """Return the region (sub-grid) for a given position."""
        return (position[0] // self.sub_grid_size, position[1] // self.sub_grid_size)

    def a_star_region_search(self, start_region, goal_region):
        """Perform region-level A* search between regions."""
# Implement A* to find the optimal path between regions
        pass

    def refine_path(self, high_level_path):
        """Refine the path within each region."""
        detailed_path = []
        for region in high_level_path:
# Perform fine-grained search within the region
            detailed_path += self.a_star_within_region(region)
        return detailed_path

    def a_star_within_region(self, region):
        """Perform A* search within a sub-grid region."""
# Use A* to navigate within the sub-grid
        pass
```

3. Meta-Agent Coordination

```
class MetaAgent:
    def __init__(self, agents, environment):
        self.agents = agents
        self.environment = environment
        self.agent_paths = {}
# Store planned paths for all agents

    def coordinate_agents(self):
        """Coordinate the movements of all agents."""
        for agent in self.agents:
            planned_path = self.plan_path(agent)
            self.agent_paths[agent] = planned_path

    def plan_path(self, agent):
        """Plan an optimal path for the agent."""
# Use A* or BFS to plan the path, considering other agents
        start, goal = agent.start, agent.goal
        return self.a_star_with_agent_avoidance(agent, start, goal)

    def a_star_with_agent_avoidance(self, agent, start, goal):
        """Implement A* considering other agents' paths to avoid collisions."""
# Implement A*
        pass

    def resolve_conflicts(self):
        """Resolve any conflicts between agents' paths."""
        for agent, path in self.agent_paths.items():
            for step in path:
# Check if any other agent is occupying the same position at the same time
                self.avoid_conflict(agent, step)

    def avoid_conflict(self, agent, step):
        """Recalculate paths if a conflict is detected."""
# Implement conflict avoidance
        pass
```