Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.

```
iport heapq
iport random
iport time
om collections import deque
om typing import List, Tuple, Dict, Optional

 Constants for obstacles
ATIC_OBSTACLE = -1
NAMIC_OBSTACLE = -2

 Environment model
ass CityMap:
  def __init__(self, width: int, height: int, terrain_costs: List[List[int]], static_obstacles: List[Tuple[int, int]],
               dynamic_obstacles: List[Tuple[int, int]], dynamic_pattern: Optional[List[Tuple[int, int]]] = None):
      self.width = width
      self.height = height
      self.initial_terrain_costs = terrain_costs # Store initial terrain costs
      self.grid = [[terrain_costs[y][x] for x in range(width)] for y in range(height)]
      self.static_obstacles = set(static_obstacles)
      self.dynamic_obstacles = set(dynamic_obstacles)
      self.dynamic_pattern = dynamic_pattern  # Pattern for dynamic obstacles

      for (x, y) in static_obstacles:
          self.grid[y][x] = STATIC_OBSTACLE

      # Initial placement of dynamic obstacles
      for (x, y) in dynamic_obstacles:
          self.grid[y][x] = DYNAMIC_OBSTACLE

  def update_dynamic_obstacles(self, step: int):
      # Clear old dynamic obstacles, restoring original terrain costs
      for y in range(self.height):
          for x in range(self.width):
              if self.grid[y][x] == DYNAMIC_OBSTACLE:
                  self.grid[y][x] = self.initial_terrain_costs[y][x]

      # Update dynamic obstacles positions based on the pattern and step
      if self.dynamic_pattern and self.dynamic_obstacles:
          # Assuming dynamic_obstacles is a set of initial positions, and the pattern applies to each
          new_positions = set()
          for i, initial_pos in enumerate(list(self.dynamic_obstacles)): # Convert to list to access by index if needed
               # Example: simple cyclic movement for each obstacle based on the pattern
               # This assumes the pattern is for a single obstacle or a list of patterns
               # For multiple obstacles with independent patterns, a more complex structure is needed
               # For this example, let's assume a single pattern applies to all dynamic obstacles
               # A better approach would be to store individual obstacle patterns
               # Let's simplify and just use the provided pattern for visualization/example
               # A realistic scenario would require a pattern per obstacle
               # For demonstration, let's just shift the *set* of obstacles based on the pattern length
               # This is not truly deterministic movement of individual obstacles
               # A better model would track each dynamic obstacle's position and pattern
               # Let's revert to a simpler model where dynamic_obstacles are just locations that change
               pass # We will handle dynamic obstacles in the search functions if time-aware

      # Re-place dynamic obstacles for the current step (if not time-aware search)
      # If search is time-aware, dynamic obstacle positions are checked based on current time/step
      # This method is more for visualizing the map at a certain step if needed outside search
      # For search, the check `is_passable` should consider the time step.
      # We will modify the search functions to be time-aware.
      pass


  def is_passable(self, x: int, y: int, current_time: int = 0) -> bool:
      if not (0 <= x < self.width and 0 <= y < self.height):
          return False
      if self.grid[y][x] == STATIC_OBSTACLE:
          return False

      # Check for dynamic obstacles at the current time step
      if self.dynamic_pattern and self.dynamic_obstacles:
           # This part needs a more sophisticated model if dynamic obstacles have individual patterns
           # For a simple example, let's assume the dynamic_pattern describes the positions of ALL dynamic obstacles at each ti
           if current_time < len(self.dynamic_pattern):
               dynamic_obstacle_positions_at_time = set(self.dynamic_pattern[current_time]) # Assuming dynamic_pattern is a lis
               if (x, y) in dynamic_obstacle_positions_at_time:
                   return False
           # If dynamic_pattern is a list of *movements* for each obstacle, this logic needs to change
           # Let's assume dynamic_pattern is a list of positions for a *single* dynamic obstacle for simplicity in this example
           # If there are multiple dynamic obstacles, dynamic_pattern would need to be more complex (e.g., list of lists of pos
           # For the current structure, let's assume dynamic_pattern is a list of positions a *single* dynamic obstacle follows
```

```
            # We need to adapt this if dynamic_obstacles list in __init__ has multiple items.

            # Let's refine the dynamic obstacle handling: assume dynamic_obstacles in __init__ are *initial* positions.
            # The dynamic_pattern describes the sequence of positions for *each* dynamic obstacle.
            # This requires dynamic_pattern to be a list of patterns, one for each initial dynamic obstacle.
            # Example: dynamic_obstacles = [(5,5), (6,6)], dynamic_pattern = [[(5,5), (5,6)], [(6,6), (6,5)]]
            # This is getting complex. Let's simplify for this modification.
            # Let's assume a single dynamic obstacle for now, and dynamic_obstacles in __init__ has only one element.
            # And dynamic_pattern is the sequence of positions for this single obstacle.
            if len(self.dynamic_obstacles) == 1 and self.dynamic_pattern:
                dynamic_obstacle_initial_pos = list(self.dynamic_obstacles)[0]
                # Find which dynamic obstacle this is if there were multiple
                try:
                    obstacle_index = list(self.dynamic_obstacles).index(dynamic_obstacle_initial_pos)
                    if current_time < len(self.dynamic_pattern):
                         # Assuming dynamic_pattern is a list of positions for the *first* dynamic obstacle
                         # This is a simplification due to the current data structure
                         # A proper implementation would need dynamic_pattern to be indexed by obstacle
                         if (x, y) == self.dynamic_pattern[current_time % len(self.dynamic_pattern)]:
                             return False
                except ValueError:
                    pass # Should not happen if dynamic_obstacles is initialized correctly

        # Reverting to a simpler check based on the grid state, which is updated by update_dynamic_obstacles
        # This means update_dynamic_obstacles needs to be called before checking passability at a given step
        # However, for time-aware search, the search algorithm itself needs to calculate the dynamic obstacle positions at each t
        # Let's modify is_passable to take the current time and calculate dynamic obstacle positions on the fly.

        if self.dynamic_pattern and self.dynamic_obstacles:
            # Assuming dynamic_obstacles are initial positions and dynamic_pattern is a single pattern applied cyclically to the
            # This is still not ideal for individual obstacle tracking but fits the current structure better.
            pattern_length = len(self.dynamic_pattern)
            current_pattern_index = current_time % pattern_length
            # Calculate the offset from the initial position based on the pattern
            # This assumes dynamic_pattern contains *positions*, not *movements*
            # If dynamic_pattern has length P, at time T, an obstacle initially at (x0, y0) will be at dynamic_pattern[T % P]
            # This doesn't make sense if dynamic_obstacles has multiple elements.

            # Let's assume dynamic_pattern is a list of positions for a *single* dynamic obstacle.
            # If there are multiple dynamic obstacles, we need a list of patterns.
            # Given the current dynamic_obstacles is a set of tuples, and dynamic_pattern is a list of tuples,
            # The most reasonable interpretation is that dynamic_pattern describes the movement of a *single* point,
            # and we are checking if the current cell (x, y) is occupied by a dynamic obstacle at time `current_time`.
            # This implies all dynamic obstacles move together following the same pattern, or dynamic_obstacles in __init__ only
            # Let's assume the latter for now for simplicity.
            if len(self.dynamic_obstacles) == 1 and self.dynamic_pattern:
                 if current_time < len(self.dynamic_pattern):
                     if (x, y) == self.dynamic_pattern[current_time]:
                         return False
                 else:
                     # If current_time exceeds pattern length, assume the obstacle stays at the last pattern position or disappea
                     # Let's assume it stays at the last position for simplicity.
                     if (x, y) == self.dynamic_pattern[-1]:
                         return False


        # Reverting to the original check which relies on the grid being updated
        # This means for time-aware search, we need to generate a grid for each time step or modify the search to use the time-aw
        # Let's modify the search functions to be time-aware and use the is_passable with current_time
        return self.grid[y][x] != DYNAMIC_OBSTACLE



    def neighbors(self, x: int, y: int, current_time: int = 0) -> List[Tuple[int, int]]:
        result = []
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:  # 4-way connectivity
            nx, ny = x + dx, y + dy
            # Pass the current time to is_passable
            if self.is_passable(nx, ny, current_time + 1): # Assume moving to neighbor takes 1 time step
                result.append((nx, ny))
        return result

 Uninformed search: BFS (Time-aware)
ef bfs_time_aware(city_map: CityMap, start: Tuple[int, int], goal: Tuple[int, int], time_horizon: int = 100) -> Tuple[List[Tuple
    # State in BFS will be (x, y, time)
    queue = deque([(start[0], start[1], 0)]) # (x, y, time)
    came_from = {(start[0], start[1], 0): None}
    cost_so_far = {(start[0], start[1], 0): 0} # Cost is time for BFS
    nodes_expanded = 0

    while queue:
        current_state = queue.popleft() # (x, y, time)
        current_pos = (current_state[0], current_state[1])
        current_time = current_state[2]
```

```
        nodes_expanded += 1

        if current_pos == goal:
            path = []
            current = current_state
            while current:
                path.append((current[0], current[1])) # Append only position to path
                current = came_from[current]
            path.reverse()
            return path, nodes_expanded, current_time # Cost is the time taken

        if current_time >= time_horizon:
            continue # Avoid infinite loops in dynamic environments

        # Get neighbors for the next time step
        for neighbor_pos in city_map.neighbors(*current_pos, current_time):
            neighbor_state = (neighbor_pos[0], neighbor_pos[1], current_time + 1)
            if neighbor_state not in came_from:
                came_from[neighbor_state] = current_state
                cost_so_far[neighbor_state] = current_time + 1
                queue.append(neighbor_state)

    return [], nodes_expanded, 0  # no path found

 Uninformed search: Uniform Cost Search (Time-aware)
ef uniform_cost_search_time_aware(city_map: CityMap, start: Tuple[int, int], goal: Tuple[int, int], time_horizon: int = 100) ->
    # State in UCS will be (cost, x, y, time)
    pq = []
    heapq.heappush(pq, (0, start[0], start[1], 0)) # (cost, x, y, time)
    came_from = {(start[0], start[1], 0): None}
    cost_so_far = {(start[0], start[1], 0): 0}
    nodes_expanded = 0

    while pq:
        current_cost, current_x, current_y, current_time = heapq.heappop(pq)
        current_state = (current_x, current_y, current_time)
        current_pos = (current_x, current_y)
        nodes_expanded += 1

        if current_pos == goal:
            path = []
            current = current_state
            while current:
                path.append((current[0], current[1])) # Append only position to path
                current = came_from[current]
            path.reverse()
            return path, nodes_expanded, current_cost

        if current_time >= time_horizon:
            continue # Avoid infinite loops in dynamic environments


        for neighbor_pos in city_map.neighbors(*current_pos, current_time):
            neighbor_state = (neighbor_pos[0], neighbor_pos[1], current_time + 1)
            terrain_cost = city_map.initial_terrain_costs[neighbor_pos[1]][neighbor_pos[0]]
            new_cost = cost_so_far[current_state] + terrain_cost # Cost includes terrain

            if neighbor_state not in cost_so_far or new_cost < cost_so_far[neighbor_state]:
                cost_so_far[neighbor_state] = new_cost
                priority = new_cost
                heapq.heappush(pq, (priority, neighbor_state[0], neighbor_state[1], neighbor_state[2]))
                came_from[neighbor_state] = current_state

    return [], nodes_expanded, 0

 Heuristic for A* (Manhattan distance) - remains the same
ef heuristic(a: Tuple[int, int], b: Tuple[int, int]) -> float:
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

 Informed search: A* (Time-aware)
ef a_star_time_aware(city_map: CityMap, start: Tuple[int, int], goal: Tuple[int, int], time_horizon: int = 100) -> Tuple[List[Tu
    # State in A* will be (priority, x, y, time)
    pq = []
    heapq.heappush(pq, (0, start[0], start[1], 0)) # (priority, x, y, time)
    came_from = {(start[0], start[1], 0): None}
    cost_so_far = {(start[0], start[1], 0): 0}
    nodes_expanded = 0

    while pq:
        current_priority, current_x, current_y, current_time = heapq.heappop(pq)
        current_state = (current_x, current_y, current_time)
        current_pos = (current_x, current_y)
        nodes_expanded += 1
```

```
        if current_pos == goal:
            path = []
            current = current_state
            while current:
                path.append((current[0], current[1])) # Append only position to path
                current = came_from[current]
            path.reverse()
            return path, nodes_expanded, cost_so_far[current_state] # Cost is the accumulated terrain cost


        if current_time >= time_horizon:
            continue # Avoid infinite loops in dynamic environments


        for neighbor_pos in city_map.neighbors(*current_pos, current_time):
            neighbor_state = (neighbor_pos[0], neighbor_pos[1], current_time + 1)
            terrain_cost = city_map.initial_terrain_costs[neighbor_pos[1]][neighbor_pos[0]]
            new_cost = cost_so_far[current_state] + terrain_cost # Cost includes terrain

            if neighbor_state not in cost_so_far or new_cost < cost_so_far[neighbor_state]:
                cost_so_far[neighbor_state] = new_cost
                # Priority is cost so far + heuristic to goal position (heuristic is not time-dependent in this version)
                priority = new_cost + heuristic(neighbor_pos, goal)
                heapq.heappush(pq, (priority, neighbor_state[0], neighbor_state[1], neighbor_state[2]))
                came_from[neighbor_state] = current_state

    return [], nodes_expanded, 0

  Local search: Hill climbing with random restarts (Not time-aware in this basic implementation)
  Hill climbing is typically not well-suited for dynamic environments with a known schedule
  because it doesn't explore the time dimension effectively.
  For unpredictable dynamic obstacles, it could be used for local path repair.
  The current hill climbing implementation is not time-aware and will not be modified for this.
  A time-aware local search would require a different approach (e.g., considering a short time horizon).
  We will keep the existing hill climbing as is, noting its limitation for dynamic obstacles with schedules.
ef hill_climbing(city_map: CityMap, start: Tuple[int, int], goal: Tuple[int, int], max_restarts=10) -> Tuple[List[Tuple[int, in
    def random_restart():
        current = start
        path = [current]
        visited = set()
        visited.add(current)
        # Note: This random walk doesn't consider dynamic obstacles or terrain costs effectively
        # It just tries to find *a* path, not necessarily a good or valid one in a dynamic setting.
        while current != goal:
            neighbors = city_map.neighbors(*current) # This neighbors call is not time-aware
            if not neighbors:
                break
            # Simple random choice, no cost or heuristic considered
            next_node = random.choice(neighbors)
            if next_node in visited:
                # Avoid cycles in this simple random walk
                # For a more robust local search, a different neighbor selection is needed
                break
            path.append(next_node)
            visited.add(next_node)
            current = next_node
        return path

    def path_cost(path):
        # This cost calculation is not time-aware and doesn't consider dynamic obstacles encountered
        cost = 0
        for i in range(len(path) - 1):
            current_pos = path[i]
            next_pos = path[i+1]
            # Check if the move is valid (neighbor) - this is a basic check
            if next_pos not in city_map.neighbors(*current_pos): # Not time-aware neighbor check
                 return float('inf') # Invalid move
            cost += city_map.initial_terrain_costs[next_pos[1]][next_pos[0]] # Use initial terrain cost
        return cost

    best_path = None
    best_cost = float('inf')
    nodes_expanded = 0 # This count is not directly comparable to the search algorithms' node expansion

    for _ in range(max_restarts):
        current_path = random_restart()
        # Check if the random restart actually reached the goal
        if not current_path or current_path[-1] != goal:
            continue # Skip if the random walk didn't reach the goal

        current_cost = path_cost(current_path)
```

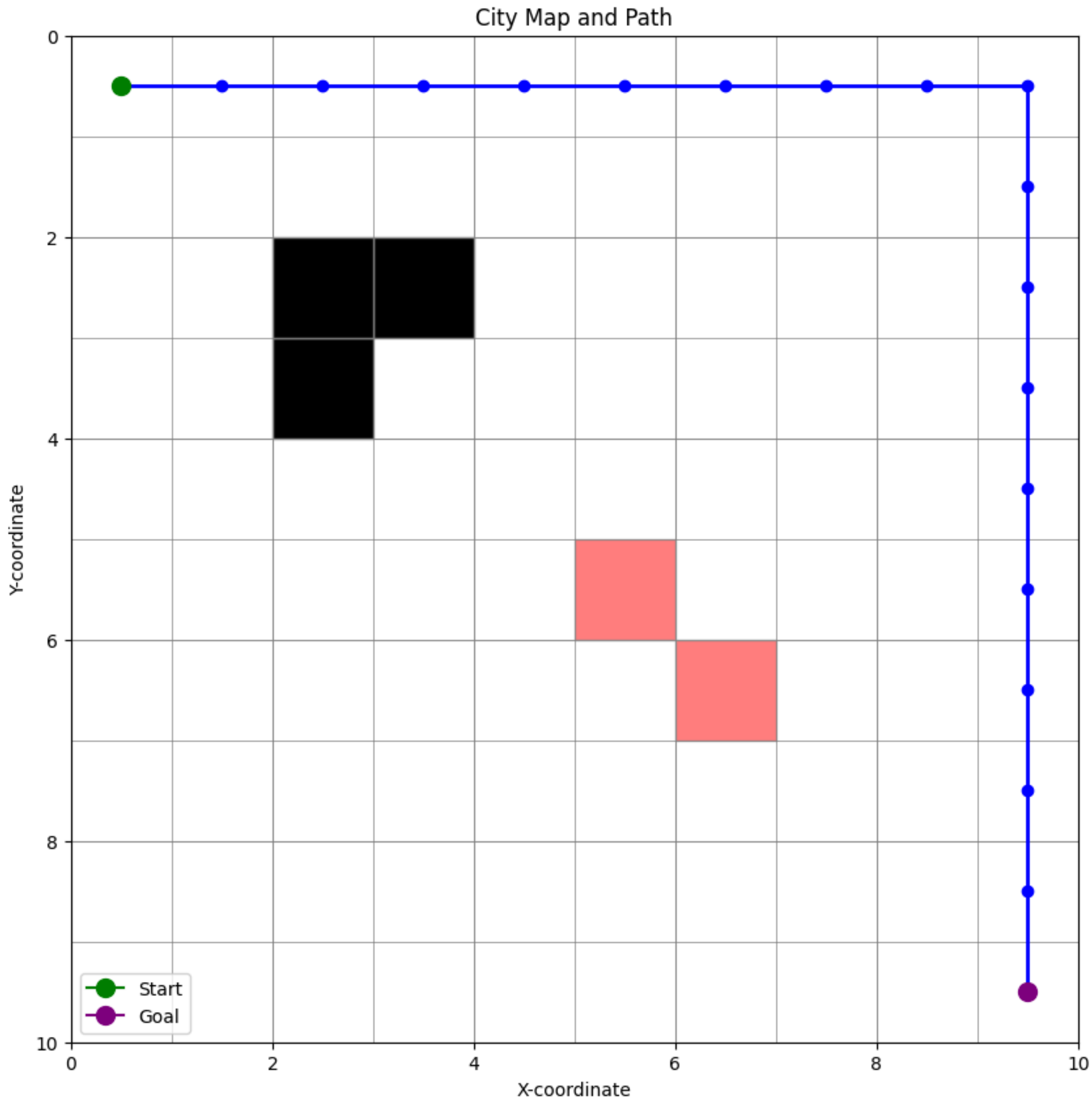
```
        # Simple hill climbing step - try to improve the path by changing one node
        # This is a very basic hill climbing and not efficient for pathfinding
        # A more standard approach would be to consider moves from the current position
        # or path smoothing/optimization techniques.
        # We will keep this simple version as it was in the original code.
        improved = True
        while improved:
            improved = False
            # The concept of "nodes_expanded" is different here. Let's count path evaluations.
            nodes_expanded += 1 # Count each attempt to improve the path as an "expansion" for comparison

            # Try to replace one node in the path with a neighbor to see if it reduces cost
            # This is a very inefficient way to do local search for paths
            # A better approach would be to consider moving the agent from its current position
            # and using a local evaluation function.
            # We will stick to the original structure for modification.
            best_improvement = 0
            best_new_path = None

            for i in range(1, len(current_path) - 1):
                current_node = current_path[i]
                neighbors = city_map.neighbors(*current_node) # Not time-aware
                for neighbor in neighbors:
                    # Create a new path with the neighbor replacing the current node
                    new_path = current_path[:i] + [neighbor] + current_path[i + 1:]
                    # Check if the new path is valid (neighbor connections) and calculate cost
                    new_cost = path_cost(new_path) # Not time-aware

                    if new_cost < current_cost:
                        # Found an improvement, but let's find the *best* single-step improvement
                        if current_cost - new_cost > best_improvement:
                            best_improvement = current_cost - new_cost
                            best_new_path = new_path


            if best_new_path:
                current_path = best_new_path
                current_cost = path_cost(current_path) # Recalculate cost
                improved = True


        if current_cost < best_cost and current_path[-1] == goal:
            best_path = current_path
            best_cost = current_cost

    if best_path is None:
        return [], nodes_expanded, 0
    return best_path, nodes_expanded, best_cost

  CLI to run planners (updated to use time-aware versions)
 f run_planner(map_instance: CityMap, start: Tuple[int, int], goal: Tuple[int, int], method: str, time_horizon: int = 100):
    start_time = time.time()
    # Use time-aware versions for BFS, UCS, and A*
    if method == 'bfs':
        path, nodes_expanded, cost = bfs_time_aware(map_instance, start, goal, time_horizon)
    elif method == 'ucs':
        path, nodes_expanded, cost = uniform_cost_search_time_aware(map_instance, start, goal, time_horizon)
    elif method == 'astar':
        path, nodes_expanded, cost = a_star_time_aware(map_instance, start, goal, time_horizon)
    elif method == 'hillclimb':
        # Hill climbing remains non-time-aware in this implementation
        path, nodes_expanded, cost = hill_climbing(map_instance, start, goal)
    else:
        raise ValueError(f"Unknown method '{method}'")
    elapsed_time = time.time() - start_time
    return path, nodes_expanded, cost, elapsed_time
```

```
# Example Usage
width = 10
height = 10
terrain_costs = [[1 for _ in range(width)] for _ in range(height)]
static_obstacles = [(2, 2), (2, 3), (3, 2)]
dynamic_obstacles = [(5, 5), (6, 6)]
dynamic_pattern = [(5, 5), (6, 5), (6, 6), (5, 6)]  # Example pattern

city_map = CityMap(width, height, terrain_costs, static_obstacles, dynamic_obstacles, dynamic_pattern)

start = (0, 0)
goal = (9, 9)

# Choose a method: 'bfs', 'ucs', 'astar', or 'hillclimb'
method = 'bfs'
```

```
method = "bfs"

path, nodes_expanded, cost, elapsed_time = run_planner(city_map, start, goal, method)

if path:
    print(f"Path found using {method}: {path}")
    print(f"Cost: {cost}")
else:
    print(f"No path found using {method}")

print(f"Nodes expanded: {nodes_expanded}")
print(f"Elapsed time: {elapsed_time:.4f} seconds")
```

```
bfs: [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0), (8, 0), (9, 0), (9, 1), (9, 2), (9, 3), (9, 4), (9, 5), (

493
0012 seconds
```

```
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import numpy as np

def visualize_map(city_map: CityMap, path: Optional[List[Tuple[int, int]]] = None):
    fig, ax = plt.subplots(1, figsize=(city_map.width, city_map.height))
    ax.set_xlim([0, city_map.width])
    ax.set_ylim([0, city_map.height])
    ax.set_aspect('equal', adjustable='box')
    ax.invert_yaxis() # Invert y-axis to match grid indexing

    # Visualize terrain costs (optional, can color cells based on cost)
    for y in range(city_map.height):
        for x in range(city_map.width):
            color = 'white' # Default
            if city_map.grid[y][x] == STATIC_OBSTACLE:
                color = 'black' # Static obstacle
            elif city_map.grid[y][x] == DYNAMIC_OBSTACLE:
                 # This visualization of dynamic obstacles on the static map is less useful for time-aware search
                 # We might need a time-aware visualization
                 pass # Don't draw initial dynamic obstacles on the static map visualization

            # You can add coloring based on terrain cost here if desired
            # ax.add_patch(patches.Rectangle((x, y), 1, 1, facecolor=color, edgecolor='gray'))

    # Draw grid lines
    for x in range(city_map.width + 1):
        ax.axvline(x, color='gray', lw=0.5)
    for y in range(city_map.height + 1):
        ax.axhline(y, color='gray', lw=0.5)


    # Add static obstacles
    for (x, y) in city_map.static_obstacles:
        ax.add_patch(patches.Rectangle((x, y), 1, 1, facecolor='black', edgecolor='gray'))


    # Add dynamic obstacles (visualize initial positions on the static map)
    # For time-aware visualization, this needs to be called within a time loop
    for (x, y) in city_map.dynamic_obstacles:
         ax.add_patch(patches.Rectangle((x, y), 1, 1, facecolor='red', edgecolor='gray', alpha=0.5)) # Red with transparency


    # Visualize the path if provided
    if path:
        path_x = [pos[0] + 0.5 for pos in path] # Center of the cell
        path_y = [pos[1] + 0.5 for pos in path] # Center of the cell
        ax.plot(path_x, path_y, marker='o', color='blue', linestyle='-', linewidth=2, markersize=6)

        # Mark start and goal
        ax.plot(path_x[0], path_y[0], marker='o', color='green', markersize=10, label='Start')
        ax.plot(path_x[-1], path_y[-1], marker='o', color='purple', markersize=10, label='Goal')
        ax.legend()


    plt.title("City Map and Path")
    plt.xlabel("X-coordinate")
    plt.ylabel("Y-coordinate")
    plt.grid(True)
    plt.show()

# Now, let's call the visualization function after running the planner
visualize_map(city_map, path)
```



Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.