Reg no.- 24BAS10083

Report on the Pathfinding Program

This notebook contains Python code for implementing and comparing different pathfinding algorithms in a simulated city environment with varying terrain costs, static obstacles, and dynamic obstacles.

Program ComponentsReport on the Pathfinding Program

This notebook contains Python code for implementing and comparing different pathfinding algorithms in a simulated city environment with varying terrain costs, static obstacles, and dynamic obstacles.

Program Components

CityMap Class:

Represents the environment with a grid of specified width and height.

Stores initial terrain costs for each cell.

Tracks the positions of static and dynamic obstacles.

Includes a dynamic_pattern to define the movement of dynamic obstacles over time.

The is_passable method checks if a cell is traversable at a given time step, considering static and dynamic obstacles.

The neighbors method returns the valid neighboring cells (4-connected) that can be moved to from a given cell at a specific time.

Search Algorithms:

bfs_time_aware (Breadth-First Search): An uninformed search algorithm that finds the shortest path in terms of the number of steps (time). It is implemented to be time-aware, considering dynamic obstacle positions at each step.

uniform_cost_search_time_aware (Uniform Cost Search): An uninformed search algorithm that finds the path with the lowest cumulative terrain cost. It is also time-aware.

a_star_time_aware (A Search)*: An informed search algorithm that uses a heuristic (Manhattan distance in this case) to guide the search and find the path with the lowest cumulative terrain cost. It is implemented to be time-aware.

hill_climbing (Hill Climbing with Random Restarts): A local search algorithm. The current implementation is not time-aware and is less suitable for

environments with deterministic dynamic obstacles. It attempts to find a path by iteratively improving a randomly generated initial path.

run_planner Function:

A utility function to execute the chosen search algorithm (bfs, ucs, astar, or hillclimb) on a given CityMap instance, start, and goal.

Records and returns the found path, the number of nodes expanded during the search, the cost of the path, and the elapsed time for the search.

visualize_map Function:

Uses matplotlib to create a visual representation of the city map.

Displays the grid, static obstacles (black squares), initial dynamic obstacle positions (red squares with transparency), the start point (green circle), the goal point (purple circle), and the found path (blue line with markers).

Example Usage and Results

The example code in the notebook demonstrates how to create a CityMap instance with defined dimensions, terrain costs, static obstacles, dynamic obstacles, and a dynamic pattern. It then sets a start and goal point and runs the astar search algorithm.

Here are the results from the example execution:

Method: astar

Path found: [(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), (9, 6), (9, 7), (9, 8), (9, 9)]

Cost: 18

Nodes expanded: 493

Elapsed time: 0.0012 seconds

The visualization shows this path on the generated map. The A* algorithm successfully found a path from (0,0) to (9,9) navigating around the static and dynamic obstacles based on their positions at each time step within the given time horizon. The cost of 18 represents the accumulated terrain cost along the path.

Implemented Features

Integer Movement Cost: The terrain_costs list allows for different movement costs for each grid cell.

Moving Obstacles: Dynamic obstacles are handled with a known deterministic schedule defined by the dynamic_pattern. The time-aware search algorithms consider these future positions.

4-Connected Movement: The neighbors method restricts movement to up, down, left, and right cells.

Conclusion

The program provides a solid foundation for experimenting with different pathfinding algorithms in a dynamic environment. The time-aware implementations of BFS, UCS, and A* can find paths that avoid moving obstacles with a known schedule. The visualization helps in understanding the environment and the found paths. Further enhancements could include more complex dynamic obstacle behaviors, different heuristics for A*, and visualization of dynamic obstacles at each time step of the path.

1. `CityMap` **Class**:
    - Represents the environment with a grid of specified width and height.
    - Stores initial terrain costs for each cell.
    - Tracks the positions of static and dynamic obstacles.
    - Includes a `dynamic_pattern` to define the movement of dynamic obstacles over time.
    - The `is_passable` method checks if a cell is traversable at a given time step, considering static and dynamic obstacles.
    - The `neighbors` method returns the valid neighboring cells (4-connected) that can be moved to from a given cell at a specific time.
2. **Search Algorithms**:
    - `bfs_time_aware` **(Breadth-First Search)**: An uninformed search algorithm that finds the shortest path in terms of the number of steps (time). It is implemented to be time-aware, considering dynamic obstacle positions at each step.
    - `uniform_cost_search_time_aware` **(Uniform Cost Search)**: An uninformed search algorithm that finds the path with the lowest cumulative terrain cost. It is also time-aware.
    - `a_star_time_aware` *(A Search)**: An informed search algorithm that uses a heuristic (Manhattan distance in this case) to guide the search and find the path with the lowest cumulative terrain cost. It is implemented to be time-aware.
    - `hill_climbing` **(Hill Climbing with Random Restarts)**: A local search algorithm. The current implementation is not time-aware and is less suitable for environments with deterministic dynamic obstacles. It attempts to find a path by iteratively improving a randomly generated initial path.
3. `run_planner` **Function**:
    - A utility function to execute the chosen search algorithm (`bfs`, `ucs`, `astar`, or `hillclimb`) on a given `CityMap` instance, start, and goal.
    - Records and returns the found path, the number of nodes expanded during the search, the cost of the path, and the elapsed time for the search.
4. `visualize_map` **Function**:

- Uses `matplotlib` to create a visual representation of the city map.
- Displays the grid, static obstacles (black squares), initial dynamic obstacle positions (red squares with transparency), the start point (green circle), the goal point (purple circle), and the found path (blue line with markers).

Example Usage and Results

The example code in the notebook demonstrates how to create a `CityMap` instance with defined dimensions, terrain costs, static obstacles, dynamic obstacles, and a dynamic pattern. It then sets a start and goal point and runs the `astar` search algorithm.

Here are the results from the example execution:

- **Method:** astar
- **Path found:** [(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), (9, 6), (9, 7), (9, 8), (9, 9)]
- **Cost:** 18
- **Nodes expanded:** 493
- **Elapsed time:** 0.0012 seconds

The visualization shows this path on the generated map. The A* algorithm successfully found a path from (0,0) to (9,9) navigating around the static and dynamic obstacles based on their positions at each time step within the given time horizon. The cost of 18 represents the accumulated terrain cost along the path.

Implemented Features

- **Integer Movement Cost:** The `terrain_costs` list allows for different movement costs for each grid cell.
- **Moving Obstacles:** Dynamic obstacles are handled with a known deterministic schedule defined by the `dynamic_pattern`. The time-aware search algorithms consider these future positions.
- **4-Connected Movement:** The `neighbors` method restricts movement to up, down, left, and right cells.

Conclusion

The program provides a solid foundation for experimenting with different pathfinding algorithms in a dynamic environment. The time-aware implementations of BFS, UCS, and A* can find paths that avoid moving obstacles with a known schedule. The visualization helps in understanding the environment and the found paths. Further enhancements could include more complex dynamic obstacle behaviors, different heuristics for A*, and visualization of dynamic obstacles at each time step of the path.