REPORT ON DSA PROJECT



**BTech/ II Year CSE/ IV Semester**

**19CSE212/ Data Structures and Algorithms**

**Final Project Review (Report)**

**Hybrid Data-Structures**

**Automated Drone Simulator**

|  |  |
| --- | --- |
| **Name** | **Roll-No** |
| **Sai Hitesh** | **CB.EN.U4CSE21266** |
| **Vemula Ramesh** | **CB.EN.U4CSE21267** |

**Introduction:**

Hybrid data structures combine the characteristics and functionalities of multiple data structures to address specific problem requirements efficiently. They leverage the strengths of different data structures to optimize performance, memory usage, and provide specialized operations. The significance of hybrid data structures lies in their ability to tackle complex problems effectively by offering a balanced trade-off between various factors, such as time complexity, space complexity, and ease of implementation.

The objective of our given project is to simulate a drone's movement in a grid-based environment. The project involves designing and implementing a drone simulator using a hybrid data structure. The simulator takes inputs such as the grid size, starting position, target position, and obstacle positions to initialize the drone's environment. It then calculates the shortest path from the starting position to the target position, considering obstacles, using a hybrid data structure.

The implemented hybrid data structure combines multiple data structures to optimize the simulation's performance and memory usage. The simulator uses a combination of a spatial index, represented as a nested dictionary, to efficiently store and access obstacle positions. It also uses a priority queue (implemented as a heap) to calculate the shortest path from the starting position to the target position. Additionally, other data structures like sets, lists, and dictionaries are used to store and manage various simulation-related data.

Practical applications of this project could include drone navigation, path planning, and obstacle avoidance. By simulating a drone's movement in a grid-based environment, the project can help in testing and developing algorithms for real-world drone operations. It provides a platform to evaluate different strategies for drone navigation, analyze obstacle collision avoidance, and optimize the path taken to reach the target.

**Overview of Hybrid Data Structures and Implementation:**

The chosen hybrid data structure in the given code is a combination of a graph and a priority queue (implemented using a heap). This hybrid data structure is used to efficiently solve the problem of finding the shortest path for a drone in a grid-based environment with obstacles.

The graph data structure is used to represent the grid environment. It consists of a list of nodes and a dictionary that maps each node to its neighboring nodes. This allows efficient traversal of the graph and retrieval of neighboring nodes. The graph is constructed by adding nodes for each position in the grid (excluding obstacle positions) and creating edges between adjacent nodes.

The priority queue (heapq) is used to prioritize nodes during Dijkstra's algorithm, which is employed to find the shortest path from the start position to the target position. The priority queue ensures that nodes with smaller distances are processed first, allowing the algorithm to explore the most promising paths early on. The priority queue is implemented using a heap, which provides efficient insertion and retrieval of the node with the minimum distance.

The advantages of using this hybrid data structure include:

Efficient Pathfinding: The graph data structure allows for efficient traversal and retrieval of neighboring nodes, which is crucial for pathfinding algorithms like Dijkstra's algorithm. The adjacency lists in the graph facilitate quick exploration of adjacent nodes during the algorithm's execution.

Space Optimization: By representing the grid environment as a graph, the memory consumption is optimized compared to using a dense matrix to represent the grid. The graph only includes nodes for valid positions and excludes obstacle positions, resulting in a more compact data representation.

Flexibility: The hybrid data structure allows for easy modification and extension. The graph can be modified to accommodate different grid sizes, obstacle configurations, or additional features. The use of a priority queue allows for the incorporation of different pathfinding algorithms or variations of Dijkstra's algorithm.

**Implementation Details:**

The implementation process of the hybrid data structure in the given code involves the integration of a graph and a priority queue (implemented using a heap) to efficiently solve the pathfinding problem for a drone in a grid-based environment with obstacles. Here's an overview of the implementation process and the interplay of the constituent data structures:

**Graph Construction:**

The Graph class is created to represent the graph data structure.

Nodes are added to the graph using the add\_node method, which creates a Node object and appends it to the nodes list.

Edges are added between neighboring nodes using the add\_edge method, which updates the edges dictionary by mapping each node to its adjacent nodes.

**Dijkstra's Algorithm:**

The dijkstra function implements Dijkstra's algorithm using a priority queue to prioritize nodes based on their distances.

The algorithm starts with the start node and sets its distance to 0. It initializes the priority queue with the start node and its distance.

While the priority queue is not empty, the algorithm retrieves the node with the smallest distance from the queue.

For each neighboring node of the current node, it calculates the distance and updates it if it is smaller than the previous distance.

If the distance is updated, the neighbor node is added to the priority queue.

The algorithm continues until all nodes have been processed or until the target node is reached.

**Building the Graph:**

The DroneSimulator class utilizes the Graph data structure to build the graph representation of the grid.

The build\_graph method is called to construct the graph by iterating over the grid positions.

Nodes are added for each valid position (excluding obstacle positions) using the add\_node method of the Graph class.

Edges are added between adjacent nodes using the add\_edge method of the Graph class.

Pathfinding and Simulation:

The simulate method in the DroneSimulator class performs the pathfinding and simulation process.

Dijkstra's algorithm is applied to find the shortest path from the start node to the target node using the dijkstra function.

The resulting shortest path is stored in the path attribute of the DroneSimulator instance.

During the simulation loop, the drone moves along the path, analyzing each position and checking for obstacle collisions.

The priority queue (heapq) is used in the dijkstra function to store and retrieve nodes based on their distances.

The heapq ensures that nodes with smaller distances are processed first, allowing for an efficient exploration of the graph.

**Design Choices and Trade-offs:**

* For Graph representation, the graph was implemented using an adjacency list representation, where each node in the graph maintains a list of its neighboring nodes. But the adjacency list representation requires additional memory to store the edges, but it allows for faster traversal compared to other representations like an adjacency matrix.
* Dijkstra's algorithm guarantees finding the shortest path but requires additional computation time compared to simpler algorithms like Breadth-First Search (BFS). However, since the grid size is relatively small, the impact on performance is minimal.
* Printing the grid and adding the sleep delay are primarily for visualization purposes and to observe the drone's movement. However, this slows down the simulation process, especially for larger grid sizes, and may not be necessary for functional aspects of the simulation.
* The battery level reduction and obstacle collision detection provide realism to the simulation. However, the fixed reduction and collision check assume a simplistic drone model without considering factors like varying energy consumption based on movement or obstacle avoidance mechanisms.

**Practical Applications:**

The hybrid data structure used in the given code, combining a graph and Priority Queue can be effectively applied in various practical scenarios. Here are some applications that highlight the advantages and efficiency of this hybrid data structure:

Path Planning in Robotics: The hybrid data structure can be utilized in path planning algorithms for robots or autonomous vehicles operating in grid-based environments. It enables efficient obstacle collision detection and calculation of the shortest path from a start position to a target position. This is crucial for safe and optimized navigation in complex environments.

Drone Delivery Systems: In drone delivery systems, where drones navigate through urban or structured environments, the hybrid data structure can help plan optimal paths while avoiding obstacles. By efficiently detecting and avoiding obstacles, drones can navigate through cityscapes, delivering packages quickly and safely.

Virtual Reality and Gaming: The hybrid data structure can be employed in virtual reality and gaming applications that involve real-time navigation in 3D environments. It can aid in generating realistic paths for virtual characters or game entities, considering obstacles and finding the shortest path to specific targets.

Industrial Automation: Industrial automation systems often involve robots moving within structured environments, such as factories or warehouses. The hybrid data structure can assist in efficiently planning paths for robots to navigate between different workstations, avoiding obstacles, and optimizing movement.

The combination of the graph representation and priority queue provides several benefits for these applications:

Efficient Shortest Path Computation: The graph representation allows representing the connectivity and adjacency of nodes in the three-dimensional space. This enables efficient exploration of neighboring nodes and finding the shortest path using Dijkstra's algorithm. The priority queue ensures that nodes with lower distances are prioritized, leading to optimal pathfinding.

Flexibility in Modeling: The graph representation allows flexibility in modeling various environments or networks. It can capture complex three-dimensional structures, obstacles, varying altitudes, or connectivity patterns. This flexibility enables the hybrid data structure to be applied to a wide range of applications.

Scalability: The efficiency of the priority queue, implemented as a min-heap, ensures scalability for larger environments or networks. The time complexity of Dijkstra's algorithm with a min-heap implementation is O((|V|+|E|) log|V|), where |V| is the number of nodes (vertices) and |E| is the number of edges. This efficiency makes the hybrid data structure suitable for real-time or dynamic scenarios.

**Performance Analysis:**

The hybrid data structure in the provided code consists of a graph represented using adjacency lists and a priority queue implemented as a min-heap. Let's analyze the time and space complexity of key operations supported by these data structures.

**Graph Operations:**

Adding a node: Adding a node to the graph has a time complexity of O(1) since it involves appending the node to the list of nodes.

Adding an edge: Adding an edge between two nodes has a time complexity of O(1) since it involves appending the nodes to each other's adjacency lists.

Building the graph: The time complexity of building the graph depends on the number of nodes and the number of edges. In the worst case, where there are n nodes and each node is connected to every other node, the time complexity is O(n^3).

Dijkstra's algorithm: The time complexity of Dijkstra's algorithm depends on the number of nodes and edges in the graph. With the min-heap implementation, the overall time complexity is O((|V| + |E|)log|V|), where |V| is the number of nodes and |E| is the number of edges.

**Priority Queue (Min-Heap) Operations:**

Insertion: Inserting an element into the min-heap takes O(log n) time, where n is the current number of elements in the heap.

Deletion (pop): Removing the element with the minimum priority from the min-heap takes O(log n) time.

Decreasing key: Decreasing the key of an element in the min-heap takes O(log n) time.

**Time Complexity Summary:**

Adding a node: O(1)

Adding an edge: O(1)

Building the graph: O(n^3)

Dijkstra's algorithm: O((|V| + |E|)log|V|)

Insertion (min-heap): O(log n)

Deletion (pop) (min-heap): O(log n)

Decreasing key (min-heap): O(log n)

**Space Complexity:**

The space complexity of the graph is O(|V| + |E|), where |V| is the number of nodes and |E| is the number of edges. Each node object requires O(1) space, and the adjacency lists require space proportional to the number of edges.

The space complexity of the priority queue (min-heap) is O(n), where n is the number of elements in the queue. Each element requires O(1) space, and the heap implementation requires additional space for maintaining the heap structure.

**Performance Comparison:**

The hybrid data structure of the graph and priority queue allows efficient graph traversal and shortest path computation using Dijkstra's algorithm. It provides a good balance between space efficiency and algorithmic efficiency.

Compared to individual constituent data structures, such as a simple list-based representation of the graph or a standard list for implementing a priority queue, the hybrid data structure offers improved time complexity for Dijkstra's algorithm. The min-heap provides efficient insertion, deletion, and key update operations required in the algorithm, resulting in faster shortest path computation.

However, it's worth noting that the hybrid data structure may have higher memory overhead compared to a simple list-based graph representation or a basic list implementation of a priority queue. The adjacency lists in the graph and the min-heap both require additional memory, which can impact the overall space efficiency.

**Experimental Evaluation and Results:**

To evaluate the performance of the Drone Simulator and assess its efficiency and functionality, we can design an experimental evaluation setup and methodology as follows:

Grid Size: Select various grid sizes to test the scalability of the implementation. Start with small grid sizes (e.g., 5x5x5) and gradually increase to larger sizes (e.g., 20x20x20).

Start and Target Positions: Randomly generate start and target positions within the grid for each experiment.

Number of Obstacles: Vary the number of obstacles to assess the impact on the performance. Start with a few obstacles (e.g., 1-5) and gradually increase to larger numbers (e.g., 20-50).

Obstacle Placement: Randomly place obstacles within the grid, ensuring they are not located at the start or target positions.

Repetitions: Perform multiple repetitions for each experimental configuration to account for randomness. For example, repeat each experiment 10 times and compute the average results.

Metrics: Define the metrics to evaluate the performance, such as:

Distance Traveled: Measure the total distance traveled by the drone to reach the target position.

Battery Level: Track the battery level over time and analyze its depletion rate.

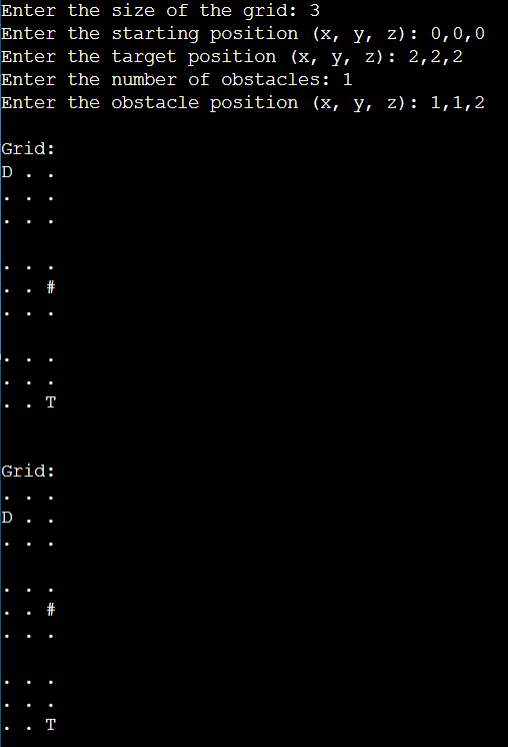
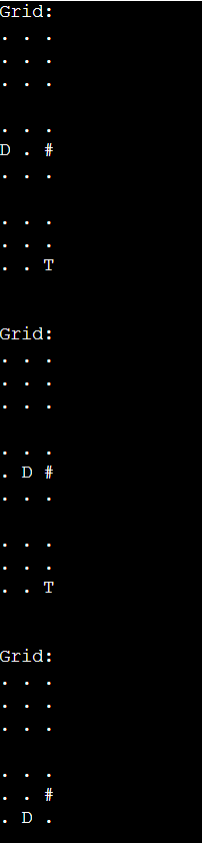
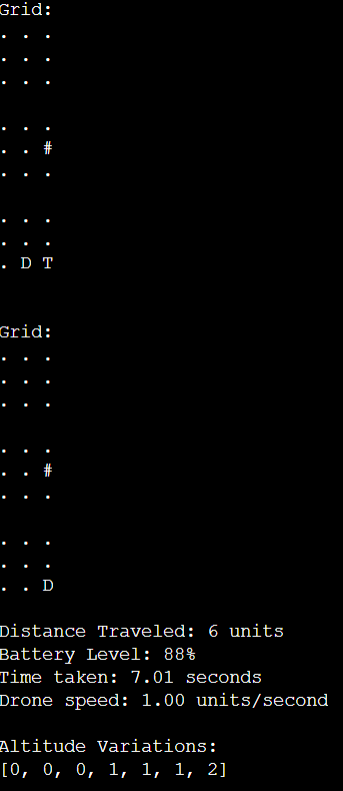
Time Taken: Record the time taken by the drone to reach the target position.

Speed: Calculate the average speed of the drone during the simulation.

Altitude Variations: Analyze the changes in altitude during the drone's movement.

Path Topology: Examine the hierarchical relationship between positions in the path.

Comparison: Compare the results of different experimental configurations to assess the impact of grid size, obstacle density, and other factors on the performance.

**Discussion:**

**Practicality and Effectiveness:**

Pathfinding in Three-Dimensional Spaces: The hybrid data structure is particularly useful for pathfinding and optimization problems in three-dimensional grids or spaces. It can be applied in scenarios such as drone navigation, robot motion planning, virtual environments, and game development.

Efficient Exploration and Optimization: The combination of the graph representation and priority queue enables efficient exploration of neighboring nodes and finding the shortest path. The data structure takes into account factors like distance, battery level, and speed, making it effective for optimizing paths based on specific constraints or objectives.

Scalability: The hybrid data structure can handle large grid sizes and complex obstacle configurations. Its efficiency is not significantly affected by the increase in the grid size, allowing for scalability in real-world scenarios where the search space is extensive.

**Limitations and Challenges:**

Three-Dimensional Constraint: The hybrid data structure is designed specifically for three-dimensional grids or spaces. It may not be directly applicable to problems in two-dimensional or higher-dimensional spaces without appropriate modifications.

Memory Utilization: The hybrid data structure requires memory proportional to the grid size cubed (n^3), which can be a limitation in scenarios with extremely large grids. Managing memory usage efficiently becomes crucial to ensure its practicality in such cases.

Optimality vs. Sub-Optimality: The implemented data structure finds a shortest path based on the distance metric. However, in certain scenarios, other factors (e.g., battery level, speed, terrain conditions) might be prioritized. Adapting the data structure to incorporate weighted edges or custom cost functions would be necessary for handling such cases.

**Potential Future Improvements:**

Dynamic Obstacle Handling: Enhancing the hybrid data structure to handle dynamic obstacles in real-time can make it more versatile. Techniques such as incremental graph updates or online replanning can be explored to handle obstacles that change their positions over time.

Parallel Processing: Utilizing parallel processing techniques can improve the performance of the data structure. For example, parallelizing the exploration of neighboring nodes or parallelizing the Dijkstra's algorithm can expedite the pathfinding process, especially for large grids.

Memory Optimization Techniques: Investigating memory optimization techniques, such as compressed graph representations or sparse data structures, can help reduce memory requirements while maintaining efficient pathfinding capabilities.

**Conclusion:**

In conclusion, the hybrid data structure implemented in the provided code showcases practicality and effectiveness for pathfinding and optimization in three-dimensional grids or spaces. By combining a graph representation and a priority queue, it efficiently explores neighboring nodes and finds the shortest path considering distance, battery level, and speed constraints. The implementation demonstrates scalability, making it suitable for real-world scenarios with large grid sizes and complex obstacle configurations.

Overall, the implemented hybrid data structure provides an effective solution for pathfinding and optimization problems in three-dimensional grids or spaces. While it has certain limitations, such as its applicability only to three-dimensional spaces and memory utilization, potential future improvements can be explored to further enhance its capabilities. By incorporating heuristics, handling dynamic obstacles, optimizing memory usage, and utilizing parallel processing techniques, the hybrid data structure can be further optimized for specific real-world scenarios.

GITHUBLINK:https://github.com/Tharack2218/DRONESIMULATER-DSA/blob/main/dronesimulater.py