PROBLEM STATEMENT : Exploration and Implementation of Motion Planning Algorithms.

SOLn :
Algorithm - 1

1. **RRT Algorithm** -

Rapidly-exploring Random Tree (RRT) is a motion planning algorithm designed for efficiently searching high-dimensional spaces by randomly sampling points and incrementally building a tree. It is widely used in robotics, autonomous systems, and pathfinding for exploring large, complex spaces, especially in situations where an exact solution is not required or is computationally infeasible.

---

2. Strengths:

- RRT quickly explores large, complex environments by extending the tree toward randomly sampled points, avoiding the need for exhaustive search.
- It scales well in high-dimensional spaces (e.g., robotic arms with multiple joints) where grid-based approaches (like A*) would be inefficient.
- The algorithm can be adapted for dynamic environments and can handle non-holonomic constraints (e.g., vehicles with steering restrictions).

3. Weaknesses:

- The paths generated by RRT are not guaranteed to be optimal or smooth. The algorithm may produce jagged, indirect routes.
- The random nature of the algorithm means that performance may vary between runs, potentially requiring many iterations to find a good solution.
- The quality of the solution heavily depends on parameters like step size and the sampling strategy, which require careful tuning.

---

4. Applicability

Obstacle Handling:

RRT can handle environments with complex, cluttered obstacles. It avoids collisions by ensuring that each new node is collision-free before adding it to the tree.

Scalability:

RRT is highly scalable in terms of dimensionality. Its performance does not degrade as sharply as grid-based methods in high-dimensional spaces, making it particularly useful for robotic systems with many degrees of freedom.

Real-time Use:

While the basic RRT is not optimised for real-time applications, modifications like RRT-Connect or RRT* can be used for real-time pathfinding in dynamic environments.

5. Performance:

Time Complexity : O(nlogn)

Space Complexity : O(n)

Note : n = number of nodes of the tree

Algorithm - 2

1. A Algorithm*

A* is a graph-based search algorithm that combines features of both Dijkstra's Algorithm and Greedy Best-First Search. It uses a heuristic to estimate the cost from any node to the goal and prioritises exploration of nodes that appear most promising. It guarantees finding the shortest path in static environments, provided the heuristic is admissible (it never overestimates the actual cost). A* is widely used in pathfinding and graph traversal, commonly applied in games, robotics, and logistics.

Use Cases:

- Robotics: Path planning in static environments with known obstacles.
- Games: Finding the shortest path between two points in a game world.
- Navigation Systems: Routing in maps with static road networks.

---

2. **Strengths**

- Optimality: A* guarantees finding the optimal path if the heuristic is admissible (the heuristic is an underestimate of the true cost to reach the goal).
- Efficiency with Heuristics: A* efficiently explores the search space by using a heuristic to focus on promising paths, reducing the number of nodes that need to be explored compared to Dijkstra's algorithm.
- Versatility: A* can be adapted for various environments by changing the heuristic. It is used for both low and high-dimensional spaces, including grid-based maps and more complex graphs.

3. **Weaknesses**

- Memory Usage: A* can be memory-intensive, as it stores all explored nodes (closed list) and nodes still to explore (open list). In large environments, this can lead to excessive memory consumption.
- Performance in Large Spaces: A* may become slow in large or high-dimensional spaces, as it expands many nodes, especially if the heuristic is weak or uninformative.
- Not Suitable for Dynamic Environments: A* is designed for static environments. It performs poorly in dynamic environments where obstacles change position, as it needs to replan from scratch.

---

4. **Applicability**

Obstacle Handling:

A* is effective in handling environments with obstacles, provided the environment is static. It expands nodes based on their cost and heuristic, ensuring that it avoids obstacles by considering the nodes with the lowest cost that do not collide with them.

Scalability:

A* works best in low to moderate-dimensional spaces (e.g., 2D or 3D grids). However, its performance deteriorates in large or high-dimensional environments, as it has to explore and store a large number of nodes.

Real-time Use:

A* is not typically used in real-time applications due to its need to explore large portions of the space. However, variants like *Theta or Lifelong Planning A***, and dynamic versions (D* Lite) can make A* more applicable to real-time and dynamic environments.

5. Performance

Time Complexity:

The time complexity of A* is $O(b^d)$ where:

- b is the branching factor (the average number of children per node),
- d is the depth of the solution.

In the worst case, A* can degenerate into an exhaustive search, which is why the choice of a good heuristic is crucial.

Space Complexity:

The space complexity is also $O(b^d)$, as A* keeps all nodes in memory, which can make it unsuitable for very large problems.

Algorithm - 3

1. Dijkstra's Algorithm

Dijkstra's Algorithm is a graph-based search algorithm used for finding the shortest path between a source node and all other nodes in a graph with non-negative edge weights. It is a fundamental algorithm in computer science and is widely used for solving shortest path problems in network routing, pathfinding, and geographic navigation. Unlike A*, Dijkstra's algorithm does not use a heuristic and explores nodes based solely on the known cost from the source node.

Use Cases:

- **Network Routing:** Finding the shortest path between routers in communication networks.
- **Navigation Systems:** Routing vehicles on road networks, where the graph represents roads and intersections.
- **Robotics:** Path planning in static environments with known distances or costs between locations.

---

2. Strengths

- **Optimality:** Dijkstra's Algorithm guarantees finding the optimal (shortest) path in graphs with non-negative edge weights. It explores nodes systematically, ensuring that no better path exists once a node has been visited.
- **Completeness:** The algorithm is complete, meaning that if a path exists between the source and target node, Dijkstra will find it. It processes all reachable nodes.
- **No Heuristic Dependency:** Unlike A*, Dijkstra's performance does not rely on a heuristic. This makes it simpler to implement and ideal in situations where no good heuristic is available.

3. Weaknesses

- **Memory Usage:** Similar to A*, Dijkstra's Algorithm can consume large amounts of memory as it needs to maintain a list of all visited and unvisited nodes. This makes it inefficient for very large graphs.

- **Slow for Large Graphs:** Dijkstra's Algorithm explores all possible paths and can become computationally expensive in large or high-dimensional graphs, as it processes all reachable nodes, regardless of whether they are on the optimal path.
- **Not Efficient for Real-time Applications:** Since Dijkstra explores all nodes indiscriminately, it is not well-suited for real-time applications or dynamic environments where re-planning is required frequently.

---

4. Applicability

Obstacle Handling:

Dijkstra's Algorithm can effectively handle environments with static obstacles by treating the obstacles as nodes with infinite cost or by excluding them from the graph. It explores nodes in a systematic fashion and finds the shortest collision-free path.

Scalability:

Dijkstra's Algorithm works well for small to medium-sized graphs. However, its performance decreases in large-scale or high-dimensional environments because it explores all nodes until the goal is reached, regardless of whether they contribute to the optimal solution.

Real-time Use:

Dijkstra is not generally used in real-time applications because it explores all reachable nodes, which can be time-consuming. However, its variants (such as D* Lite or Lifelong Planning A*) can be applied to dynamic environments.

---

5. Performance

Time Complexity:

- **Without priority queue:** $O(v^2)$, where V is the number of vertices. This occurs when using an unoptimized version of Dijkstra's algorithm, typically implemented with arrays.
- **With priority queue (e.g., using a min-heap):** *O((V+E)logV)*, where V is the number of vertices, and E is the number of edges. This optimized version reduces the time complexity, especially in sparse graphs.

Space Complexity:

The space complexity of Dijkstra's Algorithm is *O(V+E)*, where:

- V is the number of vertices (nodes),
- E is the number of edges (connections between nodes).

This accounts for storing the graph, the distance array, and the priority queue.

Algorithm - 4

1. Dynamic A (D** Lite) Algorithm:

Dynamic A* (also known as D* Lite) is an efficient variant of the A* algorithm designed for path planning in dynamic environments where the environment or obstacles change over time. It is commonly used in robotics, where robots must adapt to changes in their environment (e.g., newly discovered obstacles or moving objects). Unlike A*, which would require re-planning from scratch when the environment changes, D* Lite efficiently updates paths based on the changes, minimizing redundant computation.

Use Cases:

- **Robotics:** Path planning for autonomous robots navigating in unknown or dynamic environments.
- **Exploration Robots:** Autonomous systems that adapt to changes in a dynamic world, such as robots in search-and-rescue operations or planetary exploration.
- **Navigation Systems:** Real-time navigation for vehicles in environments where conditions change (e.g., road closures, traffic, or moving obstacles).

---

2. Strengths

- **Efficient in Dynamic Environments:** D* Lite incrementally updates the path without needing to restart the entire search process, making it highly efficient in environments that change over time.
- **Reusability of Previous Computation:** When changes occur (e.g., obstacles are added or removed), D* Lite reuses previous search results, reducing computational redundancy.
- **Optimality:** D* Lite guarantees finding the optimal path as long as the cost map is accurate and there are no negative edge weights, similar to A*.
- **Real-time Performance:** By updating only the affected portions of the graph, D* Lite can quickly adjust the path, making it well-suited for real-time applications where frequent re-planning is required.

---

3. Weaknesses

- **Memory Usage:** Similar to A*, D* Lite can consume a large amount of memory, as it must store all nodes and their associated costs, especially in large or dense environments.
- **Complexity:** D* Lite can be more complex to implement than A* due to the need for maintaining incremental updates and handling changes in the environment.
- **Handling Large Dynamic Changes:** While D* Lite is efficient for small changes in the environment, large, frequent changes can degrade its performance, requiring more extensive re-planning.

---

4. Applicability

Obstacle Handling:

D* Lite excels at handling dynamic obstacles. When obstacles are added or removed from the environment, the algorithm efficiently updates the path without needing to completely replan, making it ideal for environments where the robot must react to moving obstacles or changes.

Scalability:

D* Lite is scalable for large environments but can suffer from performance issues in very large spaces with significant dynamic changes. However, its efficiency in local updates makes it better suited than static algorithms like A* in such scenarios.

Real-time Use:

D* Lite is designed for real-time applications where the environment may change during execution. It dynamically adjusts paths on-the-fly, making it an ideal choice for autonomous robots, drones, or self-driving vehicles navigating in uncertain or changing environments.

---

5. Performance

Time Complexity:

The time complexity of D* Lite is approximately:

- **Initial Path Search:** $O(V \log V)$where V is the number of vertices. This is comparable to A* in the initial planning phase.
- **Subsequent Replanning:** Much faster than re-running A*, typically O(V) for local updates, as it only adjusts the affected parts of the graph when the environment changes.

Space Complexity:

The space complexity is:

- O(V), where V is the number of vertices. This is similar to A* since the algorithm must maintain information for all nodes in the environment.