Delivery Robot Path Planning

By:

- **Nourhan Deif 202201959**
- **Ali Adel 202202078**
- **Karim Ezz 202201972**
- **Abdulkhaliq Sarwat 202202084**

Delivery Robot Path Planning

1. Problem Modeling:

- ❖ *State space representation:*
  - Each state represents a specific location of the delivery robot in its environment (Grid).
  - The state space collects all possible locations the robot could be in along with relevant information like obstacles.

- ❖ Initial and goal states:
  - Initial state: robot‘s starting location. (0,0)
  - Goal state: the distention the robot needs to be in (29,29)
- ❖ Actions:
  - Move forward
  - Turn right, turn left, up and down
  - Backtrack to avoid obstacles.
- ❖ Transition function:
  - If the robot is situated in a certain location and decides to move ahead, the transition function would change its location to a neighboring cell in the chosen direction, taking into account any walls or obstacles present.
  - This function also computes alterations in the robot's time taken to arrive at the goal, or any time increment due to congestion and barriers.

- ❖ Problem size:
  - 30X30 Grid with 900 cells

2. *Modeling Assumptions:*
   - Obstacles*: There may be* obstacles that the robot must avoid.
   - There are random obstacles
   - The environment is fully observed the robot has all possible information about its environment

2. Search Algorithms and their performance:

- Problem overview: The present issue involves controlling a Delivery Robot on a 2D plane. The robot must locate the shortest path from a given entry point to an exit point while avoiding obstructions. The matrix exhibits the grid form of specific grid sizes. The goal of this project is to program different search strategies and compare their effectiveness in resolving this particular problem of finding a route.
- Uninformed Search Algorithms:

  ❖ How the algorithms work:

  1. **Breadth First Search (BFS)**: It visits every node in a level, one level at a time, then offers the shortest route in an unweighted grid.
  2. **Depth First Search (DFS):** It goes as far as possible along a branch before any backtracking – which is likely inefficient in finding the network's shortest path.
  3. **Uniform Cost Search (UCS):** It expands the lowest cost node first, i.e., considering the cost incurred in traversing to the next step.
  4. **Iterative deepening search (IDS):** It uses the benefits of BFS and DFS by iterative deepening depth-first search.

  ❖ Time complexity:

  1. **BFS:** Time Complexity is O(b^d), where b is the theoretical maximum number of children of a node and d is the depth of the goal.
  2. **DFS:** Looking for left and right subtrees takes O(b^d) time in the worst case. However, memory can be less consuming than BFS.
  3. **UCS:** O(b^d) is another exploration strategy that is similar to BFS but instead of expanding all nodes paths are explored based on path cost.
  4. **IDS:** The time complexity is O(b^d) as for the case of b d trees, but it is more memory efficient since only the current depth level has to be saved.

❖ Space complexity:

1. **BFS:** Uses a considerable amount of space because all the nodes at a given level are kept in memory.
2. **DFS:** Less memory-consuming in contrast to BFS, because only nodes in the existing path are in memory.
3. **UCS:** Uses similar information to that of BFS but incurs extra cost in keeping track of the distance of each node.
4. **IDS:** Most of the time efficient because only keeps the records of the nodes in the depth level of interest.

❖ Optimality:

- The four algorithms are complete in the sense that for each of the algorithms, there is state space such that if there is a solution to be found, it will be found.

❖ Completeness:

- **BFS** and **UCS** are optimal as they find the shortest path (in terms of several steps or cost).
- **DFS** is not guaranteed to be optimal as it may explore deep but suboptimal paths first.
- **IDS** is optimal as it behaves like BFS for finding the shortest path

- Heuristic Search Algorithms:
  - ❖ Heuristic functions:
    - Manhattan Distance: This is computed as a value of x and y which is the vertical and horizontal difference between a particular current node and goal node respectively. It works well for grid-based pathfinding as its horizontal and vertical forms restrict movement to only two directions.

    - Euclidean Distance: This is the straight distance between the current node and the goal in a practical sense

❖ How algorithms work:

- **Greedy Best-First Search (GBFS):** Selects the next node to be expanded based on how close it appears to the goal according to a

heuristic function. It does not ensure that an optimal path will be found but is quick and uses little memory.

- **A Search*:** Takes into account the cost to arrive at the present node, as well as the heuristic awareness of the distance Remaining to the goal. A* is an optimal and complete algorithm as long as the heuristic employed is admissible.

❖ Time complexity:
- The running time complexity for both GBFS and A* search algorithm is O(b^d), however, we see that A* generates fewer nodes than that of GBFS.

❖ Memory storage:
- **GBFS:** Often, less space is occupied when A* is not implemented especially since path cost is not factored in (only the heuristic).
- **A*:** Affords more space as it involves the storage of the path cost and the evaluation factor.

❖ Optimality:
- A* is complete under an admissible and consistent heuristic.
  On the other hand, GBFS cannot be said to be complete since such an approach, may lead to local optima and thus resting there

❖ Completeness:
- A* is said to be optimal in the case of using an admissible (does not estimate more than the actual lowest cost to reach a goal) heuristic.
- In the case of GBFS, it cannot be optimal because it can be quite greedy and take any path that deviates from the goal instead of the shortest one.

- <u>Local Search Algorithms:</u>
  ❖ How the algorithm works:

- **Hill Climbing:** A naive local search technique that progresses continually toward the steepest ascent (i.e., toward the target point). It is likely to become insipid to ever-evolving changes around it since the system may get to a local minimum.

- **Simulated Annealing:** A probabilistic order of searching space which in some instances permits movement towards higher costs in order to jump some local minima. There is a heating schedule incorporated in this method in order to melt the chances of making those worse moves when the search advances.

- **Genetic Algorithm:** It's a search algorithm based on population, which imitates the process of biological evolution and uses the operations selection, crossover, and mutation to change the solutions over several generations.

❖ Time:
- **Hill Climbing**: Typically O(n), where n is the number of states in the neighborhood.
- **Simulated Annealing**: Depends on the cooling schedule and the number of iterations.
- **Genetic Algorithm**: The complexity varies depending on the population size, mutation rate, and number of generations. Typically O(p * g), where p is the population size and g is the number of generations.

❖ Space:
- **Hill Climbing:** An algorithm with very low memory requirements - keeping only the current state and its neighbors.
- **Simulated Annealing:** A memory-intensive algorithm that keeps the current state as well as the history of the search when required.
- **Genetic Algorithm:** An algorithm that also uses extra memory for the population and evolved generations' maintenance.

❖ Completeness:
- **Hill Climbing:** Incomplete, may die in the local minima.
- **Simulated Annealing:** This can avoid dropping into local minima, however, it does not ensure that the global minimum will be located.

- **Genetic Algorithm:** It is complete but may take a long time depending on the parameters and the convergence criteria.

❖ Optimality:

- **Hill Climbing:** Non-optimal due to the reason that local minima might be encountered.
- **Simulated Annealing:** Near optimality can be reached, given enough time and a proper temperature schedule.
- **Genetic Algorithm:** Solutions are often reasonable but may necessitate many generations to attain that solution.

❖ **Summary of Results:**

- BFS and A-star are the optimal and complete algorithms, and this makes them appropriate for looking for the best path in the grid-based pathfinding problem. Greedy Best-First Search is more efficient in terms of time but less optimal and complete and often cannot provide the optimal path. Hill Climbing is effective yet it is prone to local minima, whilst Simulated Annealing and Genetic Algorithm can overcome such failure but are more costly in terms of time. A star can provide a good compromise between completeness, optimality, and time complexity.

Algorithms	Time	space	Completeness	Optimality
BFS	Time Complexity is O(b^d), where b is the theoretical maximum number of children of a node and d is the depth of the goal.	(O(b^d)) as it stores all nodes at a level	Always complete it and find a solution if one exists.	Optima, for unweighted graph and it find the shortest path
DFS	Worst case time complexity O(b^m), it can be inefficient for large or infinite search spaces as it may explore unnecessary paths.	Worst case space complexity O(bm), Memory-efficient compared to algorithms like Breadth-First Search.	Incomplete if the search space is infinite or if there are cycles without proper handling	Non-optimal. DFS does not inherently aim to find the best solution but rather the first solution it encounters.
UCS	Time complexity O(b^(1+(C/ε)), reflecting optimality by expanding lowest path cost	Inefficient space complexity O(b^(1+(C/ε)) because all frontier and explored nodes stored in memory	complete	Guarantees highest optimality between uninformed search algorithms
IDS	Time Complexity O(b^d)	Space Complexity O(bd), memory efficient algorithm as it saves only the current path	complete only if the search space is infinite	Guaranteed optimality if each step cost is represented by 1.
GBFS	Explores nodes based on h(n)	Uses less memory, focuses on nodes closest to the goal	Complete, but depends on heuristic	Not guaranteed

A*	Explores nodes based on f(n) = g(n) + h(n) ,so take time	Requires storing all generated nodes	Complete with an admissible heuristic	Guaranteed if the heuristic is admissible
Hill Climbing	Time depends on the number of states in the neighborhood and the speed of progressing toward the goal.	Has low space complexity as it only keeps the current state and its neighbors in memory.	In complete because it may stuck in a local minimum and fail to find the solution	Not optimal; it may miss the best path due to local optima.
Simulated Annealing	Time depends on the cooling schedule and the number of iterations required to converge to a solution.	has low space complexity but may need slightly more memory to track probabilities or search history.	Not guaranteed but has a higher chance of finding a solution than Hill Climbing as it allows occasional moves to worse states.	Not guaranteed but has a higher chance of finding a solution than Hill Climbing as it allows occasional moves to worse states.
genetic algorithm	Time complexity depends on the population size P, number of generations G, and the evaluation cost of the fitness function F. Typically, $O(P \cdot G \cdot F)$	Space complexity is O(P), where P is the population size. Higher memory consumption compared to DFS for large populations.	Incomplete by nature. It does not guarantee finding the exact or best solution, especially in finite time.	