Module 03

Introduction to Optimal Pathfinding

- Optimal pathfinding involves finding the most efficient route from a starting point to a goal state in a graph or grid.
- Efficiency is often measured in terms of distance, time, cost, etc.
- Different algorithms can be used to find optimal paths, each with its own strengths and weaknesses.

Branch and Bound

- Branch and Bound is a systematic method for finding optimal solutions to combinatorial optimization problems.
- It explores the search space by branching into different paths and uses bounding techniques to prune the search tree.
- In pathfinding, it can be used to find the shortest path by exploring promising paths and discarding those that are not optimal.
- It guarantees finding the optimal solution but may have high time complexity for large graphs.

A* Algorithm

- A* (pronounced "A star") is a popular heuristic search algorithm used for pathfinding and graph traversal.
- It uses both the cost to reach a node and an estimate of the cost from that node to the goal to guide the search.
- A* maintains two lists: open list (nodes to be evaluated) and closed list (nodes already evaluated).
- It selects nodes to expand based on a heuristic function, which estimates the cost from the current node to the goal.
- A* is guaranteed to find the optimal path if certain conditions are met (admissibility and consistency of the heuristic).

Algorithm

```
Initialization:

    Create an open list and a closed list. Initially, the open list contains only the starti
    Assign a cost of 0 to the starting node and an estimated cost (heuristic) to reach the g

Main Loop:

    While the open list is not empty:
        Select the node with the lowest total cost (sum of the path cost from the start node
        If the current node is the goal node, the path has been found. Terminate the search.
        Otherwise, move the current node from the open list to the closed list to mark it as
```

```
Expansion:

Expand the current node by considering all its neighboring nodes (successors).
    For each successor:
        Calculate the cost of reaching that successor from the start node (known as the g-
value).
        Calculate the estimated cost to reach the goal from the successor (known as the heu
value).
        Compute the total cost f(n) = g(n) + h(n) for the successor.
        If the successor is already in the open list and the new f-
value is lower, update its cost and parent.
        If the successor is not in the open list, add it to the open list with the computed

Termination:

    If the open list becomes empty and the goal has not been reached, there is no path from

Backtracking:

Once the goal is reached, reconstruct the path by backtracking from the goal node through th
```

Example

IDA Algorithm:*

- IDA* (Iterative Deepening A*) is a variant of the A* algorithm designed to address memory constraints.
- It avoids storing the entire search tree in memory by performing a depth-first search with iterative deepening.
- At each iteration, it limits the search depth based on a threshold, gradually increasing the threshold until the goal is found.
- IDA* guarantees finding the optimal solution like A* but uses less memory, making it suitable for memory-constrained environments.
- It sacrifices some efficiency for memory savings compared to A*, especially in domains with high branching factors.

```
The IDA* algorithm includes the following steps:

- Start with an initial cost limit.
    - The algorithm begins with an initial cost limit, which is usually set to the heuristi

- Perform a depth-first search (DFS) within the cost limit.
    - The algorithm performs a DFS search from the starting node until it reaches a node wi

- Check for the goal node.
    - If the goal node is found during the DFS search, the algorithm returns the optimal pa
- Update the cost limit.
    - If the goal node is not found during the DFS search, the algorithm updates the cost li

- Repeat the process until the goal is found.
    - The algorithm repeats the process, increasing the cost limit each time until the goal
```

Problem Decomposition

- Problem decomposition involves breaking down a complex problem into smaller, more manageable sub-problems or components. This technique allows for a systematic approach to understanding and solving the larger problem by focusing on its constituent parts. Here's how it typically works:
- *Identify the Problem:* Clearly define the problem you're trying to solve. This could be a technical issue, a business challenge, or any other complex situation.
- *Break it Down:* Analyze the problem and break it down into smaller, more manageable components. These components should be relatively independent and represent distinct aspects of the problem.
- *Hierarchical Structure:* Organize the components hierarchically, with higher-level components representing broader aspects of the problem and lower-level components representing finer details.

Goal Tree

- A goal tree is a hierarchical representation of goals and subgoals required to achieve a specific objective. It illustrates the relationships between different goals in a tree-like structure.
- Each node in the tree represents a goal, and the edges represent dependencies or relationships between goals. Goals at higher levels of the tree are typically broader, while goals at lower levels are more specific.
- *Example:* In a robotics project, the goal tree might include high-level goals like "Navigate to Destination" and lower-level goals like "Avoid Obstacles," "Find Optimal Path," and "Update Position."

AND-OR Graph

- An AND-OR graph is a graphical representation used to model the relationships between different actions or conditions in AI systems. It consists of nodes representing actions or conditions and edges representing dependencies or relationships.
- Nodes in an AND-OR graph can be labeled as either AND-nodes or OR-nodes. AND-nodes represent a logical AND relationship between their child nodes, meaning all child nodes must be satisfied. OR-nodes represent a logical OR relationship between their child nodes, meaning at least one child node must be satisfied.

- *Example:* In a planning problem, an AND-OR graph might represent different actions that can be taken to achieve a goal, with AND-nodes representing concurrent actions and OR-nodes representing alternative actions.

Algorithm 1: Pseudocode of AO* Algorithm

```
Data: Graph, StartNode
Result: The minimum cost path from StartNode to GoalNode
CurrentNode ← StartNode
while There is a new path with lower cost from StartNode to the
  GoalNode do
    calculate the cost of path from the current node to the goal
      node through each of its successor nodes;
    if the successor node is connected to other successor nodes by
      AND-ARCS then
        sum up the cost of all paths in the AND-ARC;
        return the total cost;
    else
        calculate the cost of the single path in the OR side;
        return the single cost;
    end
    find the minimum cost path
    CurrentNode ← SuccessorNodeOfMinimumCostPath
    if CurrentNode has no successor node then
        do the backpropagation and correct the estimated costs;
        CurrentNode ← StartNode
        return CurrentNode, New estimated costs;
    else
        return null;
    end
    return The minimum cost path;
end
```

AO* Algorithm
AO* Algorithm