1. Problem-Solving Agents

Definition:
A **problem-solving agent** is an AI system that **decides what actions to take** in order to achieve a specific goal. It **thinks before it acts**, unlike a simple reflex agent.

How it works:

1. It **defines the goal** clearly.
2. It **describes the current state** of the environment.
3. It **figures out possible actions** (operators) it can take.
4. It **searches** for a sequence of actions (called a plan) that leads to the goal.
5. It **executes** that plan.

Example:
Imagine a robot in a maze trying to reach the exit. It will:

- See where it is now.
- Decide where it wants to go (goal).
- Look at all paths (possible actions).
- Choose the best path (search).
- Move step-by-step until it reaches the exit.

2. Well-Defined Problems & Solutions

Well-defined problem:

A problem is **well-defined** when it is clearly and completely specified.

It has 5 key components:

1. **Initial state** – Where the agent starts.
2. **Actions (operators)** – What the agent can do.
3. **Transition model** – What happens when an action is taken.
4. **Goal test** – How to check if the goal is reached.
5. **Path cost** – How much it costs to take a path (e.g., steps, time).

> ✅ If a problem has all these, it's **well-defined**. If some parts are missing or vague, it's **ill-defined**.

Well-defined solution:

- A solution is a **sequence of actions** that transforms the initial state to a goal state.
- An **optimal solution** is the one with the **lowest path cost**.

Example:

Problem: Find a route from Kolkata to Delhi.

- Initial state: You are in Kolkata.
- Actions: Drive, take a train, or fly.
- Transition model: If you fly, you reach in 2 hrs; train takes 18 hrs.
- Goal test: You're in Delhi.
- Path cost: Time, money, or distance.

3. Formulating Problems

What it means:

Formulating a problem means **converting a real-world task into a well-defined problem** that an AI agent can solve.

Steps to formulate:

1. Identify the **initial state**.
2. Clearly define the **goal**.
3. List all **possible actions**.
4. Understand the **result of each action** (transition model).
5. Decide how to **test for the goal**.
6. Assign a **cost** to each action if needed.

Example (8-puzzle):

- Initial state: Random arrangement of 8 tiles.
- Goal: Arrange tiles in order (1–8).
- Actions: Move the blank tile up/down/left/right.
- Transition model: Each move changes the state.
- Goal test: Tiles are in order.
- Path cost: Number of moves taken.

Uninformed search algorithms is also known as blind search algorithms, are a class of search algorithms that do not use any domain-specific knowledge about the problem being solved.

- Uninformed search algorithms rely on the information provided in the problem definition, such as the initial state, actions available in each state, and the goal state.
- These are called "blind" because they do not have a heuristic function to guide the search towards the goal instead, they explore the search space systematically.
- Uninformed search algorithms provide basic search strategies for exploring problem spaces where no additional knowledge is available beyond the problem definition.
- These algorithms are important for solving a wide range of problems in AI, such as pathfinding, puzzle solving, and state-space search.

While these algorithms may not always be the most efficient, they provide a baseline for understanding and solving complex problems in AI.

Types of Uninformed Search Algorithms

1. Breadth-First Search (BFS)

Breadth-First Search explores all nodes at the current depth before moving to the next level. It uses a queue (FIFO) to keep track of nodes, ensuring that the shortest path is found in an unweighted graph.

Key Features:

- Guarantees the shortest path if the cost is uniform.
- Uses more memory as it stores all child nodes.\

2. Depth-First Search (DFS)

Depth-First Search (DFS) explores as far as possible along each path before backtracking. It uses a stack (LIFO) and is more memory-efficient than Breadth-First Search (BFS) but does not guarantee the shortest path.

Key Features:

- Efficient for deep exploration.
- Can get stuck in infinite loops if cycles exist.

Depth Limited Search for AI

Last Updated : 22 May, 2024

Depth Limited Search is a key algorithm used in the problem space among the strategies concerned with artificial intelligence. The article provides a comprehensive overview of the Depth-Limited Search (DLS) algorithm, explaining its concept, applications, and implementation in solving pathfinding problems in robotics, while also addressing frequently asked questions.

Introducing Depth Limited Search (DLS)

Depth Limited Search is a modified version of DFS that imposes a limit on the depth of the search. This means that the algorithm will only explore nodes up to a certain depth, effectively preventing it from going down excessively deep paths that are unlikely to lead to the goal. By setting a maximum depth limit, DLS aims to improve efficiency and ensure more manageable search times.

How Depth Limited Search Works

1. **Initialization**: Begin at the root node with a specified depth limit.
2. **Exploration**: Traverse the tree or graph, exploring each node's children.
3. **Depth Check**: If the current depth exceeds the set limit, stop exploring that path and backtrack.
4. **Goal Check**: If the goal node is found within the depth limit, the search is successful.
5. **Backtracking**: If the search reaches the depth limit or a leaf node without finding the goal, backtrack and explore other branches.

Applications of Depth Limited Search in AI

1. **Pathfinding in Robotics:** DLS is employed for nonholonomic motion planning of robots in the presence of obstacles. By imposing restriction on the depth it makes the robot stop after exploring a particular depth of an area and restricting the robot from too much wandering.
2. **Network Routing Algorithms:** A DLS can be implemented to compute paths between nodes in computer networks restricting the number of hops to prevent loops.
3. **Puzzle Solving in AI Systems:** DLS can be used to solve puzzles such as the 8-puzzle or Sudoku by manipulating possible moves a fixed number of times that reduces how many steps are taken in the search.
4. **Game Playing:** In AI for games, instead, DLS can be used to plan forward a few moves up to a certain level of depth to help decide how much effort to put into a given decision.

Applications of Depth Limited Search in AI

1. **Pathfinding in Robotics:** DLS is employed for nonholonomic motion planning of robots in the presence of obstacles. By imposing restriction on the depth it makes the robot stop after exploring a particular depth of an area and restricting the robot from too much wandering.
2. **Network Routing Algorithms:** A DLS can be implemented to compute paths between nodes in computer networks restricting the number of hops to prevent loops.
3. **Puzzle Solving in AI Systems:** DLS can be used to solve puzzles such as the 8-puzzle or Sudoku by manipulating possible moves a fixed number of times that reduces how many steps are taken in the search.
4. **Game Playing:** In AI for games, instead, DLS can be used to plan forward a few moves up to a certain level of depth to help decide how much effort to put into a given decision.

Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS)

Last Updated : 20 Sep, 2024

There are two common ways to traverse a graph, BFS and DFS. Considering a Tree (or Graph) of huge height and width, both BFS and DFS are not very efficient due to following reasons.

1. **DFS** first traverses nodes going through one adjacent of root, then next adjacent. The problem with this approach is, if there is a node close to root, but not in first few subtrees explored by DFS, then DFS reaches that node very late. Also, DFS may not find shortest path to a node (in terms of number of edges).
2. **BFS** goes level by level, but requires more space. The space required by DFS is O(d) where d is depth of tree, but space required by BFS is O(n) where n is number of nodes in tree (Why? Note that the last level of tree can have around n/2 nodes and second last level n/4 nodes and in BFS we need to have every level one by one in queue).

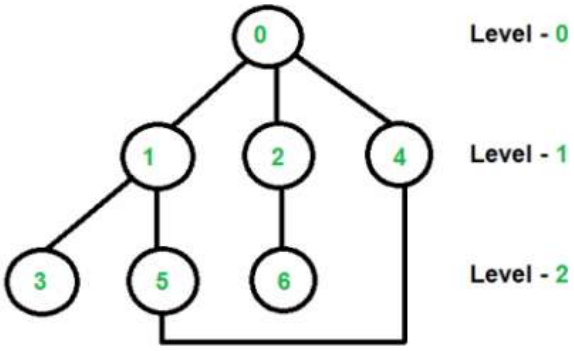
IDDFS combines depth-first search's space-efficiency and breadth-first search's fast search (for nodes closer to root).

How does IDDFS work?

IDDFS calls DFS for different depths starting from an initial value. In every call, DFS is restricted from going beyond given depth. So basically we do DFS in a BFS fashion.

An important thing to note is, we visit top level nodes multiple times. The last (or max depth) level is visited once, second last level is visited twice, and so on. It may seem expensive, but it turns out to be not so costly, since in a tree most of the nodes are in the bottom level. So it does not matter much if the upper levels are visited multiple times. Below is implementation of above algorithm

Depth	Iterative Deepening Depth First Search
0	0
1	0 1 2 4
2	0 1 3 5 2 6 4 5
3	0 1 3 5 4 2 6 4 5 1



The explanation of the above pattern is left to the readers.

A comparison table between DFS, BFS and IDDFS

	Time Complexity	Space Complexity	When to Use ?
DFS	$O(b^d)$	O (d)	=> Don't care if the answer is closest to the starting vertex/root. => When graph/tree is not very big/infinite.
BFS	$O(b^d)$	$O(b^d)$	=> When space is not an issue => When we do care/want the closest answer to the root.
IDDFS	$O(b^d)$	O(bd)	=> You want a BFS, you don't have enough memory, and somewhat slower performance is accepted. In short, you want a BFS + DFS.