AI-Module-2-Important-Topics

For more notes visit

https://rtpnotes.vercel.app

Playlist to refer

https://youtube.com/playlist?list=PLnzz0gSUYIN3mo-LB-l2Vfadz6HiPi8Po&feature=shared

1. Well Defined Problem

A well-defined problem is described using the following components:

1. **Initial State**
   - The starting point or condition where the agent begins.
     Example: Starting in Ernakulam.
2. **Actions**
   - The set of possible moves or decisions the agent can take from a given state.
     - For a specific state `s`, `ACTIONS(s)` returns all actions applicable in that state.
       Example: From Ernakulam, the actions might be:
       - Go to Thrissur
       - Go to Palakkad
       - Go to Kozhikode
3. **Transition Model**
   - A description of how each action changes the state, defined by a function `RESULT(s, a)` that returns the new state after taking action `a` in state `s`.

Example: If the agent is in Ernakulam and takes the action `Go to Thrissur`, the result is the state "In Thrissur."

4. **Successor State**
   - Any state that can be reached from a given state by performing one action.
     Example: The successor of Ernakulam after `Go to Thrissur` is Thrissur.
5. **State Space**
   - The set of all possible states that can be reached from the initial state through any sequence of actions.
   - It forms a graph where:
     - **Nodes** represent states.
     - **Links** represent actions between states.
6. **Path in State Space**
   - A sequence of states connected by actions.
     Example: Ernakulam → Thrissur → Palakkad.
7. **Goal Test**
   - A check to determine if the agent has reached the desired goal state.
     Example: The goal state might be "In Chennai."
8. **Path Cost Function**
   - A numeric value representing the total cost of a path, calculated by summing up the costs of individual actions along the path.
     - **Step Cost:** The cost of taking a specific action `a` in state `s` to reach state `s'`, denoted as `c(s, a, s')`.
9. **Solution**
   - A sequence of actions that leads from the initial state to the goal state.
     - The quality of a solution is measured by its **path cost**.
     - An **optimal solution** is one with the lowest path cost among all possible solutions.

This structured approach ensures that problems are clearly defined and can be solved systematically.

Example: Traveling in Romania

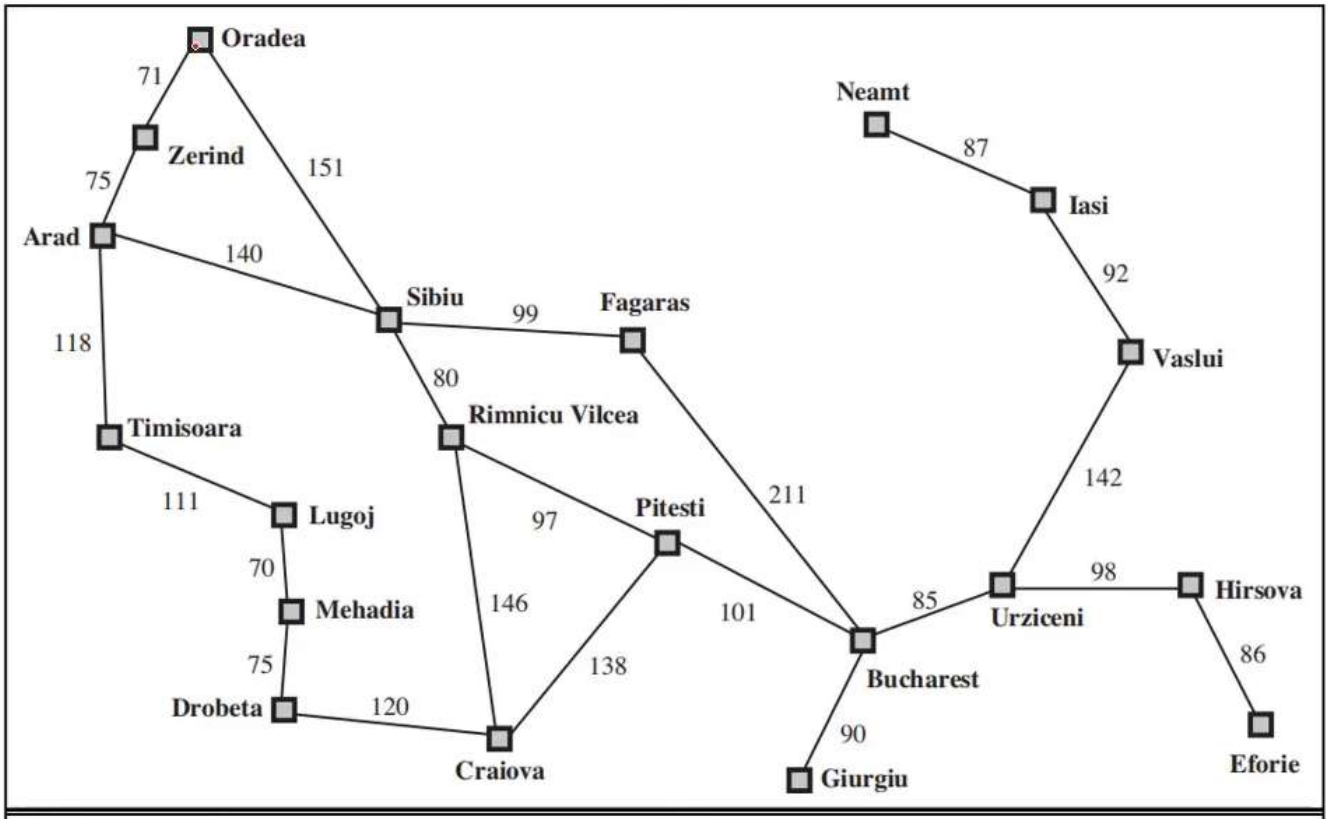


Figure 3.2 A simplified road map of part of Romania.

You are on vacation in Romania, currently in **Arad**, and need to catch a flight from **Bucharest** tomorrow.

Steps to Solve the Problem

1. **Problem:**
   - Find a way to get from **Arad** to **Bucharest**.
2. **Initial State:**
   - Start in **Arad**.
3. **Actions (Operators):**
   - Drive from one city to another.
4. **State Space:**
   - The cities you can visit, e.g., {Sibiu, Fagaras, Timisoara}.
5. **Goal Test:**
   - Check if you are in **Bucharest**.
6. **Path Cost Function:**
   - Use a map to calculate the total driving cost (e.g., time, distance, or fuel).
7. **Solution:**

- A sequence of cities that gets you to Bucharest.
  - Example paths:
    - **Arad → Sibiu → Fagaras → Bucharest**
    - **Arad → Timisoara → Lugoj → Mehadia → Craiova → Pitesti → Bucharest**

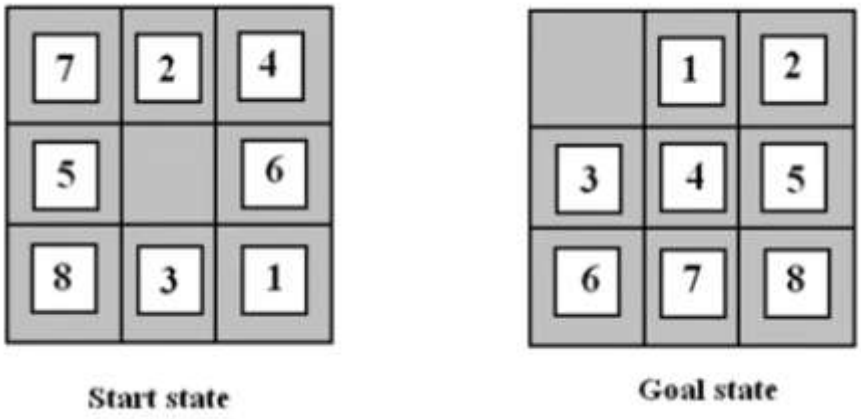
8. **State Space Map:**
   - Shows possible routes and cities:
     - Example: **{Arad → Sibiu → Fagaras → Bucharest}**

Single state problem formulation

A problem is defined using **four key elements**:

1. **Initial State**
   - The starting point.
     Example: "At Arad."
2. **Actions (Successor Function)**
   - Defines possible actions and resulting states from a given state.
     - Represented as `S(x)` = set of action-state pairs.
       Example: From Arad:
       - `<Arad → Zerind, Zerind>`
       - `<Arad → Sibiu, Sibiu>`
3. **Goal Test**
   - Determines if the goal has been achieved.
     - **Explicit Goal:** Direct condition like "At Bucharest."
     - **Implicit Goal:** Based on specific rules, like "Checkmate" in chess.
4. **Path Cost (Additive)**
   - The total cost of reaching a goal, calculated by summing step costs.
     - **Step Cost:** The cost of taking a specific action, denoted as `c(x, a, y)` and assumed to be ≥ 0.
       Example: Distance traveled or number of moves made.

2. 8 Puzzle problem



The **8-puzzle** is a sliding puzzle where tiles move into a blank space to reach a specific goal arrangement.

1. **States:**
   - Each state shows where all 8 tiles and the blank space are located in the 3x3 grid.
2. **Initial State:**
   - Any arrangement of tiles can be the starting point.
3. **Actions:**
   - Move the blank space **Left**, **Right**, **Up**, or **Down**.
   - Some moves may not be possible depending on the blank's position.
4. **Transition Model:**
   - Defines what happens after an action.
     - Example: If the blank is moved **Left**, the tile next to it switches places with the blank.
5. **Goal Test:**
   - Checks if the current state matches the goal arrangement of tiles.
6. **Path Cost:**
   - Each move costs **1** step, so the total cost is the number of moves taken to solve the puzzle

3. Uninformed search strategies

- Uninformed search strategies don't have extra information about the problem or goal beyond what is given in the problem description.

- These strategies only generate new states (successors) and check if they are the goal. They don’t know where the goal is, so they explore blindly until they find it.
- They **don't** use any extra clues about where the goal might be.
- They just keep exploring different paths until they find a solution.

1. Breadth-First Search

How BFS Works:

1. Start from the **root node**.
2. Expand all its **direct successors** (children).
3. Move to the next level and expand their successors, continuing level by level.
4. Repeat until the goal is found.

Key Points:

- **Explores Level by Level:**
  BFS expands all nodes at one level before moving to the next.
- **FIFO Queue (First In, First Out):**
  - Nodes are added to the back of the queue as they are discovered (new nodes).
  - Older nodes (shallower) are expanded first.
- **Shallowest Path First:**
  BFS always finds the shortest path (in terms of levels) to any node because it processes nodes in order of their depth.
- **Goal Test on Generation:**
  - The goal check is done **when a node is created**, not when it’s expanded.

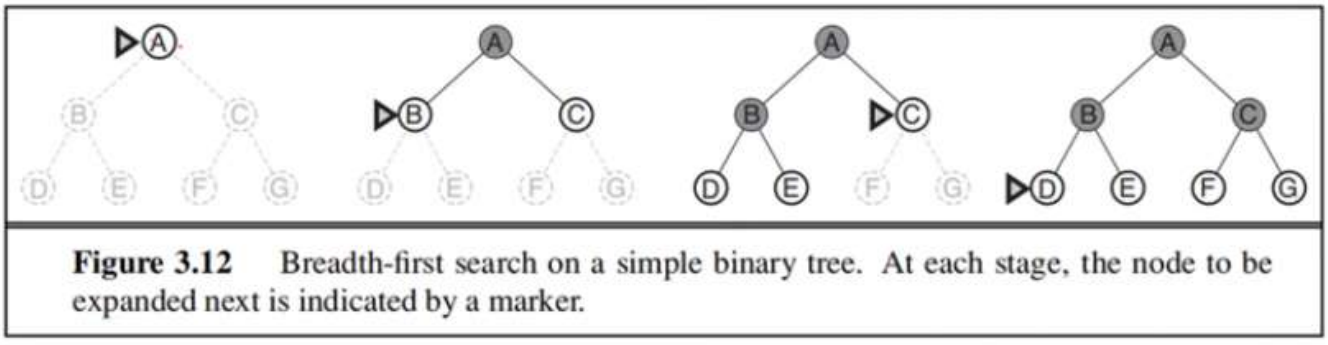


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

2. Uniform Cost Search

- Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost g(n). This is done by storing the frontier as a priority queue ordered by g.
- The goal test is applied to a node when it is selected for expansion because the first goal node that is generate may be on a suboptimal path
- A test is added in case a better path is found to a node currently on frontier

Example: Finding the Path from Sibiu to Bucharest

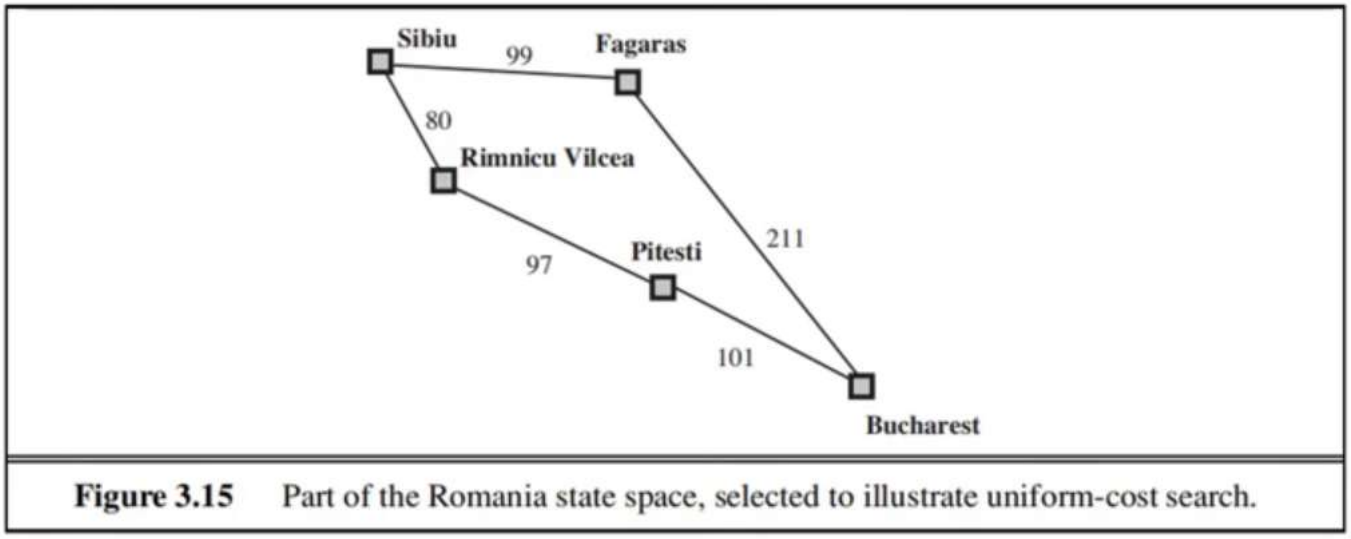


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

1. **Starting Point:**
   - The goal is to travel from **Sibiu** to **Bucharest**.
2. **Expanding Sibiu:**
   - Successors of **Sibiu** are:
     - **Rimnicu Vilcea** (cost: 80)
     - **Fagaras** (cost: 99)
3. **Expanding the Least-Cost Node (Rimnicu Vilcea):**
   - Expand **Rimnicu Vilcea** (cost: 80).
   - Add **Pitesti** as a successor with a total cost of **80 + 97 = 177**.
4. **Expanding the Next Least-Cost Node (Fagaras):**
   - Expand **Fagaras** (cost: 99).
   - Add **Bucharest** as a successor with a total cost of **99 + 211 = 310**.
5. **Finding the Goal Node:**

- The goal node (**Bucharest**) is found, but the algorithm continues to search for a better path.

6. **Expanding Pitesti:**
   - Expand **Pitesti** (cost: 177).
   - Add another path to **Bucharest** with a total cost of **80 + 97 + 101 = 278**.
7. **Comparing Paths:**
   - Compare the new path to **Bucharest** (cost: 278) with the previous path (cost: 310).
   - The new path is cheaper, so the old path is discarded.
8. **Selecting the Best Path:**
   - **Bucharest** with a cost of **278** is selected, and the solution is returned.

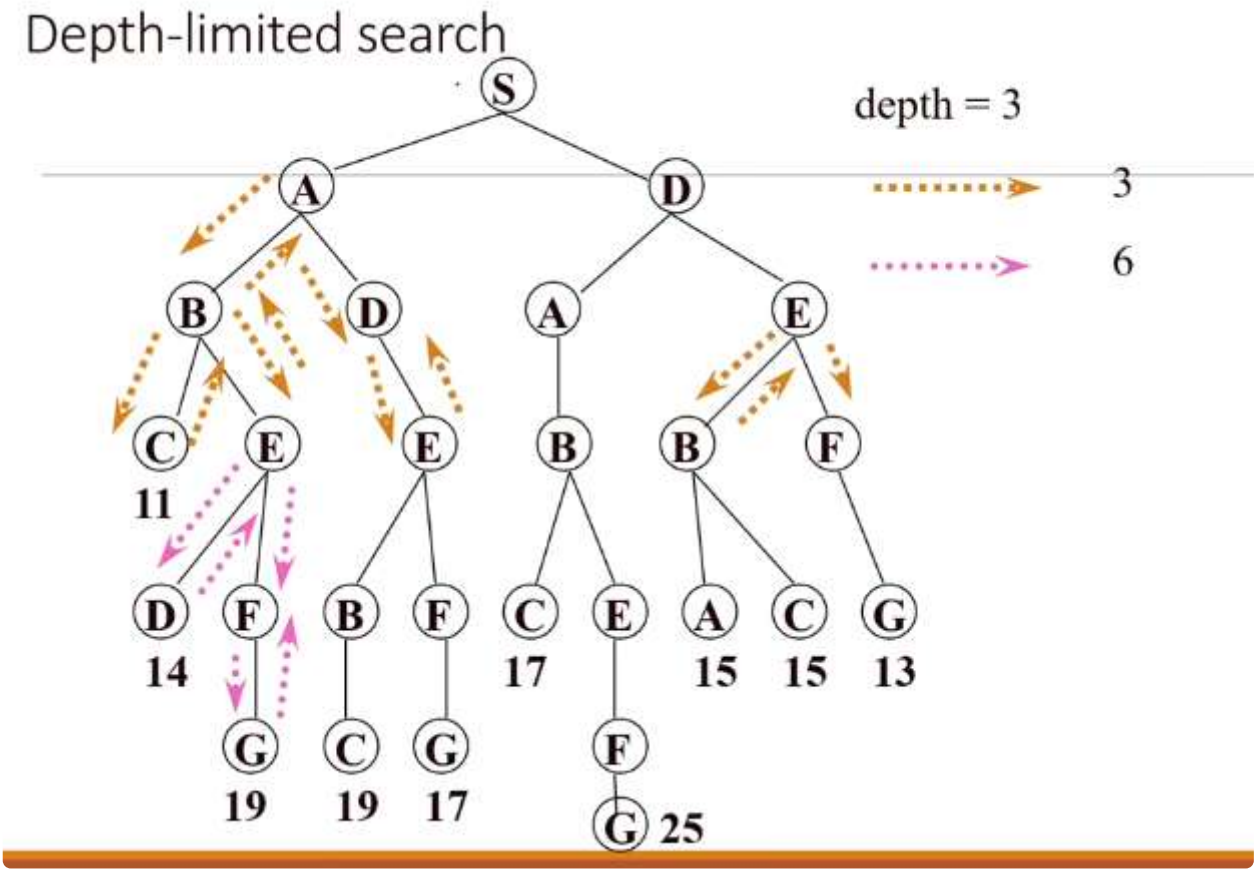
3. Depth Limited Search

Depth-Limited Search is like Depth-First Search but stops when it reaches a predefined depth limit (**L**).

Key Features:

1. **Predetermined Depth Limit:**
   - Nodes beyond the depth limit are **not explored**.
   - Helps prevent getting stuck in **infinite paths**.
2. **Potential Issues:**
   - If the depth limit (**L**) is **too small**:
     - The goal might not be reachable.
   - If the depth limit (**L**) is **too large**:
     - Similar problems as regular Depth-First Search (e.g., inefficiency).
3. **Special Case:**
   - Regular Depth-First Search is a specific case of DLS where the limit (**L**) is set to **infinity**.

Example



- Let set the depth limit as 3.
- We got from S->A->B->C
  - The depth is now 3, stopping here, and backtracking
- C-> B ->E
  - Depth is 3 again, backtracking
- E->B->A->D->E
  - Depth is 3 again, backtracking
- And it goes on

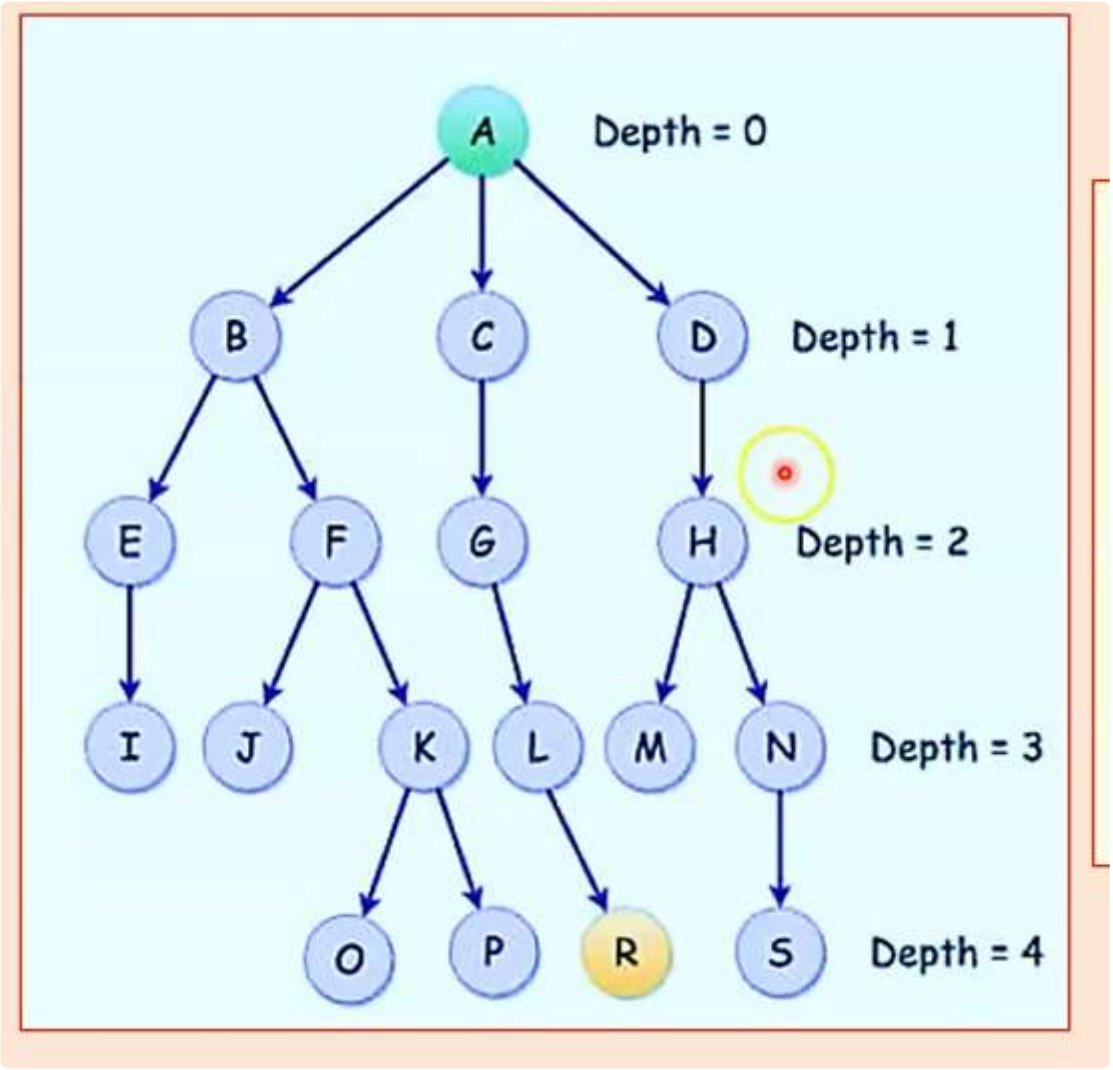
4. Iterative deepening

Iterative Deepening Depth-First Search (IDDFS) is a search algorithm that combines the strengths of both Depth-First Search (DFS) and Breadth-First Search (BFS).

How it works:

- IDDFS works by performing a series of Depth-First Searches, each with an increasing "depth limit."
- It starts with a shallow search and gradually goes deeper until it finds the goal (or solution).
- For each iteration, it performs DFS up to a certain depth. If the goal isn't found, it increases the depth and repeats the search.
- **Where it's helpful:**
  - It's useful in situations where the search space is large, and we don't know how deep the solution is (for example, in game trees or puzzles).
- **Advantages:**
  - **Memory-efficient** like DFS.
  - **Fast to find a solution** like BFS.
- **Disadvantages:**
  - It **repeats some work** from previous iterations, since every time it increases the depth limit, it has to search the same nodes as before.
- **Time complexity:**
  - If the branching factor is `b` (how many children each node has), and `d` is the depth of the goal node, the time complexity is around `O(b^d)`.

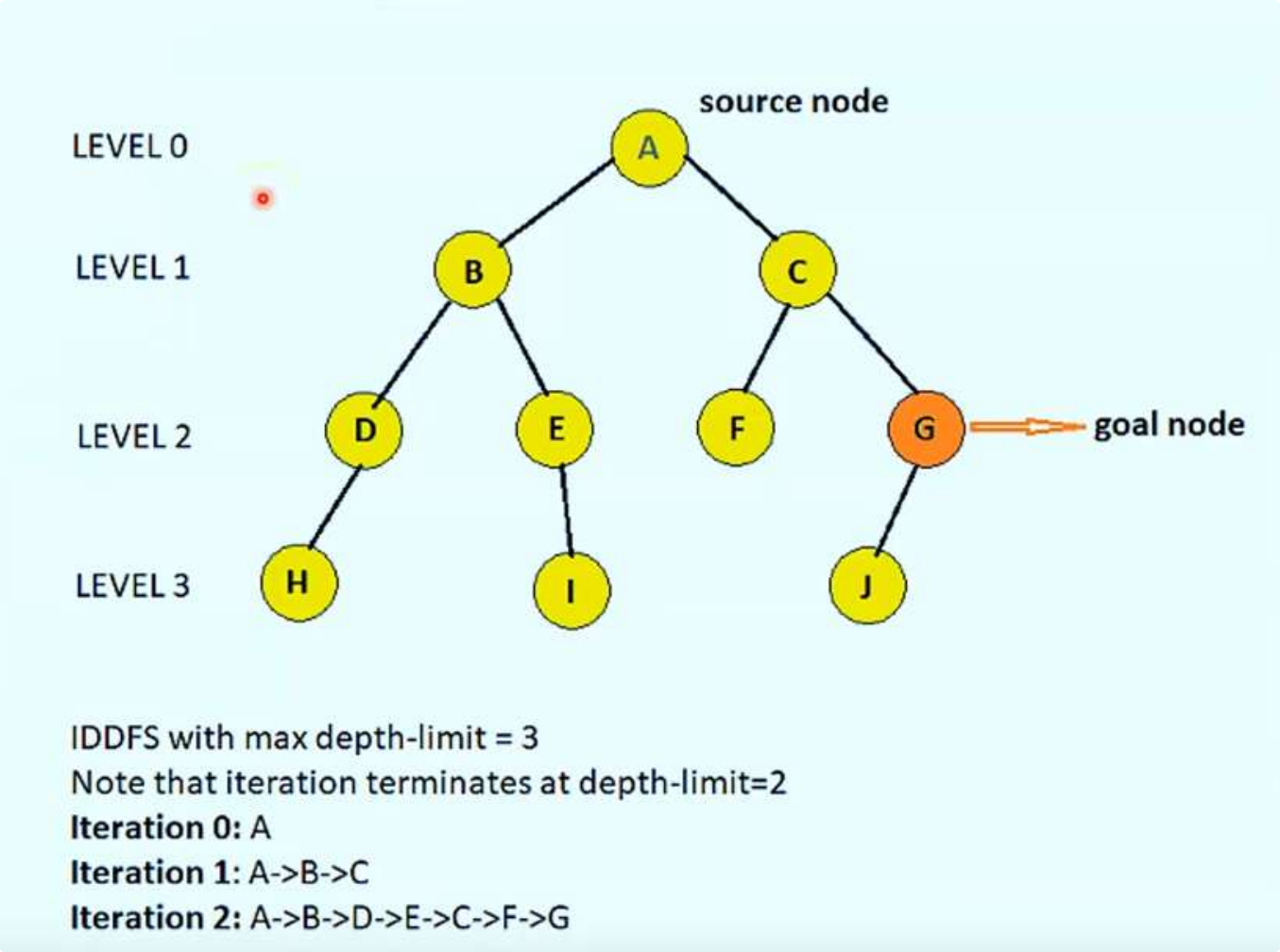
Example 1



- Our goal is to reach R
- As you can see there are different levels of depth, like depth = 0, depth = 1 ... depth = 4
- We will traverse this tree by each depth level
- Depth level 0
  - Only A is there
  - A
- Depth level 1
  - We will go from Left to right and go upto the depth level
  - A,B,C,D
- Depth Level 2
  - A,B,E,F,C,G,D,H
- Depth Level 3
  - A,B,E,I,F,J,K,C,G,L,D,H,M,N
- Depth Level 4

- A,B,E,I,F,J,K,O,P,C,G,L,R,D,H,M,N,S

Example 2



4. Informed search or heuristic search

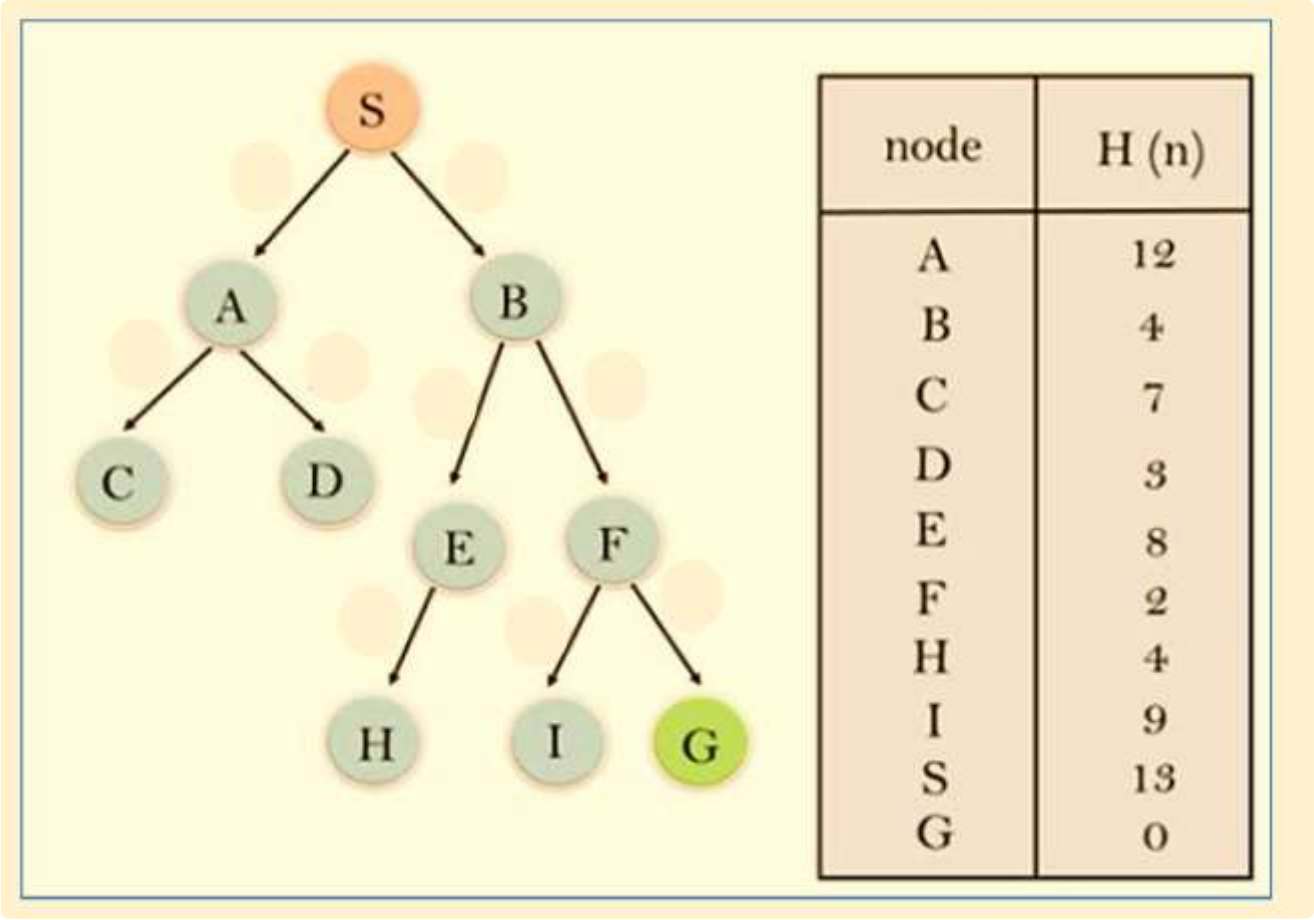
Problem specific knowledge beyond the definition of the problem itself, can find solutions more efficiently than an uninformed strategy

Best-First Search (Greedy Search)

- It tries to combine the strengths of two common search strategies: **depth-first search** (which goes deep into one path) and **breadth-first search** (which explores all options at the same level first).
- The algorithm uses the heuristic function to pick the "best" option at every step.
- It expands the node that seems closest to the goal based on the heuristic value.

- For example, **if you're navigating a city and you think a particular road seems fastest, you'll take that road even if it may not lead to the goal directly.**

Example



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

- Here you can see the nodes (A,B,C ...G) and their heuristic values (12,4,7 ... 0)
- **Our goal is to reach from S to G.**
- First we start from S
  - Here A and B are successor nodes
  - We visited S, so we put it to closed List
  - We discovered 2 successor Nodes A and B, so adding them to Open List
  - Open List = [A,B]
  - Closed List = [S]
- From A and B, the one with smaller heuristic value is B (Its value is 4)
  - We are visiting B, so modifying the lists
  - Open List = [A]
  - Closed List = [S,B]
- Visiting B

    - The successor nodes are E and F
    - Open List = [A,E,F]
    - Closed List = [S,B]
- From E and F, the one with smaller heuristic value is F (Its value is 2)
    - Visiting F
    - Open List = [A,E]
    - Closed List = [S,B,F]
- Visiting F
    - The successor nodes are I and G
    - Open List = [A,E,I,G]
    - Closed List = [S,B,F]
- Visiting G (Heuristic is 0 and its our goal)
    - Open List = [A,E,I,G]
    - Closed List = [S,B,F,G]
- So our final solution path is
    - S -> B -> F -> G

A* Algorithm

- A* (A-star) algorithm is a smart way to find the shortest path between two points. Think of it like navigating a map to find the best route from your house to a friend's place
- But instead of just checking the distance, you also consider how easy or difficult it is to walk along the path.

Key Concepts:

1. **g(n)**: This is the actual cost to reach a point (node) from the starting position.
2. **h(n)**: This is the estimated cost to reach the destination (goal) from that point (node). It’s like a guess based on how far the destination might be.
3. **f(n) = g(n) + h(n)**: A* combines both the actual cost to reach a point and the estimated cost to reach the goal from that point.

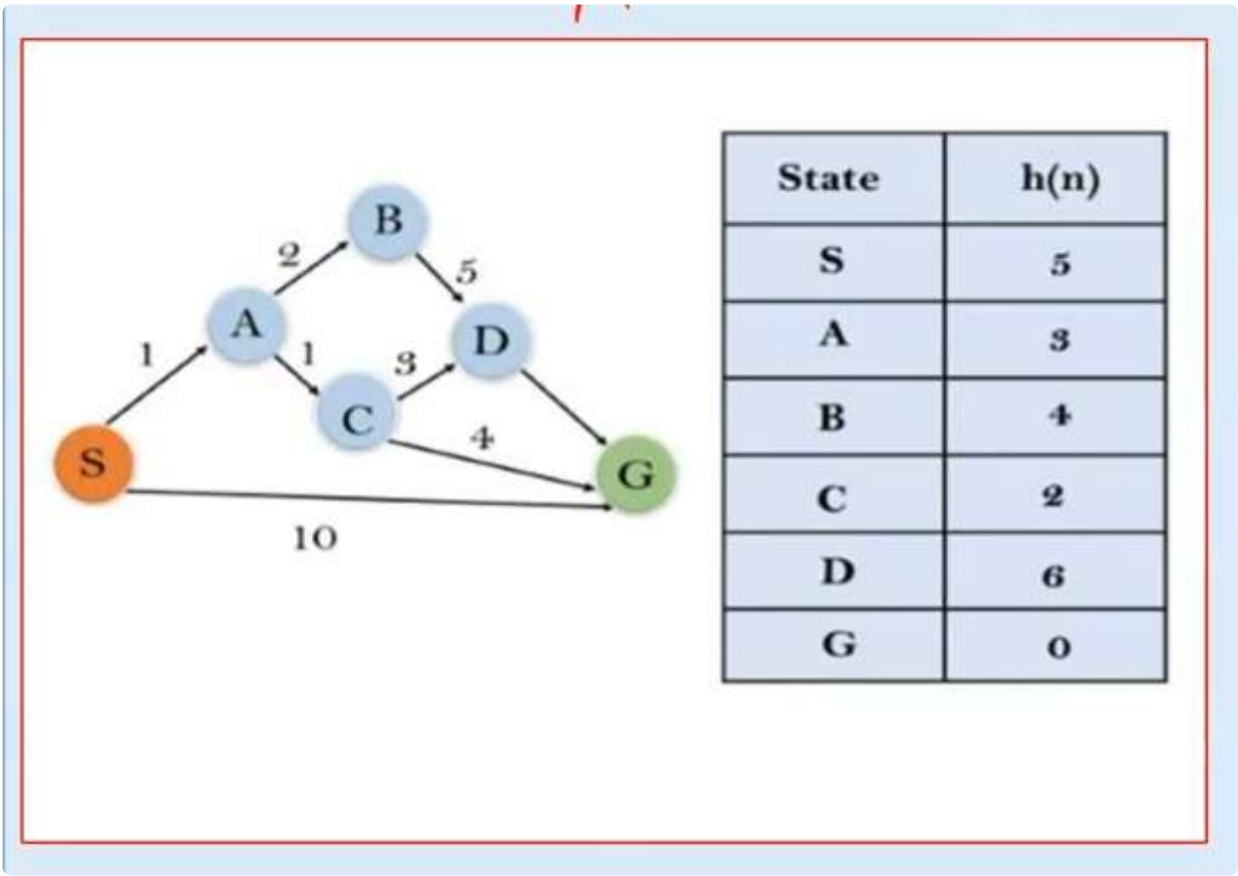
How it Works:

- The algorithm checks all possible routes, but it tries to be smart by expanding the paths that seem most promising. It does this by looking at the sum of the actual distance traveled (g) and the estimated remaining distance (h).
- A* explores the path with the **smallest f(n)**, which balances the known distance and the guess for the remaining distance.

Steps of the Algorithm:

1. Start at the initial point and place it in a list called the **OPEN list** (the list of nodes to be explored).
2. Pick the node from the OPEN list with the smallest f(n) value.
3. If this node is the goal, you're done! If not, generate all possible next steps (successor nodes) from this point.
4. Add these new points to the OPEN list to explore later.
5. Move the current node to a **CLOSED list** (so you don't check it again).
6. Repeat until you find the goal or no more points can be explored.

Example



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

- Our goal is to go from S to G

    - From S, there are 2 paths
    - S -> A
        - We need to calculate f(n)
        - **f(n) = g(n) + h(n)**
        - g(n) is the distance from S to A (From the graph we can see that the distance is 1)
        - h(n) is the heuristic value (The h(n) value of A is 3 from the table)
        - f(n) = 1+3 = 4
    - S -> G
        - **f(n) = g(n) + h(n)**
        - f(n) = 10+0= 10
    - When comparing f(n) of A and G, A is smaller with the value of 4. So we will check the nodes from A
- From A we have 2 paths
    - S -> A -> B
        - **f(n) = g(n) + h(n)**
        - f(n) = 3 + 4 = 7 (We got that 3 from S-> A = 1 and A->B = 2, S->A->B = 1+2 = 3)
    - S -> A -> C
        - f(n) = 2 + 2 = 4
    - f(n) of C is smaller
- From C we have 2 paths
    - S -> A -> C -> D
        - f(n) = 5 + 6 = 11
    - S -> A -> C -> G
        - f(n) = 6 + 0 = 6
    - f(n) of G is smaller.
- Since we reached G, we need to compare with the earlier nodes to see if they are smaller
    - S -> G f(n) = 10
    - S -> A -> B f(n) = 7
    - S -> A -> C -> G f(n) = 6
- S -> A -> C -> G f(n) = 6 is the smallest
- So our best path is
    - S -> A -> C -> G

Advantages

- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

Disadvantages

• It does not always produce the shortest path as it mostly based on heuristics and approximation.
• A *search algorithm has some complexity issues.*
• *The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

5. Different Heuristic Functions

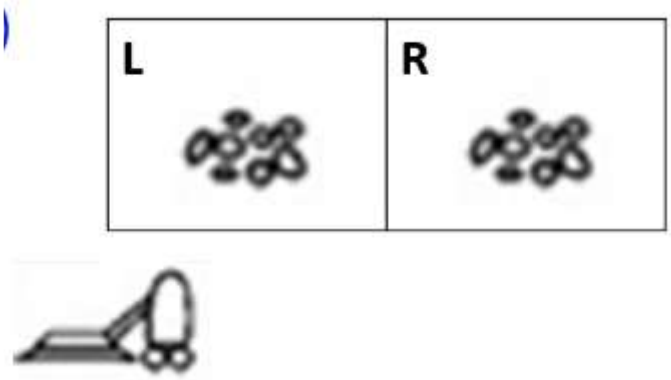
Admissible Heuristic

- An admissible heuristic is a function that estimates the cost from the current state to the goal, but it never overestimates this cost.
- In other words, it is "optimistic." For example, if you’re trying to find the shortest path to a destination, an admissible heuristic will always **guess either the exact distance or something less, but never more.**
- Think of it like planning a road trip: if your GPS tells you the trip will take no more than 5 hours, you know you’ll either arrive in exactly 5 hours or sooner—but not later.

Consistent Heuristic (Monotonic)

- A consistent heuristic, has an additional property: the estimated cost to reach the goal must always be less than or equal to the estimated cost from the current node to a neighbor, plus the actual cost to move from the current node to that neighbor.

6. Vaccum World Problem



- There are two dirty square blocks and we need to clean these dirty square blocks by using a vaccum cleaner
- Percepts: Location and Content
  - Eg (L,Dirty)
- Possible actions for vacuum cleaner
  - Move Left (L)
  - Move Right (R)
  - Suck (S)
  - No Operation (NoOp)

Initial State

- Vacuum Agent can be in any state(Left or Right)

Successor function

- Successor function generates legal states resulting from applying the 3 actions {Left, RIght and Suck}

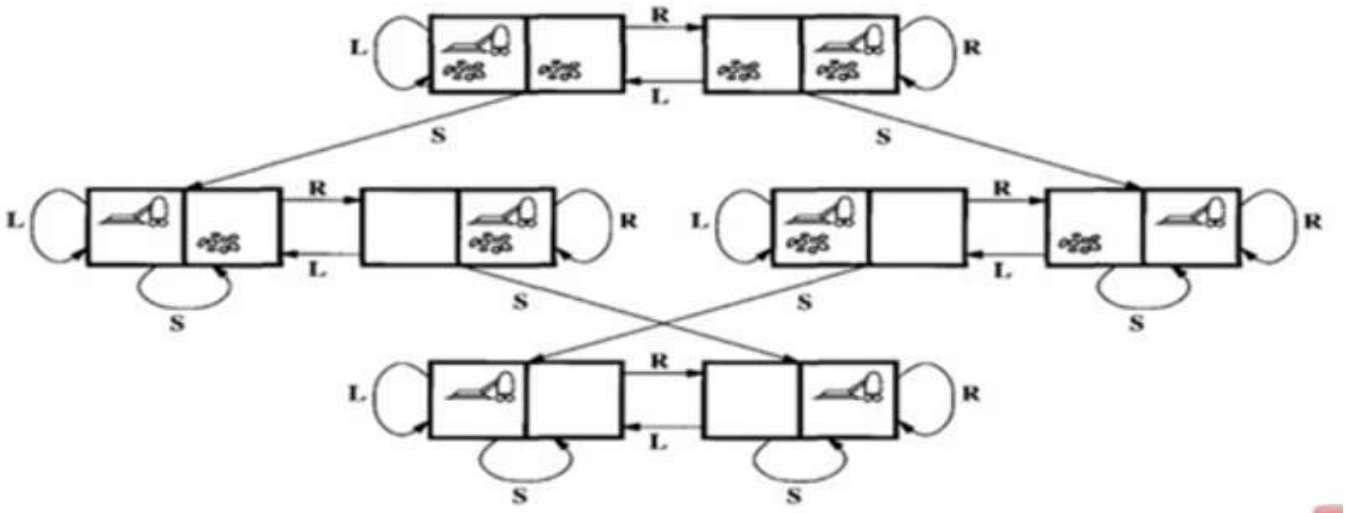
Goal Test

- Checks whether all squares are clean

Path Cost

- Each step costs 1, so the path cost is the sum of steps in the path from Initial state to goal state

Transition Diagram



- First 2 boxes are dirty
  - We can do the following operations
    - Suck the dirt
    - Move left
    - Move right
  - Suppose we Suck the dirt
    - Now one box is clean
    - We can move right
  - Moving Right
    - There is dirt on the second box, which can be sucked
  - After sucking the dirt, we have 2 clean squares
- Since we did this in 3 steps, the path cost is 3