

UNIT – 1

Introduction to AI: Intelligent Agents, problem-Solving Agents, Searching for Solutions. **Uninformed Search Strategies:** Breadth-first search, Uniform cost search, Depth-first search, Iterative deepening Depth-first search, Bidirectional search.

Informed (Heuristic) Search Strategies: Greedy best-first search, A* search, Heuristic Functions,

Beyond Classical Search: Hill-climbing search, Simulated annealing search, Local Search in Continuous Spaces

Intelligent Agents

An **intelligent agent** is an entity that perceives its environment, processes that information, and takes actions to achieve specific goals or objectives. Intelligent agents are commonly found in the fields of artificial intelligence (AI), robotics, and autonomous systems..

There are a few key components that define an intelligent agent:

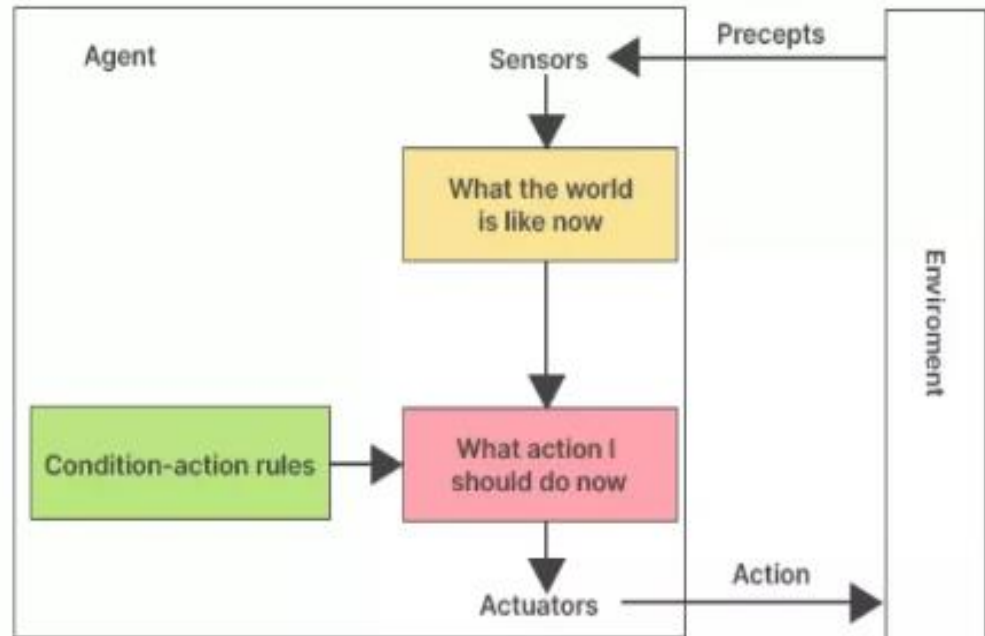
1. **Perception:** The ability to gather information about the environment through sensors or other means (e.g., cameras, microphones, or input data).
2. **Reasoning:** The agent uses algorithms or models to process the information it gathers and make decisions about what actions to take. This can involve learning from past experiences or making predictions about future outcomes.
3. **Action:** Based on its reasoning, the agent performs actions to influence or interact with its environment. These actions can be physical (e.g., moving a robot) or digital (e.g., sending an email, interacting with software).
4. **Autonomy:** Intelligent agents can operate without constant human intervention. They often make decisions based on pre-defined rules, machine learning models, or adaptive algorithms.
5. **Goal-Oriented:** Intelligent agents typically have a goal or a set of goals they work towards. These goals could be predefined by a human or could evolve as the agent learns and adapts over time.

Types of Intelligent Agents:

- **Simple Reflex Agents:** These agents make decisions based on the current state of the environment, using a set of predefined rules (e.g., if condition X is true, do action Y).
- **Model-Based Reflex Agents:** These agents build a model of the environment to account for partial knowledge, making their actions more adaptable.
- **Goal-Based Agents:** These agents take actions based on the pursuit of specific goals and can plan and adapt their actions accordingly.
- **Utility-Based Agents:** These agents aim to maximize a utility function, selecting actions that provide the highest value or satisfaction based on their goals.

- **Learning Agents:** These agents can improve their performance by learning from experience and adapting their behavior over time.

Simple Reflex Agents:



A **simple reflex agent** is a type of intelligent agent that makes decisions and performs actions based solely on the current state of the environment, using a set of predefined rules. This kind of agent doesn't maintain an internal model of the environment or keep track of past states.

Characteristics of Simple Reflex Agents:

1. **Perception:** It observes the environment and gathers information only about the current state.
2. **Action:** Based on the current state, the agent takes a predefined action as defined by a rule or condition.
3. **No Memory:** The agent doesn't remember past states or actions. It only reacts to what's happening right now.
4. **Rule-Based:** The agent operates based on a set of condition-action rules, also known as "production rules" or "if-then" rules. If a specific condition is met, the agent performs an associated action.

Structure of a Simple Reflex Agent:

The typical structure of a simple reflex agent consists of:

1. **Sensors:** These gather information about the current environment.
2. **Condition-Action Rules (Production Rules):** These are predefined rules that specify what actions to take in response to certain conditions or states.

3. **Actuators:** The components that carry out the action in the environment based on the decision made.

Example of a Simple Reflex Agent:

A vacuum cleaning robot designed to clean a room. It might follow these rules:

- **If the current tile is dirty, then clean the tile.**
- **If the current tile is clean, then move to the next tile.**

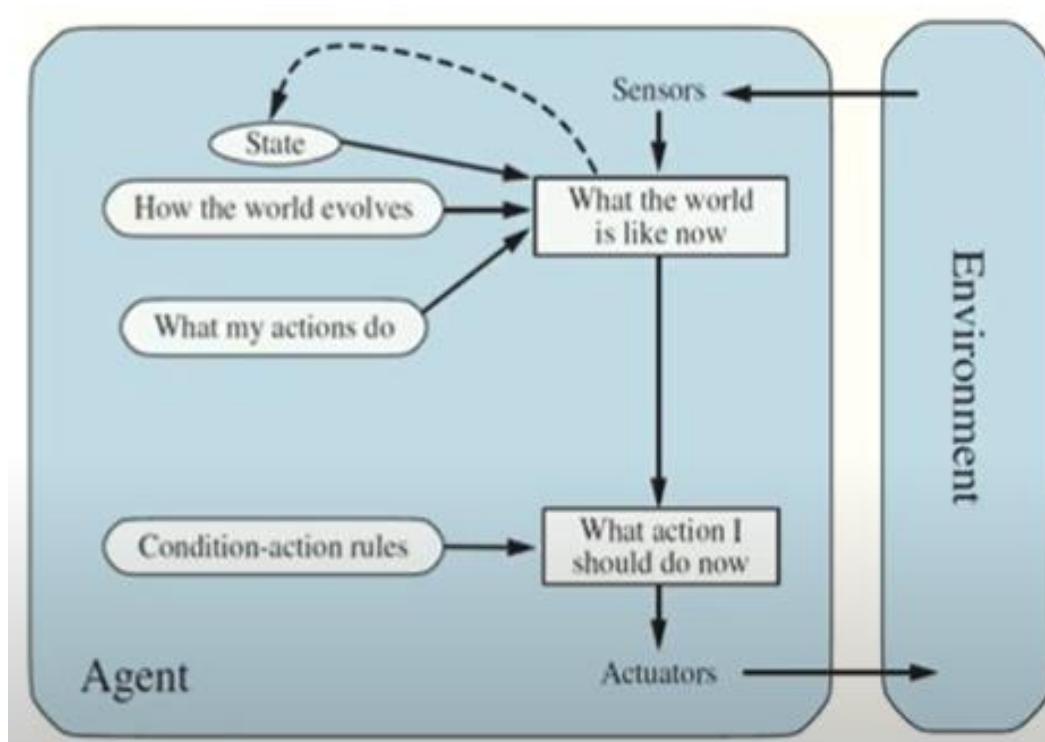
Advantages:

- **Simplicity:** Simple reflex agents are easy to design and implement since they don't require sophisticated algorithms or memory management.
- **Efficiency in Simple Environments:** They work well in environments that are predictable and don't require long-term planning or complex decision-making.

Limitations:

- **Limited Flexibility:** Simple reflex agents can only respond to the current situation, so they may not handle unexpected events or complex environments well.
- **No Learning:** They don't learn from past experiences, which means they can't improve their behavior over time.
- **No Consideration of History:** They don't maintain any memory or history of previous states, so they can't adapt to patterns or changes over time.

Model-Based Reflex Agents :



A **model-based agent** is an intelligent system that uses an internal model (a kind of "map") of the environment to make better decisions. Unlike simple reflex agents, which only react to the present, model-based agents can remember what happened in the past and use that knowledge to decide on actions.

Features of Model based agent:

1. **Internal Model:**
 - It keeps track of what the world is like, even if it can't see everything. It updates the model based on the information it gets.
2. **Perception and Action:**
 - It uses sensors to perceive the environment (like a camera or a temperature sensor).
 - It then uses its model to decide what to do next.
3. **State Representation:**
 - The agent knows not just what's happening right now, but also what happened before. This helps it plan for the future.
4. **Reasoning:**
 - The agent uses its model to predict the outcomes of its actions and make better choices.

Example:

A **robot vacuum cleaner** in a room:

- It uses sensors to detect dirt and obstacles.
- It builds a **model** of the room, remembering where it's already cleaned and where there are obstacles.
- If it gets stuck, it knows where to go next based on its model.

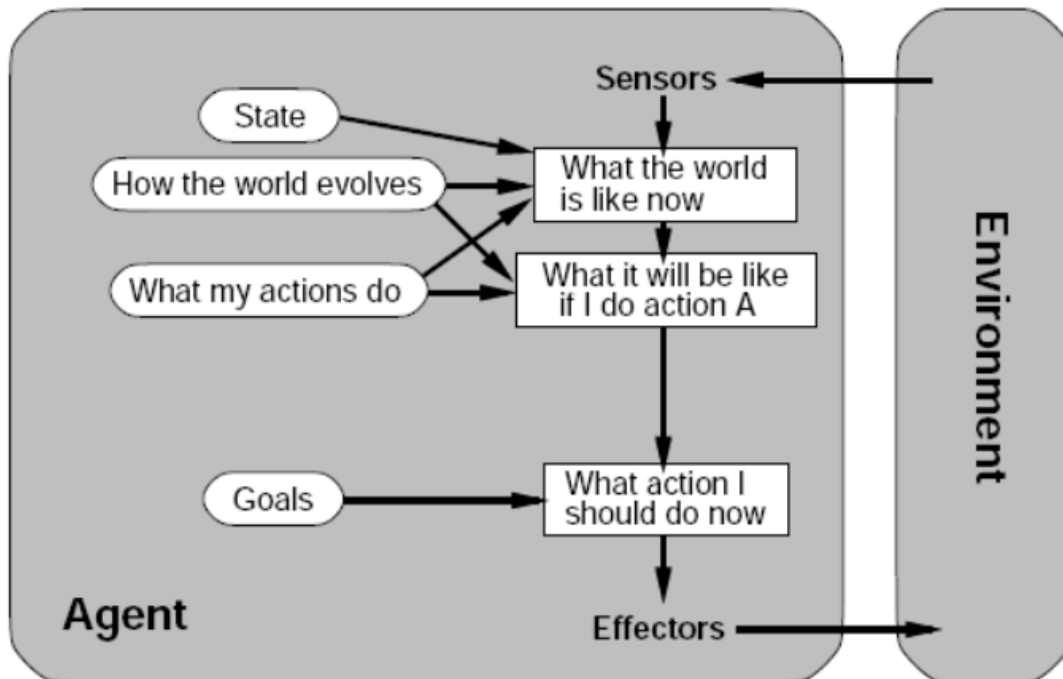
Advantages:

- **Handles Partial Information:** It can make decisions even when it can't see the whole environment.
- **More Flexible:** It can adapt to changes because it "remembers" what happened before.
- **Better Decision-Making:** The agent can predict the outcome of its actions.

Disadvantages:

- **More Complex:** It needs more memory and computing power to maintain its model.
- **Accuracy Issues:** If the model is wrong or outdated, the agent could make poor decisions.

Goal based agent :



A **goal-based agent** is an intelligent system that makes decisions based on **goals** it wants to achieve. Unlike reflex agents (which just react to the environment), goal-based agents plan ahead and choose actions that help them reach a specific goal.

Key Features:

1. **Goals:**
 - The agent has one or more goals it wants to achieve, like reaching a destination or solving a problem.
2. **Decision-Making:**
 - It doesn't just react to the current situation; it plans actions to reach its goal.
3. **Action Planning:**
 - The agent chooses actions based on what will bring it closer to its goal. It might even consider multiple steps ahead.
4. **Problem Solving:**
 - The agent may need to find a sequence of actions to achieve its goal (i.e., it plans ahead).

Example:

Imagine a **robot** in a maze trying to find the exit:

- **Goal:** Reach the exit of the maze.
- The robot considers different paths and chooses the one that brings it closer to the exit.
- It may avoid dead ends and backtrack if necessary.

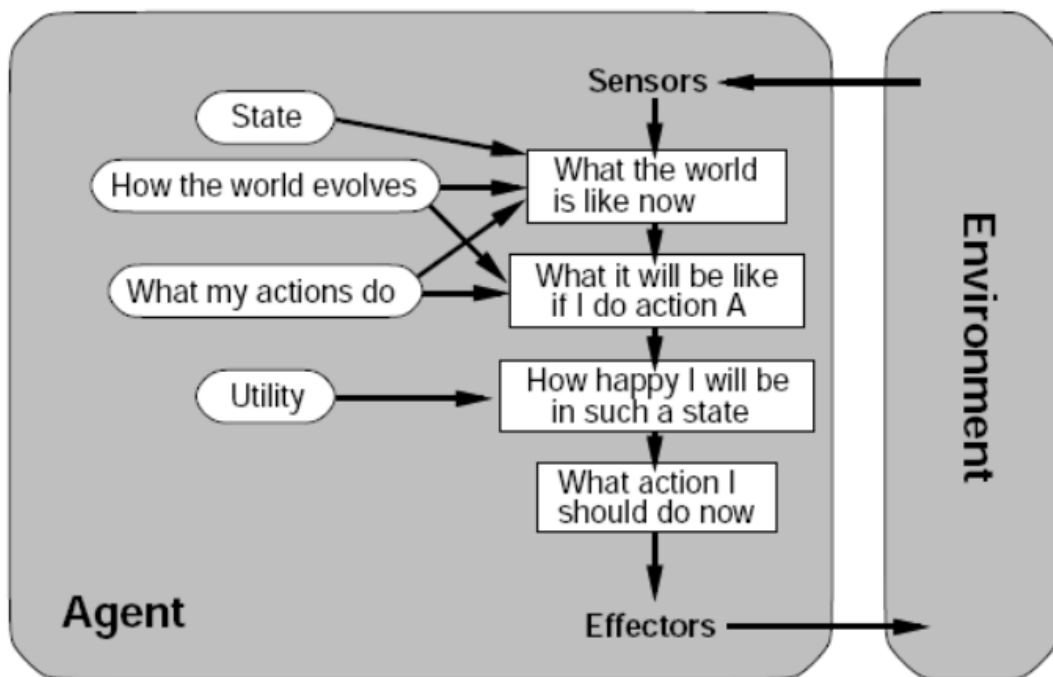
Advantages:

- **Flexible:** It can adapt to different goals and environments.
- **Efficient:** It can plan ahead to achieve its goals in the most effective way.

Disadvantages:

- **Complexity:** Planning can be computationally expensive.
- **Requires More Information:** The agent needs to know its environment well to plan correctly.

Utility based agent



A **utility-based agent** is an intelligent system that chooses actions based on **how much satisfaction or "utility"** it will get from different outcomes. It doesn't just aim for a goal, but tries to pick the action that gives it the **highest satisfaction** or benefit.

Features of Utility – based agent:

1. **Utility:**
 - The agent assigns a value (called "utility") to different outcomes. The higher the utility, the better the outcome is for the agent.
2. **Decision-Making:**
 - The agent chooses actions that maximize its utility. It evaluates different actions and picks the one that gives the best result.

3. **Flexibility:**

- It can handle situations where multiple goals or outcomes are possible and not just choose one goal but the one that offers the best overall satisfaction.

4. **Balance:**

- Unlike goal-based agents that aim to reach a single goal, utility-based agents balance multiple factors to make decisions that are the most beneficial.

Example:

A **robot** trying to decide where to go:

- **Options:** It can go left, right, or straight.
- Each option has a **utility score**:
 - Left might have a high utility (food), but a low utility (danger) might be on the right.
 - The robot picks the path with the **highest utility** overall.

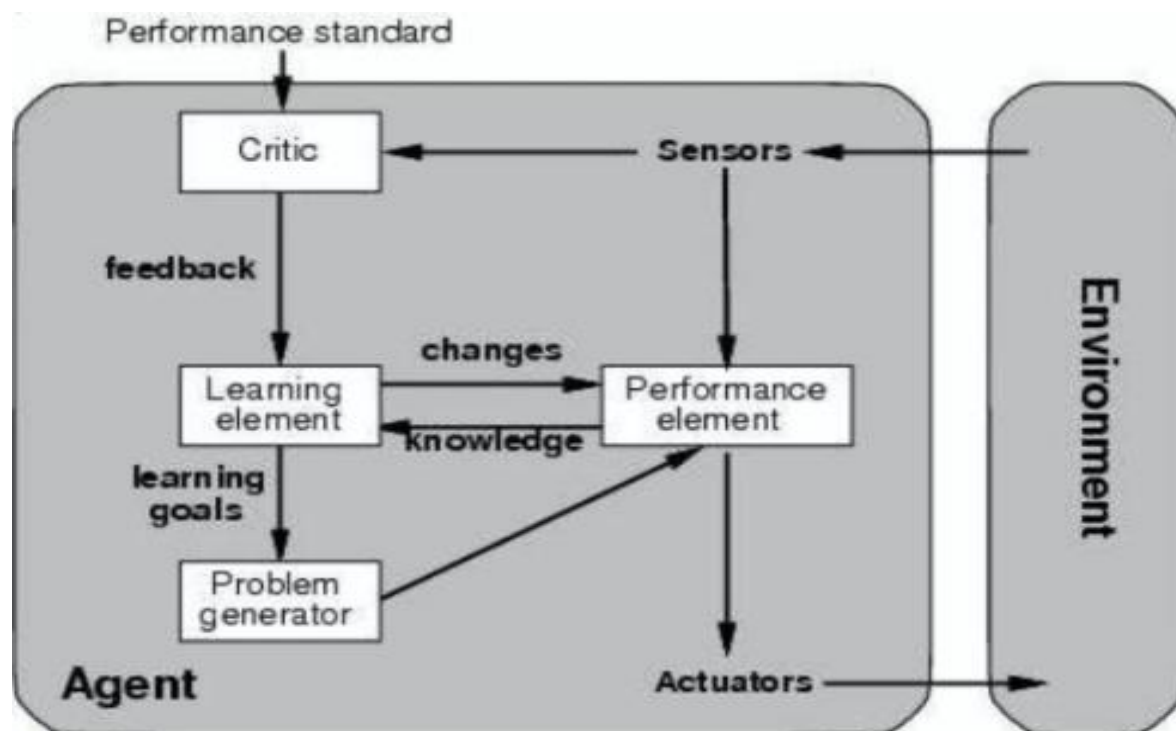
Advantages:

- **Maximizes Satisfaction:** The agent doesn't just achieve goals; it maximizes the best possible outcome.
- **Flexible Decision-Making:** It works well when there are multiple competing goals or choices.

Disadvantages:

- **Complexity:** Calculating utility for different outcomes can be difficult.
- **Requires Detailed Information:** The agent needs to know the utility of different actions and outcomes.

Learning agent:



A **learning agent** is an intelligent system that **improves its performance** over time by learning from its experiences. It **adapts** and gets better at making decisions based on feedback from its actions.

Features of Learning agent:

1. **Learning from Experience:**
 - The agent improves by **observing the results** of its actions and using that information to make better choices in the future.
2. **Feedback:**
 - It gets feedback (positive or negative) based on how well it did, and adjusts its behavior to improve.
3. **Adaptation:**
 - The agent can **change** how it makes decisions to be more effective, based on what it has learned.
4. **Goal:**
 - The main goal is to **maximize performance** and improve over time.

Example:

A robot vacuum cleaner:

- At first, it might make mistakes and miss some spots.
- Over time, it **learns** the best paths to clean the room by remembering where it missed spots before.
- It **adjusts** its cleaning pattern based on its previous experiences to clean the room more efficiently.

Advantages:

- **Improves Over Time:** The more the agent learns, the better it performs.
- **Adaptable:** It can handle new situations by learning from its experiences.

Disadvantages:

- **Takes Time:** Learning takes time, so it may not be effective right away.
- **Requires Data:** The agent needs enough experiences (data) to learn properly.

Problem Solving Agent

A **problem-solving agent** is an intelligent agent that tries to solve a problem by **searching** through possible actions and selecting the one that leads to the goal.

Key Concepts:

1. **Goal:**
 - The agent has a **goal** it wants to achieve. The goal defines what success looks like.
2. **States:**
 - The **state** represents the current situation or configuration of the world.
 - The agent begins in an initial state and works towards reaching the goal state.
3. **Actions:**
 - The agent performs **actions** to move from one state to another. Each action takes the agent from one state to another, and the set of possible actions defines the agent's **action space**.
4. **Search:**
 - Problem-solving agents use **search** algorithms to explore different states. They try different actions and explore different paths to find a sequence of actions that leads to the goal.
 -
5. **Plan:**
 - The agent creates a **plan**, a sequence of actions, to reach the goal state from the initial state.

Problem-Solving Process:

1. **Formulate the Problem:**
 - Define the initial state, actions, and goal state. This step transforms the real-world problem into a well-structured format that can be solved by a search algorithm.
2. **Search for a Solution:**
 - Use a search algorithm to find a path or sequence of actions that leads from the initial state to the goal state.
3. **Execute the Plan:**
 - Once the sequence of actions is found, the agent carries out the actions to achieve the goal.

Example:

- A **robot** wants to move from its starting point to the goal in a maze.
 - **Initial State:** The robot's current position in the maze.
 - **Actions:** Moving up, down, left, or right.
 - **Goal:** Reach the exit of the maze.
 - The agent uses a search algorithm to find the optimal path and then follows it to reach the exit.

Types of Problems:

- **Well-Defined Problems:** The goal, initial state, and actions are clearly defined (e.g., navigating a maze).
- **Ill-Defined Problems:** These are more open-ended with no clear solution path, making them harder to solve.

Advantages:

- **Structured:** Provides a clear, systematic way to solve problems.
- **Generalizable:** Can be applied to a wide range of problems as long as they can be formulated as a search problem.

Disadvantages:

- **Computation Cost:** Searching large problem spaces can be slow and resource-intensive.
- **Requires Well-Defined Goals:** It works best for problems with clear goals and well-defined rules.

Searching for Solutions :

Search is the process of exploring a space of possible solutions to find the best one. In problem-solving, this involves finding a sequence of actions that lead from an initial state to the goal state.

1. State Space:

- A **state space** is a set of all possible states the agent can be in, along with actions that move the agent from one state to another.
- The goal is to navigate through this space to reach the **goal state**.

2. Search Tree:

- The **search tree** represents the states and actions as a tree, where each node is a state, and edges represent actions that lead from one state to another.

3. Search Problem:

- The problem consists of:
 - **Initial State:** The starting point.
 - **Actions:** The available actions that move the agent from one state to another.
 - **Transition Model:** A description of how actions change the current state.
 - **Goal Test:** A way to check if the agent has reached its goal.

Path Cost: The cost associated with a path of actions.

Types of Search:

1. Uninformed Search:

- **Uninformed search** methods do not have any additional information beyond the problem description. They explore the state space systematically.
- Examples:
 - **Breadth-First Search:** Explores all nodes at the present depth level before moving on to nodes at the next depth level.
 - **Depth-First Search:** Explores as deep as possible along a branch before backtracking.

2. Informed Search (Heuristic Search):

- **Informed search** uses additional information (a **heuristic**) to guide the search toward more promising paths.
- Example:
 - *A Search**: Combines the actual cost to reach a state and an estimate of the cost to get to the goal, choosing the most promising path.

Example: Consider a **robot in a grid**:

- **Initial State:** The robot's starting position.
- **Goal State:** The robot's destination.
- **Actions:** Moving up, down, left, or right.
- The robot uses a search algorithm to explore possible paths and find the most efficient one to the goal.

Advantages:

- **Systematic Exploration:** Search algorithms ensure that all possibilities are considered.
- **Can Be Applied to Various Problems:** Search is applicable to many kinds of problems, like navigation, puzzles, and games.

Disadvantages:

- **Computational Cost:** Searching through large state spaces can be slow and require a lot of memory.
- **Complexity:** Some problems have so many possible states that search becomes impractical.

Uninformed Search Strategies

Uninformed search strategies do not use any domain-specific knowledge (like heuristics) beyond the problem definition. They search the state space systematically to find a solution.

1. Breadth-first search
2. Depth-first search
3. Uniform cost search
4. Iterative deepening Depth-first search
5. Bidirectional search

1. Breadth First Search :

- Explores all the nodes at the present depth level before moving on to nodes at the next depth level.
- It explores the state space layer by layer, ensuring that the shallowest (least-cost) solution is found first.
- **Time Complexity:** $O(b^d)$ where b is the branching factor and d is the depth of the shallowest solution.
- **Space Complexity:** $O(b^d)$ as it stores all nodes at each depth level.
- **Pros:** Guaranteed to find the shortest path (least-cost solution).
- **Cons:** Memory intensive, as it needs to store all nodes in the current level.

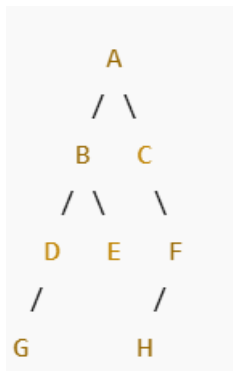
BFS Algorithm

1. **Initialize:**
 - Create an empty **queue** and enqueue the starting node (initial state).
 - Create a **visited set** to keep track of visited nodes and mark the start node as visited.
2. **While the queue is not empty:**
 - Dequeue the front node from the queue.
 - **For each neighbor** of the current node:
 - If the neighbor has **not been visited**:
 - Mark it as visited.
 - **Enqueue** the neighbor to the queue.
3. **Repeat** until:
 - The goal node is found, or
 - The queue is empty (all nodes have been visited).
4. **Return the solution** (if found) or indicate no solution.

Example :

Let's say we want to find the shortest path from node A to node G.

- **Start Node:** A
- **Goal Node:** G



The queue and visited nodes evolve as follows:

1. **Queue:** [A]
Visited: {A}
2. Dequeue A, visit its neighbors (B, C), and enqueue them.
Queue: [B, C]
Visited: {A, B, C}
3. Dequeue B, visit its neighbors (D, E), and enqueue them.
Queue: [C, D, E]
Visited: {A, B, C, D, E}
4. Dequeue C, visit its neighbor (F), and enqueue it.
Queue: [D, E, F]
Visited: {A, B, C, D, E, F}
5. Dequeue D, visit its neighbor (G), and enqueue it.
Queue: [E, F, G]
Visited: {A, B, C, D, E, F, G}
6. **Goal found!** We stop, and reconstruct the path from A to G.
Path: A -> B -> D -> G

2. Depth First Search :

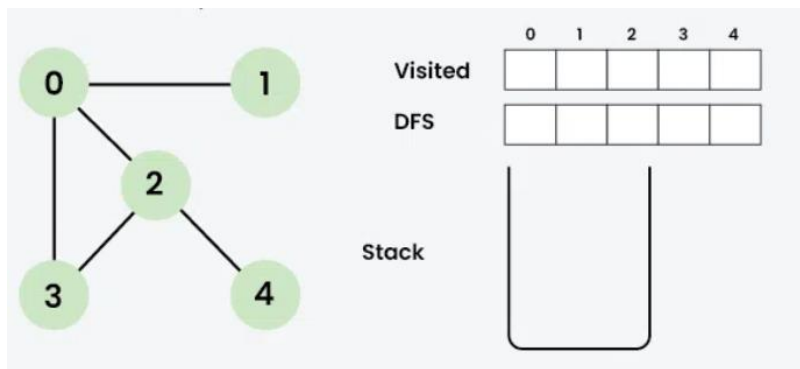
Depth-First Search (DFS) is an uninformed search algorithm that explores as far down a branch as possible before backtracking. It dives deeper into the graph, visiting nodes one by one, and only backtracks when it reaches a dead-end or when it has explored all the nodes on the current path.

DFS Algorithm (Pseudocode)

1. **Initialize:**
 - Create a stack and push the starting node (initial state) onto the stack.
 - Create a **visited set** to track visited nodes and mark the start node as visited.
2. **While the stack is not empty:**
 - Pop the top node from the stack.
 - **If the node is the goal:**
 - Return the solution or path.
 - Otherwise, **explore** all unvisited neighbors of the current node:
 - For each neighbor, mark it as visited and push it onto the stack.
3. **Repeat** until the goal node is found, or the stack is empty (no solution exists).

Example :

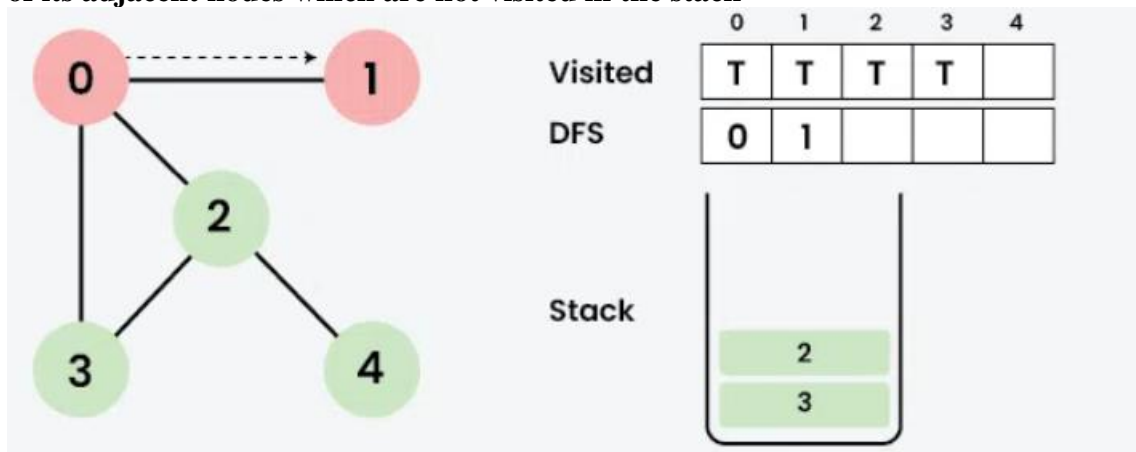
Initially Stack and Visited array are empty



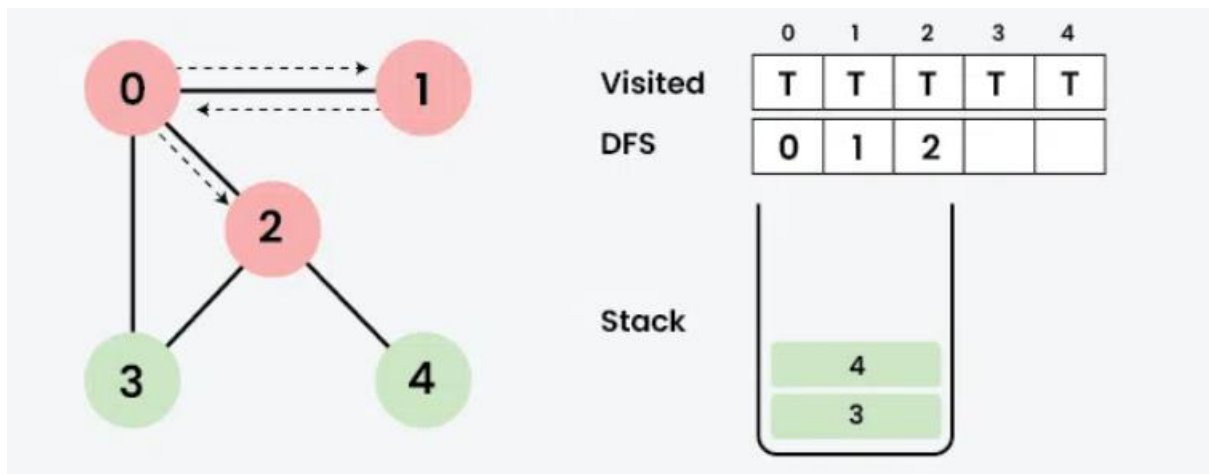
Visit 0 and put its adjacent nodes which are not visited yet into the stack



Now node 1 at the top of the stack , so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack



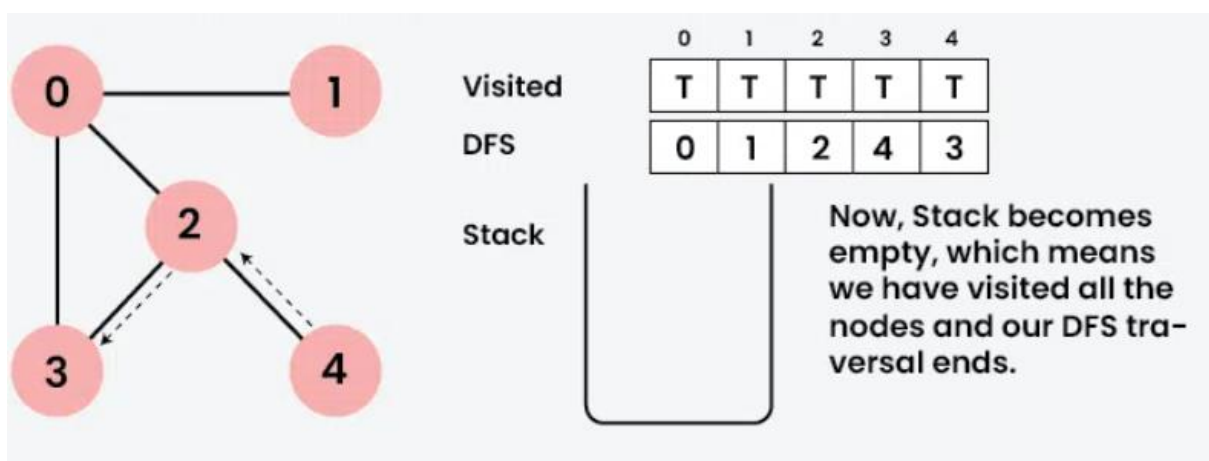
Now, node 2 at the top of the stack, visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack



Now, node 4 at the top of the stack, So visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



Now, node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack



Difference between DFS and BFS

Feature	DFS	BFS
Exploration Strategy	DFS, explores deep before backtracking	BFS, explores level by level
Data Structure	Stack (LIFO)	Queue(FIFO)
Path Finding	Does not guarantee shortest path	Guarantees the shortest path
Space Complexity	$O(b^*m)$, where m is the maximum depth	$O(b^d)$, where d is the depth of the solution
Time Complexity	$O(b^m)$, where m is the maximum depth	$O(b^d)$, where d is the depth of the solution
Memory Efficiency	More memory-efficient for deep trees	Can be memory-intensive for large graphs
Completeness	Not guaranteed in infinite graphs	Guaranteed to find a solution if one exists
Common use cases	Deep tree search, solving puzzles, topological sorting	Shortest path in un weighted graphs, level-order traversal

3.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 .
- UCS explores the search tree by expanding the least-cost node first.
- It uses a **priority queue** to keep track of nodes to explore. The priority queue orders the nodes by their cumulative path cost from the start node, ensuring that the node with the least cost is expanded first.
- UCS is **guaranteed** to find the optimal (least-cost) path, as it explores paths incrementally by increasing cost.

Algorithm:

```

function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier  $\leftarrow$  a priority queue ordered by PATH-COST, with node as the only element
  explored  $\leftarrow$  an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node  $\leftarrow$  POP(frontier) /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        add child.STATE to explored
        frontier  $\leftarrow$  INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child

```

Properties of UCS:

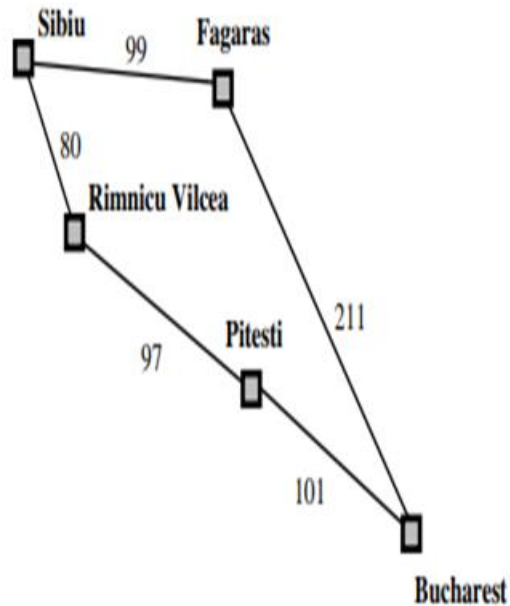
- **Optimal:** UCS guarantees finding the least-cost path, making it an optimal search algorithm when the cost of edges varies.
- **Complete:** UCS is complete, meaning it will find a solution if one exists, as long as the graph does not contain infinite cost paths.
- **Works with Non-negative Edge Weights:** UCS is only applicable to graphs with non-negative edge weights since it relies on the assumption that the cost to reach a node is always increasing.

Time Complexity:

- $O((E+V)\log[V])$, where E is the number of edges and V is the number of nodes.
- This is because for each edge, we may perform an insert and extract operation on the priority queue, both of which take $O(\log[V])$ time.

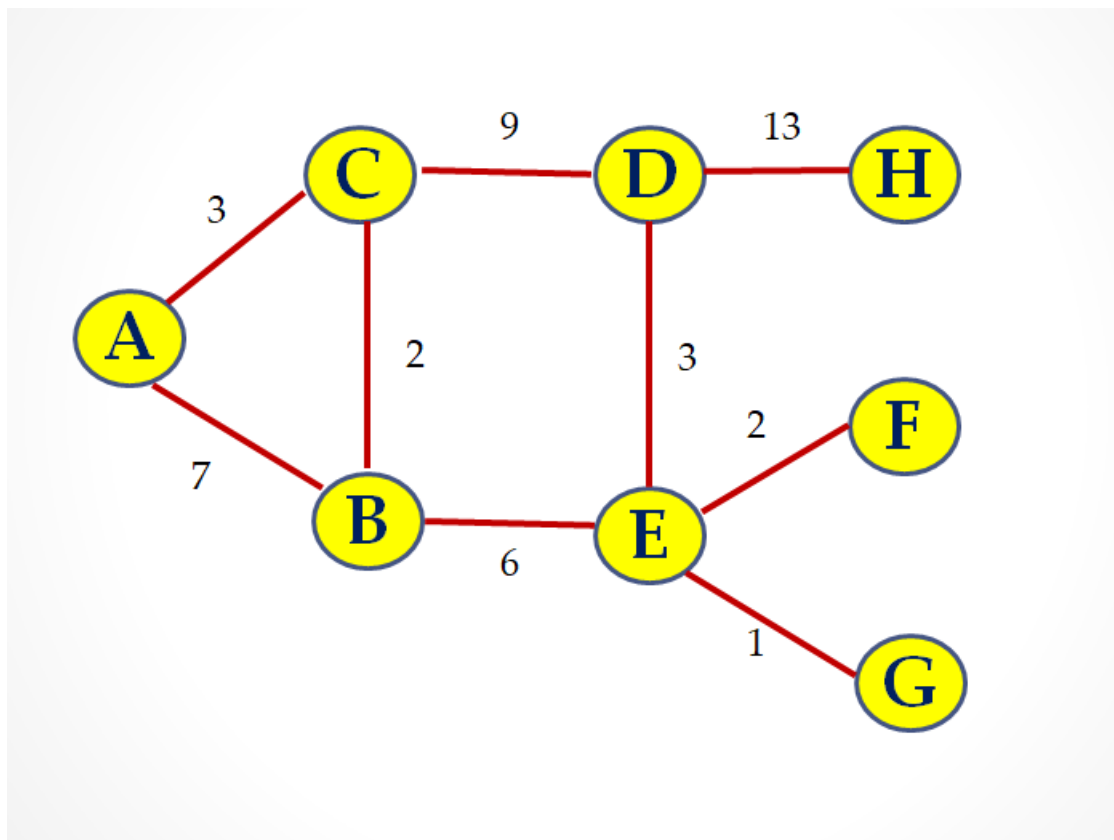
Space Complexity:

- $O(V)$, because we store all the nodes in the frontier and the cost_so_far and parent dictionaries.



- **The problem is to get from Sibiu to Bucharest.**
- The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99 respectively.
- The least-cost node, Rimnicu Vilcea, is expanded next,
- Adding Pitesti with cost **$80+97 = 177$** .
- The least-cost node is now Fagaras, so it is expanded, adding Bucharest with
- cost **$99+211 = 310$** .
- Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with
- cost **$80+97+101 = 278$**
- Now the algorithm checks to see if this new path is better than the old one
- Then the old one is discarded.
- Bucharest, now with **g-cost 278**, is selected for expansion and the solution is returned

Example 2: Start Node is **A** , Goal Node is **G**



4. Iterative deepening Depth-first search:

Iterative Deepening Depth-First Search (IDDFS) is a combination of two search strategies: **Depth-First Search (DFS)** and **Breadth-First Search (BFS)**. It addresses the limitations of DFS by performing DFS repeatedly with increasing depth limits. It performs multiple DFS iterations, each with a progressively larger depth limit, until it finds the solution

- ☐ IDDFS performs **Depth-First Search (DFS)** iteratively, but with a **depth limit** for each iteration.
- ☐ In each iteration, it starts from the root node and performs DFS up to a specific depth limit.
- ☐ After completing one DFS iteration, the depth limit is increased by 1, and a new DFS is performed from the root node with this new depth limit.
- ☐ This continues until the solution is found or the search space is fully explored

Advantages of IDDFS:

1. **Memory Efficiency:** Unlike BFS, which can be memory-intensive (storing all nodes at a given level), IDDFS only needs memory proportional to the **depth of the tree**, which is $O(bd)$, where b is the branching factor and d is the depth of the deepest node.
2. **Completeness:** IDDFS is **guaranteed to find a solution** (if one exists), just like BFS, because it explores all possible nodes at increasing depths.

3. **Optimality:** If the cost of each step is the same, IDDFS will find the **shortest path** to the goal, similar to BFS.
4. **Works in Infinite Search Spaces:** IDDFS can work in infinite or very large search spaces, where BFS would run out of memory.

Limitations of IDDFS:

1. **Repeated Exploration:** The same nodes are explored repeatedly in each iteration, which makes IDDFS less efficient than other algorithms like UCS when costs vary.
2. **Not Optimal for Weighted Graphs:** IDDFS does not work well for graphs with weighted edges because it does not consider the path cost at each depth limit.

Algorithm :

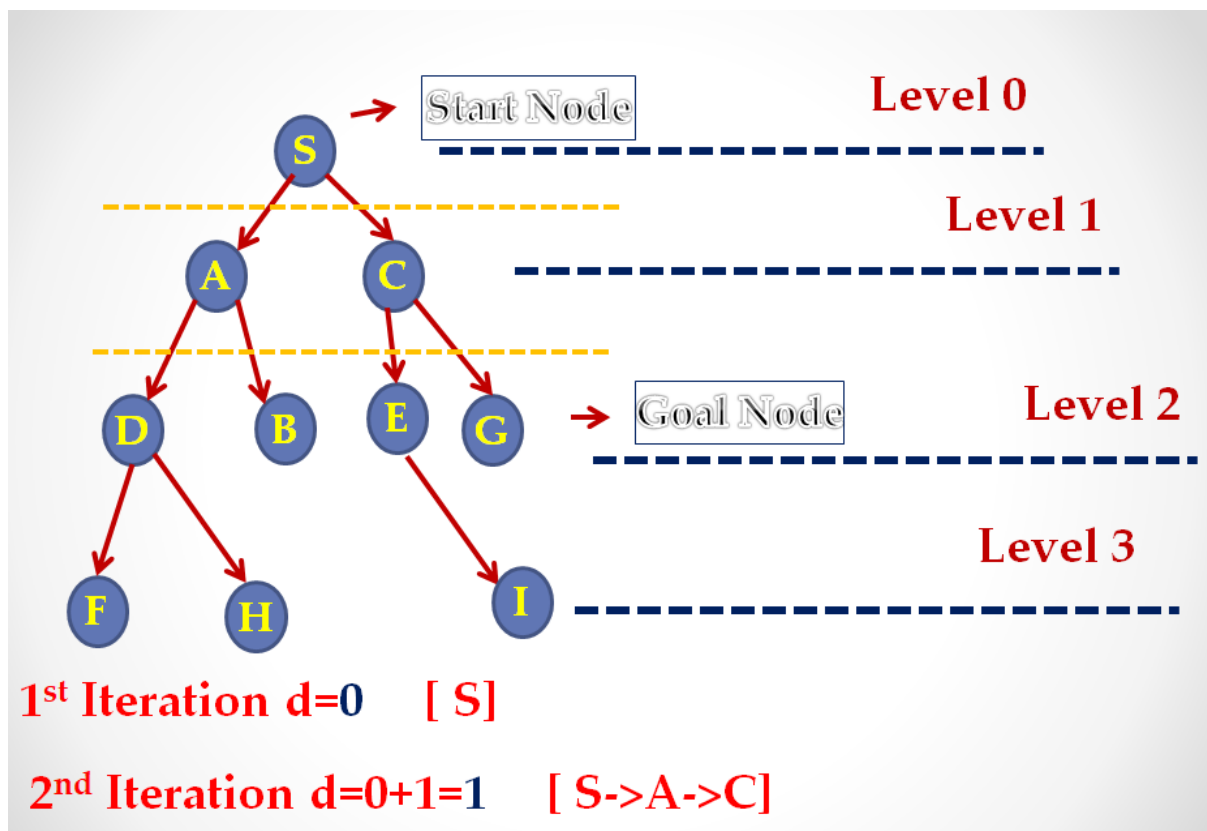
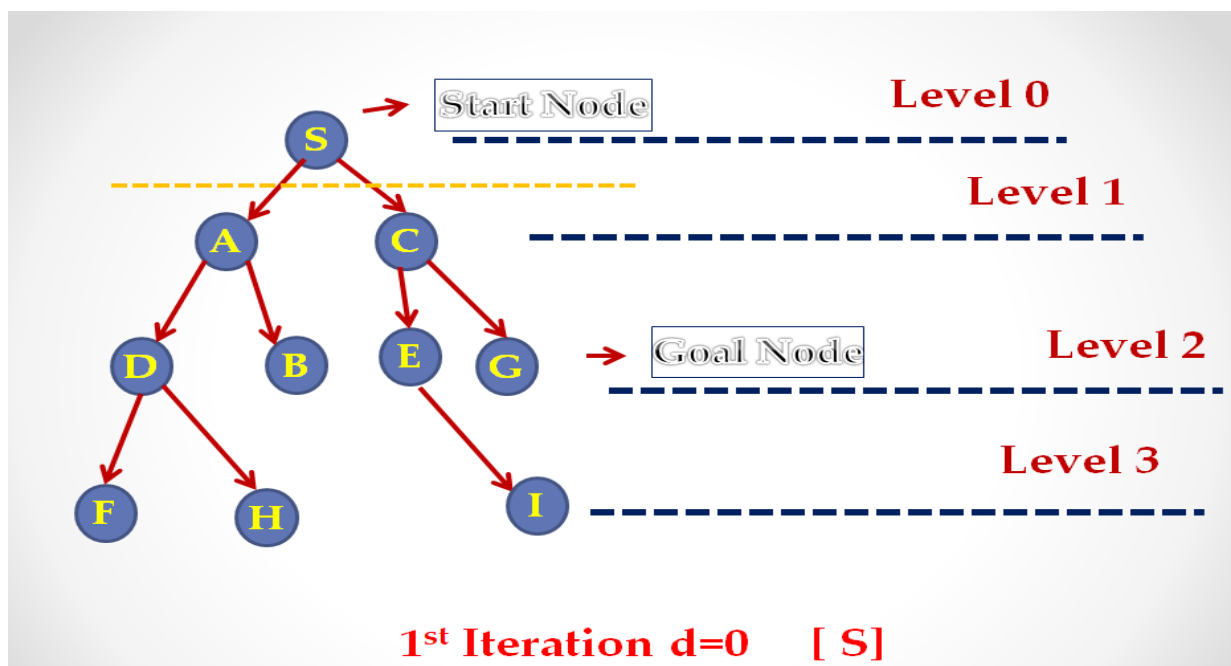
```

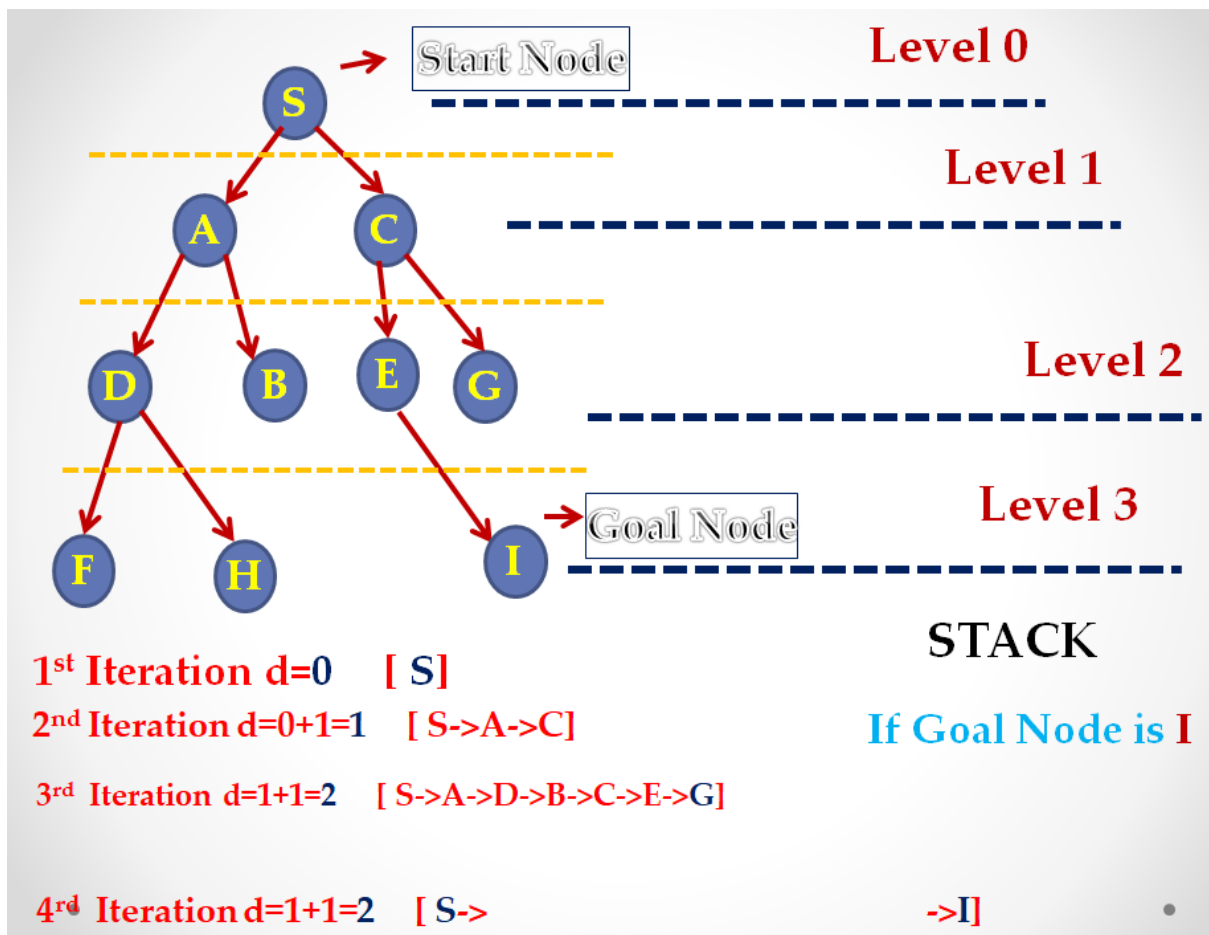
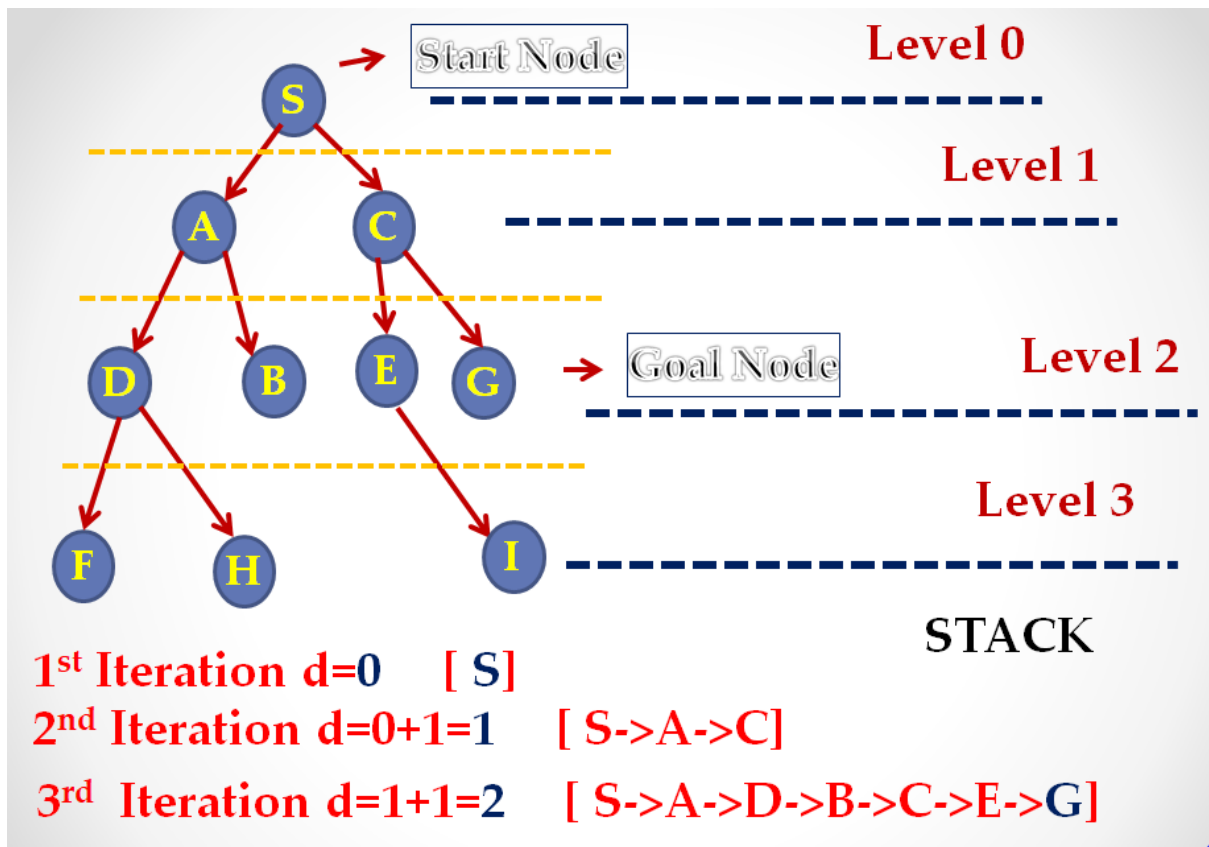
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  inputs: problem, a problem

  for depth  $\leftarrow$  0 to  $\infty$  do
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
    if result  $\neq$  cutoff then return result

```

Example :





Depth Limited Search :

- The problem of unbounded trees can be alleviated by supplying depth-first search with a predetermined depth limit L .
- That is, nodes at depth L are treated as if they have no successors.
- This approach is called depth-limited search.
- The depth limit solves the infinite-path problem.
- Unfortunately, it also introduces an additional source of incompleteness if we choose l
- $l < d$, that is, the shallowest goal is beyond the depth limit.
- Depth-limited search will also be non-optimal if we choose $l > d$.
- Working is similar to DFS but with a predetermined limit.
- Helps in solving the problem of **DFS : Infinite Path**

Termination Conditions :

- A) Failure Value : There is no solution
- B) Cutoff Failure : Terminates on reaching predetermined depth.....No Solution.

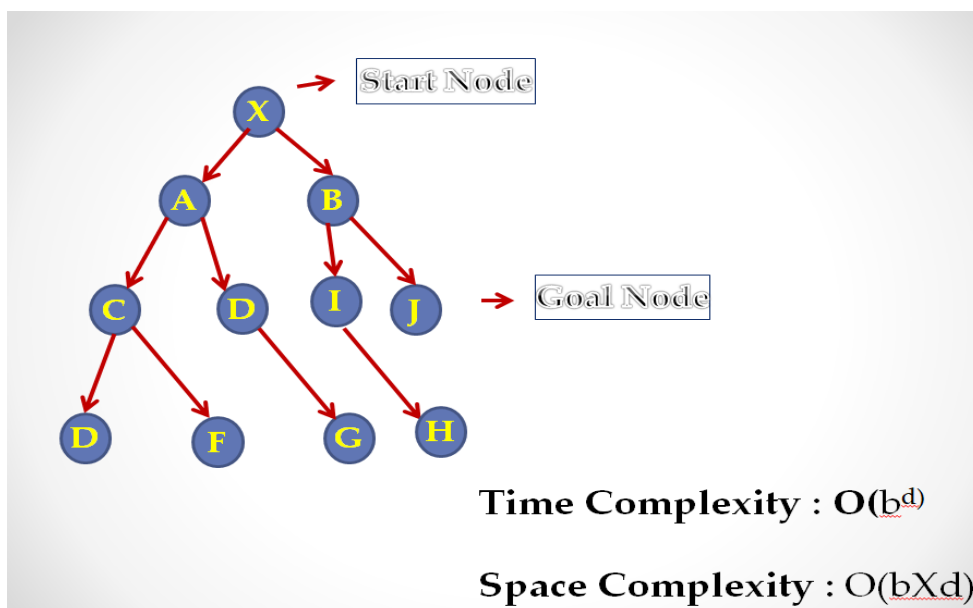
Advantages :

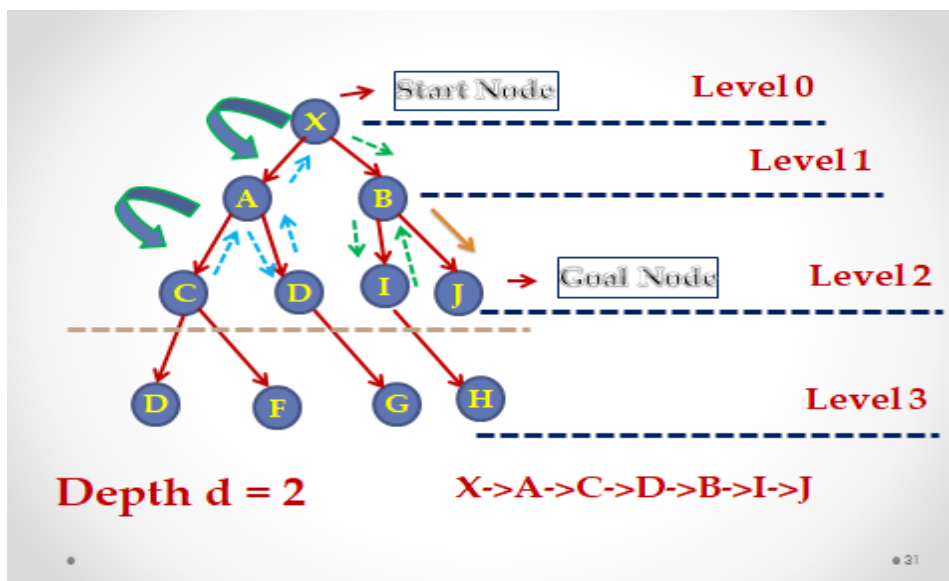
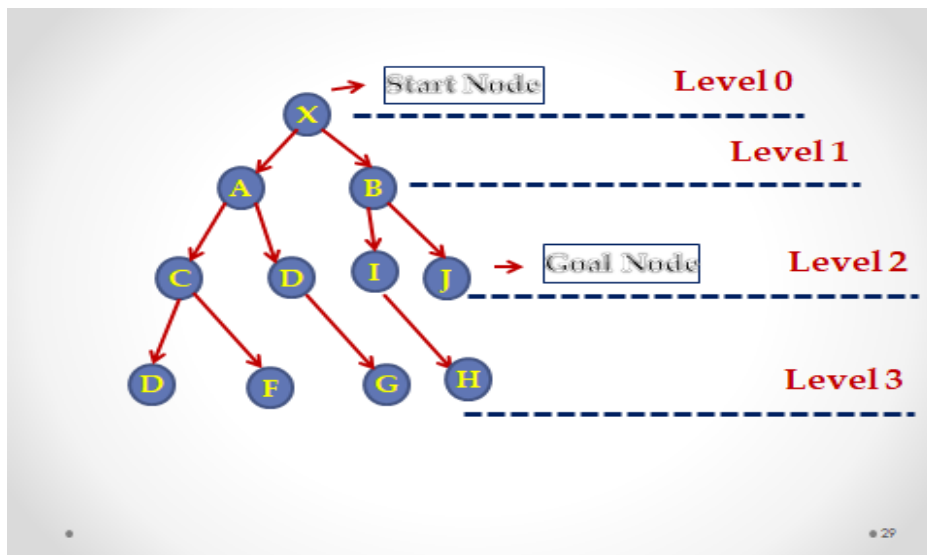
- Memory Efficient

Disadvantages :

- Can be terminated without finding solution (Incompleteness)
- Not Optimal

Example :





Algorithm :

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  cutoff_occurred?  $\leftarrow$  false
  if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
    result  $\leftarrow$  RECURSIVE-DLS(successor, problem, limit)
    if result = cutoff then cutoff_occurred?  $\leftarrow$  true
    else if result  $\neq$  failure then return result
  if cutoff_occurred? then return cutoff else return failure
```

Figure 3.13 A recursive implementation of depth-limited search.

5. Bidirectional Search :

The **bidirectional** search is to run two simultaneous searches-one forward from the initial state and the other backward from the goal, stopping when the two searches meet in the middle

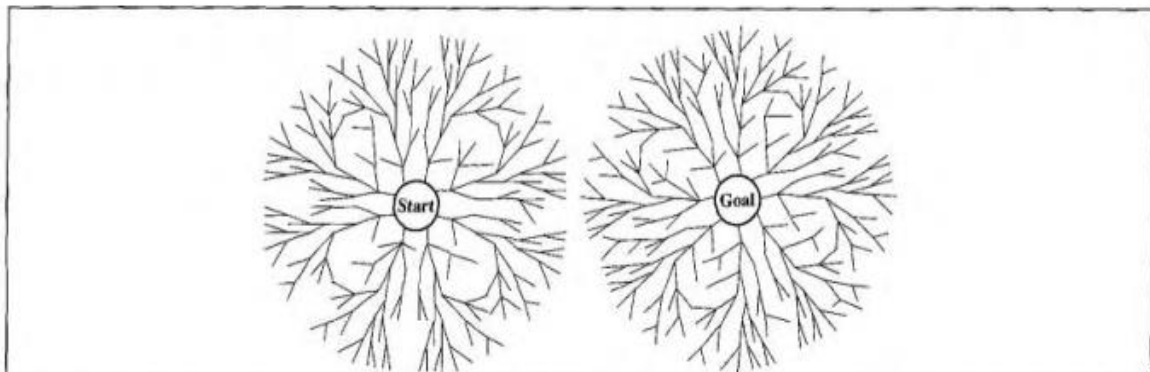
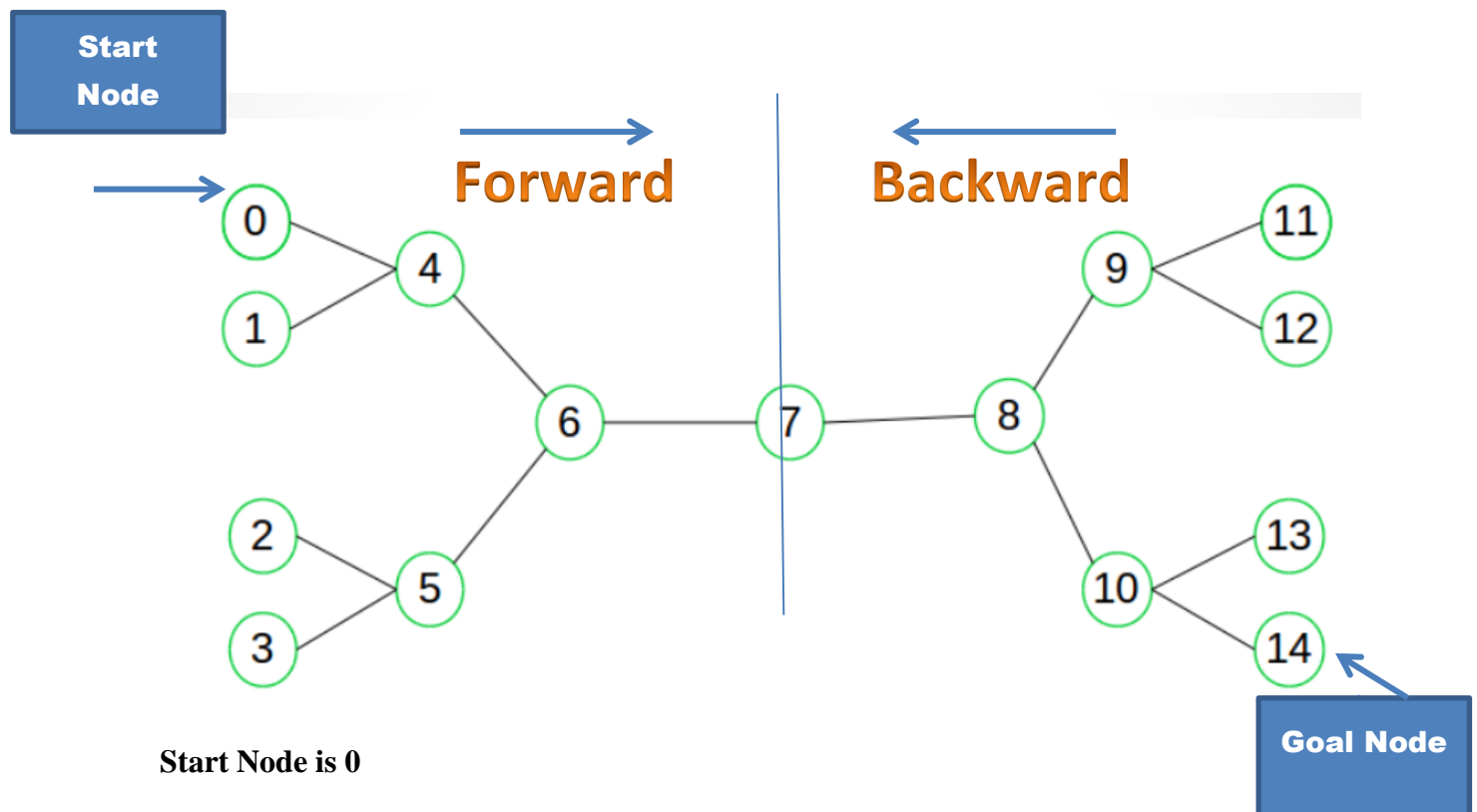


Figure 3.16 A schematic view of a bidirectional search that is about to succeed, when a branch from the start node meets a branch from the goal node.

- Two different Searches are run simultaneously
 - Forward (Start to Goal)
 - Backward (Goal to Start)
 - Single Search Graph is replaced with two small graphs.
 - Any search technique can be used (BFS, DFS,...)
 - Search STOP condition : When Graphs intersect

The motivation is that $d^{d/2} + b^{d/2}$ is much less than b^d , or in the figure, the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.

- **Forward Search:**
 - Start from the initial node (source node) and expand nodes in the usual way until a solution is found or a pre-defined depth is reached.
- **Backward Search:**
 - Simultaneously, start from the goal node and expand nodes in the reverse direction (towards the start).
- **Meeting in the Middle:**
 - The search terminates when the forward search and backward search meet at a common node. The solution can be found by combining the paths from both directions.



Start Node is 0

Goal Node is 14

Forward :

Step 1 : 0 → 4

Step 2 : 4 → 6

Step 3 : 6 → 7

Forward : 0 → 4 → 6 → 7

Backward : 14 → 10 → 8 → 7

Backward :

Step 1 : 14 → 10

Step 2 : 10 → 8

Step 3 : 8 → 7

Final Path from Start Node 0 to Goal Node 14

Comparing uninformed search strategies

Figure 3.17 compares search strategies in terms of the four evaluation criteria set forth in Section 3.4.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes ^a	Yes ^{a,b}	No	No	Yes ^a	Yes ^{a,d}
Time	$O(b^{d+1})$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes ^c	Yes	No	No	Yes ^c	Yes ^{c,d}

Figure 3.17 Evaluation of search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; ℓ is the depth limit. Superscript caveats are as follows: ^a complete if b is finite; ^b complete if step costs $\geq \epsilon$ for positive ϵ ; ^c optimal if step costs are all identical; ^d if both directions use breadth-first search.

Informed (Heuristic) Search Strategies

Informed (Heuristic) Search Strategies are a class of search algorithms in Artificial Intelligence that use domain-specific knowledge to guide their search for a solution.

Informed search strategies use a **heuristic function** to estimate how "close" a given state is to the goal state. This "informed guess" helps them prioritize which paths to explore, leading to much more efficient problem-solving in large or complex search spaces.

Heuristic Function ($h(n)$)

$h(n)$ is an estimated cost (or distance, or number of steps) from a given node n to the nearest goal state.

It provides an "informed guess" about the remaining path cost to the goal. A good heuristic function can significantly reduce the number of nodes the algorithm needs to explore, thereby saving time and computational resources.

Heuristic functions are problem-specific.

Heuristics **are not always perfect**, they are estimates, and sometimes they can be inaccurate. The quality of the heuristic heavily influences the performance and optimality of the search algorithm.

Greedy best-first search :

Greedy Best-First Search (GBFS) is a search algorithm that is used to find the path to the goal by **prioritizing nodes that are closest to the goal** based on a heuristic function.

- It evaluates nodes by using just the heuristic function: $f(n) = h(n)$.
- GBFS uses a **heuristic function** $h(n)$ to estimate the **cost** of reaching the goal from a node n .
- It explores nodes by choosing the one with the **lowest heuristic value** (i.e., the one that appears to be closest to the goal according to the heuristic).
- GBFS **does not** take into account the cost incurred to reach the node (like in Uniform Cost Search); it only focuses on the heuristic estimate of the distance to the goal.
- Route-finding problems in Romania, using the straightline distance heuristic, which we will call **h_{SLD}** .
- If the goal is Bucharest, we will need to know the straight line distances to Bucharest.

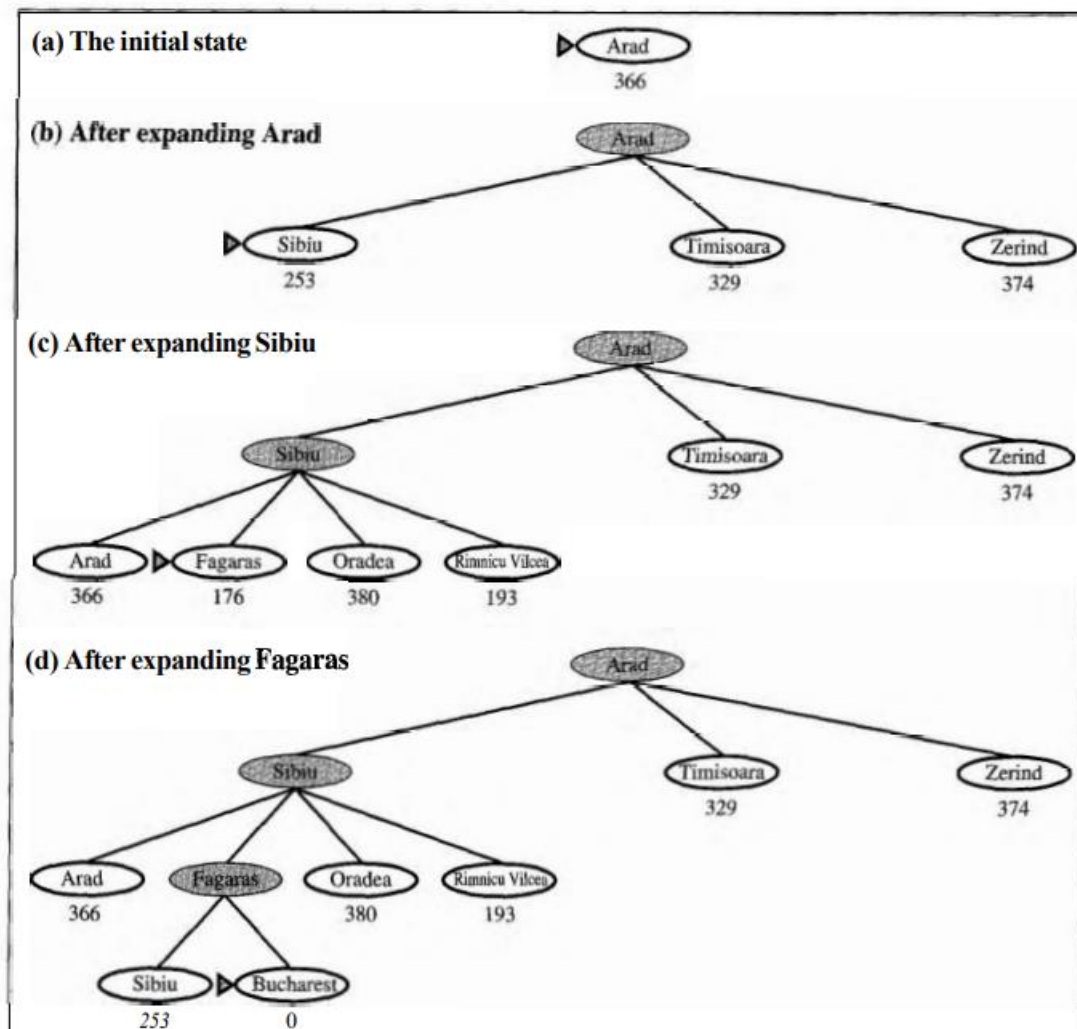
Example,

$h_{SLD}(\text{In}(\text{Arad})) = 366$. Notice that the values of h_{SLD} cannot be computed from the problem description itself.

Moreover, it takes a certain amount of experience to know that h_{SLD} is correlated with actual road distances and is, therefore, a useful heuristic.

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figure 4.1 Values of h_{SLD} —straight-line distances to Bucharest.



The above diagram shows stages in a greedy best-first search for Bucharest using the straight-line distance heuristic hSLD. Nodes are labeled with their h-values.

A* Search Algorithm :

:

- A* Search algorithm is informed search algorithm.
- Used to find the optimal path from the initial state to the goal state.
- A* search is used to find the shortest path from a **start state to a goal state** by evaluating nodes
- based on a cost function.
- A* Search algorithm evaluates nodes by using the function $f(n) = g(n) + h(n)$

$$f(n) = g(n) + h(n)$$

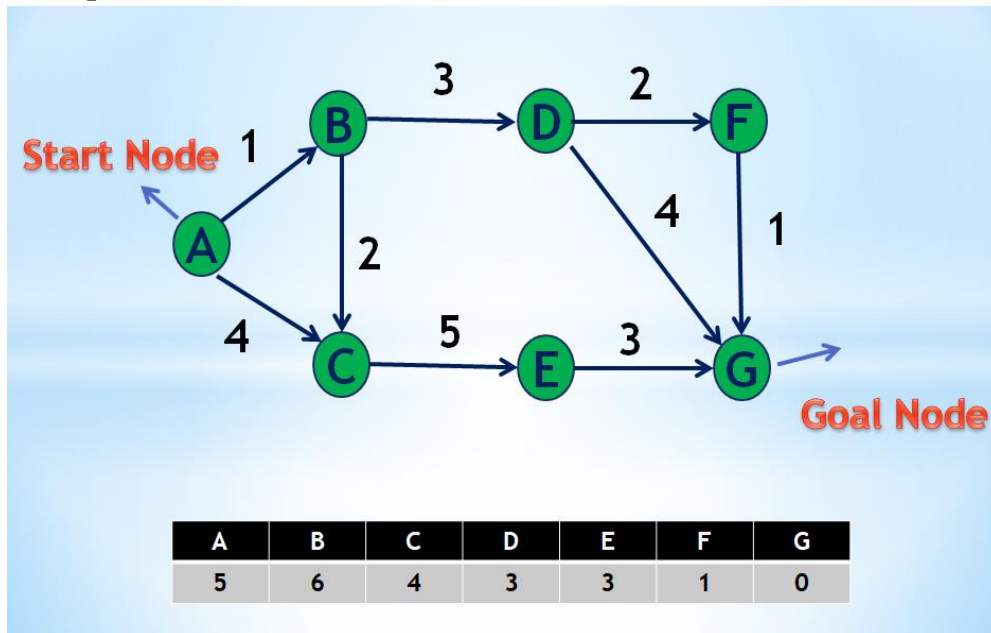
Where:

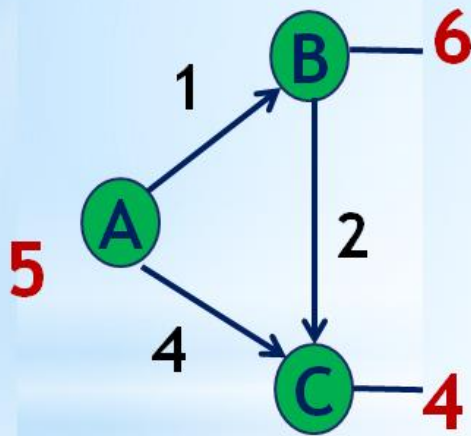
- $f(n)$ is the total estimated cost of the cheapest solution through node n .
- $g(n)$ is the Cost from *initial state to the state at the current node n*
- $h(n)$ is Estimated cost from the state at node n to a goal state.

Components of A*:

1. **Start Node:** The initial state of the problem.
2. **Goal Node:** The target state that you want to reach.
3. **Cost Function:** The function $f(n)$ which combines the cost to reach the node $g(n)$ and the heuristic estimate $h(n)$.

Example :





A	B	C	D	E	F	G
5	6	4	3	3	1	0

A->A : $g(n)+h(n)$ [**A to A**]

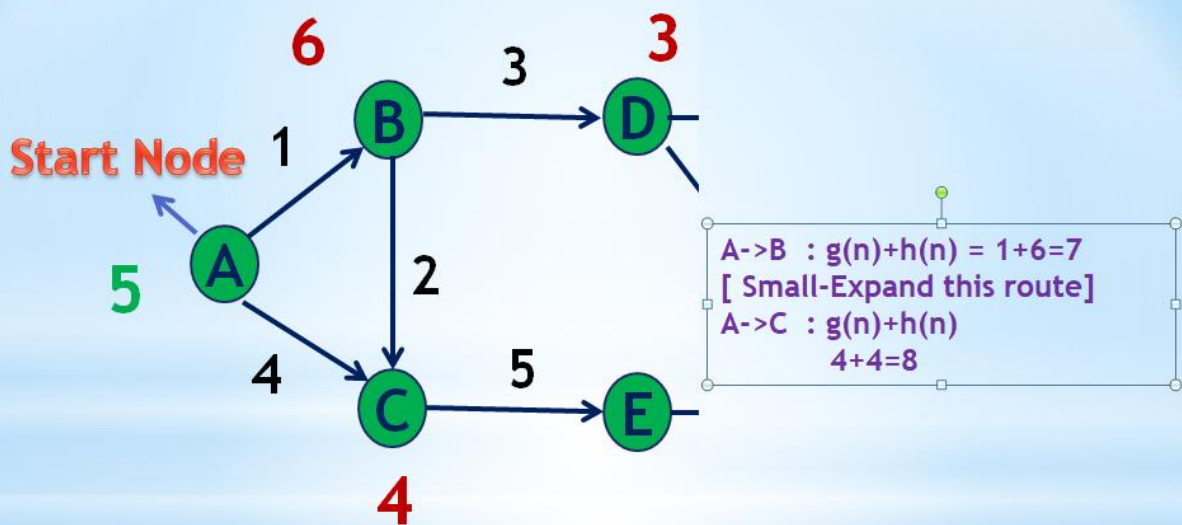
$$0+5=5$$

A->B : $g(n)+h(n)$

$$1+6=7 \text{ [Small Value.... Expand this route]}$$

A->C : $g(n)+h(n)$

$$4+4=8$$



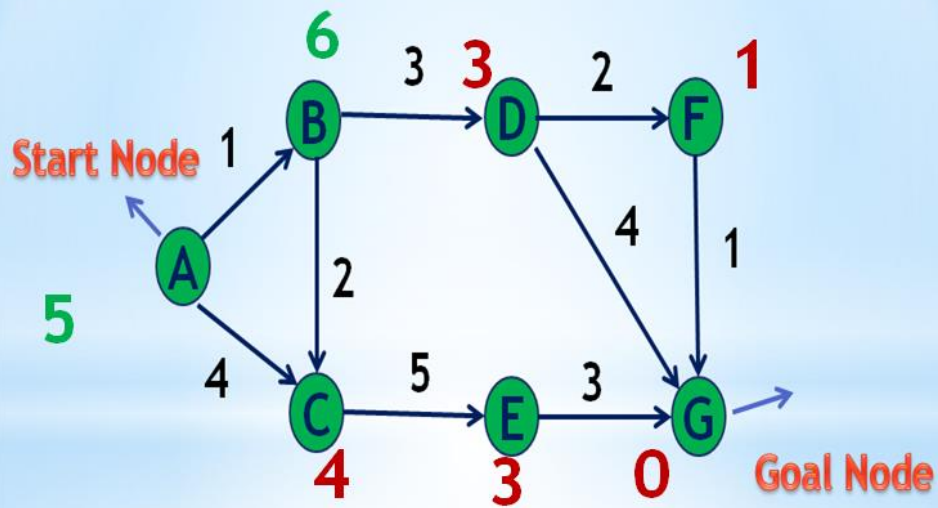
A	B	C	D	E	F	G
5	6	4	3	3	1	0

B->D : $g(n)+h(n)$

$$1+3+3=7$$

B->C : $g(n)+h(n)$

$$1+2+4=7 \text{ [small Value.... Expand this route]}$$



A	B	C	D	E	F	G
5	6	4	3	3	1	0

C -> **E** : $g(n)+h(n)$
 $1+2+5+3=11$

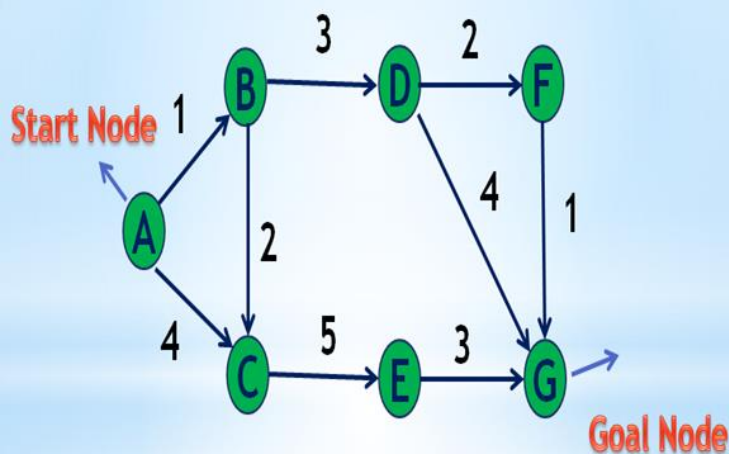
D -> **G** : $1+3+4+0 = 8$

D -> **F** : $1+3+2+1 = 7$ [**Small Value**.... Expand this route]

B->**D** : $g(n)+h(n)$
 $1+3+3=7$

B->**C** : $g(n)+h(n)$
 $1+2+4=7$

[same Value....
 Expand this route]



A	B	C	D	E	F	G
5	6	4	3	3	1	0

F -> **G** : $1+3+2+1+0 = 7$

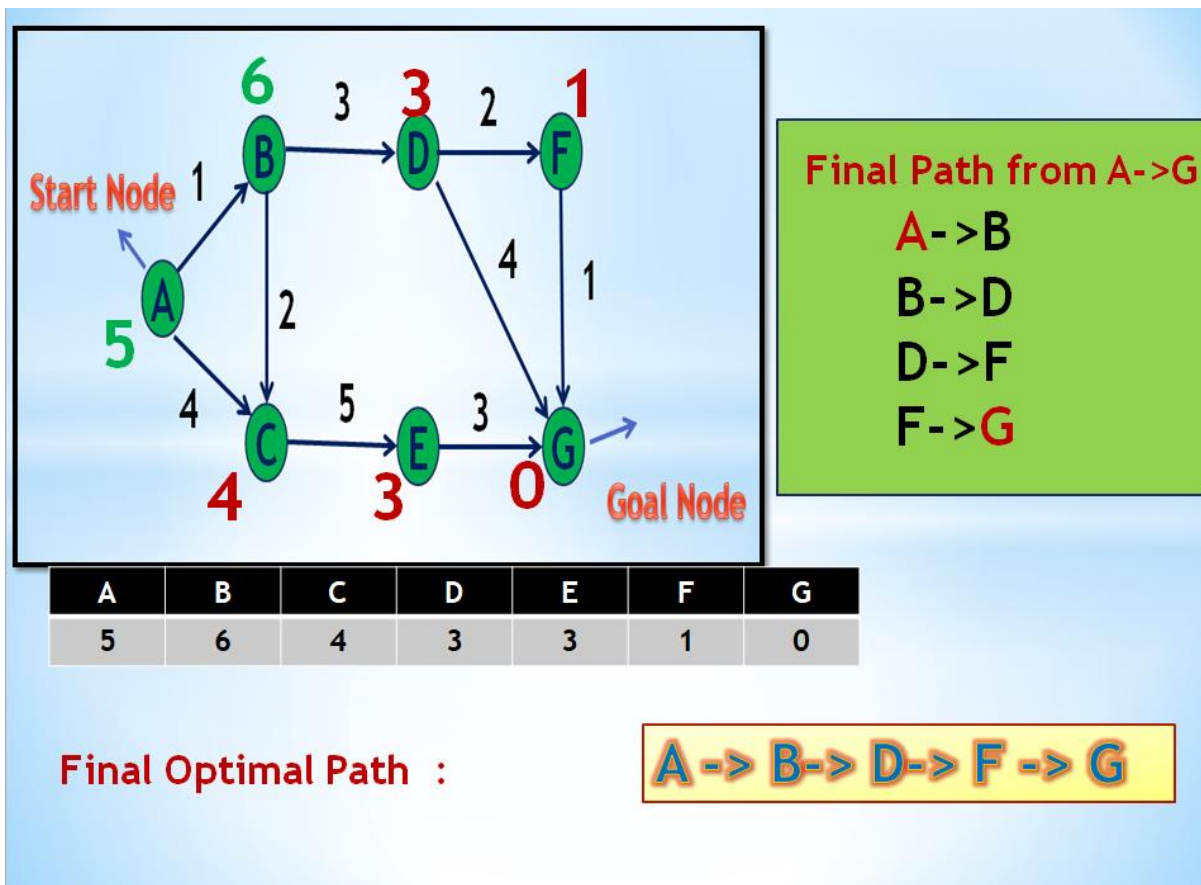
C-> **E** : $g(n)+h(n)$
 $1+2+5+3=11$

D -> **G** : $1+3+4+0 = 8$

D -> **F** : $1+3+2+1 = 7$

[Small Value.... Expand this route]

[**Small Value**.... GOAL NODE reached i.e., **F**]



Applications of A*

- **Path finding in Games and Robotics:**
A* is extensively used in the *gaming industry* to control characters in *dynamic environments*, as well as in robotics for navigating between points.
- **Network Routing:**
In telecommunications, A* helps in determining the *shortest routing path* that *data packets* should take to *reach the destination*.
- **AI and Machine Learning:**
A* can be used in planning and *decision-making* algorithms, where multiple stages of decisions and movements need to be evaluated.

Advantages of A*

- **Optimality:**
When equipped with an admissible heuristic, A* is guaranteed to find the shortest path to the goal.
- **Completeness:**
A* will always find a solution if one exists.
- **Flexibility:**

By adjusting heuristics, A* can be adapted to a wide range of problem settings and constraints

- **Time Complexity:** In the worst case, A* can explore all nodes, leading to a time complexity of $O(b^d)$, where b is the branching factor and d is the depth of the solution. However, if the heuristic is good (i.e., close to the true cost), A* can dramatically reduce the number of nodes explored.
- **Space Complexity:** Since A* must store all nodes in memory (both open and closed lists), the space complexity is also $O(b^d)$

Algorithm :

□ **Initialization:**

- Create an open list (priority queue) to store nodes that need to be evaluated. This list will store nodes based on their $f(n)$ value.
- Create a closed list to store nodes that have already been evaluated.
- Set the **start node** with $f(\text{start}) = g(\text{start}) + h(\text{start})$, where $g(\text{start}) = 0$ and $h(\text{start})$ is the heuristic estimate to the goal.

□ **Loop** (until the open list is empty or the goal is found):

- **Select the node n with the lowest $f(n)$ from the open list.** If n is the goal node, terminate and return the path.
- Move node n from the open list to the closed list.
- **For each neighbor of n , do the following:**
 - ✚ If the neighbor is in the closed list, skip it (it has already been evaluated).
 - ✚ Calculate $g(n)$, the cost to reach the neighbor from the start.
 - ✚ If the neighbor is not in the open list or has a lower $g(n)$ value, update its $f(n)$ value and set the current node n as its parent. If it's not in the open list, add it.

□ **Goal Test:**

- When the goal node is found, reconstruct the path from the start node to the goal by following the parent pointers from the goal node back to the start node.

Heuristic Functions

- **Definition:** A heuristic function is a function that estimates the cost of the cheapest path from a given state to the goal state in a search problem. Heuristics are used to guide search algorithms toward the goal more efficiently than uninformed search algorithms.
- **Heuristic functions** are strategies or methods that guide the search process in AI algorithms by providing estimates of the most promising path to a solution.
- **Heuristic functions** transform complex problems into more manageable sub problems by providing estimates that guide the search process.
- This approach is particularly effective in **AI planning**, where the goal is to sequence actions that lead to a desired outcome.
- **Informed search** algorithms use heuristic functions to estimate the cost of reaching the goal, significantly improving search efficiency.
- **Heuristic functions are used** in informed search algorithms, such as A* search, greedy best-first search, and hill-climbing, to prioritize exploring paths that seem most promising based on the estimated cost.

Role of Heuristic Functions in AI :

- **Efficiency:** They reduce the search space, leading to faster solution times.
- **Guidance:** They provide a sense of direction in large problem spaces, avoiding unnecessary exploration.
- **Practicality:** They offer practical solutions in situations where exact methods are computationally prohibitive.

Heuristic functions are particularly useful in various problem types, including:

Path finding Problems: Path finding problems, such as navigating a maze or finding the shortest route on a map, benefit greatly from heuristic functions that estimate the distance to the goal.

Constraint Satisfaction Problems: In constraint satisfaction problems, such as scheduling and **puzzle-solving**, heuristics help in selecting the most promising variables and values to explore.

Optimization Problems: Optimization problems, like the **traveling salesman problem**, use heuristics to find near-optimal solutions within a reasonable time frame.

Characteristics of Heuristic Functions

1. **Admissibility:**

- A heuristic is **admissible** if it never overestimates the true cost to reach the goal from a given state.
- An admissible heuristic ensures that the algorithm, such as A* search, will always find the optimal solution, if one exists.

2. Consistency (or Monotonicity):

- A heuristic is **consistent** if, for every node n and every successor n' of n , the estimated cost of reaching the goal from n is no greater than the cost of getting from n to n' plus the estimated cost of reaching the goal from n' :

$$h(n) \leq c(n, n') + h(n')$$

where $h(n)$ is the heuristic estimate at node n and $c(n, n')$ is the cost to get from n to n' .

- A consistent heuristic is always admissible, but the reverse is not necessarily true.

3. Types of Heuristics:

- **Domain-Specific Heuristics:** These are heuristics tailored to a specific problem domain, based on domain knowledge.
- **Domain-Independent Heuristics:** These heuristics do not rely on domain-specific knowledge and are often based on simpler, general criteria (e.g., Manhattan distance for grid-based problems).
- **Relaxed Problem Heuristics:** A heuristic derived from a simplified or relaxed version of the problem. For example, in the 8-puzzle problem, a heuristic might be based on how many tiles are out of place, ignoring the complex constraints of the puzzle.

Examples :

1. Traveling Salesman Problem (TSP):

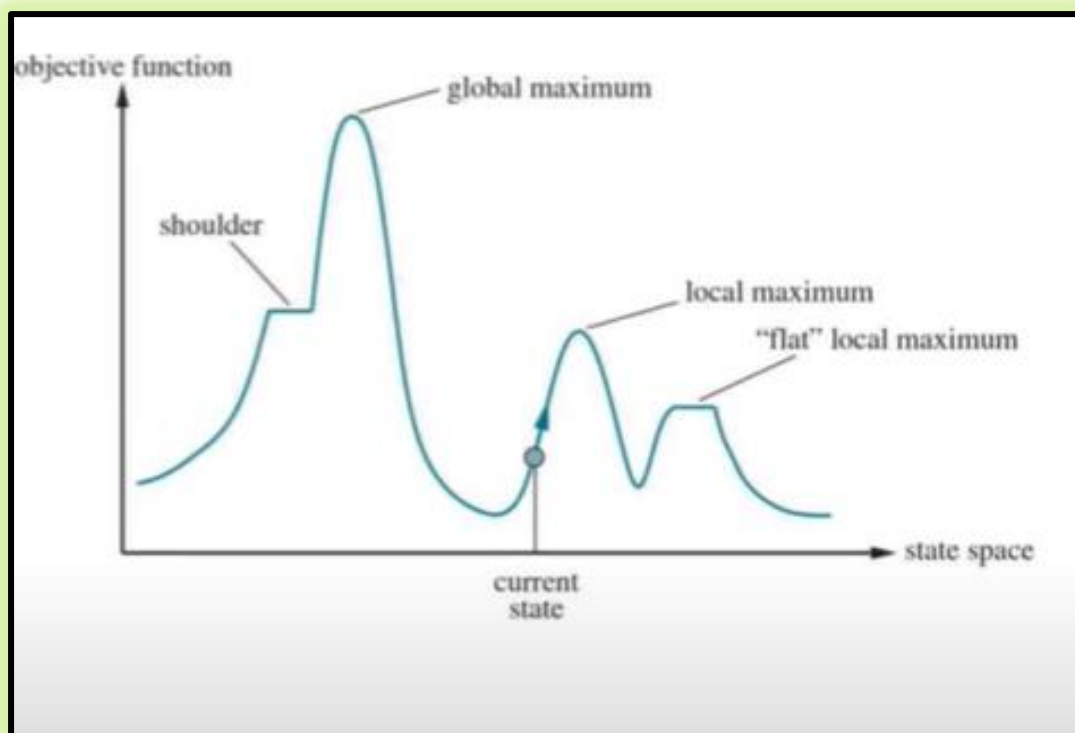
- A common heuristic for the TSP might involve the **minimum spanning tree** of the cities or **straight-line distance** (Euclidean distance) between cities.

2. 8-Puzzle Problem:

- A simple heuristic for the 8-puzzle problem is the **Manhattan distance**, which measures the sum of the distances of each tile from its goal position, moving only horizontally or vertically.
- Another heuristic could be the **number of misplaced tiles**, which counts how many tiles are not in their goal position.

Hill Climbing Search:

- Hill Climbing algorithm is a **Heuristic Search** algorithm which continuously moves in the direction of **increasing value** to find the peak of the mountain or best solution to the problem.
- It keeps track of one current state and on each iteration moves to the neighboring state with highest value – i.e., it heads in the direction that provides the **steepest ascent**.
- In this algorithm, when it reaches a peak value where no neighbor has a higher value, then it terminates.
- It is also called greedy local search as it only searches its good immediate neighbor state and not beyond that.
- Hill Climbing is mostly used when a good heuristic is available



❖ **Different regions in the state space landscape :**

- **Local Maximum** is a state which better than its neighbor states, but there is also another state which is higher than it.
- **Global Maximum** is the best possible state of state space landscape. It has the highest value of objective function.
- **Current State** is a state in a landscape diagram where an agent is currently present.

- **Flat Local Maximum** is a flat space in the landscape where all the neighbor states of current states have the same value.
- **Shoulder** is a plateau region which has an uphill edge.

Algorithm

- **Initial State:** Start with an arbitrary or random solution (initial state).
- **Neighboring States:** Identify neighboring states of the current solution by making small adjustments (mutations or tweaks).
- **Move to Neighbor:** If one of the neighboring states offers a better solution (according to some evaluation function), move to this new state.
- **Termination:** Repeat this process until no neighboring state is better than the current one. At this point, you've reached a local maximum or minimum (depending on whether you're maximizing or minimizing).

```

function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← problem.INITIAL
  while true do
    neighbor ← a highest-valued successor state of current
    if VALUE(neighbor) ≤ VALUE(current) then return current
    current ← neighbor

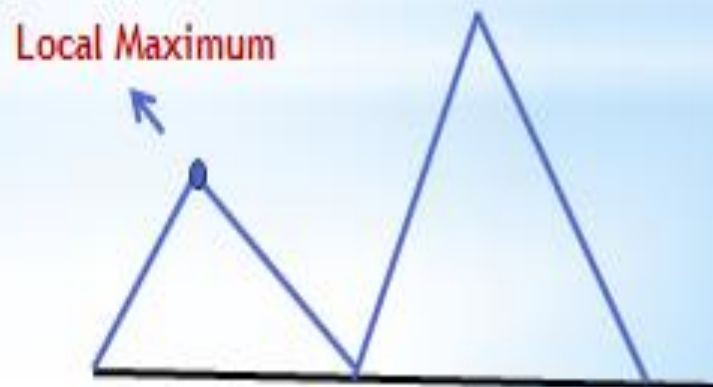
```

Problems in Hill Climbing Algorithm :

1. **Local Maximum** : A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum

Solution :

- Backtracking technique can be a solution of the local maximum in state space landscape.
- Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.

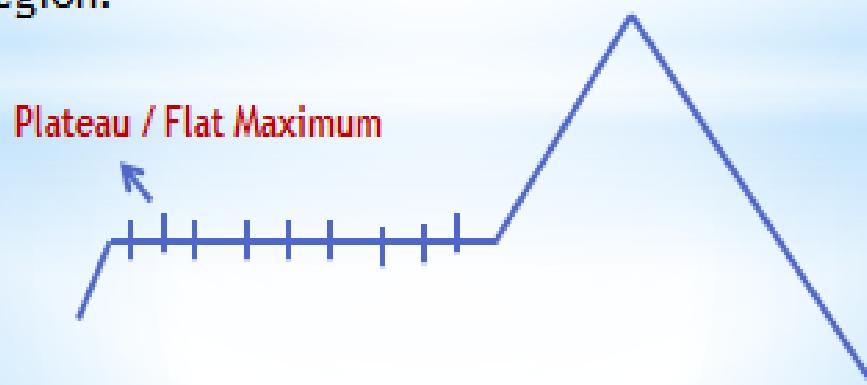


2. Plateau

- A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move.
- A **hill-climbing** search might be lost in the plateau area.

Solution : The solution for the plateau is to take big steps while searching, to solve the problem.

Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.



3. Ridges :

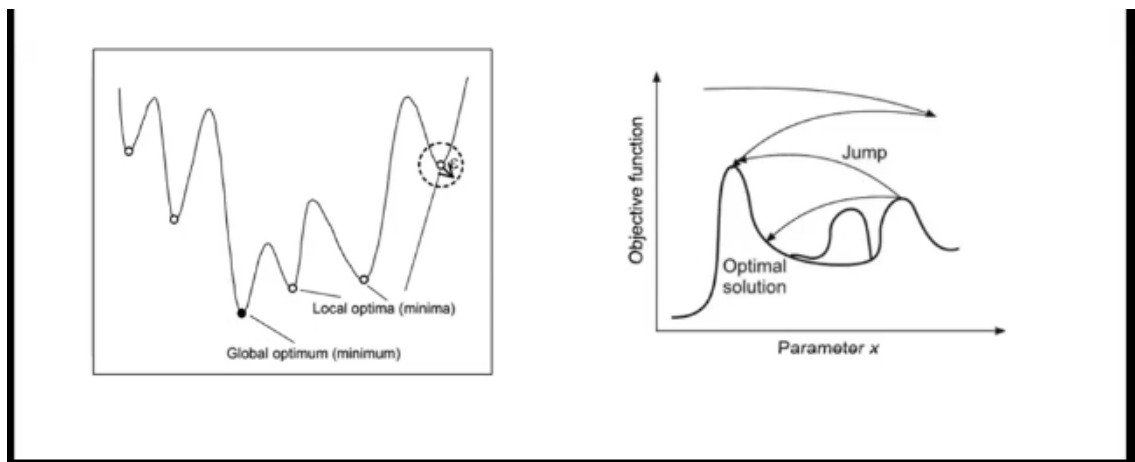
- A ridge is a special form of the local maximum.
- It has an area , which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.

Solution : With the use of bidirectional search, or by moving in different directions, we can improve this problem



Simulated annealing search :

- Simulated Annealing is an optimization algorithm designed to search for an optimal or near-optimal solution in a large solution space.
- The name and concept are derived from the process of annealing in metallurgy, where a material is heated and then slowly cooled to remove defects and achieve a stable crystalline structure.
- In Simulated Annealing, the "heat" corresponds to the degree of randomness in the search process, which decreases over time (cooling schedule) to refine the solution.
- The method is widely used in combinatorial optimization, where problems often have numerous local optima that standard techniques like gradient descent might get stuck in.
- Simulated Annealing excels in escaping these local minima by introducing controlled randomness in its search, allowing for a more thorough exploration of the solution space.



Local Search in Continuous Spaces :

Local search in continuous spaces aims to find the global optimum (minimum or maximum) of an objective function $f(x)$, where x is a vector of continuous variables $x=(x_1,x_2,\dots,x_n)$ and $x_i \in \mathbb{R}$.

Main Concepts in Continuous Local Search :

1. **Objective Function ($f(x)$):** The function that we want to optimize (minimize or maximize).
2. **Search Space:** The multi-dimensional space where x resides, often defined by bounds on each variable (e.g., $x_i \in [L_i, U_i]$).
3. **Neighborhood:** For a given point x , its neighborhood consists of points that are "close" to x according to some distance metric (e.g., Euclidean distance).
4. **Gradient:** If the function is differentiable, the gradient $\nabla f(x)$ points in the direction of the steepest ascent (for maximization) or descent (for minimization).
5. **Step Size (Learning Rate):** How far to move in the chosen direction in each iteration. A crucial parameter for many methods.

Challenges in Continuous Spaces:

- **Infinite Search Space:** Unlike discrete problems with a finite number of states, continuous spaces have infinitely many points, making exhaustive search impossible.
- **Local Optima:** The objective function might have multiple local minima (for minimization problems) or maxima (for maximization problems). Local search algorithms are prone to getting stuck in these local optima, failing to find the global optimum.
- **Derivatives:** The objective function might not be differentiable, or its derivatives might be expensive to compute or not well-behaved.

Search Algorithms in Continuous Spaces :

The algorithms generally fall into **two** categories:

a) Gradient-based

b) Gradient-free

a) Gradient-based

- i) **Gradient Descent (for Minimization) / Gradient Ascent (for Maximization)**
- ii) **Stochastic Gradient Descent (SGD)**
- iii) **Newton's Method (and Quasi-Newton Methods like BFGS, L-BFGS)**
- iv) **Conjugate Gradient Method**

i. **Gradient Descent (for Minimization) / Gradient Ascent (for Maximization)**

Move in the direction opposite to the gradient (for minimization) or in the direction of the gradient (for maximization). The gradient indicates the direction of the steepest slope.

ii. **Stochastic Gradient Descent (SGD):**

Particularly useful for optimizing functions that are sums of many components (e.g., loss functions in machine learning where the loss is summed over many data points). Instead of calculating the gradient over the entire dataset, it calculates it on a small random subset (a "mini-batch") or even a single data point.

iii. **Newton's Method (and Quasi-Newton Methods like BFGS, L-BFGS):**

Uses the second derivative (Hessian matrix) to find the optimal step size and direction, aiming to jump directly to the minimum in a quadratic function.

iv. **Conjugate Gradient Method:**

Iteratively refines the search direction to ensure that each new search direction is "conjugate" to the previous ones, meaning it doesn't undo progress made in previous steps.

b) Gradient-free

- i) **Hill Climbing (for continuous spaces)**
- ii) **Simulated Annealing**
- iii) **Nelder-Mead Simplex Method (Amoeba Method)**
- iv) **Random Search / Basin Hopping**

- i) **Hill Climbing (for continuous spaces)** : Generate a random neighbor within a small radius of the current point. If the neighbor has a better objective function value, move to it. Repeat.
- ii) **Simulated Annealing** : Inspired by the annealing process in metallurgy. It's a metaheuristic that allows occasional "uphill" moves (worsening the objective) with a certain probability, especially early in the search. This probability decreases over time (temperature "cools"), making it less likely to accept worse solutions later
- iii) **Nelder-Mead Simplex Method** : Maintains a "simplex" (a geometric figure of $n+1$ points in an n -dimensional space). In each step, it tries to replace the worst point in the simplex with a better one using operations like reflection, expansion, contraction, and shrinking.
- iv) **Random Search/Basin Hopping** : Periodically jump to a completely random point in the search space to try and escape local optima, followed by a local optimization step from that new point.