Unit- 3
Problem-Solving and Search Strategies

Parul® University

Topics:

- Problem-Solving and Search Strategies
- Production Systems and AI
- Uninformed and informed search algorithms
- Adversarial search techniques in game-playing AI

Parul® University

Problem Solving

- By problem solving we mean that, the agent is in a desired, is in some situation and wants to be in some desired situation.
- So, given to desired situation and the task of the agent is to make a series of decisions or a series of moves, which will transform the given situation to the desired situation.
- Problem solving in artificial intelligence may be characterized as a systematic search through a range of possible actions in order to reach some predefined goal or solution.
- In AI problem solving by search algorithms is quite common technique.
- In the coming age of AI, it will have big impact on the technologies of the robotic sand path finding. It is also widely used in travel planning.

Search Strategies

- A search algorithm takes a problem as input and returns the solution in the form of an action sequence.
- Once the solution is found, the actions it recommends can be carried out. This phase is called as the execution phase. After formulating a goal and problem to solve the agent cells a search procedure to solve it.
- **A problem can be defined by 5 components.**

a) The initial state : The state from which agent will start.

b) The goal state : The state to be finally reached.

c) The current state : The state at which the agent is present after starting from the initial state.

d) Successor function : It is the description of possible actions and their outcomes.

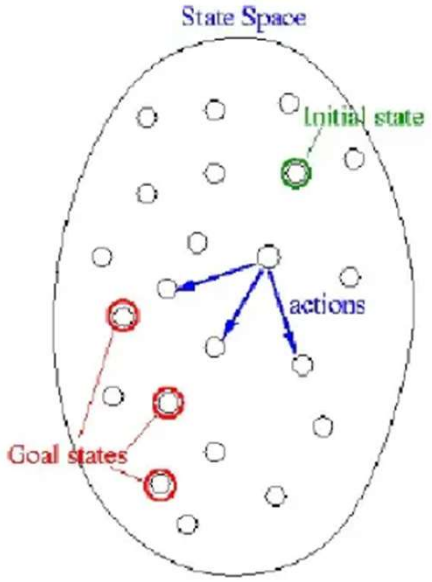
e) Path cost : It is a function that assigns a numeric cost to each path.

Defining Search Problem

- A search problem consists of the following:
- S: the full set of states space
- S_0 : the initial state
- A:S →S is a set of operators (Action States)
- G is the set of final states. Note that G ⊆ S (Goal States)

The Search Problem is to find a sequence of actions which transforms the agent from the initial state to a goal state g ∈ G.

A search problem is represented by a 4-tuple {S , S0 , A,G}.



Parul® University

Search Framework

- *How real the world problem is*

1. State Space Search: how a real-world problem is mapped into a state space and then we can have goals to meet through search algorithms

I. Uninformed/Blind Search : No domain specific information is available
II. Informed/Heuristic search: functions to guide the search procedural

2. Problem Reduction Search: problems which could be decomposed into sub problems in a flavor similar to dynamic problems. In addition, how best to solve a problem

3. Game tree Search: gives idea how chess playing game works

4. Advances:
1. Memory Bounded search
2. Multi Objective Search
3. Learning how to search

Parul® University

Search Algorithm

1. **Initialize**: set OPEN=s, CLOSED={}
2. **Fail:** If OPEN={}, terminate with failure
3. **Select:** Select a state from OPEN and save in CLOSED
4. **Terminate**: If n∈G, terminate with success
5. **Expand:** generate the successor of n

   For each successor, m, insert m in OPEN only if

   m∈[OPEN∪CLOSED]
6. **Loop**: Goto step 2

Parul® University

Evaluating Search Strategies

- To find minimum set of operators to reach to goal
- We will look at the following three factors to measure performance, characteristics and efficiency of various search strategies.

1. **Completeness:** Is the strategy guaranteed to find a solution if one exists?
2. **Optimality:** Does the solution have low cost or the minimal cost?
3. What is the **search cost** associated with the time and memory required to find a solution?
   a. **Time complexity**: Time taken (number of nodes expanded) (worst or average case) to find a solution.
   b. **Space complexity**: Space used by the algorithm measured in terms of the maximum size of fringe

Search Techniques

Parul® University

- Generate-And-Test, Hill Climbing, Best-First Search, Problem Reduction, Constraint Satisfaction, Means-Ends Analysis. Heuristic search, Hill Climbing, Best first search, mean and end analysis, Constraint Satisfaction, A* and AO* Algorithm, Knowledge Representation: Basic concepts, Knowledge representation Paradigms, Propositional Logic, Inference Rules in Propositional Logic, Knowledge representation using Predicate logic, Predicate Calculus, Predicate and arguments, ISA hierarchy, Frame notation, Resolution, Natural Deduction

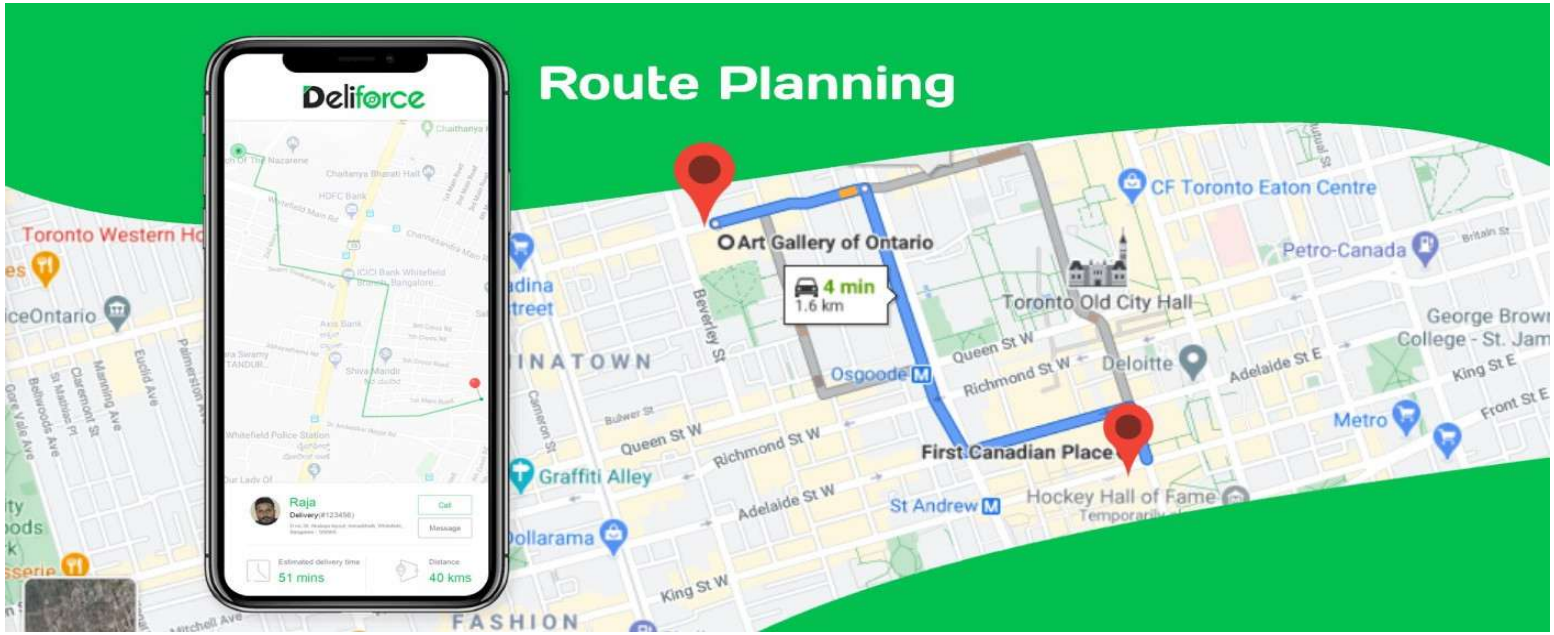
Search techniques are fundamental to problem-solving in Artificial Intelligence (AI). They equip AI agents with the ability to navigate through a vast amount of possibilities (search space) to find the optimal solution that achieves a specific goal.

Key components of a search problem in AI:

State Space: The collection of all possible states the agent can be in.
Start State: The agent's initial state.
Goal State: The state the agent desires to reach.
Transition Function: A function that defines the legal actions that can move the agent from one state to another.
Cost Function: A function that assigns a cost to each transition.

Parul®
University

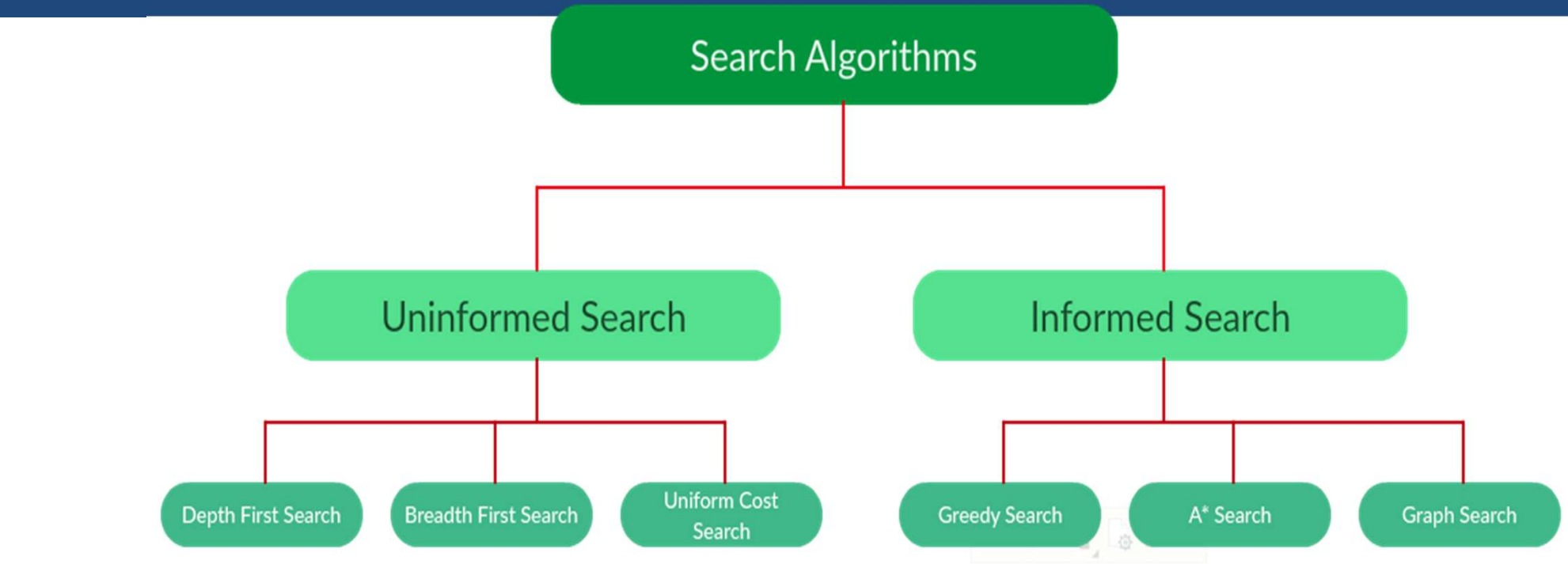
DIGITAL LEARNING CONTENT



- Following are the four essential properties of search algorithms to compare the efficiency of these algorithms:
- **Completeness:** A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.
- **Optimality:** If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.
- **Time Complexity:** Time complexity is a measure of time for an algorithm to complete its task.
- **Space Complexity:** It is the maximum storage space required at any point during the search, as the complexity of the problem.

- **Types of search algorithms:**

- There are far too many powerful search algorithms out there to fit in a single article. Instead, this article will discuss *six* of the fundamental search algorithms, divided into *two* categories, as shown below.


Search Algorithms
Uninformed Search
Informed Search
Depth First Search
Breadth First Search
Uniform Cost Search
Greedy Search
A* Search
Graph Search

Uninformed Search Algorithms:

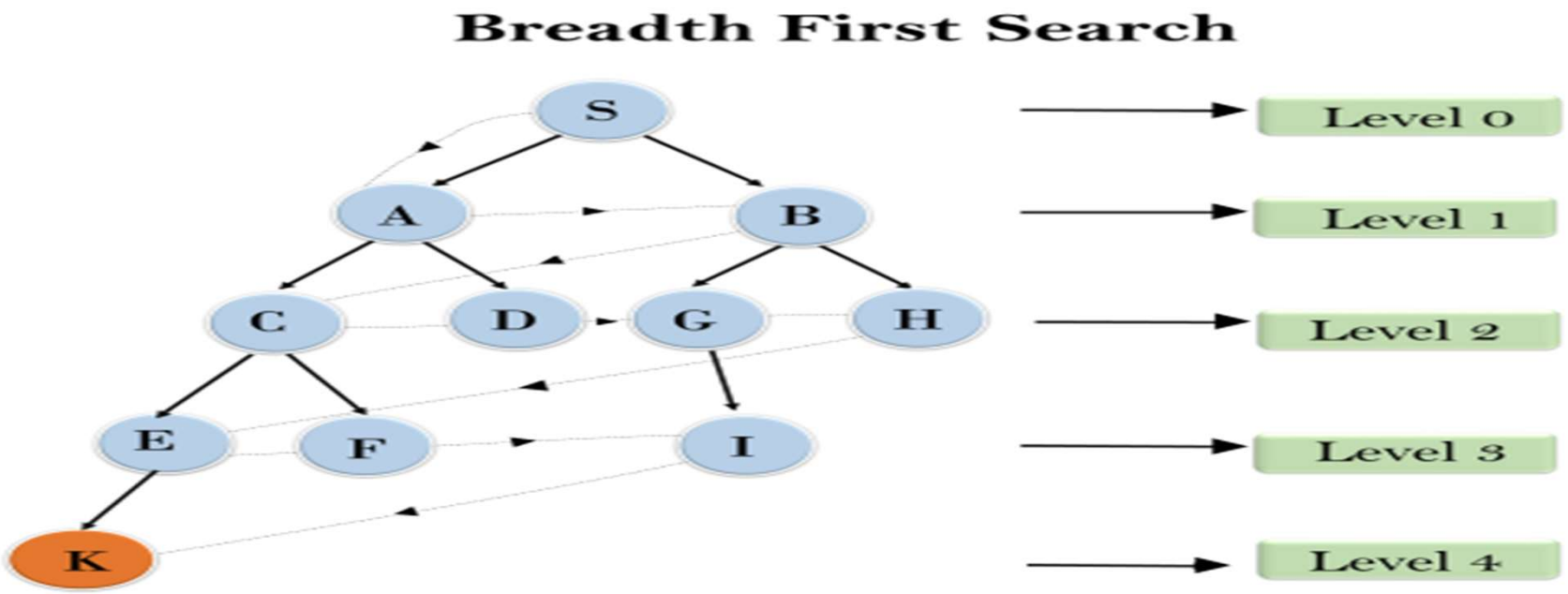
- The search algorithms in this section have no additional information on the goal node other than the one provided in the problem definition.
- The plans to reach the goal state from the start state differ only by the order and/or length of actions. Uninformed search is also called **Blind search**.
- These algorithms can only generate the successors and differentiate between the goal state and non goal state.

Informed Search Algorithms:

- Here, the algorithms have information on the goal state, which helps in more efficient searching. This information is obtained by something called a *heuristic*.
- In an informed search, a heuristic is a *function* that estimates how close a state is to the goal state. For example – Manhattan distance, Euclidean distance, etc. (Lesser the distance, closer the goal.)

Breadth First Search

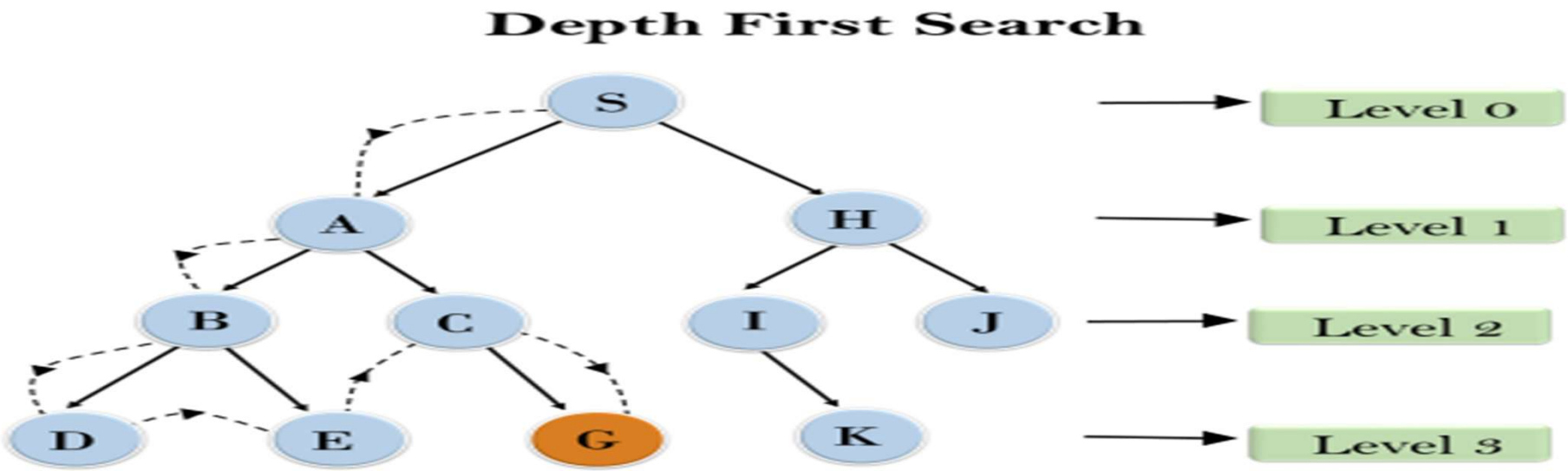
- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.



S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K

Depth-first search (DFS)

- Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures.
- The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.
- It uses last in- first-out strategy and hence it is implemented using a stack.

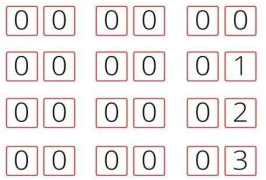


Root node--->Left node ----> right node.

Generate and Test Search

- It is a heuristic search technique based on Depth First Search with Backtracking which guarantees to find a solution
- **Algorithm**
- *Generate a possible solution.*
- *Test to see if this is a actual solution by comparing the chosen point or the endpoint*
- *If a solution is found, quit. Otherwise go to Step 1*

Properties of Good Generators:



- ***Complete:*** *they should generate all the possible solutions*
- ***Non Redundant:*** *Good Generators should not yield a duplicate solution*
- ***Informed:*** *Good Generators have the knowledge about the search space*
- In this case, one way to find the required pin is to generate all the solutions in a brute force manner for example,
- The total number of solutions in this case is $(100)^3$ which is approximately 1M.
- Now consider using heuristic function where we have domain knowledge that every number is a prime number between 0-99 then the possible number of solutions are $(25)^3$ which is approximately 15,000.

Hill Climbing

- A hill-climbing algorithm is a local search algorithm that moves continuously upward (increasing) until the best solution is attained.
- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

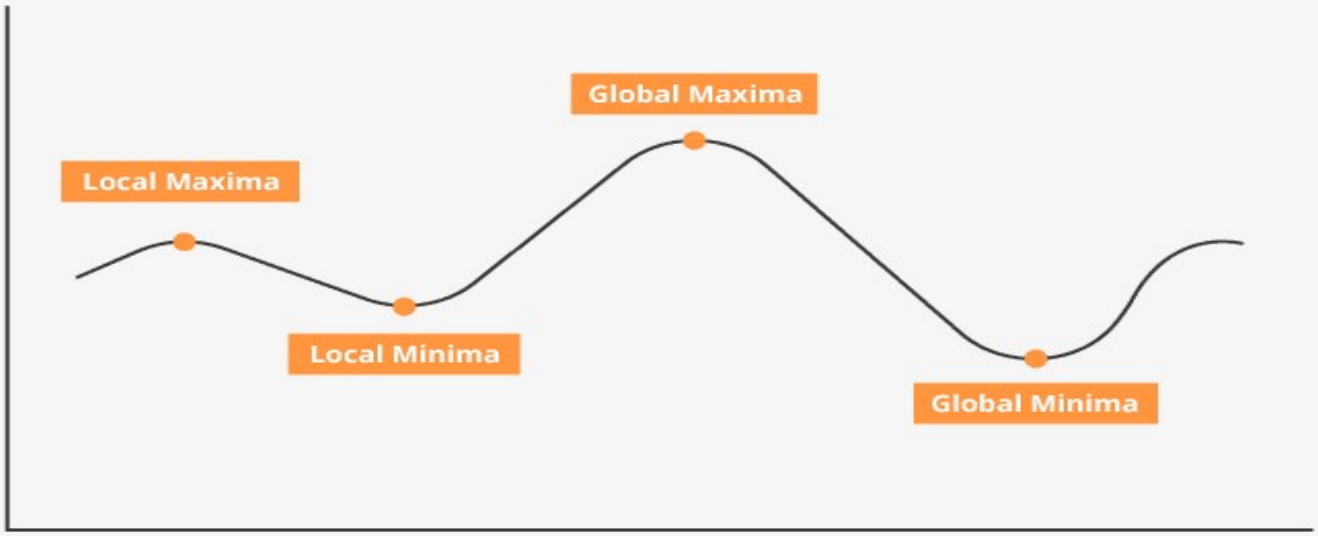
Algorithm(Hill Climbing)

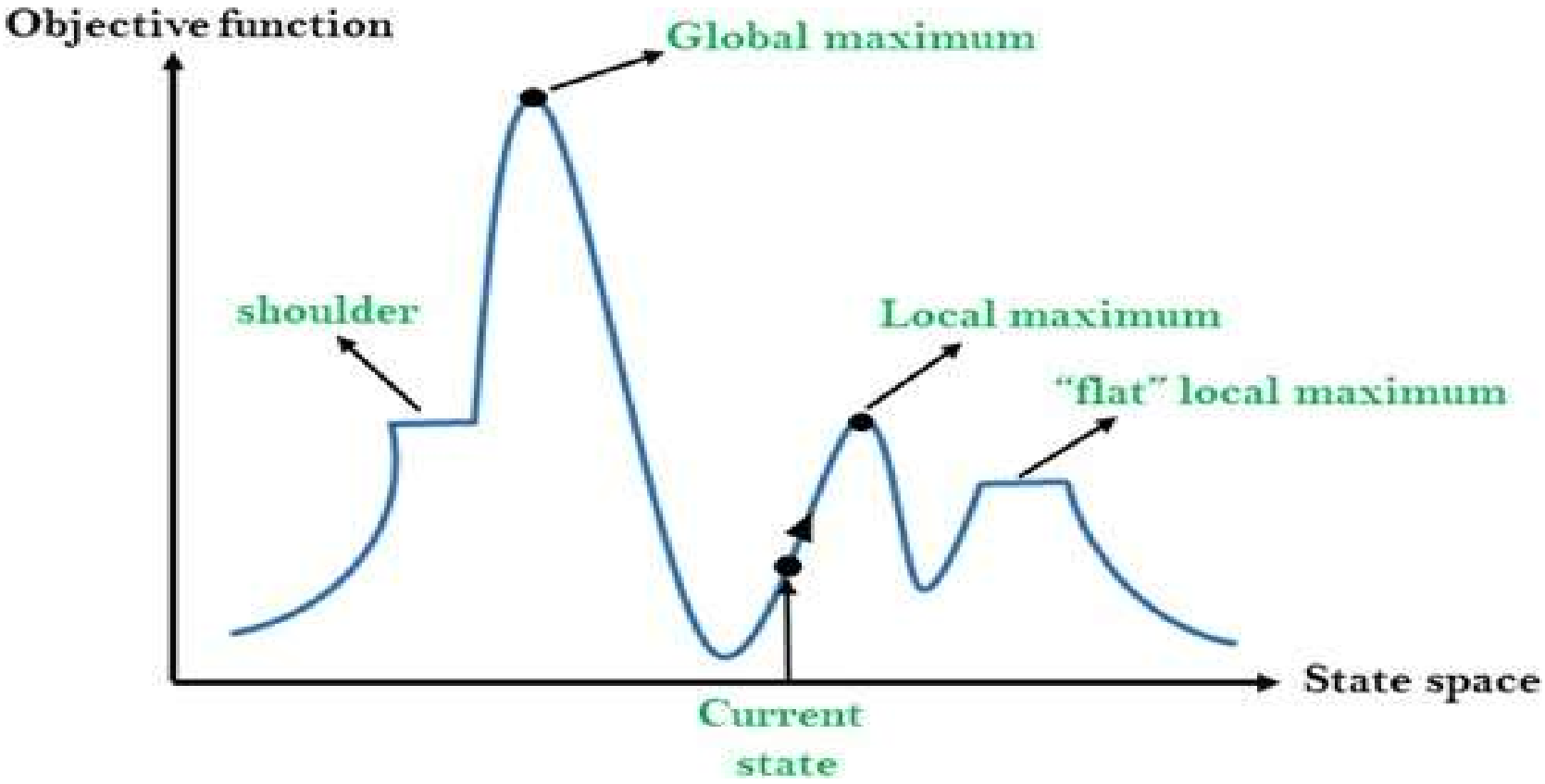
- 1. Examine the current state, If Current State = Goal State, Return success and exit
- Else if New state is better than current state then Goto New state
- return to step 1
- Exit

Different regions in the state space landscape:

- **Local Maximum:** Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.
- **Global Maximum:** Global maximum is the best possible state of state space landscape. It has the highest value of objective function.
- **Current state:** It is a state in a landscape diagram where an agent is currently present.
- **Flat local maximum:** It is a flat space in the landscape where all the neighbor states of current states have the same value.
- **Shoulder:** It is a plateau region which has an uphill edge.

Hill Climbing in Artificial Intelligence




Objective function
Global maximum
shoulder
Local maximum
"flat" local maximum
State space
Current
state

Advantage of Hill Climbing algorithm:

- simple and intuitive algorithm that is easy to understand and implement.
- efficient in finding local optima good choice for problems where a good solution is needed quickly.
- The algorithm can be easily modified and extended

Disadvantages of Hill Climbing algorithm

- It can get stuck in local maxima, and may never find the global maxima of the problem.
- a poor initial solution may result in a poor final solution.
- does not explore the search space very thoroughly
- It may be less effective than other optimization algorithms, such as genetic algorithms or simulated annealing, for certain types of problems.

Best-First Search

Informed Search Algorithms

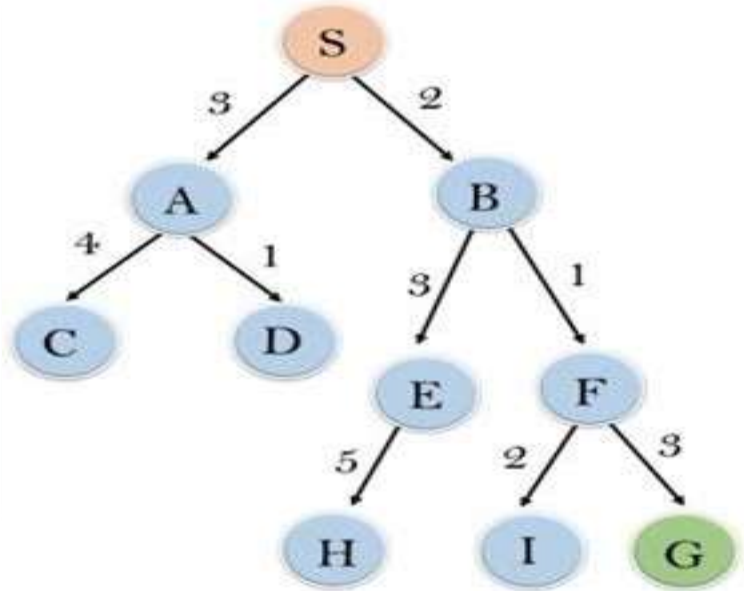
Heuristics function(used in Informed Search, and it finds the most promising path**)**

Greedy Search(always selects the path which appears best at that moment)

Best first search algorithm:

- **Step 1:** Place the starting node into the OPEN list.
- **Step 2:** If the OPEN list is empty, Stop and return failure.
- **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
- **Step 4:** Expand the node n, and generate the successors of node n.
- **Step 5:** Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- **Step 6:** For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
- **Step 7:** Return to Step 2.

Eg:



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

- Advantages:
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.
- Disadvantages:
- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.
- **Time Complexity:** The worst case time complexity of Greedy best first search is $O(b^m)$.
- **Space Complexity:** The worst case space complexity of Greedy best first search is $O(b^m)$.
- Where b is the number of legal moves at each point and m is the maximum depth of the tree.

Problem Reduction

- divide and conquer strategy
- decomposing it into smaller sub-problems
- sub-solutions can then be recombined to get a solution as a whole
- is called **Problem Reduction**.
- method generates arcs: AND arc(connected) and OR arc

- AND Arc eg: Charging a phone
- Phone & charger & power supply
- OR arc eg: Unlocking a phone
- Face or Pin or fingerprint

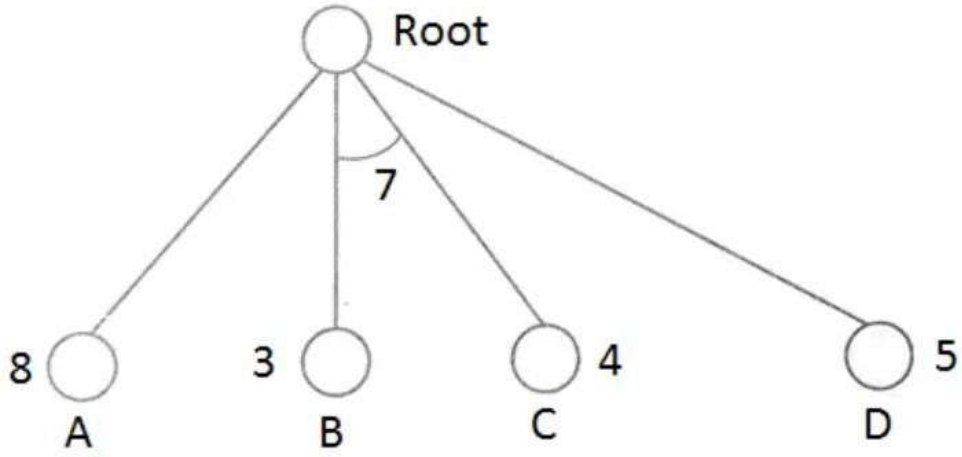


Fig: AND / OR Tree

- What is Heuristics?
- A heuristic is a technique that is used to solve a problem faster than the classic methods. These techniques are used to find the approximate solution of a problem when classical methods do not. Heuristics are said to be the problem-solving techniques that result in practical and quick solutions.
- Heuristics are strategies that are derived from past experience with similar problems. Heuristics use practical methods and shortcuts used to produce the solutions that may or may not be optimal, but those solutions are sufficient in a given limited timeframe.

Constraint Satisfaction

- constraint satisfaction means *solving a problem under certain constraints or rules.*
- *problem is solved when its values satisfy certain constraints or rules of the problem*
- Constraint satisfaction depends on three components, namely:
- **X:** It is a set of variables.
- **D:** It is a set of domains where the variables reside. There is a specific domain for each variable.
- **C:** It is a set of constraints which are followed by the set of variables.

- The constraint value consists of a pair of **{scope, rel}**. The **scope** is a tuple of variables which participate in the constraint and **rel** is a relation which includes a list of values which the variables can take to satisfy the constraints of the problem.
- C1=((v1,v2) (v1<>v2))
- Constraint1 =scope=V1 &V2 where V1<>V2

Means-Ends Analysis Algorithm

- We have studied the strategies which can reason either in forward or backward
- but a mixture of the two directions is used to solve complex problems
- It limits the search
- Works

1.First, evaluate the difference between Initial State and final State. 2.Select the various operators which can be applied for each difference. 3.Apply the operator at each difference, which reduces the difference between the current state and goal state.

Algorithm for Means-Ends Analysis:

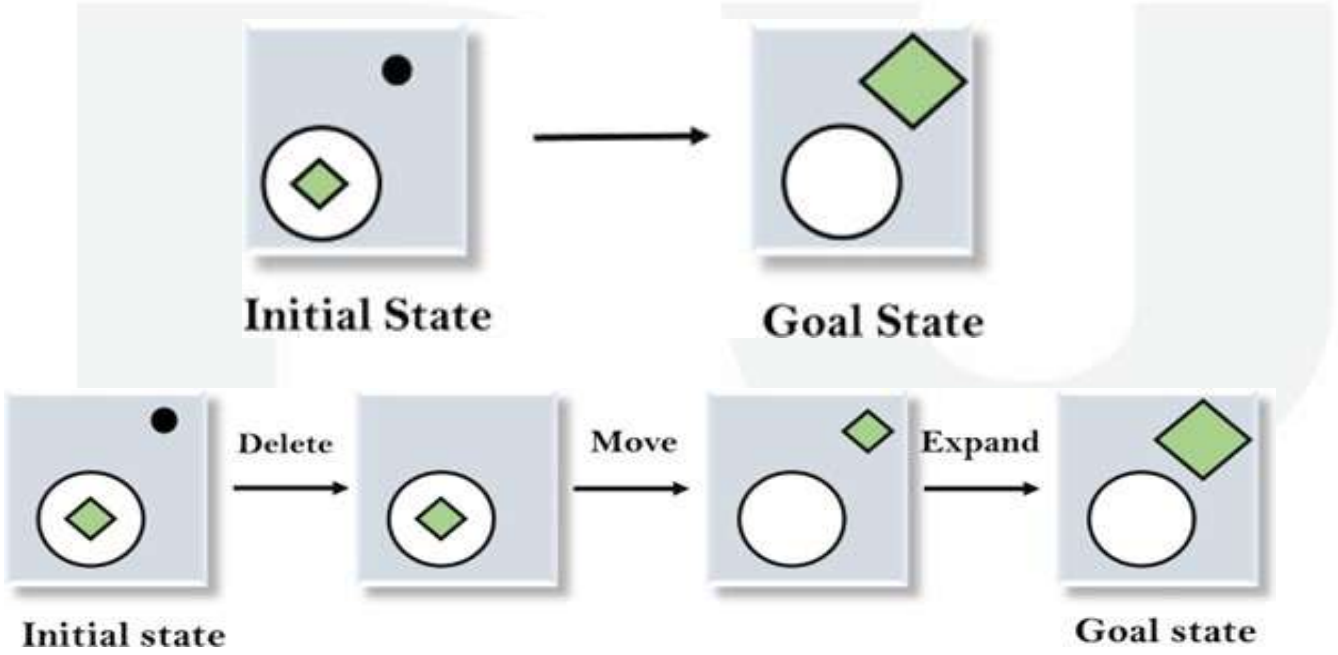
- **tep 1:** Compare CURRENT to GOAL, if there are no differences between both then return Success and Exit.
- **Step 2:** Else, select the most significant difference and reduce it by doing the following steps until the success or failure occurs.
  - Select a new operator O which is applicable for the current difference, and if there is no such operator, then signal failure.
  - Attempt to apply operator O to CURRENT. Make a description of two states.
    i) O-Start, a state in which O?s preconditions are satisfied.
    ii) O-Result, the state that would result if O were applied In O-start.
  - If

(First-Part <------ MEA (CURRENT, O-START)

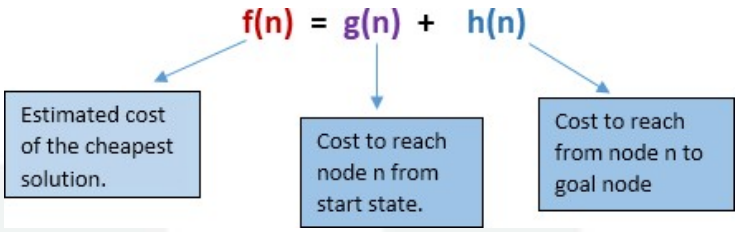
And

(LAST-Part <----- MEA (O-Result, GOAL), are successful, then signal Success and return the result of combining FIRST-PART, O, and LAST-PART.

Example of Mean-Ends Analysis:



A* Algorithm



- A* search is the most commonly known form of best-first search.
- used to find the shortest path between two nodes in a graph
- Fitness Function(F Score) = Actual Cost + Estimated Cost
- It is widely used in various applications, including robotics, video games, and route planning systems.
- The cost function combines the actual cost of reaching a node from the start node (known as the "g-score") and an estimated cost of reaching the goal node from the current node

Algorithm of A* search:

- **Step1:** Place the starting node in the OPEN list.
- **Step 2:** Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
- **Step 3:** Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise
- **Step 4:** Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.
- **Step 5:** Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
- **Step 6:** Return to **Step 2**.

Advantages of A*

- Optimality: A* guarantees finding the shortest path between a given start and goal node, provided that certain conditions are met
- Efficiency: A* is typically more efficient than brute-force search algorithms like breadth-first search or depth-first search, especially in large graphs.
- Flexibility: A* can be applied to a wide range of problems, including pathfinding in grids, maps, or networks
- Speed: A* is often faster than other uninformed search algorithms because it intelligently prioritizes the exploration of nodes.

Disadvantages of A*

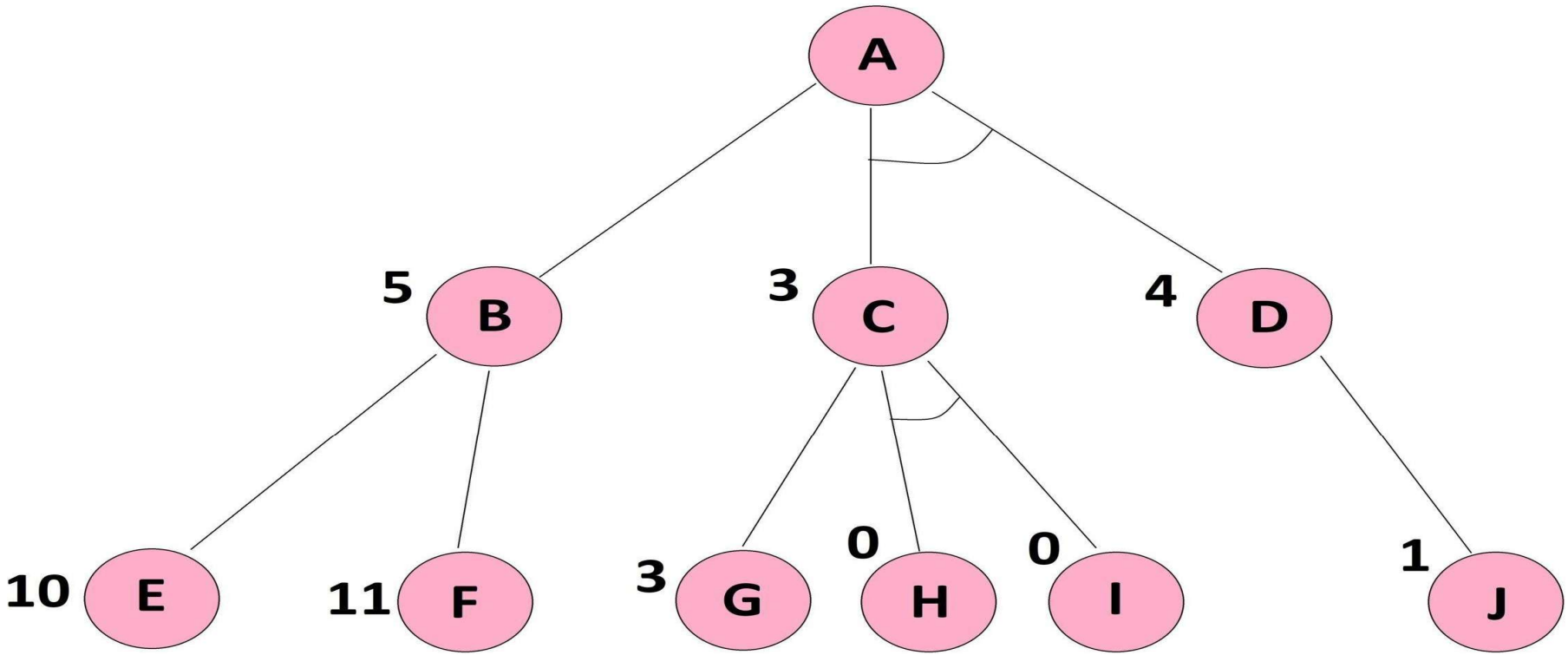
- Heuristic Accuracy: The performance of A* heavily relies on the accuracy of the heuristic function used
- Memory Requirements: A* needs to keep track of the nodes in the open and closed sets during the search process.
- Computational Complexity: Although A* is generally efficient, its worst-case time complexity is exponential. In graphs with many possible paths or when using a poor heuristic, A* can degrade in performance. In such cases, alternative algorithms like Dijkstra's algorithm (which guarantees optimality but lacks the heuristic-guided search) might be more suitable.
- Path Reoptimization: A* assumes that the environment and costs remain static throughout the search process

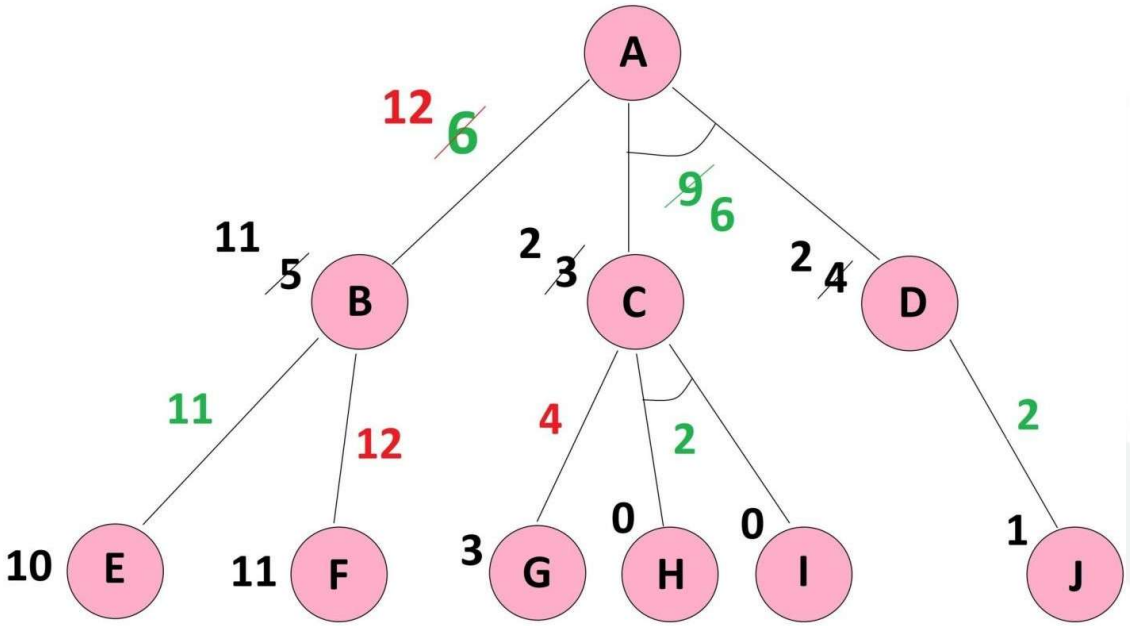
Difference between A* and Dijkstra's algorithm (Extra Not in Syllabus, need not learn)

1. Goal-directed search: A* is a goal-directed search algorithm, meaning it incorporates a heuristic function that estimates the cost from each node to the goal. This heuristic guides the search and helps prioritize the exploration of nodes that are more likely to lead to the goal. On the other hand, Dijkstra's algorithm is an uninformed search algorithm that explores all nodes uniformly without considering any heuristic information.
2. Optimality: A* guarantees finding the shortest path from the start node to the goal node, given an admissible heuristic function. It achieves optimality by considering both the cost of reaching each node from the start and the estimated cost to the goal. Dijkstra's algorithm, on the other hand, also finds the shortest path from the start node to all other nodes in the graph but does not consider a heuristic. It explores all nodes until it reaches the goal, resulting in a higher computational cost in some cases.
3. Memory usage: A* typically uses more memory than Dijkstra's algorithm because it needs to store additional information like the heuristic values for each node. A* maintains two sets: an open set and a closed set, whereas Dijkstra's algorithm only needs a priority queue or a min-heap to prioritize nodes based on their tentative distances.
4. Time complexity: In terms of time complexity, both algorithms have a similar worst-case scenario. They both have a time complexity of O((V + E) log V), where V is the number of nodes and E is the number of edges in the graph. However, in practice, A* tends to be more efficient due to its goal-directed nature and the ability to prune unpromising paths using the heuristic information.
5. Application domain: A* is commonly used in pathfinding problems where the goal is known in advance and finding the shortest path is the primary objective. It is suitable for problems like grid-based navigation, game AI, and route planning. Dijkstra's algorithm, on the other hand, is more general and applicable in scenarios where finding the shortest path from a source node to all other nodes is required, without considering any goal-directed information.

AO* Algorithm

- AND OR
- AO* (Anytime Repairing A*)
- it is an **extension** of the A* algorithm
- AO* is useful in scenarios where the **entire graph is not known in advance**, or the **graph can change dynamically** over time.
- begins with a limited knowledge of the graph and **continuously refines its solution as it acquires more information.**
- The key idea behind AO* is the concept of "**consistent subgraphs**." Instead of having a single graph representing the entire problem space, AO* maintains multiple subgraphs, each representing a consistent subset of the problem space. As the algorithm explores new areas of the graph, it expands and repairs the subgraphs to incorporate the new information.


A
5
B
3
C
4
D
10
E
11
F
3
G
0
H
0
I
1
J



- This updated P(A-C-D) with the cost of 6 is still less than the updated P(A-B) with the cost of 12, and therefore, the minimum cost path from A to the goal node goes from P(A-C-D) by the cost of 6

Ex
:

- .....
- ...S.....
- .....
- ..XXXXX.....
- .....X.....
- .....X.....
- .....X...G.....
- .....

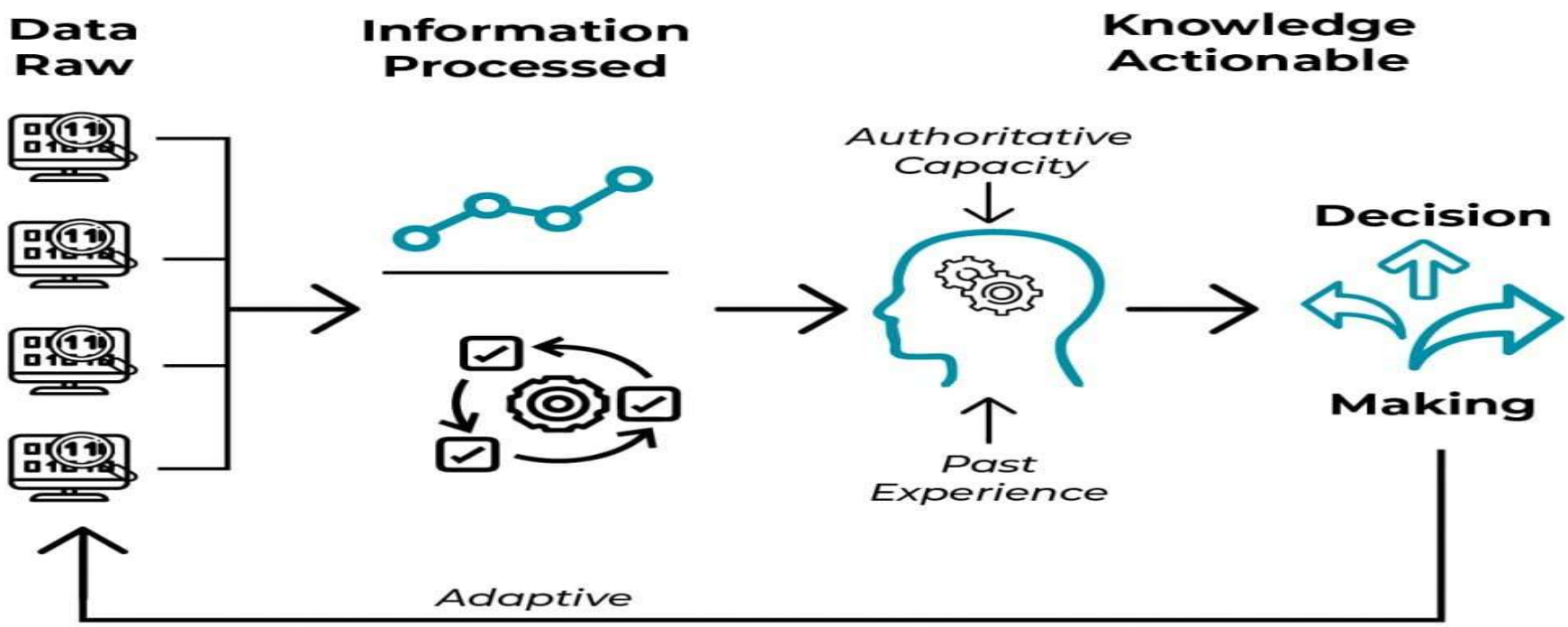
Legend:
S - Start position
G - Goal position
X – Obstacle

- In the initial stage, the **agent's visibility is limited to a 3x3 grid** centered around its current position.
- As the **agent moves forward, new portions of the graph become visible**, and the AO* algorithm incrementally expands the subgraph, incorporating the new information.

A* and AO* Algorithm

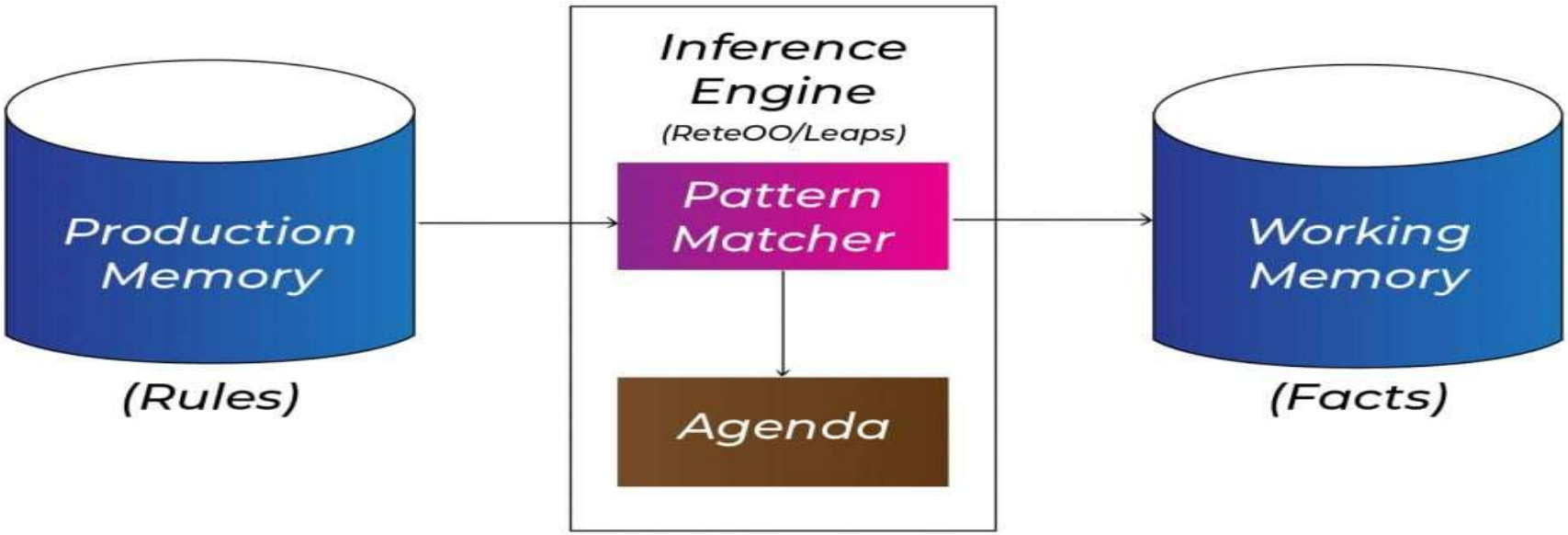
- AO* uses the same cost function as A*, combining the g-score and h-score to determine the priority of nodes. **The main difference lies in the *handling of incomplete information* and the *ability to refine the solution* as more information is gathered.**
- In summary, **A* is a widely used algorithm for finding the shortest path in a graph**, while **AO* extends** A* to handle dynamic or **incomplete graphs by incrementally refining the solution based on new information.**

ANALYTIXLABS



ANALYTIXLABS

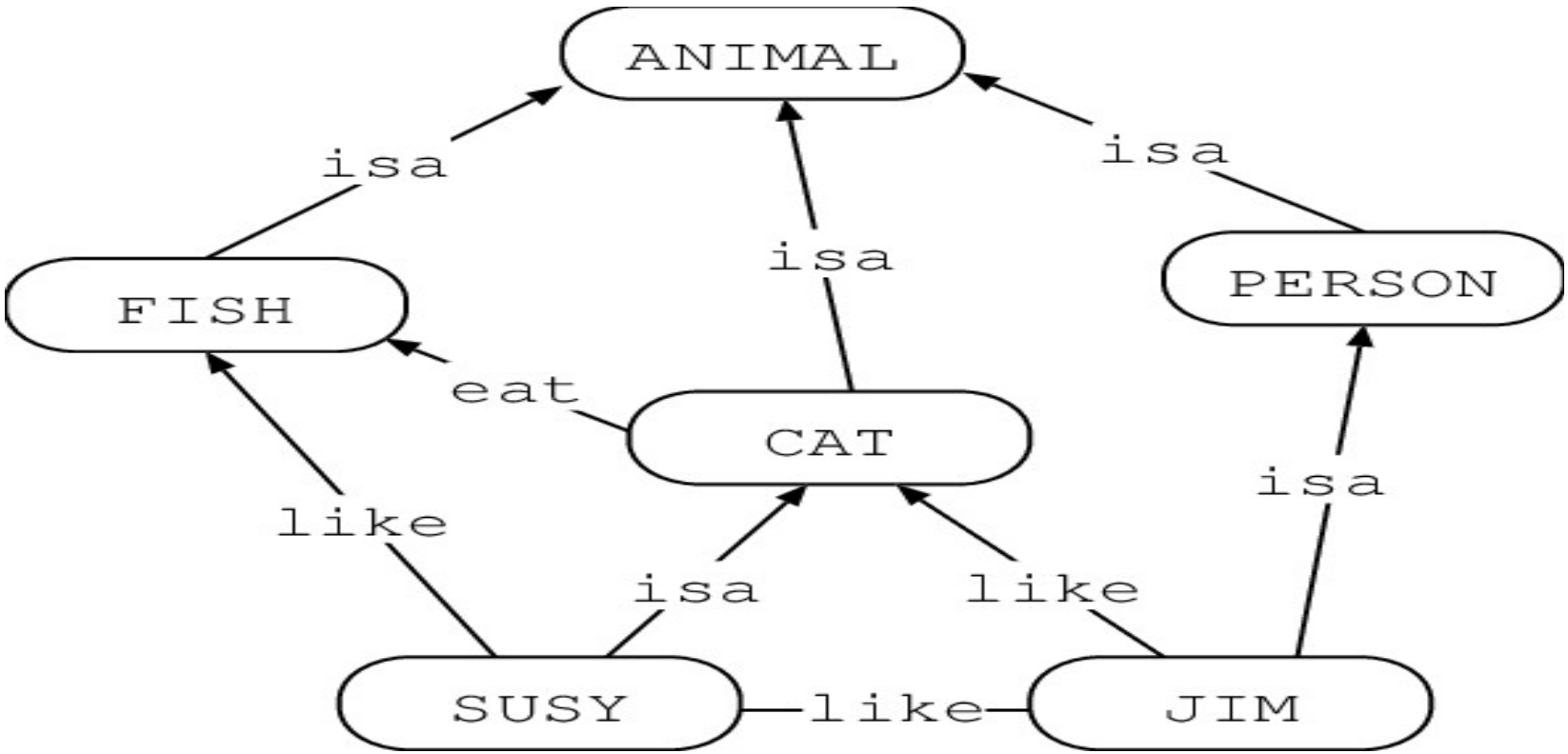
PRODUCTION RULES

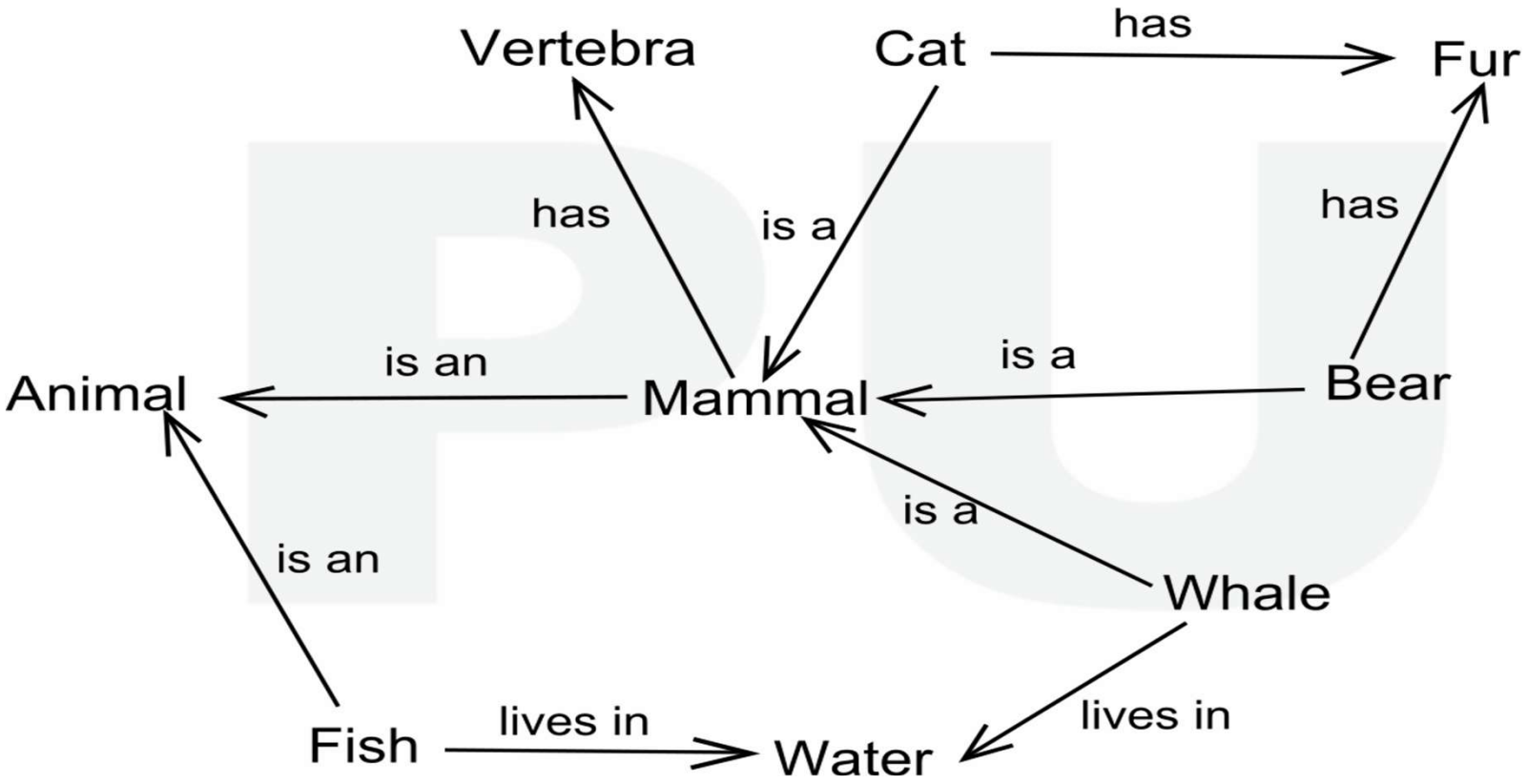


Knowledge representation Paradigms

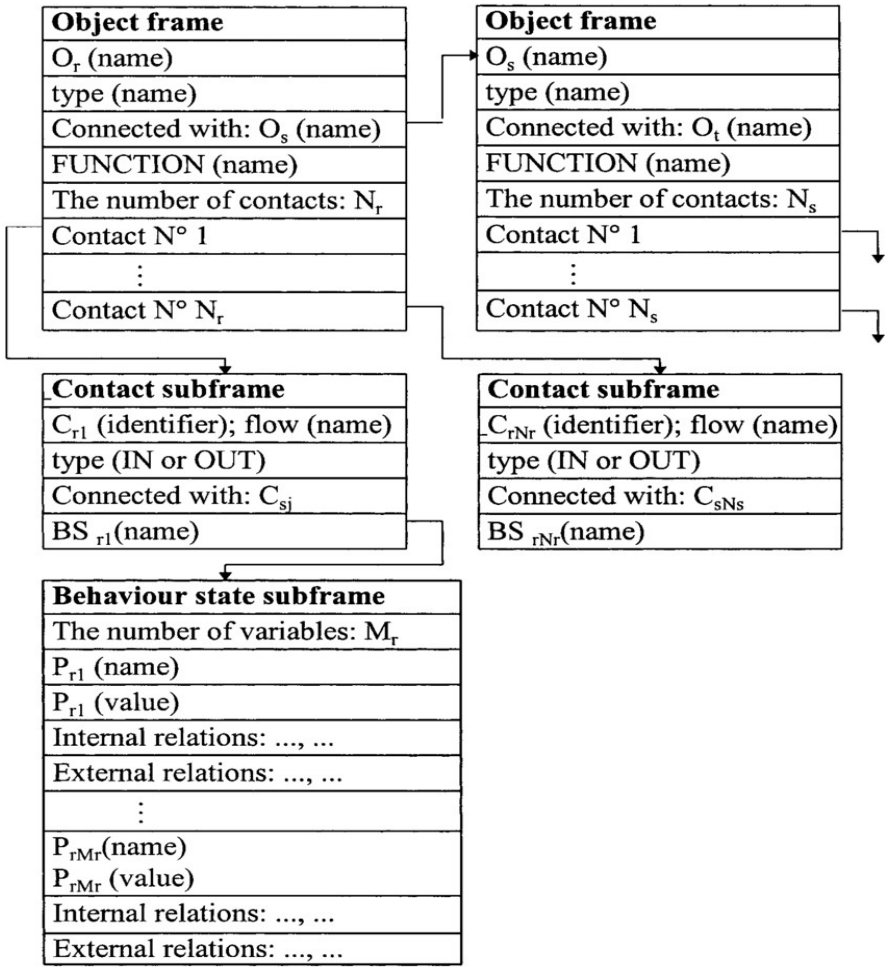
- Knowledge representation paradigms are the different ways that knowledge can be encoded for use by computer programs. These paradigms are crucial in artificial intelligence (AI) for tasks like problem-solving, reasoning, and decision-making.
- Here are some of the most common knowledge representation paradigms:
- Logical Knowledge Representation: This paradigm uses formal logic to represent knowledge. Propositions, which are statements that can be true or false, are used to represent facts. Inferences are made by applying logical rules to these

- Semantic Networks: Semantic networks represent knowledge as a graph of nodes and links. Nodes represent concepts, and links represent relationships between concepts. Semantic networks are good for representing relationships between entities and for inheritance reasoning.


ANIMAL
isa
isa
isa
FISH
PERSON
eat
CAT
like
isa
isa
like
SUSY
like
JIM


Vertebra
Cat
has
Fur
has
is a
has
is an
is a
Animal
Mammal
Bear
is a
is an
Whale
Fish
lives in
Water
lives in

- **Frames:** Frames are a type of data structure that group related information together. A frame typically consists of a set of slots, each of which has a name and a value. Frames are good for representing stereotypical knowledge about objects, events, and situations.


Object frame
O_r (name)
type (name)
Connected with: O_s (name)
FUNCTION (name)
The number of contacts: N_r
Contact N° 1
Contact N° N_r
Object frame
O_s (name)
type (name)
Connected with: O_t (name)
FUNCTION (name)
The number of contacts: N_s
Contact N° 1
Contact N° N_s
Contact subframe
C_{r1} (identifier); flow (name)
type (IN or OUT)
Connected with: C_{sj}
BS $_{r1}$(name)
Contact subframe
C_{rNr} (identifier); flow (name)
type (IN or OUT)
Connected with: C_{sNs}
BS $_{rNr}$(name)
Behaviour state subframe
The number of variables: M_r
P_{r1} (name)
P_{r1} (value)
Internal relations: ..., ...
External relations: ..., ...
P_{rMr}(name)
P_{rMr} (value)
Internal relations: ..., ...
External relations: ..., ...

- **Production Rules:** Production rules are a type of knowledge representation that consists of a set of condition-action pairs. The condition part of a rule specifies a set of circumstances that must be true for the rule to fire. The action part of a rule specifies the action that should be taken when the rule fires. Production rules are good for representing procedural knowledge, such as how to solve a problem or how to perform a task.
- **Knowledge Base:** A knowledge base is a repository that stores the represented knowledge. structured database, a collection of rules, an ontology, or a combination of various representations.
- Effective knowledge representation is essential for building intelligent systems that can understand, reason, learn, and communicate.

Propositional Logic, Inference Rules in Propositional Logic

- propositions are statements that can be either true or false
- Logical Operators in Propositional Logic:
- Negation (¬): eg: This is not 2040
- Conjunction (A): AND: Pay fees AND Use Gym
- Disjunction (V): (OR): Either lecture or Phone
- Implication (→) represents a conditional statement, "if p, then q.“
- Eg: A: "It is raining." B: "I will take an umbrella.“ A → B
- Biconditional (↔): "p if and only if q.“
- Eg A: "I have a valid ticket." B: "I can enter the concert."

Knowledge representation using Predicate logic

- Predicate logic, extends propositional logic
- variables, quantifiers, and predicates
- Simple Predicate: Ashoka was a man man(Ashoka) (class, member)
- Ashoka was Maharashtrian Maharashtrian(Ashoka)
- Quantified Statement:
- Universal Quantifier: ∀x Animal(x) - Represents that "For all x, x is an animal.“ means every x is an Animal
- Existential Quantifier: ∃x Carnivore(x) - Represents that "There exists an x that is a carnivore.“ means out of all x atleast 1 x is carnivore

- Rules and Implications:
  - Rule: Carnivore(x) → Eats(x, y) - Represents that "If x is a carnivore, then x eats y."
- Negation:
  - Negation: ¬Carnivore(x) - Represents that "It is not true that x is a carnivore."
- Complex Statements:
  - Compound Statement: Animal(x) A Carnivore(x) - Represents that "x is an animal and x is a carnivore."
  - Implication: Animal(x) → ∃y Eats(x, y) - Represents that "If x is an animal, then there exists a y that x eats."

Predicate Calculus, Predicate and arguments

- Predicates: Predicates represent properties or relationships between objects.
- Statements:
  1. Animal(Dog): This statement asserts that Dog is an animal.
- Loves(John, Mary): This statement asserts that John loves Mary
- Universal Quantifier:
- ∀x Animal(x): This statement asserts that "For all x, x is an animal." It implies that every entity in the domain is an animal.
- Existential Quantifier:
- ∃x Loves(x, Dog): This statement asserts that "There exists an x such that x loves Dog." It implies that there is at least one entity that loves the Dog.

ISA hierarchy((Is-a) hierarchy)

- It represents the hierarchical relationships between concepts
- is commonly used in object-oriented programming and knowledge-based systems.
- Here's an example of an ISA hierarchy for animals:

1. Animal (top-level class)
    1. Mammal
        1. Carnivore
        2. Herbivore
    2. Reptile
    3. Bird
    4. Fish

Frame notation

- frame is a record like structure
- collection of attributes and its values to describe an entity

Slots	Filters
Title	Artificial Intelligence
Genre	Computer Science
Author	Peter Norvig
Edition	Third Edition
Year	1996
Page	1152

Slots	Filter
Name	Peter
Profession	Doctor
Age	25
Marital status	Single
Weight	78

- Advantages of frame representation:

1.The frame knowledge representation makes the programming easier by **grouping the related data.**

2.The frame representation is comparably **flexible and used by many applications in AI.**

3.It is **very easy to add** slots for **new attribute and relations**.

4.It is **easy** to include **default data** and to search for **missing values**.

5.Frame representation is **easy to understand** and visualize.

- **Disadvantages of frame representation**:

1.In frame system inference mechanism is **not be easily processed**.

2.Inference mechanism cannot be smoothly proceeded representation.

3.Frame representation has a **much generalized approach**.

- Steps for Resolution:

1.Conversion of facts into first-order logic.

2.Convert FOL statements into CNF

3.Negate the statement which needs to prove (proof by contradiction)

4.Draw resolution graph (unification).

5. Whenever we get an empty clause, stop and report the original theorem is true.

Example #1

1. If It is sunny and warm day you will enjoy
2. If it is raining you will get wet
3. It is warm day
4. It is raining
5. It is sunny

Goal: You will enjoy

Prove: enjoy

Step 1 : Conversion to first order logic

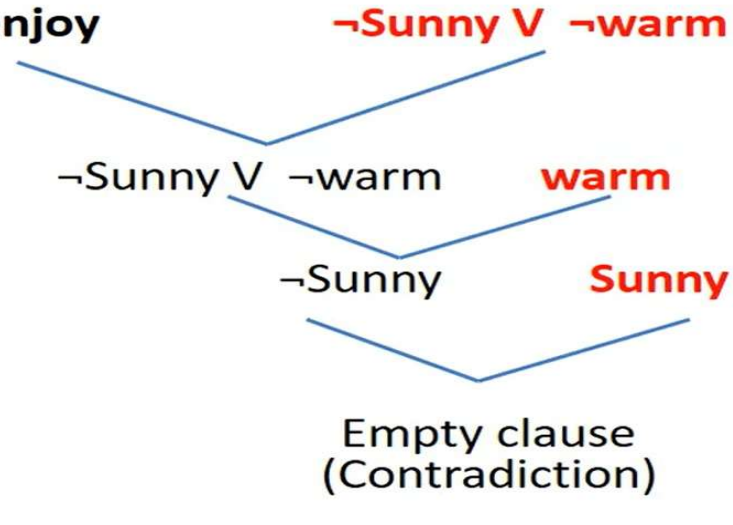
- If It is sunny and warm day you will enjoy
  Sunny ∧ warm → enjoy
- If it is raining you will get wet
  raining → wet
- It is warm day
  warm
- It is raining
  raining
- It is sunny
  Sunny

Step 2 : conversion to CNF

- **Sunny ∧ Warm → enjoy**
  Eliminate implication:
  ¬(Sunny ∧ warm) V enjoy
  Moving negation inside
  ¬Sunny V ¬warm V enjoy
- **raining → wet**
  Eliminate implication:
  ¬raining V wet
- **warm**
- **raining**
- **Sunny**

Step: 3 & 4 Resolution graph

- Negate the statement to be proved: **¬enjoy**
- Now take the statements one by one and create resolution graph.



After applying Proof by Refutation (Contradiction) on the goal, the problem is solved, and it has terminated with a **Null clause** (∅). **Hence, the goal is achieved.**

Example #2

- Consider the following Knowledge Base:

1. The humidity is high or the sky is cloudy.
2. If the sky is cloudy, then it will rain.
3. If the humidity is high, then it is hot.
4. It is not hot.

Goal: It will rain.

Step 1 : Conversion to first order logic

1. **The humidity is high or the sky is cloudy.**
2. If the sky is cloudy, then it will rain.
3. If the humidity is high, then it is hot.
4. It is not hot.

- **Let P : The humidity is high**
- **Let Q: sky is cloudy**

The humidity is high or the sky is cloudy

P V Q

Conversion to first order logic...

1. **The humidity is high or the sky is cloudy.**
2. **If the sky is cloudy, then it will rain.**
3. If the humidity is high, then it is hot.
4. It is not hot.

Goal: It will rain.

- **Let Q: sky is cloudy**
- **Let R: it will rain**

If the sky is cloudy, then it will rain

$Q \rightarrow R$

Conversion to first order logic...

1. The humidity is high or the sky is cloudy.
2. If the sky is cloudy, then it will rain.
3. If the humidity is high, then it is hot.
4. It is not hot.

- **Goal:** It will rain.

- Let P : The humidity is high
- Let S: it is hot

If the humidity is high, then it is hot

$P \rightarrow S$

Conversion to first order logic...

1. The humidity is high or the sky is cloudy.
2. If the sky is cloudy, then it will rain.
3. If the humidity is high, then it is hot.
4. It is not hot.

- **Goal:** It will rain.

- Let S: it is hot

It is not hot

¬S

Statements in first order logic

1. The humidity is high or the sky is cloudy.
2. If the sky is cloudy, then it will rain.
3. If the humidity is high, then it is hot.
4. It is not hot.

- P V Q
- Q→R
- P→S
- ¬S

- **Goal:** It will rain. **(R)**

Convert it to CNF

- P V Q
- Q→R
- P→S
- ¬S

- P V Q
- ¬Q V R
- ¬P V S
- ¬S

Negation of Goal (¬R): It will not rain.

Statements in first order logic

1. The humidity is high or the sky is cloudy.
2. If the sky is cloudy, then it will rain.
3. If the humidity is high, then it is hot.
4. It is not hot.

- P V Q
- Q→R
- P→S
- ¬S

- **Goal:** It will rain. **(R)**

Convert it to CNF

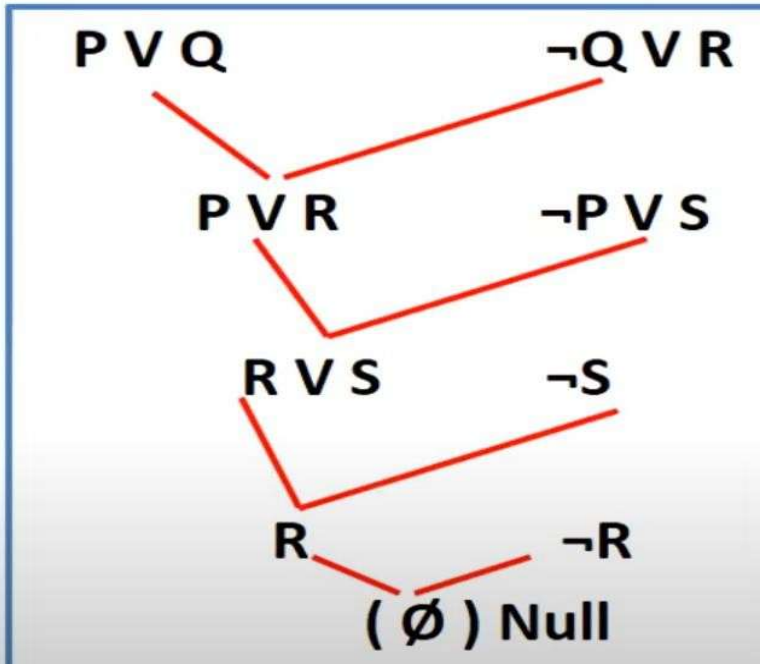
- P V Q
- Q→R
- P→S
- ¬S

- P V Q
- ¬Q V R
- ¬P V S
- ¬S

Negation of Goal (¬R): It will not rain.

Resolution graph

- P V Q
- ¬Q V R
- ¬P V S
- ¬S
- ¬R (Goal)



Example #3

1. Sunil likes all kind of food.
2. Apple and vegetable are food
3. Anything anyone eats and not killed is food.
4. Anil eats peanuts and still alive
5. Sohan eats everything that Anil eats.

- **Prove by resolution that:**

  Sunil likes peanuts

Conversion of Facts/Statements into FOL

1. Sunil likes all kind of food.
2. Apple and vegetable are food
3. Anything anyone eats and not killed is food.
4. Anil eats peanuts and still alive
5. Sohan eats everything that Anil eats
6. Sunil likes peanuts.

1. ∀x : food(x)→likes(Sunil, x)
2. food(apple) ∧ food(vegetable)
3. ∀x ∀y : eats(x , y) ∧ ¬ killed(x)→food(y)
4. eats(Anil , Peanut) ∧ alive(Anil)
5. ∀x: eats(Anil, x) → eats(Sohan , x)
6. Likes(Sunil, Peanut)

- ∀x : ¬ killed(x)→alive(x)
- ∀x : alive (x)→¬killed (x)

(added predicates)

Conversion of FOL into CNF: Elimination of →

1. ∀x : food(x)→likes(Sunil, x)
2. food(apple) ∧ food(vegetable)
3. ∀x ∀y : eats(x , y) ∧ ¬ killed(x)→food(y)
4. eats(Anil , Peanut) ∧ alive(Anil)
5. ∀x: eats(Anil, x) → eats(Sohan , x)
6. ∀x : ¬ killed(x)→alive(x)
7. ∀x : alive (x)→¬killed (x)
8. Likes(Sunil, Peanut)

1. ∀x ¬ food(x) V likes(Sunil, x)
2. food(Apple) ∧ food(vegetables)
3. ∀x ∀y ¬ [eats(x, y) ∧ ¬ killed(x)] V food(y)
4. eats (Anil, Peanuts) ∧ alive(Anil)
5. ∀x ¬ eats(Anil, x) V eats(Sohan, x)
6. ∀x¬ [¬ killed(x)] V alive(x)
7. ∀x ¬ alive(x) V ¬ killed(x)
8. likes(Sunil, Peanuts)

Move negation (¬) inwards and rewrite

1. ∀x ¬ food(x) V likes(Sunil, x)
2. food(Apple) ∧ food(vegetables)
3. ∀x ∀y ¬ [eats(x, y) ∧ ¬ killed(x)] V food(y)
4. eats (Anil, Peanuts) ∧ alive(Anil)
5. ∀x ¬ eats(Anil, x) V eats(Sohan, x)
6. ∀x¬ [¬ killed(x)] V alive(x)
7. ∀x ¬ alive(x) V ¬ killed(x)
8. likes(Sunil, Peanuts).

1. ∀x ¬ food(x) V likes(Sunil, x)
2. food(Apple) ∧ food(vegetables)
3. ∀x ∀y ¬ eats(x, y) V killed(x) V food(y)
4. eats (Anil, Peanuts) ∧ alive(Anil)
5. ∀x ¬ eats(Anil, x) V eats(Sohan, x)
6. ∀x killed(x) V alive(x)
7. ∀x ¬ alive(x) V ¬ killed(x)
8. likes(Sunil, Peanuts).

Drop Universal quantifiers

1. ∀x ¬ food(x) V likes(Sunil, x)
2. food(Apple) Λ food(vegetables)
3. ∀y ∀z ¬ eats(y, z) V killed(y) V food(z)
4. eats (Anil, Peanuts) Λ alive(Anil)
5. ∀w¬ eats(Anil, w) V eats(Sohan, w)
6. ∀g killed(g) V alive(g)
7. ∀k ¬ alive(k) V ¬ killed(k)
8. likes(Sunil, Peanuts).

1. ¬ food(x) V likes(Sunil, x)
2. food(Apple)
3. food(vegetables)
4. ¬ eats(y, z) V killed(y) V food(z)
5. eats (Anil, Peanuts)
6. alive(Anil)
7. ¬ eats(Anil, w) V eats(Sohan, w)
8. killed(g) V alive(g)
9. ¬ alive(k) V ¬ killed(k)
10. likes(Sunil, Peanuts)

Statements "food(Apple) Λ food(vegetables)" and "eats (Anil, Peanuts) Λ alive(Anil)" can be written in two separate statements.

Negate the statement to be proved

- Prove by resolution that: **Sunil likes peanuts.**

  likes(Sunil, Peanuts).

  ¬likes(Sunil, Peanuts)

Resolution graph