

UNIT I

Introduction to AI:

What is Artificial Intelligence?

In today's world, technology is growing very fast, and we are getting in touch with different new technologies day by day.

Here, one of the booming technologies of computer science is Artificial Intelligence which is ready to create a new revolution in the world by making intelligent machines. The Artificial Intelligence is now all around us. It is currently working with a variety of subfields, ranging from general to specific, such as self-driving cars, playing chess, proving theorems, playing music, Painting, etc.

AI is one of the fascinating and universal fields of Computer science which has a great scope in

Artificial Intelligence is composed of two words **Artificial** and **Intelligence**, where Artificial defines "*man-made*," and intelligence defines "*thinking power*", hence AI means "*a man-made thinking power*."

So, we can define AI as:

"It is a branch of computer science by which we can create intelligent machines which can behave like a human, think like humans, and able to make decisions."

Artificial Intelligence exists when a machine can have human based skills such as learning, reasoning, and solving problems

With Artificial Intelligence you do not need to preprogram a machine to do some work, despite that you can create a machine with programmed algorithms which can work with own intelligence, and that is the awesomeness of AI.

It is believed that AI is not a new technology, and some people says that as per Greek myth, there were Mechanical men in early days which can work and behave like humans.

Why Artificial Intelligence?

Before Learning about Artificial Intelligence, we should know that what is the importance of AI and why should we learn it. Following are some main reasons to learn about AI:

- With the help of AI, you can create such software or devices which can solve real-world problems very easily and with accuracy such as health issues, marketing, traffic issues, etc.

- With the help of AI, you can create your personal virtual Assistant, such as Cortana, Google Assistant, Siri, etc.
- With the help of AI, you can build such Robots which can work in an environment where survival of humans can be at risk.
- AI opens a path for other new technologies, new devices, and new Opportunities.

Goals of Artificial Intelligence

Following are the main goals of Artificial Intelligence:

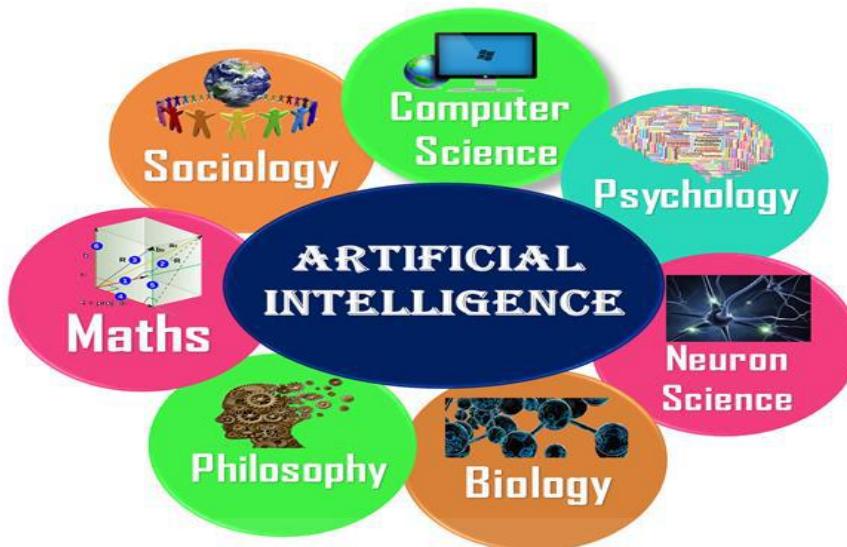
1. Replicate human intelligence
2. Solve Knowledge-intensive tasks
3. An intelligent connection of perception and action
4. Building a machine which can perform tasks that requires human intelligence such as:
 - Proving a theorem
 - Playing chess
 - Plan some surgical operation
 - Driving a car in traffic
5. Creating some system which can exhibit intelligent behavior, learn new things by itself, demonstrate, explain, and can advise to its user.

What Comprises to Artificial Intelligence?

Artificial Intelligence is not just a part of computer science even it's so vast and requires lots of other factors which can contribute to it. To create the AI first we should know that how intelligence is composed, so the Intelligence is an intangible part of our brain which is a combination of **Reasoning, learning, problem-solving perception, language understanding, etc.**

To achieve the above factors for a machine or software Artificial Intelligence requires the following discipline:

- Mathematics
- Biology
- Psychology
- Sociology
- Computer Science
- Neurons Study
- Statistics



Advantages of Artificial Intelligence

Following are some main advantages of Artificial Intelligence:

- **High Accuracy with less errors:** AI machines or systems are prone to less errors and high accuracy as it takes decisions as per pre-experience or information.
- **High-Speed:** AI systems can be of very high-speed and fast-decision making, because of that AI systems can beat a chess champion in the Chess game.
- **High reliability:** AI machines are highly reliable and can perform the same action multiple times with high accuracy.
- **Useful for risky areas:** AI machines can be helpful in situations such as defusing a bomb, exploring the ocean floor, where to employ a human can be risky.
- **Digital Assistant:** AI can be very useful to provide digital assistant to the users such as AI technology is currently used by various E-commerce websites to show the products as per customer requirement.
- **Useful as a public utility:** AI can be very useful for public utilities such as a self-driving car which can make our journey safer and hassle-free, facial recognition for security purpose, Natural language processing to communicate with the human in human-language, etc.

Disadvantages of Artificial Intelligence

Every technology has some disadvantages, and the same goes for Artificial intelligence. Being so advantageous technology still, it has some disadvantages which we need to keep in our mind while creating an AI system. Following are the disadvantages of AI:

- **High Cost:** The hardware and software requirement of AI is very costly as it requires lots of maintenance to meet current world requirements.
- **Can't think out of the box:** Even we are making smarter machines with AI, but still they cannot work out of the box, as the robot will only do that work for which they are trained, or programmed.

- **No feelings and emotions:** AI machines can be an outstanding performer, but still it does not have the feeling so it cannot make any kind of emotional attachment with human, and may sometime be harmful for users if the proper care is not taken.
- **Increase dependency on machines:** With the increment of technology, people are getting more dependent on devices and hence they are losing their mental capabilities.
- **No Original Creativity:** As humans are so creative and can imagine some new ideas but still AI machines cannot beat this power of human intelligence and cannot be creative and imaginative.

Prerequisite

Before learning about Artificial Intelligence, you must have the fundamental knowledge of following so that you can understand the concepts easily:

- Any computer language such as C, C++, Java, Python, etc.(knowledge of Python will be an advantage)
- Knowledge of essential Mathematics such as derivatives, probability theory, etc.

Future of Artificial Intelligence:

1. Health Care Industries

India is 17.7% of the worlds' population that makes it the second-largest country in terms of China's population. Health care facilities are not available to all individuals living in the country. It is because of the lack of good doctors, not having good infrastructure, etc. Still, there are people who couldn't reach to doctors/ hospitals. AI has the ability to provide the facility to detect disease based on symptoms; even if you don't go to the doctor, AI would read the data from Fitness band/medical history of an individual to analyze the pattern and suggest proper medication and even deliver it on one's fingertips just through cell-phone.

As mentioned earlier Google's deep mind has already beaten doctors in detecting fatal diseases like breast cancer. It's not far away when AI will be detecting common disease as well as providing proper suggestions for medication. The consequences of this could be: no need for doctors in the long term result in JOB reduction.

2. AI in Education

The development of a country depends on the quality of education youth is getting. Right now, we can see there are lots of courses are available on AI. But in the future AI is going to transform the classical way of education. Now the world doesn't need skilled labourers for manufacturing industries, which is mostly replaced by robots and automation. The education system could be quite effective and can be according to the individual's personality and ability. It would give chance brighter students to shine and to imbecile a better way to cop up.

Right Education can enhance the power of individuals/nations; on the other hand, misuse of the same could lead to devastating results.

3. AI in Finance

Quantification of growth for any country is directly related to its economic and financial condition. As AI has enormous scope in almost every field, it has great potential to boost individuals' economic health and a nation. Nowadays, the AI algorithm is being used in managing equity funds.

An AI system could take a lot number of parameters while figuring out the best way to manage funds. It would perform better than a human manager. AI-driven strategies in the field of finance are going to change the classical way of trading and investing. It could be devastating for some fund managing firms who cannot afford such facilities and could affect business on a large scale, as the decision would be quick and abrupt. The competition would be tough and on edge all the time.

4. AI in Military and Cybersecurity

AI-assisted Military technologies have built autonomous weapon systems, which won't need humans at all hence building the safest way to enhance the security of a nation. We could see robot Military in the near future, which is as intelligent as a soldier/ commando and will be able to perform some tasks.

AI-assisted strategies would enhance mission effectiveness and will provide the safest way to execute it. The concerning part with AI-assisted system is that how it performs algorithm is not quite explainable. The deep neural networks learn faster and continuously keep learning the main problem here would be explainable AI. It could possess devastating results when it reaches in the wrong hands or makes wrong decisions on its own.

Intelligent Agents:

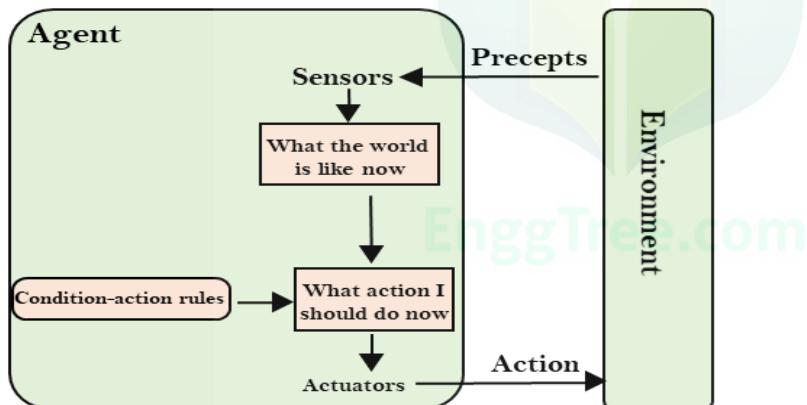
Types of AI Agents or (Structure of Agents):

Agents can be grouped into five classes based on their degree of perceived intelligence and capability. All these agents can improve their performance and generate better action over the time. These are given below:

- Simple Reflex Agent
- Model-based reflex agent
- Goal-based agents
- Utility-based agent
- Learning agent

1. Simple Reflex agent:

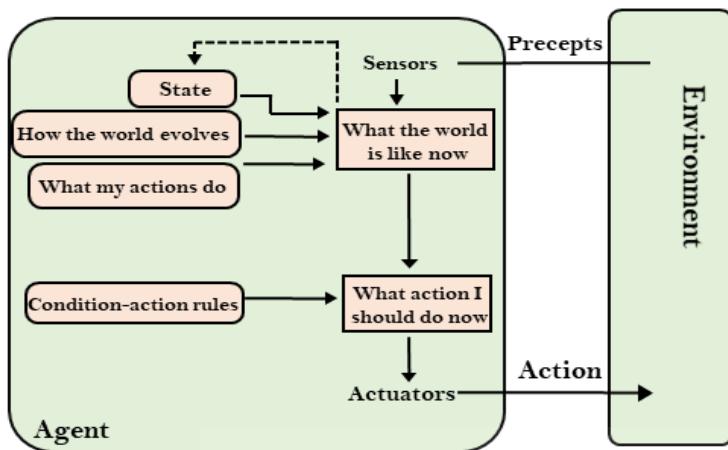
- The Simple reflex agents are the simplest agents. These agents take decisions on the basis of the current percepts and ignore the rest of the percept history.
- These agents only succeed in the fully observable environment.
- The Simple reflex agent does not consider any part of percepts history during their decision and action process.
- The Simple reflex agent works on Condition-action rule, which means it maps the current state to action. Such as a Room Cleaner agent, it works only if there is dirt in the room.
- Problems for the simple reflex agent design approach:
 - They have very limited intelligence
 - They do not have knowledge of non-perceptual parts of the current state
 - Mostly too big to generate and to store.
 - Not adaptive to changes in the environment.



2. Model-based reflex agent

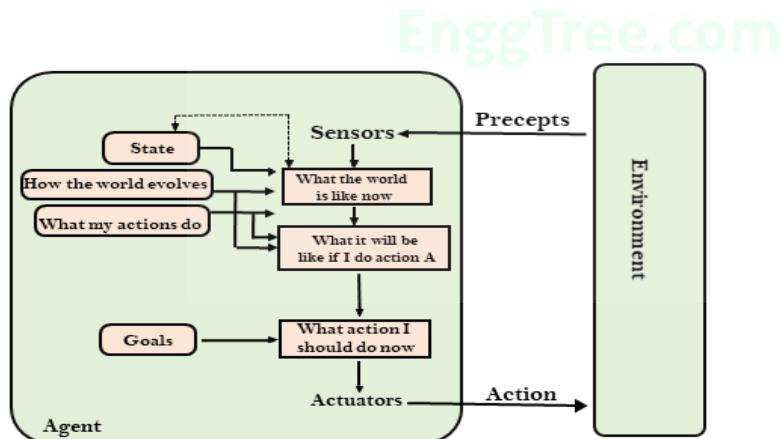
- The Model-based agent can work in a partially observable environment, and track the situation.
- A model-based agent has two important factors:
 - **Model:** It is knowledge about "how things happen in the world," so it is called a Model-based agent.
 - **Internal State:** It is a representation of the current state based on percept history.
- These agents have the model, "which is knowledge of the world" and based on the model they perform actions.
- Updating the agent state requires information about:
 - a. How the world evolves

- b. How the agent's action affects the world.



3. Goal-based agents

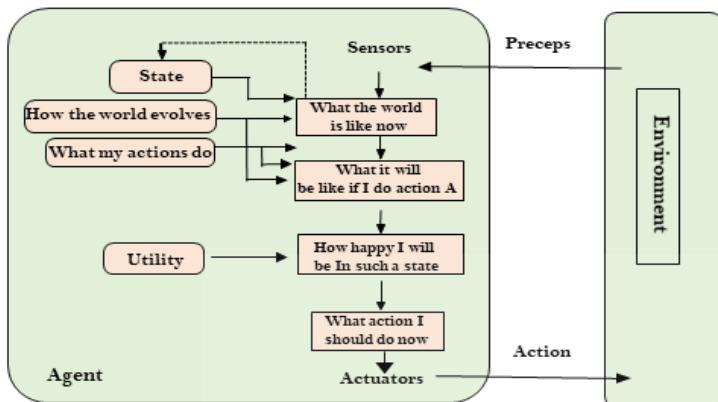
- The knowledge of the current state environment is not always sufficient to decide for an agent to what to do.
- The agent needs to know its goal which describes desirable situations.
- Goal-based agents expand the capabilities of the model-based agent by having the "goal" information.
- They choose an action, so that they can achieve the goal.
- These agents may have to consider a long sequence of possible actions before deciding whether the goal is achieved or not. Such considerations of different scenario are called searching and planning, which makes an agent proactive.



4. Utility-based agents

- These agents are similar to the goal-based agent but provide an extra component of utility measurement which makes them different by providing a measure of success at a given state.

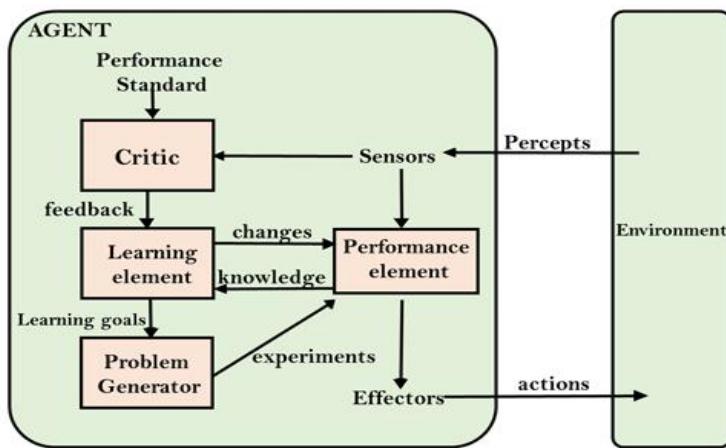
- Utility-based agent act based not only goals but also the best way to achieve the goal.
- The Utility-based agent is useful when there are multiple possible alternatives, and an agent has to choose in order to perform the best action.
- The utility function maps each state to a real number to check how efficiently each action achieves the goals.



5. Learning Agents

- A learning agent in AI is the type of agent which can learn from its past experiences, or it has learning capabilities.
- It starts to act with basic knowledge and then able to act and adapt automatically through learning.
- A learning agent has mainly four conceptual components, which are:
 - a. **Learning element:** It is responsible for making improvements by learning from environment
 - b. **Critic:** Learning element takes feedback from critic which describes that how well the agent is doing with respect to a fixed performance standard.
 - c. **Performance element:** It is responsible for selecting external action
 - d. **Problem generator:** This component is responsible for suggesting actions that will lead to new and informative experiences.

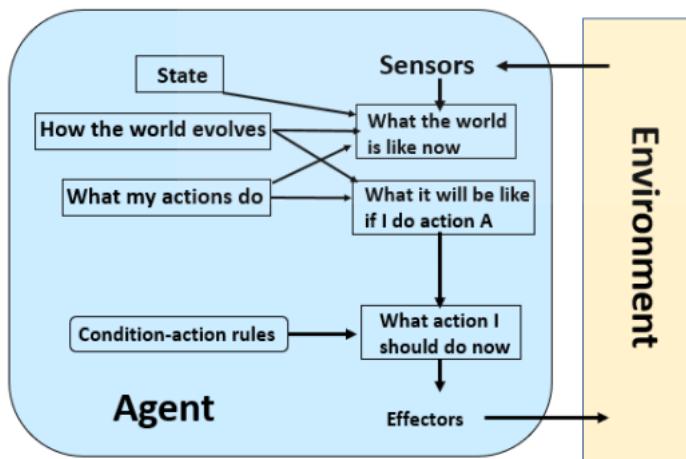
Hence, learning agents are able to learn, analyze performance, and look for new ways to improve the performance.



Nature of Environments:

The environment is the **Task Environment (problem)** for which the **Rational Agent is the solution**. Any task environment is characterised on the basis of PEAS.

- Performance** – What is the performance characteristic which would either make the agent successful or not. For example, as per the previous example clean floor, optimal energy consumption might be performance measures.
- Environment** – Physical characteristics and constraints expected. For example, wood floors, furniture in the way etc
- Actuators** – The physical or logical constructs which would take action. For example for the vacuum cleaner, these are the suction pumps
- Sensors** – Again physical or logical constructs which would sense the environment.



Rational Agents could be physical agents like the one described above or it could also be a program that operates in a non-physical environment like an operating system. Imagine a bot

web site operator designed to scan Internet news sources and show the interesting items to its users, while selling advertising space to generate revenue.

As another example, consider an online tutoring system

Agent	Performance	Environment	Actuator	Sensor
Math E learning system	SLA defined score on the test	Student, Teacher, parents	Computer display system for exercises, corrections, feedback	Keyboard, Mouse

Environments can further be classified into various buckets. This would help determine the intelligence which would need to be built in the agent. These are

- **Observable** – Full or Partial? If the agents sensors get full access then they do not need to pre-store any information. Partial may be due to inaccuracy of sensors or incomplete information about an environment, like limited access to enemy territory
- **Number of Agents** – For the vacuum cleaner, it works in a single agent environment but for driver-less taxis, every driver-less taxi is a separate agent and hence multi agent environment
- **Deterministic** – The number of unknowns in the environment which affect the predictability of the environment. For example, floor space for cleaning is mostly deterministic, the furniture is where it is most of the time but taxi driving on a road is non-deterministic.
- **Discrete** – Does the agent respond when needed or does it have to continuously scan the environment. Driver-less is continuous, online tutor is discrete
- **Static** – How often does the environment change. Can the agent learn about the environment and always do the same thing?
- **Episodic** – If the response to a certain precept is not dependent on the previous one i.e. it is stateless (static methods in Java) then it is discrete. If the decision taken now influences the future decisions then it is a sequential environment.

Agents in Artificial Intelligence

An AI system can be defined as the study of the rational agent and its environment. The agents sense the environment through sensors and act on their environment through actuators. An AI agent can have mental properties such as knowledge, belief, intention, etc.

What is an Agent?

An agent can be anything that perceives its environment through sensors and act upon that environment through actuators. An Agent runs in the cycle of **perceiving, thinking, and acting**. An agent can be:

- **Human-Agent:** A human agent has eyes, ears, and other organs which work for sensors and hand, legs, vocal tract work for actuators.
- **Robotic Agent:** A robotic agent can have cameras, infrared range finder, NLP for sensors and various motors for actuators.

- **Software Agent:** Software agent can have keystrokes, file contents as sensory input and act on those inputs and display output on the screen.

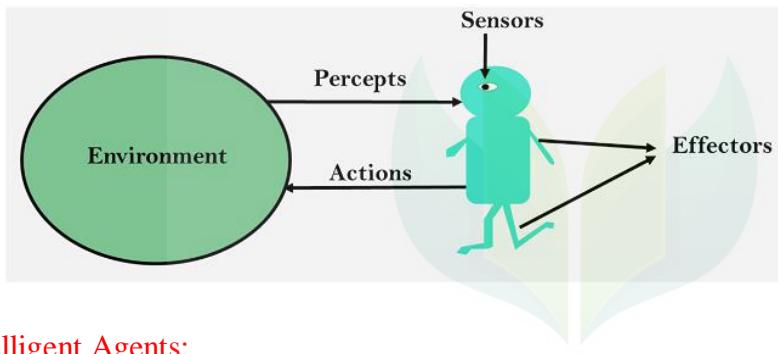
Hence the world around us is full of agents such as thermostat, cellphone, camera, and even we are also agents.

Before moving forward, we should first know about sensors, effectors, and actuators.

Sensor: Sensor is a device which detects the change in the environment and sends the information to other electronic devices. An agent observes its environment through sensors.

Actuators: Actuators are the component of machines that converts energy into motion. The actuators are only responsible for moving and controlling a system. An actuator can be an electric motor, gears, rails, etc.

Effectors: Effectors are the devices which affect the environment. Effectors can be legs, wheels, arms, fingers, wings, fins, and display screen.



Intelligent Agents:

An intelligent agent is an autonomous entity which act upon an environment using sensors and actuators for achieving goals. An intelligent agent may learn from the environment to achieve their goals. A thermostat is an example of an intelligent agent.

Following are the main four rules for an AI agent:

- **Rule 1:** An AI agent must have the ability to perceive the environment.
- **Rule 2:** The observation must be used to make decisions.
- **Rule 3:** Decision should result in an action.
- **Rule 4:** The action taken by an AI agent must be a rational action.

Rational Agent:

A rational agent is an agent which has clear preference, models uncertainty, and acts in a way to maximize its performance measure with all possible actions.

A rational agent is said to perform the right things. AI is about creating rational agents to use for game theory and decision theory for various real-world scenarios.

For an AI agent, the rational action is most important because in AI reinforcement learning algorithm, for each best possible action, agent gets the positive reward and for each wrong action, an agent gets a negative reward.

Note: Rational agents in AI are very similar to intelligent agents.

Rationality:

The rationality of an agent is measured by its performance measure. Rationality can be judged on the basis of following points:

- Performance measure which defines the success criterion.
- Agent prior knowledge of its environment.
- Best possible actions that an agent can perform.
- The sequence of percepts.

Note: Rationality differs from Omniscience because an Omnipotent agent knows the actual outcome of its action and act accordingly, which is not possible in reality.

Structure of an AI Agent

The task of AI is to design an agent program which implements the agent function. The structure of an intelligent agent is a combination of architecture and agent program. It can be viewed as:

$$\text{Agent} = \text{Architecture} + \text{Agent program}$$

Following are the main three terms involved in the structure of an AI agent:

Architecture: Architecture is machinery that an AI agent executes on.

Agent Function: Agent function is used to map a percept to an action

$$f:P^* \rightarrow A$$

Agent program: Agent program is an implementation of agent function. An agent program executes on the physical architecture to produce function f.

PEAS Representation

PEAS is a type of model on which an AI agent works upon. When we define an AI agent or rational agent, then we can group its properties under PEAS representation model. It is made up of four words:

- **P:** Performance measure
- **E:** Environment
- **A:** Actuators

- **S:** Sensors

Here performance measure is the objective for the success of an agent's behavior.

PEAS for self-driving cars:

Let's suppose a self-driving car then PEAS representation will be:

Performance: Safety, time, legal drive, comfort

Environment: Roads, other vehicles, road signs, pedestrian

Actuators: Steering, accelerator, brake, signal, horn

Sensors: Camera, GPS, speedometer, odometer, accelerometer, sonar.

Example of Agents with their PEAS representation

Agent	Performance measure	Environment	Actuators	Sensors
1. Medical Diagnose	<input type="radio"/> Healthy patient <input type="radio"/> Minimized cost	<input type="radio"/> Patient <input type="radio"/> Hospital <input type="radio"/> Staff	<input type="radio"/> Tests <input type="radio"/> Treatments	Keyboard (Entry of symptoms)
2. Vacuum Cleaner	<input type="radio"/> Cleanliness <input type="radio"/> Efficiency <input type="radio"/> Battery life <input type="radio"/> Security	<input type="radio"/> Room <input type="radio"/> Table <input type="radio"/> Wood floor <input type="radio"/> Carpet <input type="radio"/> Various obstacles	<input type="radio"/> Wheels <input type="radio"/> Brushes <input type="radio"/> Vacuum Extractor	<input type="radio"/> Camera <input type="radio"/> Dirt detection sensor <input type="radio"/> Cliff sensor <input type="radio"/> Bump Sensor <input type="radio"/> Infrared Wall Sensor
3. Part - picking Robot	<input type="radio"/> Percentage of parts in correct bins.	<input type="radio"/> Conveyor belt with parts, <input type="radio"/> Bins	<input type="radio"/> Jointed Arms <input type="radio"/> Hand	<input type="radio"/> Camera <input type="radio"/> Joint angle sensors.

Agent Environment in AI:

An environment is everything in the world which surrounds the agent, but it is not a part of an agent itself. An environment can be described as a situation in which an agent is present.

The environment is where agent lives, operate and provide the agent with something to sense and act upon it. An environment is mostly said to be non-feministic.

Features of Environment

Environment can have various features from the point of view of an agent:

1. Fully observable vs Partially Observable
2. Static vs Dynamic
3. Discrete vs Continuous
4. Deterministic vs Stochastic
5. Single-agent vs Multi-agent
6. Episodic vs sequential
7. Known vs Unknown
8. Accessible vs Inaccessible

1. Fully observable vs Partially Observable:

- If an agent sensor can sense or access the complete state of an environment at each point of time then it is a **fully observable** environment, else it is **partially observable**.
- A fully observable environment is easy as there is no need to maintain the internal state to keep track history of the world.
- An agent with no sensors in all environments then such an environment is called as **unobservable**.

2. Deterministic vs Stochastic:

- If an agent's current state and selected action can completely determine the next state of the environment, then such environment is called a deterministic environment.
- A stochastic environment is random in nature and cannot be determined completely by an agent.
- In a deterministic, fully observable environment, agent does not need to worry about uncertainty.

3. Episodic vs Sequential:

- In an episodic environment, there is a series of one-shot actions, and only the current percept is required for the action.
- However, in Sequential environment, an agent requires memory of past actions to determine the next best actions.

4. Single-agent vs Multi-agent

- If only one agent is involved in an environment, and operating by itself then such an environment is called single agent environment.
- However, if multiple agents are operating in an environment, then such an environment is called a multi-agent environment.
- The agent design problems in the multi-agent environment are different from single agent environment.

5. Static vs Dynamic:

- If the environment can change itself while an agent is deliberating then such environment is called a dynamic environment else it is called a static environment.
- Static environments are easy to deal because an agent does not need to continue looking at the world while deciding for an action.
- However for dynamic environment, agents need to keep looking at the world at each action.
- Taxi driving is an example of a dynamic environment whereas Crossword puzzles are an example of a static environment.

6. Discrete vs Continuous:

- If in an environment there are a finite number of percepts and actions that can be performed within it, then such an environment is called a discrete environment else it is called continuous environment.
- A chess game comes under discrete environment as there is a finite number of moves that can be performed.
- A self-driving car is an example of a continuous environment.

7. Known vs Unknown

- Known and unknown are not actually a feature of an environment, but it is an agent's state of knowledge to perform an action.
- In a known environment, the results for all actions are known to the agent. While in unknown environment, agent needs to learn how it works in order to perform an action.
- It is quite possible that a known environment to be partially observable and an Unknown environment to be fully observable.

8. Accessible vs Inaccessible

- If an agent can obtain complete and accurate information about the state's environment, then such an environment is called an Accessible environment else it is called inaccessible.
- An empty room whose state can be defined by its temperature is an example of an accessible environment.
- Information about an event on earth is an example of Inaccessible environment.

Search Algorithms in Artificial Intelligence:

Search algorithms are one of the most important areas of Artificial Intelligence.

Problem-solving agents:

In Artificial Intelligence, Search techniques are universal problem-solving methods. **Rational agents** or **Problem-solving agents** in AI mostly used these search strategies or algorithms to

solve a specific problem and provide the best result. Problem-solving agents are the goal-based agents and use atomic representation. In this topic, we will learn various problem-solving search algorithms.

Search Algorithm Terminologies:

- **Search:** Searching is a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:
 - a. **Search Space:** Search space represents a set of possible solutions, which a system may have.
 - b. **Start State:** It is a state from where agent begins **the search**.
 - c. **Goal test:** It is a function which observe the current state and returns whether the goal state is achieved or not.

Search tree: A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.

Actions: It gives the description of all the available actions to the agent.

Transition model: A description of what each action do, can be represented as a transition model.

Path Cost: It is a function which assigns a numeric cost to each path.

Solution: It is an action sequence which leads from the start node to the goal node.

Optimal Solution: If a solution has the lowest cost among all solutions.

Properties of Search Algorithms:

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.

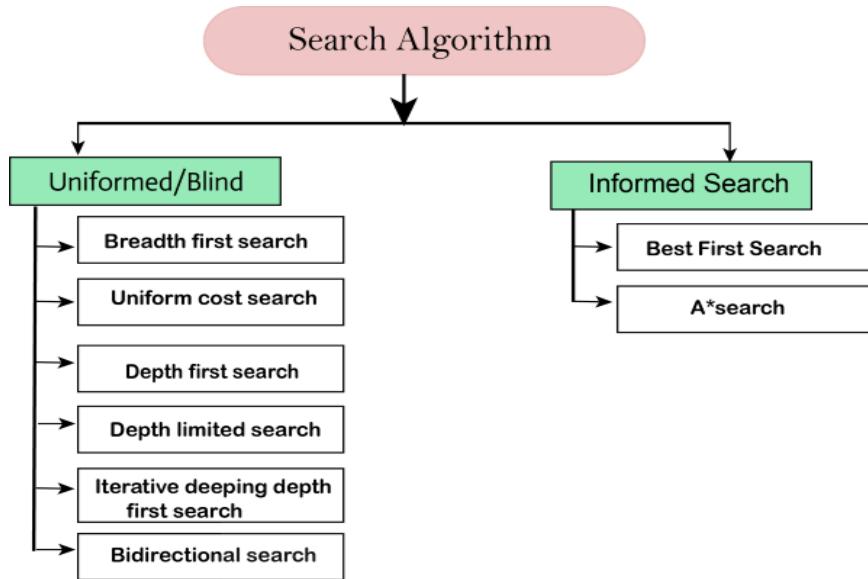
Optimality: If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.

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

Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.



Uninformed Search Algorithms:

Uninformed search is a class of general-purpose search algorithms which operates in brute force-way. Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called blind search.

Following are the various types of uninformed search algorithms:

1. **Breadth-first Search**
2. **Depth-first Search**
3. **Depth-limited Search**
4. **Iterative deepening depth-first search**
5. **Uniform cost search**
6. **Bidirectional Search**

1. 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.

Advantages:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

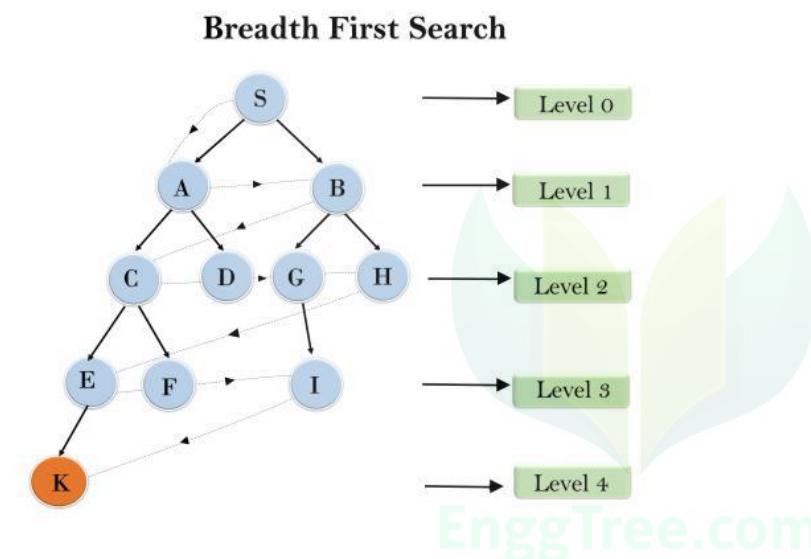
Disadvantages:

- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

Example:

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

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



Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d = depth of shallowest solution and b is a node at every state.

$$T(b) = 1+b^2+b^3+\dots+b^d= O(b^d)$$

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.

Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

2. Depth-first Search

- Depth-first search is a recursive algorithm for traversing a tree or graph data structure.

- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

Note: Backtracking is an algorithm technique for finding all possible solutions using recursion.

Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage:

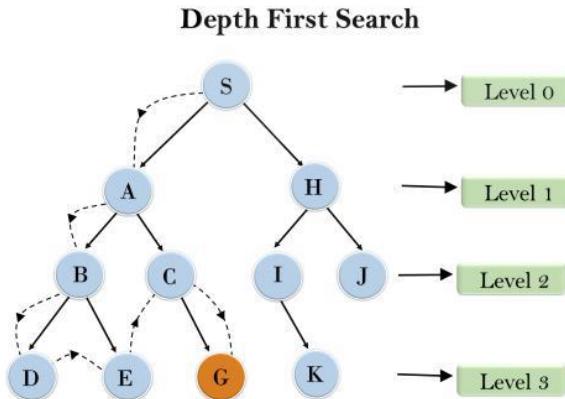
- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Example:

In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:

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

It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.



Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

$$T(n) = 1 + n^2 + n^3 + \dots + n^m = O(n^m)$$

Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)

Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.

Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

3. Depth-Limited Search Algorithm:

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

Depth-limited search can be terminated with two Conditions of failure:

- Standard failure value: It indicates that problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit.

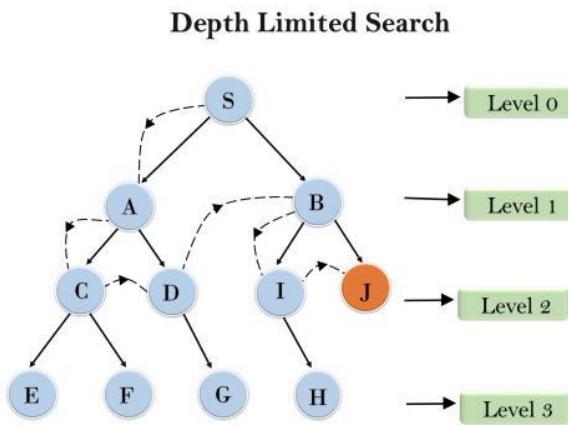
Advantages:

Depth-limited search is Memory efficient.

Disadvantages:

- Depth-limited search also has a disadvantage of incompleteness.

Example:



Completeness: DLS search algorithm is complete if the solution is above the depth-limit.

Time Complexity: Time complexity of DLS algorithm is $O(b^l)$.

Space Complexity: Space complexity of DLS algorithm is $O(b \times l)$.

Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if $l > d$.

4. Uniform-cost Search Algorithm:

Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs from the root node. It can be used to solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

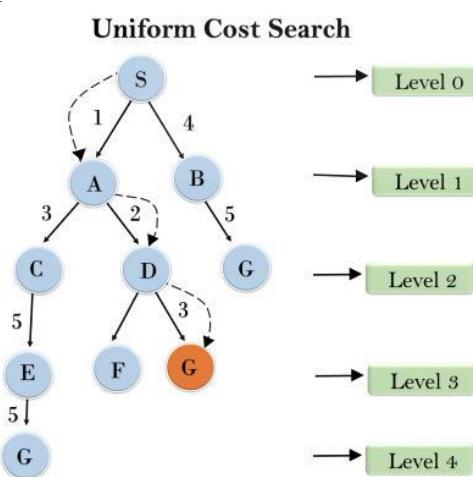
Advantages:

- Uniform cost search is optimal because at every state the path with the least cost is chosen.

Disadvantages:

- It does not care about the number of steps involved in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Example:



Completeness:

Uniform-cost search is complete, such as if there is a solution, UCS will find it.

Time Complexity:

Let C^* is Cost of the optimal solution, and ϵ is each step to get closer to the goal node. Then the number of steps is $= C^*/\epsilon + 1$. Here we have taken $+1$, as we start from state 0 and end to C^*/ϵ .

Hence, the worst-case time complexity of Uniform-cost search is $O(b^{1 + \lceil C^*/\epsilon \rceil})$.

Space Complexity:

The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $O(b^{1 + \lceil C^*/\epsilon \rceil})$.

Optimal:

Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

5. Iterative deepening depth-first Search:

The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.

This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.

This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.

The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Advantages:

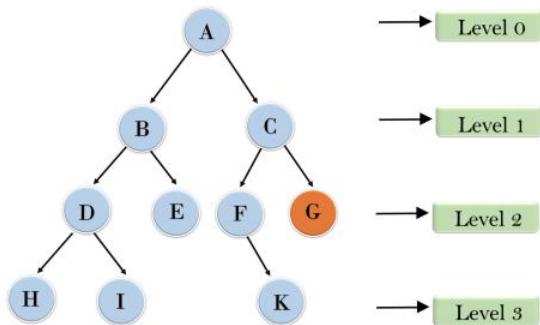
- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

Disadvantages:

- The main drawback of IDDFS is that it repeats all the work of the previous phase.

Example:

Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:

Iterative deepening depth first search

1'st Iteration----> A

2'nd Iteration----> A, B, C

3'rd Iteration----->A, B, D, E, C, F, G

4'th Iteration----->A, B, D, H, I, E, C, F, K, G

In the fourth iteration, the algorithm will find the goal node.

Completeness:

This algorithm is complete is ifthe branching factor is finite.

Time Complexity:

Let's suppose b is the branching factor and depth is d then the worst-case time complexity is $O(b^d)$.

EnggTree.com

Space Complexity:

The space complexity of IDDFS will be $O(bd)$.

Optimal:

IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

6. Bidirectional Search Algorithm:

Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node. Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.

Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Advantages:

- Bidirectional search is fast.
- Bidirectional search requires less memory

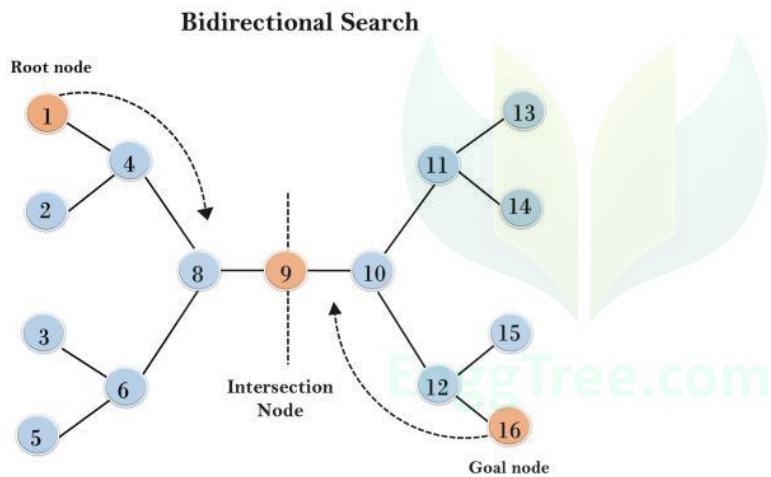
Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- **In bidirectional search, one should know the goal state in advance.**

Example:

In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.

The algorithm terminates at node 9 where two searches meet.



Completeness: Bidirectional Search is complete if we use BFS in both searches.

Time Complexity: Time complexity of bidirectional search using BFS is $O(b^d)$.

Space Complexity: Space complexity of bidirectional search is $O(b^d)$.

Optimal: Bidirectional search is Optimal.

Informed Search Algorithms:

So far we have talked about the uninformed search algorithms which looked through search space for all possible solutions of the problem without having any additional knowledge about search space. But informed search algorithm contains an array of knowledge such as how far we

are from the goal, path cost, how to reach to goal node, etc. This knowledge help agents to explore less to the search space and find more efficiently the goal node.

The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Heuristics function: Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by $h(n)$, and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Admissibility of the heuristic function is given as:

$$h(n) \leq h^*(n)$$

Pure Heuristic Search:

Pure heuristic search is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value $h(n)$. It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.

On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues until a goal state is found.

In the informed search we will discuss two main algorithms which are given below:

- **Best First Search Algorithm(Greedy search)**
- **A* Search Algorithm**

1.) Best-first Search Algorithm (Greedy Search):

Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

$$f(n) = g(n).$$

Where, $h(n)$ = estimated cost from node n to the goal.

The greedy best first algorithm is implemented by the priority queue.

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.

Advantages:

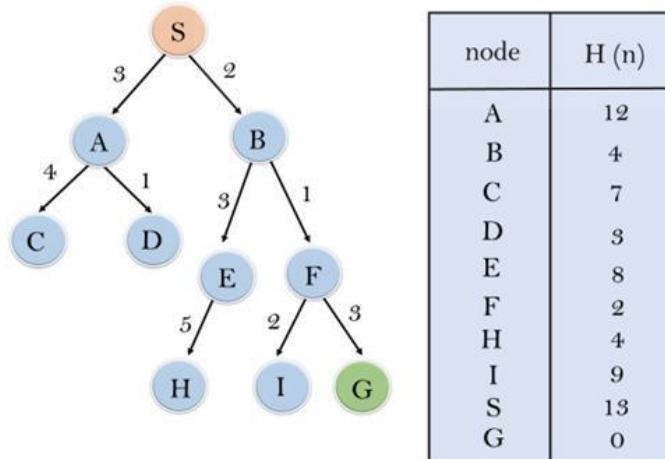
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

Disadvantages:

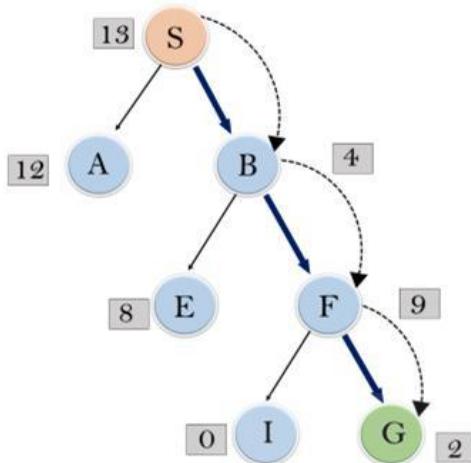
- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.

Example:

Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function $f(n)=h(n)$, which is given in the below table.



In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists. Following are the iteration for traversing the above example.



Expand the nodes of S and put in the CLOSED list

Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration 2: Open [E, F, A], Closed [S, B]
: Open [E, A], Closed [S, B, F]

Iteration 3: Open [I, G, E, A], Closed [S, B, F]
: Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: **S----> B---->F----> G**

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, m is the maximum depth of the search space.

Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

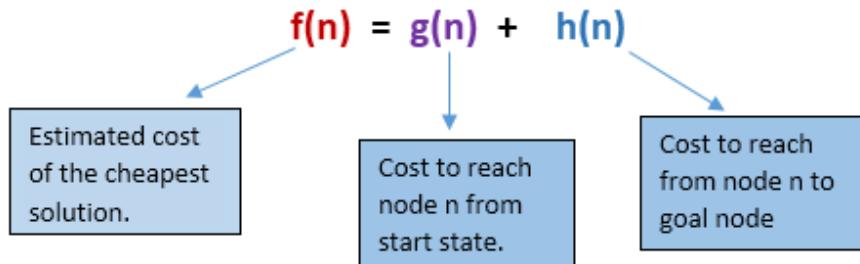
Optimal: Greedy best first search algorithm is not optimal.

2.) A* Search Algorithm:

A* search is the most commonly known form of best-first search. It uses heuristic function $h(n)$, and cost to reach the node n from the start state $g(n)$. It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm

expands less search tree and provides optimal result faster. A* algorithm is similar to UCS except that it uses $g(n)+h(n)$ instead of $g(n)$.

In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a **fitness number**.



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:

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

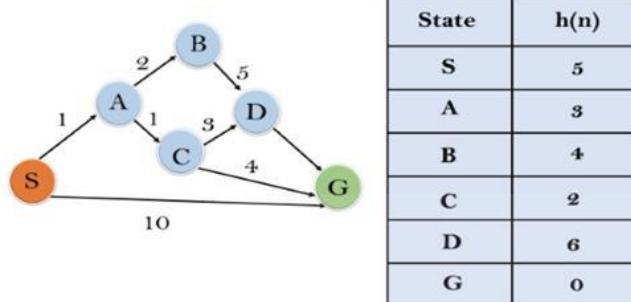
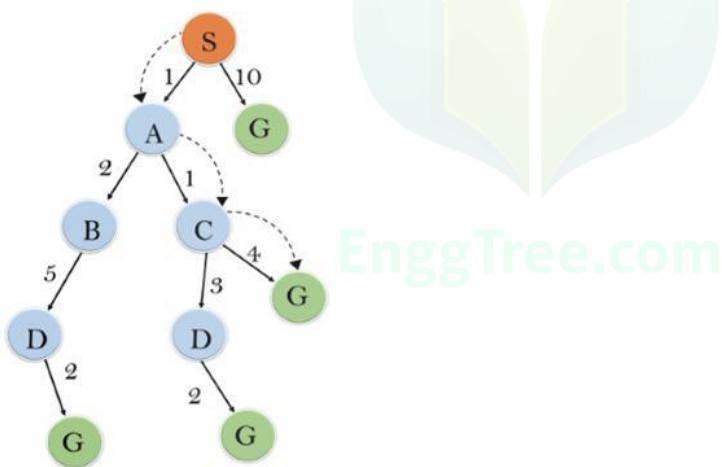
Disadvantages:

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

Example:

In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the $f(n)$ of each state using the formula $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach any node from start state.

Here we will use OPEN and CLOSED list.

**Solution:**

Initialization: $\{(S, 5)\}$

Iteration1: $\{(S \rightarrow A, 4), (S \rightarrow G, 10)\}$

Iteration2: $\{(S \rightarrow A \rightarrow C, 4), (S \rightarrow A \rightarrow B, 7), (S \rightarrow G, 10)\}$

Iteration3: $\{(S \rightarrow A \rightarrow C \rightarrow G, 6), (S \rightarrow A \rightarrow C \rightarrow D, 11), (S \rightarrow A \rightarrow B, 7), (S \rightarrow G, 10)\}$

Iteration 4 will give the final result, as **S--->A--->C--->G** it provides the optimal path with cost 6.

Points to remember:

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition $f(n) \leq h(n)$

Complete: A* algorithm is complete as long as:

- Branching factor is finite.
- Cost at every action is fixed.

Optimal: A* search algorithm is optimal if it follows below two conditions:

- **Admissible:** the first condition requires for optimality is that $h(n)$ should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
- **Consistency:** Second required condition is consistency for only A* graph-search.

If the heuristic function is admissible, then A* tree search will always find the least cost path.

Time Complexity: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d . So the time complexity is $O(b^d)$, where b is the branching factor.

Space Complexity: The space complexity of A* search algorithm is $O(b^d)$

1. Heuristic Search Strategies:**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.

optimal, but those solutions are sufficient in a given limited timeframe.

Why do we need heuristics?

Heuristics are used in situations in which there is the requirement of a short-term solution. On facing complex situations with limited resources and time, Heuristics can help the companies to make quick decisions by shortcuts and approximated calculations. Most of the heuristic methods involve mental shortcuts to make decisions on past experiences.

The heuristic method might not always provide us the finest solution, but it is assured that it helps us find a good solution in a reasonable time.

Based on context, there can be different heuristic methods that correlate with the problem's scope. The most common heuristic methods are - trial and error, guesswork, the process of elimination, historical data analysis. These methods involve simply available information that is not particular to the problem but is most appropriate. They can include representative, affect, and availability heuristics.

Heuristic search techniques in AI (Artificial Intelligence)

We can perform the Heuristic techniques into two categories:

Direct Heuristic Search techniques in AI

It includes Blind Search, Uninformed Search, and Blind control strategy. These search techniques are not always possible as they require much memory and time. These techniques search the complete space for a solution and use the arbitrary ordering of operations.

The examples of Direct Heuristic search techniques include Breadth-First Search (BFS) and Depth First Search (DFS).

Weak Heuristic Search techniques in AI

It includes Informed Search, Heuristic Search, and Heuristic control strategy. These techniques are helpful when they are applied properly to the right types of tasks. They usually require domain-specific information.

The examples of Weak Heuristic search techniques include Best First Search (BFS) and A*.

Before describing certain heuristic techniques, let's see some of the techniques listed below:

- Bidirectional Search
- A* search
- Simulated Annealing
- Hill Climbing
- Best First search
- Beam search



First, let's talk about the Hill climbing in Artificial intelligence.

Hill Climbing Algorithm

It is a technique for optimizing the mathematical problems. Hill Climbing is widely used when a good heuristic is available.

It is a local search algorithm that continuously moves in the direction of increasing elevation/value to find the mountain's peak or the best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value. Traveling-salesman Problem is one of the widely discussed examples of the Hill climbing algorithm, in which we need to minimize the distance traveled by the salesman.

It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that. The steps of a simple hill-climbing algorithm are listed below:

Step 1: Evaluate the initial state. If it is the goal state, then return success and Stop.

Step 2: Loop Until a solution is found or there is no new operator left to apply.

Step 3: Select and apply an operator to the current state.

Step 4: Check new state:

If it is a goal state, then return to success and quit.

Else if it is better than the current state, then assign a new state as a current state.

Else if not better than the current state, then return to step2.

Step 5: Exit.

Best first search (BFS)

This algorithm always chooses the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It lets us to take the benefit of both algorithms. It uses the heuristic function and search. With the help of the best-first search, at each step, we can choose the most promising node.

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 the goal node, then return success and stop the search, else continue to next step.

Step 6: For each successor node, the 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 lists, then add it to the OPEN list.

Step 7: Return to Step 2.

A* Search Algorithm

A* search is the most commonly known form of best-first search. It uses the heuristic function $h(n)$ and cost to reach the node n from the start state $g(n)$. It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently.

It finds the shortest path through the search space using the heuristic function. This search algorithm expands fewer search tree and gives optimal results faster.

Algorithm of A* search:

Step 1: 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.

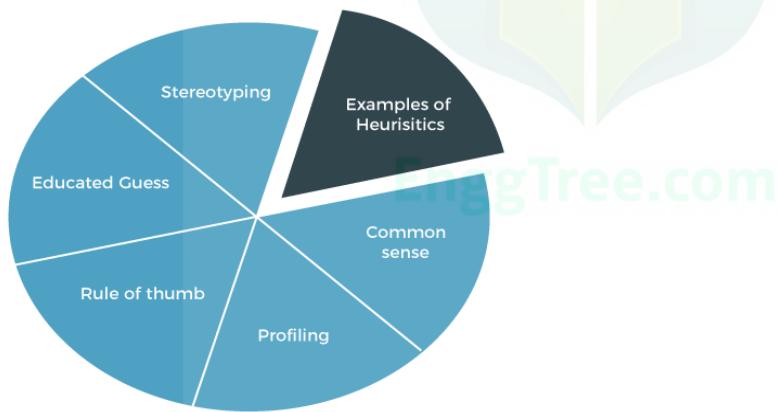
Step 3: Select the node from the OPEN list which has the smallest value of the evaluation function ($g+h$). If node n is the goal node, then return success and stop, otherwise.

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 the evaluation function for n' and place it into the 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.

Examples of heuristics in everyday life



Some of the real-life examples of heuristics that people use as a way to solve a problem:

- **Common sense:** It is a heuristic that is used to solve a problem based on the observation of an individual.
- **Rule of thumb:** In heuristics, we also use a term rule of thumb. This heuristic allows an individual to make an approximation without doing an exhaustive search.
- **Working backward:** It lets an individual solve a problem by assuming that the problem is already being solved by them and working backward in their minds to see how much a solution has been reached.
- **Availability heuristic:** It allows a person to judge a situation based on the examples of similar situations that come to mind.

- **Familiarity heuristic:** It allows a person to approach a problem on the fact that an individual is familiar with the same situation, so one should act similarly as he/she acted in the same situation before.
- **Educated guess:** It allows a person to reach a conclusion without doing an exhaustive search. Using it, a person considers what they have observed in the past and applies that history to the situation where there is not any definite answer has decided yet.

Types of heuristics

There are various types of heuristics, including the availability heuristic, affect heuristic and representative heuristic. Each heuristic type plays a role in decision-making. Let's discuss about the Availability heuristic, affect heuristic, and Representative heuristic.

Availability heuristic

Availability heuristic is said to be the judgment that people make regarding the likelihood of an event based on information that quickly comes into mind. On making decisions, people typically rely on the past knowledge or experience of an event. It allows a person to judge a situation based on the examples of similar situations that come to mind.

Representative heuristic

It occurs when we evaluate an event's probability on the basis of its similarity with another event.

Example: We can understand the representative heuristic by the example of product packaging, as consumers tend to associate the products quality with the external packaging of a product. If a company packages its products that remind you of a high quality and well-known product, then consumers will relate that product as having the same quality as the branded product.

So, instead of evaluating the product based on its quality, customers correlate the products quality based on the similarity in packaging.

Affect heuristic

It is based on the negative and positive feelings that are linked with a certain stimulus. It includes quick feelings that are based on past beliefs. Its theory is one's emotional response to a stimulus that can affect the decisions taken by an individual.

When people take a little time to evaluate a situation carefully, they might base their decisions based on their emotional response.

Example: The affect heuristic can be understood by the example of advertisements. Advertisements can influence the emotions of consumers, so it affects the purchasing decision of a consumer. The most common examples of advertisements are the ads of fast food. When fast-food companies run the advertisement, they hope to obtain a positive emotional response that pushes you to positively view their products.

If someone carefully analyzes the benefits and risks of consuming fast food, they might decide that fast food is unhealthy. But people rarely take time to evaluate everything they see and generally make decisions based on their automatic emotional response. So, Fast food companies present advertisements that rely on such type of Affect heuristic for generating a positive emotional response which results in sales.

Limitation of heuristics

Along with the benefits, heuristic also has some limitations.

- Although heuristics speed up our decision-making process and also help us to solve problems, they can also introduce errors just because something has worked accurately in the past, so it does not mean that it will work again.
- It will hard to find alternative solutions or ideas if we always rely on the existing solutions or heuristics.

2. Heuristic Functions in Artificial Intelligence:

Heuristic Functions in AI: As we have already seen that an informed search make use of heuristic functions in order to reach the goal node in a more prominent way. Therefore, there are several pathways in a search tree to reach the goal node from the current node. The selection of a good heuristic function matters certainly. *A good heuristic function is determined by its efficiency. More is the information about the problem, more is the processing time.*

Some toy problems, such as 8-puzzle, 8-queen, tic-tac-toe, etc., can be solved more efficiently with the help of a heuristic function. Let's see how

Consider the following 8-puzzle problem where we have a start state and a goal state. Our task is to slide the tiles of the current/start state and place it in an order followed in the goal state. There can be four moves either **left, right, up, or down**. There can be several ways to convert the current/start state to the goal state, but, we can use a heuristic function $h(n)$ to solve the problem more efficiently.

1	2	3
8	6	
7	5	4

Start State

1	2	3
8		4
7	6	5

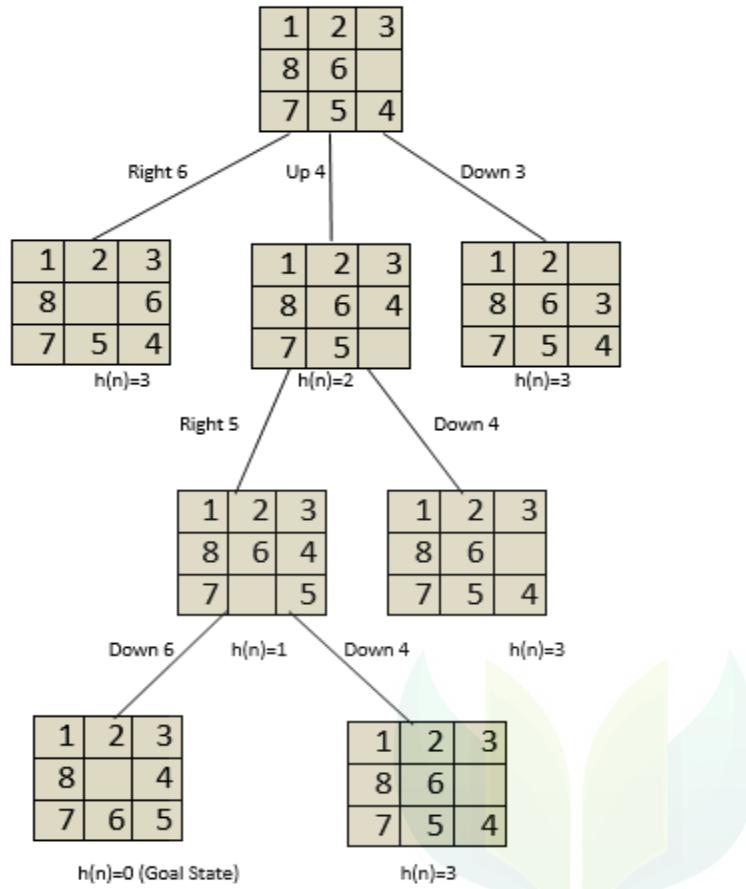
Goal State

A heuristic function for the 8-puzzle problem is defined below:

$h(n)$ =Number of tiles out of position.

So, there is total of three tiles out of position i.e., 6,5 and 4. Do not count the empty tile present in the goal state). i.e. $h(n)=3$. Now, we require to minimize the value of **$h(n) =0$** .

We can construct a state-space tree to minimize the $h(n)$ value to 0, as shown below:



It is seen from the above state space tree that the goal state is minimized from $h(n)=3$ to $h(n)=0$. However, we can create and use several heuristic functions as per the requirement. It is also clear from the above example that a heuristic function $h(n)$ can be defined as the information required to solve a given problem more efficiently. The information can be related to the **nature of the state, cost of transforming from one state to another, goal node characteristics, etc.**, which is expressed as a heuristic function.

3. Local Search Algorithms and Optimization Problem:

The **informed** and **uninformed search** expands the nodes systematically in two ways:

- keeping different paths in the memory and
- selecting the best suitable path,

Which leads to a solution state required to reach the goal node. But beyond these "**classical search algorithms**," we have some "**local search algorithms**" where the path cost does not matter, and only focus on solution-state needed to reach the goal node.

A local search algorithm completes its task by traversing on a single current node rather than multiple paths and following the neighbors of that node generally.

Although local search algorithms are not systematic, still they have the following two advantages:

- Local search algorithms use a very little or constant amount of memory as they operate only on a single path.
- Most often, they find a reasonable solution in large or infinite state spaces where the classical or systematic algorithms do not work.

Does the local search algorithm work for a pure optimized problem?

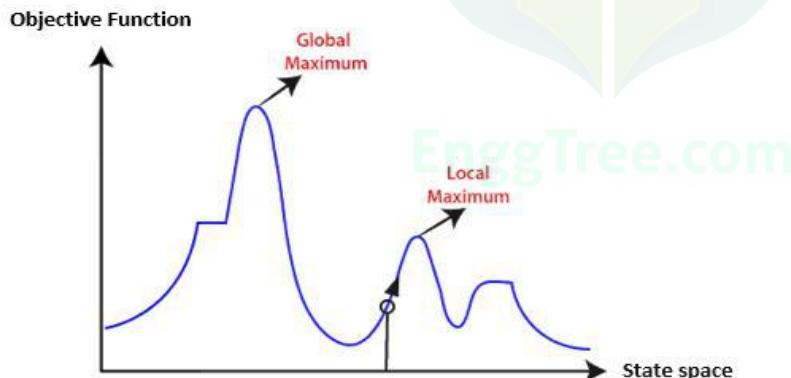
Yes, the local search algorithm works for pure optimized problems. A pure optimization problem is one where all the nodes can give a solution. But the target is to find the best state out of all according to the **objective function**. Unfortunately, the pure optimization problem fails to find high-quality solutions to reach the goal state from the current state.

Note: An objective function is a function whose value is either minimized or maximized in different contexts of the optimization problems. In the case of search algorithms, an objective function can be the path cost for reaching the goal node, etc.

Working of a Local search algorithm

Let's understand the working of a local search algorithm with the help of an example:
Consider the below state-space landscape having both:

- **Location:** It is defined by the state.
- **Elevation:** It is defined by the value of the objective function or heuristic cost function.



A one-dimensional state-space landscape in which elevation corresponds to the objective function

The local search algorithm explores the above landscape by finding the following two points:

- **Global Minimum:** If the elevation corresponds to the cost, then the task is to find the lowest valley, which is known as **Global Minimum**.
- **Global Maxima:** If the elevation corresponds to an objective function, then it finds the highest peak which is called as **Global Maxima**. It is the highest point in the valley.

We will understand the working of these points better in Hill-climbing search.

Below are some different types of local searches:

- Hill-climbing Search
- Simulated Annealing
- Local Beam Search

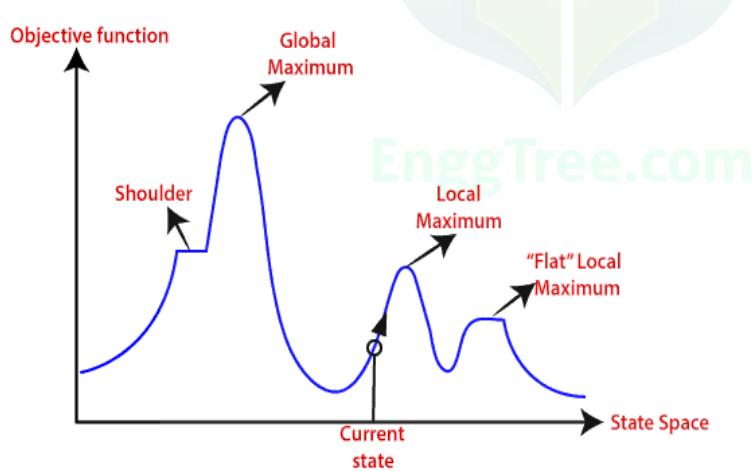
Hill Climbing Algorithm in AI

Hill Climbing Algorithm: Hill climbing search is a local search problem. *The purpose of the hill climbing search is to climb a hill and reach the topmost peak/ point of that hill.* It is based on the **heuristic search technique** where the person who is climbing up on the hill estimates the direction which will lead him to the highest peak.

State-space Landscape of Hill climbing algorithm

To understand the concept of hill climbing algorithm, consider the below landscape representing the **goal state/peak** and the **current state** of the climber. The topographical regions shown in the figure can be defined as:

- **Global Maximum:** It is the highest point on the hill, which is the goal state.
- **Local Maximum:** It is the peak higher than all other peaks but lower than the global maximum.
- **Flat local maximum:** It is the flat area over the hill where it has no uphill or downhill. It is a saturated point of the hill.
- **Shoulder:** It is also a flat area where the summit is possible.
- **Current state:** It is the current position of the person.

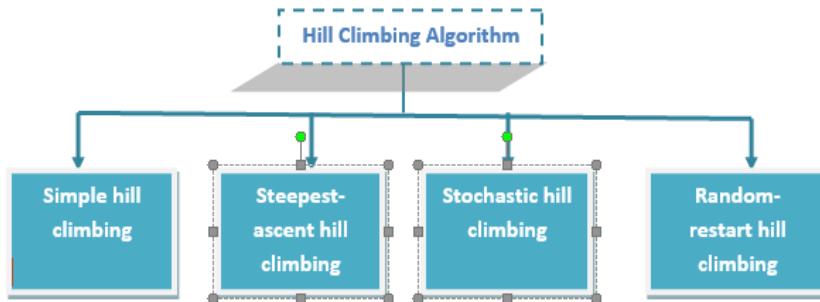


A one-dimensional state-space landscape in which elevation corresponds to the objective function

Types of Hill climbing search algorithm

There are following types of hill-climbing search:

- Simple hill climbing
- Steepest-ascent hill climbing
- Stochastic hill climbing
- Random-restart hill climbing



Simple hill climbing search

Simple hill climbing is the simplest technique to climb a hill. The task is to reach the highest peak of the mountain. Here, the movement of the climber depends on his move/steps. If he finds his next step better than the previous one, he continues to move else remain in the same state. This search focus only on his previous and next step.

Simple hill climbing Algorithm

1. Create a **CURRENT** node, **NEIGHBOUR** node, and a **GOAL** node.
2. If the **CURRENT node=GOAL node**, return **GOAL** and terminate the search.
3. Else **CURRENT node<= NEIGHBOUR node**, move ahead.
4. Loop until the goal is not reached or a point is not found.

Steepest-ascent hill climbing

Steepest-ascent hill climbing is different from simple hill climbing search. Unlike simple hill climbing search, It considers all the successive nodes, compares them, and choose the node which is closest to the solution. Steepest hill climbing search is similar to **best-first search** because it focuses on each node instead of one.

Note: Both simple, as well as steepest-ascent hill climbing search, fails when there is no closer node.

Steepest-ascent hill climbing algorithm

1. Create a **CURRENT** node and a **GOAL** node.
2. If the **CURRENT node=GOAL node**, return **GOAL** and terminate the search.
3. Loop until a better node is not found to reach the solution.
4. If there is any better successor node present, expand it.
5. When the **GOAL** is attained, return **GOAL** and terminate.

Stochastic hill climbing

Stochastic hill climbing does not focus on all the nodes. It selects one node at random and decides whether it should be expanded or search for a better one.

Random-restart hill climbing

Random-restart algorithm is based on **try and try strategy**. It iteratively searches the node and selects the best one at each step until the goal is not found. The success depends most commonly on the shape of the hill. If there are few plateaus, local maxima, and ridges, it becomes easy to reach the destination.

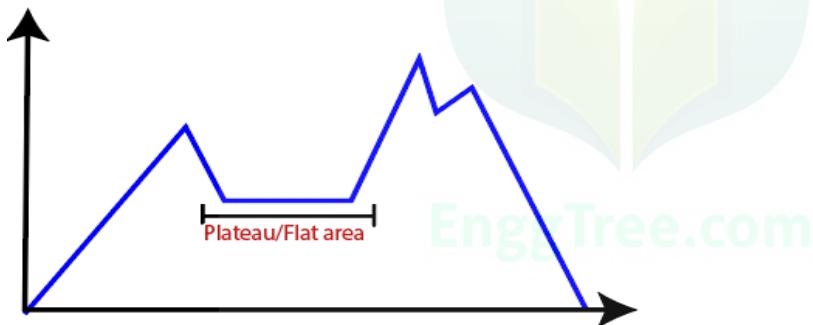
Limitations of Hill climbing algorithm

Hill climbing algorithm is a fast and furious approach. It finds the solution state rapidly because it is quite easy to improve a bad state. But, there are following limitations of this search:

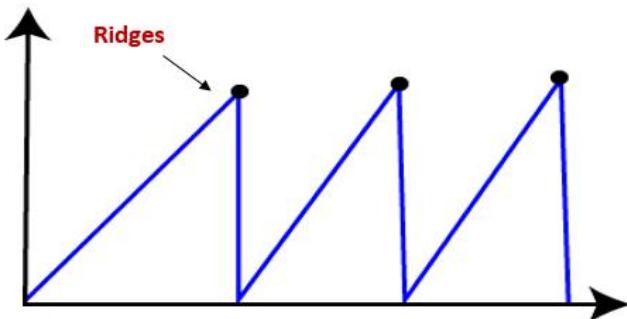
- **Local Maxima:** It is that peak of the mountain which is highest than all its neighboring states but lower than the global maxima. It is not the goal peak because there is another peak higher than it.



- **Plateau:** It is a flat surface area where no uphill exists. It becomes difficult for the climber to decide that in which direction he should move to reach the goal point. Sometimes, the person gets lost in the flat area.



- **Ridges:** It is a challenging problem where the person finds two or more local maxima of the same height commonly. It becomes difficult for the person to navigate the right point and stuck to that point itself.



Simulated Annealing

Simulated annealing is similar to the hill climbing algorithm. It works on the current situation. It picks **a random move** instead of picking **the best move**. If the move leads to the improvement of the current situation, it is always accepted as a step towards the solution state, else it accepts the move having **a probability less than 1**. This search technique was first used in **1980** to solve **VLSI layout** problems. It is also applied for factory scheduling and other large optimization tasks.

Local Beam Search

Local beam search is quite different from random-restart search. It keeps track of **k** states instead of just one. It selects **k** randomly generated states, and expand them at each step. If any state is a goal state, the search stops with success. Else it selects the best **k** successors from the complete list and repeats the same process. In random-restart search where each search process runs independently, but in local beam search, the necessary information is shared between the parallel search processes.

Disadvantages of Local Beam search

- This search can suffer from a lack of diversity among the **k** states.
- It is an expensive version of hill climbing search.

4. Local search in continuous space:

- A local search is first conducted in the continuous space **until a local optimum is reached**. It then switches to a discrete space that represents a discretization of the continuous model to find an improved solution from there.
- The process continues switching between the two problem formulations until no further improvement can be found in either.
- To perform local search in continuous state space we need techniques from calculus.
- The main technique to find a minimum is called gradient descent.

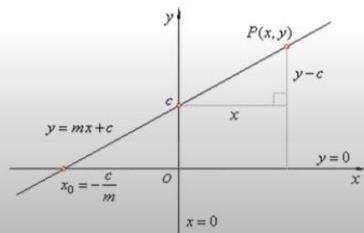
Gradient Descent

- A gradient measures how much the output of a function changes if you change the inputs a little bit.
- In machine learning, a gradient is a derivative of a function that has more than one input variable. Known as the slope of a function in mathematical terms, the gradient simply measures the change in all weights with regard to the change in error.
- Gradient descent is an iterative optimization algorithm to find the minimum of a function.
- Gradient Descent is an optimization algorithm for finding a local minimum of a differentiable function. Gradient descent is simply used in machine learning to find the values of a function's parameters (coefficients) that minimize a cost function as far as possible.

Linear Regression

- In statistics, linear regression is a linear approach to modelling the relationship between a dependent variable and one or more independent variables.

$$Y = m X + c$$



- Loss Function –**

The loss is the error in our predicted value of m and c . Our goal is to minimize this error to obtain the most accurate value of m and c .

Mean Squared Error function to calculate the loss:

Step 1 : Find the difference between the actual y and predicted y value($y = mx + c$), for a given x .

Step 2 : Square this difference.

Step 3 : Find the mean of the squares for every value in X .

- Mean Squared Error Equation**

$$E = \frac{1}{n} \sum_{i=0}^n (y_i - \bar{y}_i)^2$$

- Substituting the value of \bar{y}_i**

$$E = \frac{1}{n} \sum_{i=0}^n (y_i - (mx_i + c))^2$$

Gradient Descent Algorithm

- Applying gradient descent to **m** and **c** and approach it step by step:

Step 1 : Initially let **m** = 0 and **c** = 0. Let **L** be our learning rate. This controls how much the value of **m** changes with each step. **L** could be a small value like 0.0001 for good accuracy.

Step 2 : Calculate the partial derivative of the loss function with respect to **m**, and plug in the current values of **x**, **y**, **m** and **c** in it to obtain the derivative value **D**.

$$D_m = \frac{1}{n} \sum_{i=0}^n 2(y_i - (mx_i + c))(-x_i)$$

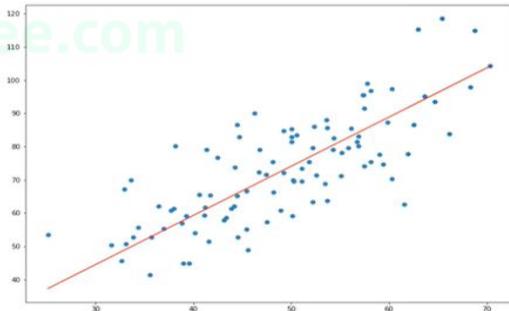
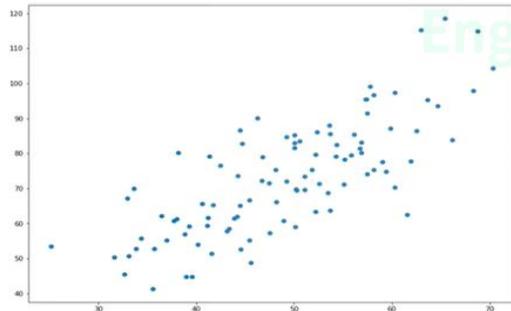
$$D_m = \frac{-2}{n} \sum_{i=0}^n x_i(y_i - \bar{y}_i)$$

Step 3 : Now we update the current value of **m** and **c** using the following equation:

$$m = m - L \times D_m$$

$$c = c - L \times D_c$$

Step 4 : We repeat this process until our loss function is a very small value or ideally 0 (which means 0 error or 100% accuracy). The value of **m** and **c** that we are left with now will be the optimum values.



5. Search with non-deterministic actions:

In Deterministic Environment the agent knows that a particular action will take it from state S1 to another state S2

In Non-Deterministic Environment the agent is not sure that a particular action would take from state S1 to S2

Fully Observable Environment is the one in which the agent has sensors which gives the complete information about the Environment to the agent

Partially Observable Environment is the one in which the agent has incomplete information of the Environment, due to faulty sensors or noisy sensors

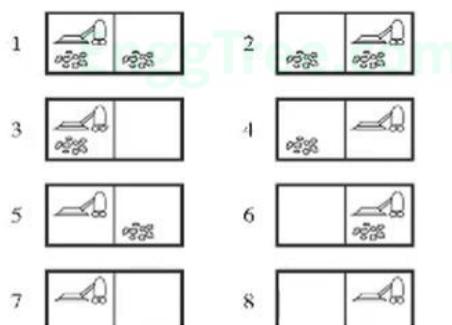
The erratic vacuum world

This is an example of Non-deterministic action

We consider a vacuum world having an error in the Suck operation, there are two erratic operations w r t Suck operation

1) When Suck operation is done on a dirty square the action cleans the square and sometimes also cleans up dirt in an adjacent square

2) When Suck operation is applied to a clean square the action sometimes deposits dirt on it



With Non-Deterministic action, if the agent takes the suck action in state 1 then the agent can be either in state 5 or state 7

There must be a contingency plan to get to the goal, as a sequence of actions will not be sufficient

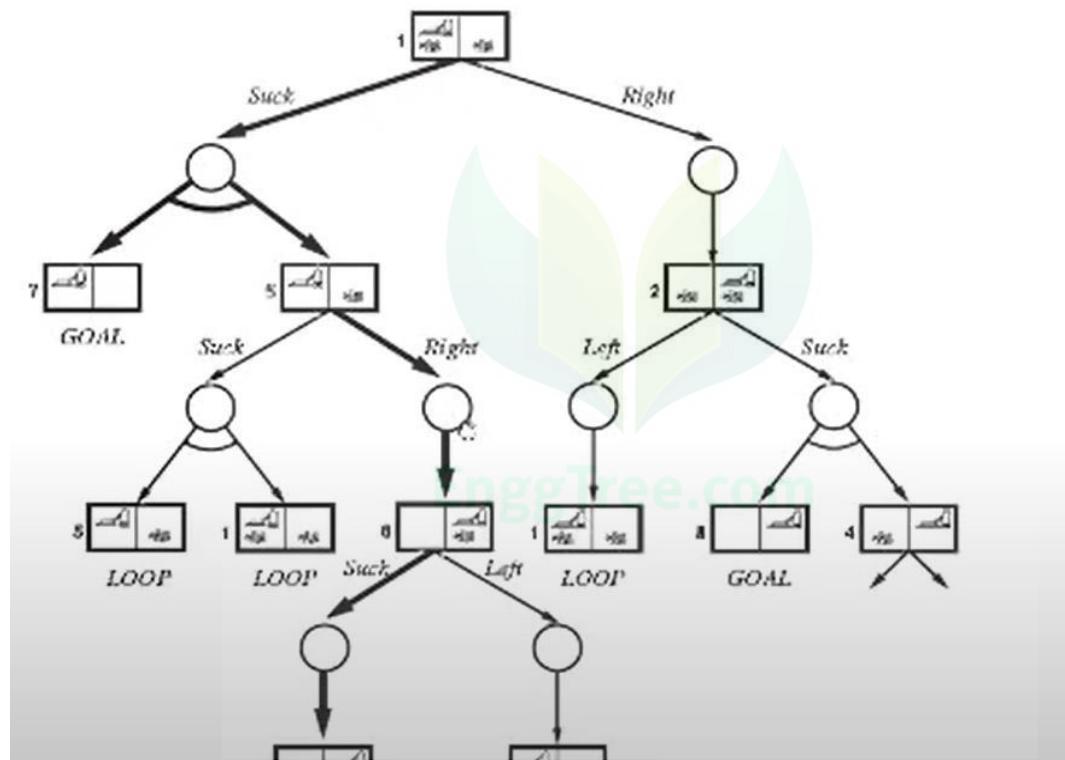
[Suck, if State = 5 then [Right, Suck] else []].

AND-OR search trees

In a deterministic environment, the branching depends only on agents choice of action, such nodes are called as OR nodes (In Vacuum world the possible actions are Left OR Right OR Suck).

In a nondeterministic environment, branching also depends on the environments choice of the outcome, such nodes are called as AND nodes

Hence we have AND-OR search tree in such cases



AND-OR search trees

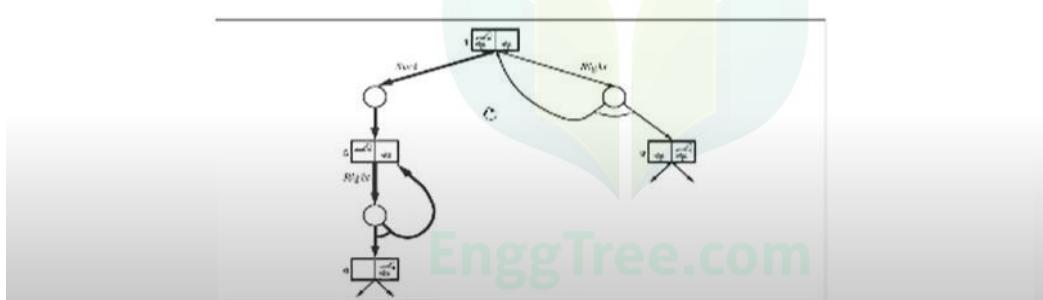
A solution for an AND-OR search problem is a subtree that

- (1) Has a goal node at every leaf
- (2) Specifies one action at each of its OR nodes
- (3) Includes every outcome branch at each of its AND nodes.

AND-OR search can be explored using recursive Depth-First search or by breadth-first or best-first methods

Slippery Vacuum-World

Another case of Non-Deterministic action in which the Right action fails sometimes. When the agent is in the Left region, and takes the Right action, it may go to the right region or may remain in the Left region if the operation fails



6. Search in partial observable environments:

- Partial observability: The agent's percepts do not suffice to pin down the exact state
- In such environment an action may lead to one of several possible outcomes—even if the environment is deterministic.
- To solve partially observable problems the concept of the “belief state” is utilized
- The belief state represents the agent’s current belief about the possible physical states it might be in, given the sequence of actions and percepts up to that point
- The belief state concept can be studied in two different scenarios;
 - Searching with no observation
 - Searching with observations

Searching with no observation

- Sensorless or conformant agents: percepts provide no information at all
- They are surprisingly useful, because they don't rely on sensors working properly
- Sensorless Vacuum world (geography known, not location & dirt distribution)
 - Initial state could be any element of the set {1, 2, 3, 4, 5, 6, 7, 8}
 - Action right will cause it to be in one of the states {2, 4, 6, 8}
 - The action sequence [Right, Suck] will always end up in one of the states {4, 8}.
 - Sequence [Right, Suck, Left, Suck] is guaranteed to reach the goal state 7 irrespective of the start state
- To solve sensorless problems, space of belief-states is searched rather than physical
- Believe-state space is fully observable as it is always known to the agent

Searching with no observation

- Suppose the underlying physical problem P is defined by ACTIONS_p , RESULT_p , GOAL-TEST_p , and STEP-COST_p .
- **Belief states:** If P has N states, then the sensorless problem has up to 2^N states
- **Initial state:** Typically the set of all states in P
- **Actions:** If a belief state $b = \{s_1, s_2\}$, and $\text{ACTIONS}_p(s_1) \neq \text{ACTIONS}_p(s_2)$; legal actions?
 - Take union of all the actions, given that illegal action has no effect on the environment
 - Otherwise intersection, i.e. the set of actions legal in all the states (illegal action might be the end of the world)
- **Transition model:**
 - with deterministic actions $b' = \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ and } s \in b\}$, sometimes called prediction
 - with non-deterministic actions $b' = \text{RESULT}(b, a) = \{s' : s' \in \text{RESULT}_P(s, a) \text{ and } s \in b\} = \bigcup_{s \in b} \{\text{RESULT}_P(s, a)\}$ union of all
- **Goal test:** a belief state is a goal state if all physical states in it satisfy GOAL-TEST_p .
- **Path cost:** Same action can have different costs in different states of a belief state, an assumption can make the formulation easy, i.e. the cost of an action is the same in all states

Searching with observation

- In partial observations, usually several states could have produced any given percept
- The percept [A, Dirty] is produced by state 3 as well as by state 1 in local-sensing vacuum (where only a position sensor and a local dirt sensor but no global dirt sensor)
- In such case the initial belief state for the local-sensing vacuum world will be {1,3}
- ACTIONS , STEP-COST , and GOAL-TEST are same as for sensorless problems
- Transition model is a bit different and consists of three stages
- **Prediction:** given the action a in belief state b , $b' = \text{PREDICT}(b, a)$, same as for sensorless
- **Observation prediction:** set of percepts that could be observed in the predicted belief state

$$\text{POSSIBLE-PERCEPTS}(b') = \{o : o = \text{PERCEP}(s) \text{ and } s \in b'\}$$
- **Update:** for each possible percept, the belief state that would result from the percept

$$b_o = \text{UPDATE}(b, o) = \{s : o = \text{PERCEP}(s) \text{ and } s \in b\}$$

updated belief state b_o can be no larger than the predicted belief state b' ;

Searching with observation

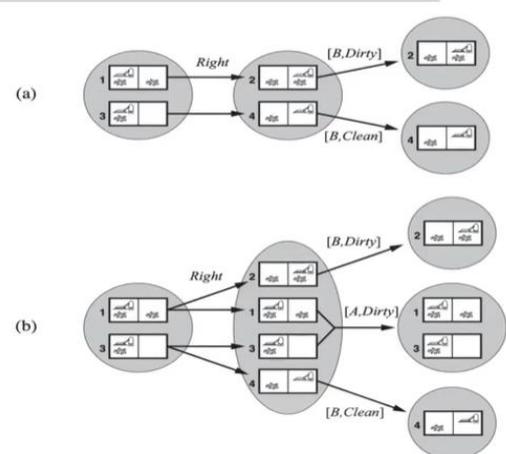
➤ Putting these three stages together, to get the transition model

RESULTS(b, a) = { b_o : $b_o = \text{UPDATE}(\text{PREDICT}(b, a), o)$ and $\text{POSSIBLE-PERCEPTS}(\text{PREDICT}(b, a))$ }

➤ The next belief state is achieved a given action a the subsequent possible percepts

➤ Figure a, shows transition model for local-sensir vacuum world

➤ Figure b, shows transition model for slippery vacuum world

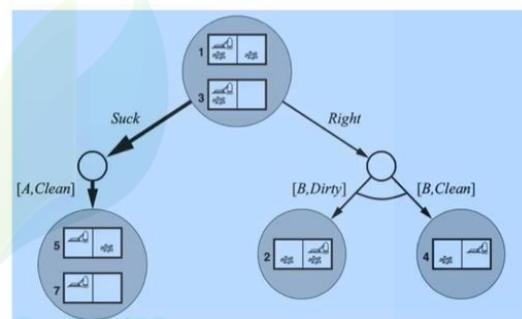


Solving partially observable problems

➤ Given the formulation discussed previously, the AND-OR search algorithm can be applied directly to derive a solution

➤ The solution is the conditional plan that tests the belief state rather than the actual state

➤ The first level of the AND-OR search tree for a problem in the local-sensing vacuum is



Online Search Problems

➤ The offline search algorithms compute a complete solution before setting foot in the real world and then execute the solution

➤ Online search takes an action, then it observes the environment and computes the next

➤ Suitable for dynamic or semidynamic domains where there is a penalty for sitting around and computing too long

➤ Also useful for

- Unknown environments
- Non-deterministic domains

➤ Examples

- Web search
- Autonomous vehicle

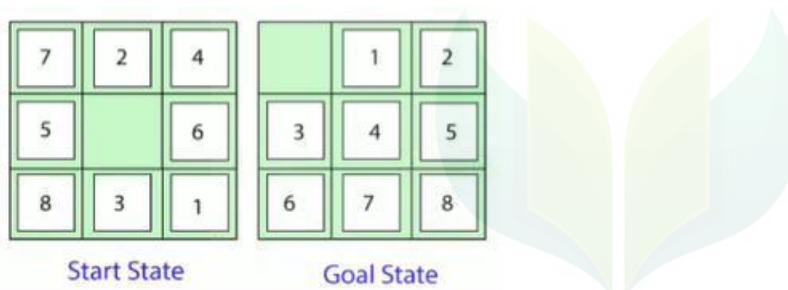
7. Online Search Agents and Unknown Environments:

Online Search Agents And Unknown Environments

An agent is anything that can perceive its environment through sensors and acts upon that environment through effectors.

Offline Search Agents- The agents who compute a complete solution before setting foot in the real world and then execute the solution is called Offline search Agents.

Online Search Agents- The online search works on the concept of interleave computation and action. In the real world, Online Search Agent first takes an action, then it observes the environment and computes the next action. Online search is a good idea in unknown environments, where the agent does not know what states exist or what its actions do.



Online Search Agents And Unknown Environments

Online search problem –

Assume deterministic (If the next state is completely determined by the current action of the agent, then the environment is deterministic.), fully observable environment.

Agent only knows:

Actions(s) – list of actions allowed in state s .

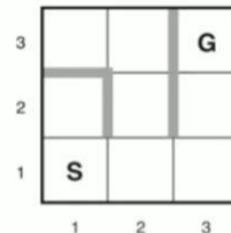
Step-cost function $c(s, a, s')$ – cannot be used until agent knows that s' is the outcome of doing a .

Goal-Test(s) -In particular, it doesn't know Result (s, a) except by actually being in s and doing a , then observing s' .

$s = \text{state}$
 $a = \text{action}$
 $s' = \text{state after doing action } a$
 $c = \text{Step cost function}$

Online Search Agents And Unknown Environments

Online search problem



In this Maze, the agent starts with 'S' state i.e. (1, 1) and then leads to (1, 2), likewise moving up leading it then achieves goal state 'G'. The going down will lead it to (1, 1) i.e. start state. The degree of achieving goal state fastly can be improved.

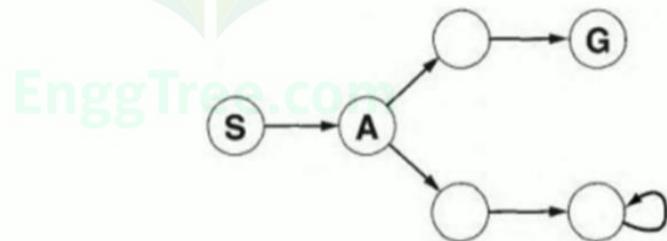
Competitive ratio = Online cost / Best cost.

Online path cost: total cost of the path that the agent actually travels.

Best cost: cost of the shortest path “if it knew the search space in advance”

Competitive ratio means shortest path. The competitive ratio must be very small.

Online search problem



Consider two goal states as shown in figure.

In this fig, online search algorithm visits states S and A, both the state spaces are identical so it must be explored in both cases. One link is to goal state (G) and other lead to dead end on both states spaces.

Online depth first search agents

```

function ONLINE-DFS-AGENT( $s'$ ) returns an action
  inputs:  $s'$ , a percept that identifies the current state
  persistent:  $result$ , a table indexed by state and action, initially empty
     $untried$ , a table that lists, for each state, the actions not yet tried
     $unbacktracked$ , a table that lists, for each state, the backtracks not yet tried
     $s, a$ , the previous state and action, initially null

  if GOAL-TEST( $s'$ ) then return stop
  if  $s'$  is a new state (not in  $untried$ ) then  $untried[s'] \leftarrow \text{ACTIONS}(s')$ 
  if  $s$  is not null then
     $result[s, a] \leftarrow s'$ 
    add  $s$  to the front of  $unbacktracked[s']$ 
  if  $untried[s']$  is empty then
    if  $unbacktracked[s']$  is empty then return stop
    else  $a \leftarrow$  an action  $b$  such that  $result[s', b] = \text{POP}(unbacktracked[s'])$ 
  else  $a \leftarrow \text{POP}(untried[s'])$ 
   $s \leftarrow s'$ 
  return  $a$ 

```

$s = \text{state}$
 $a = \text{action}$
 $s' = \text{state after doing action } a$
 $c = \text{Step cost function}$

1. Game Theory:

Game theory is basically a branch of mathematics that is used to typical strategic interaction between different players (agents), all of which are equally rational, in a context with predefined rules (of playing or maneuvering) and outcomes. Every player or agent is a rational entity who is selfish and tries to maximize the reward to be obtained using a particular strategy. All the players abide by certain rules in order to receive a predefined payoff- a reward after a certain outcome. Hence, a GAME can be defined as a set of players, actions, strategies, and a final payoff for which all the players are competing. Game Theory has now become a describing factor for both Machine Learning algorithms and many daily life situations.

Consider the SVM (Support Vector Machine) for instance. According to Game Theory, the SVM is a game between 2 players where one player challenges the other to find the best hyperplane after providing the most difficult points for classification. The final payoff of this game is a solution that will be a trade-off between the strategic abilities of both players competing.

Nash equilibrium:

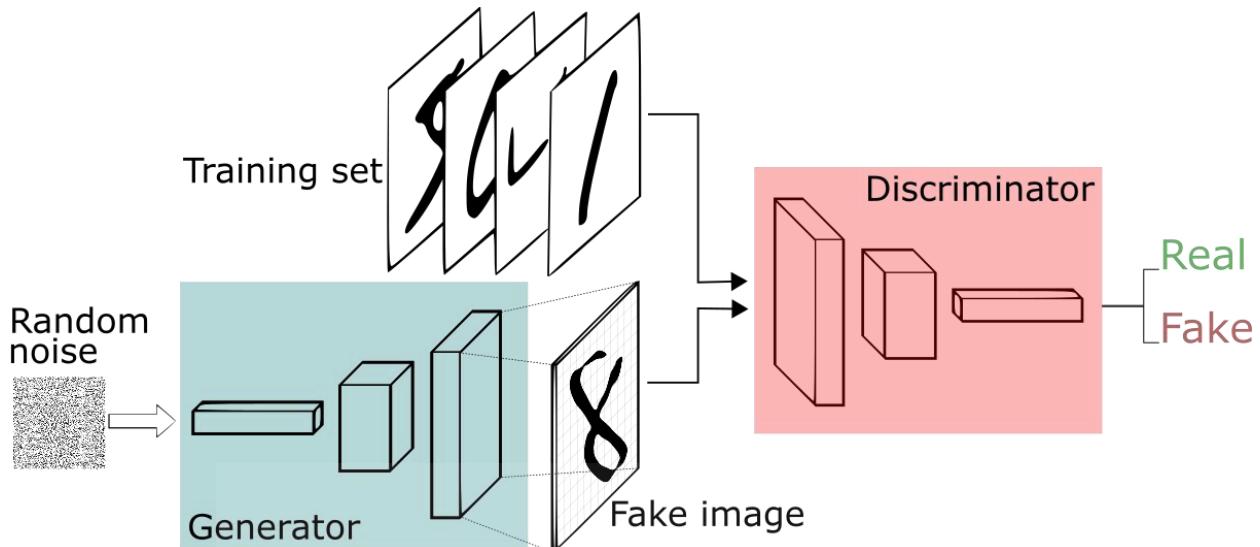
Nash equilibrium can be considered the essence of Game Theory. It is basically a state, a point of equilibrium of collaboration of multiple players in a game. Nash Equilibrium guarantees maximum profit to each player.

Let us try to understand this with the help of Generative Adversarial Networks (GANs).

What is GAN?

It is a combination of two neural networks: the Discriminator and the Generator. The Generator Neural Network is fed input images which it analyzes and then produces new sample images, which are made to represent the actual input images as close as possible. Once the images have been produced, they are sent to the Discriminator Neural Network. This neural network judges the images sent to it and classifies them as generated images and actual input images. If the image is classified as the original image, the DNN changes its parameters of judging. If the image is classified as a generated image, the image is rejected and returned to the GNN. The GNN then alters its parameters in order to improve the quality of the image produced.

This is a competitive process which goes on until both neural networks do not require to make any changes in their parameters and there can be no further improvement in both neural networks. This state of no further improvement is known as NASH EQUILIBRIUM. In other words, GAN is a 2-player competitive game where both players are continuously optimizing themselves to find a Nash Equilibrium.



But how do we know if the game has reached Nash Equilibrium?

In any game, one of the agents is required to disclose their strategy in front of the other agents. After the revelation, if none of the players changes their strategies, it is understood that the game has reached Nash Equilibrium.

Now that we are aware of the basics of Game Theory, let us try to understand how Nash Equilibrium is attained in a simultaneous game. There are many examples but the most famous is the Prisoner's Dilemma. There are some more examples such as the Closed-bag exchange Game, the Friend or For Game, and the iterated Snowdrift Game.

In all these games, two players are involved and the final payoff is a result of a decision that has to be made by both players. Both players have to make a choice between defection and co-operation. If both players cooperate, the final payoff will turn out to be positive for both. However, if both defect, the final payoff will be negative for both players. If there is a combination of one player defecting and the other co-operating, the final payoff will be positive for one and negative for another.

Here, Nash Equilibrium plays an important role. Only if both players jot out a strategy that benefits each other and provide both with a positive payoff, the solution to this problem will be optimal.

There are many more real examples and a number of pieces of code that try to solve this dilemma. The basic essence, however, is the attainment of the Nash Equilibrium in an uncomfortable situation.

Where is GAME THEORY now?

Game Theory is increasingly becoming a part of the real-world in its various applications in areas like public health services, public safety, and wildlife. Currently, game theory is being used in adversary training in GANs, multi-agent systems, and imitation and reinforcement learning. In the case of perfect information and symmetric games, many Machine Learning and

Deep Learning techniques are applicable. The real challenge lies in the development of techniques to handle incomplete information games, such as Poker. The complexity of the game lies in the fact that there are too many combinations of cards and the uncertainty of the cards being held by the various players.

Types of Games:

Currently, there are about 5 types of classification of games. They are as follows:

1. Zero-Sum and Non-Zero Sum Games: In non-zero-sum games, there are multiple players and all of them have the option to gain a benefit due to any move by another player. In zero-sum games, however, if one player earns something, the other players are bound to lose a key payoff.
2. Simultaneous and Sequential Games: Sequential games are the more popular games where every player is aware of the movement of another player. Simultaneous games are more difficult as in them, the players are involved in a concurrent game. BOARD GAMES are the perfect example of sequential games and are also referred to as turn-based or extensive-form games.



Perfect Information game.

4. Asymmetric and Symmetric Games: Asymmetric games are those in which each player has a different and usually conflicting final goal. Symmetric games are those in which all players have the same ultimate goal but the strategy being used by each is completely different.
5. Co-operative and Non-Co-operative Games: In non-co-operative games, every player plays for himself while in co-operative games, players form alliances in order to achieve the final goal.

2. Optimal Decisions in Games:

Humans' intellectual capacities have been engaged by games for as long as civilization has existed, sometimes to an alarming degree. Games are an intriguing subject for AI researchers because of their abstract character. A game's state is simple to depict, and actors are usually limited to a small number of actions with predetermined results. Physical games, such as croquet and ice hockey, contain significantly more intricate descriptions, a much wider variety of possible actions, and rather ambiguous regulations defining the legality of activities. With the exception of robot soccer, these physical games have not piqued the AI community's interest.

Games are usually intriguing because they are difficult to solve. Chess, for example, has an average branching factor of around 35, and games frequently stretch to 50 moves per player, therefore the search tree has roughly 35100 or 10154 nodes (despite the search graph having

“only” about 1040 unique nodes). As a result, games, like the real world, necessitate the ability to make some sort of decision even when calculating the best option is impossible.

Inefficiency is also heavily punished in games. Whereas a half-efficient implementation of A search will merely take twice as long to complete, a chess software that is half as efficient in utilizing its available time will almost certainly be beaten to death, all other factors being equal. As a result of this research, a number of intriguing suggestions for making the most use of time have emerged.

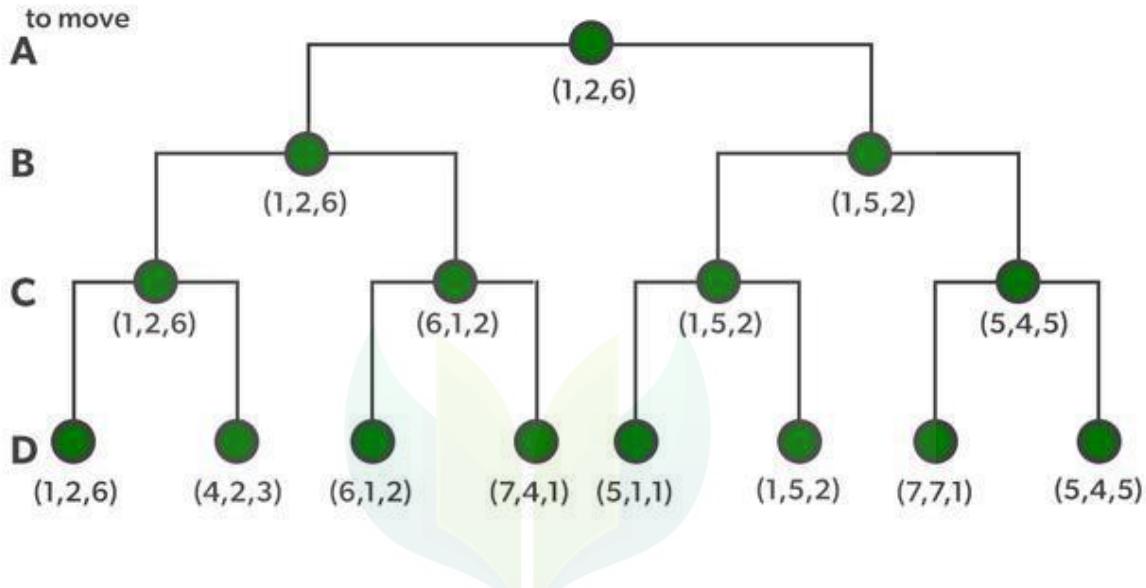
Optimal Decision Making in Games

Let us start with games with two players, whom we’ll refer to as MAX and MIN for obvious reasons. MAX is the first to move, and then they take turns until the game is finished. At the conclusion of the game, the victorious player receives points, while the loser receives penalties. A game can be formalized as a type of search problem that has the following elements:

- S0: The initial state of the game, which describes how it is set up at the start.
- Player (s): Defines which player in a state has the move.
- Actions (s): Returns a state’s set of legal moves.
- Result (s, a): A transition model that defines a move’s outcome.
- Terminal-Test (s): A terminal test that returns true if the game is over but false otherwise. Terminal states are those in which the game has come to a conclusion.
- Utility (s, p): A utility function (also known as a payout function or objective function) determines the final numeric value for a game that concludes in the terminal state s for player p. The result in chess is a win, a loss, or a draw, with values of +1, 0, or 1/2. Backgammon’s payoffs range from 0 to +192, but certain games have a greater range of possible outcomes. A zero-sum game is defined (confusingly) as one in which the total reward to all players is the same for each game instance. Chess is a zero-sum game because each game has a payoff of 0 + 1, 1 + 0, or 1/2 + 1/2. “Constant-sum” would have been a preferable name, 22 but zero-sum is the usual term and makes sense if each participant is charged 1.

The game tree for the game is defined by the beginning state, ACTIONS function, and RESULT function—a tree in which the nodes are game states and the edges represent movements. The figure below depicts a portion of the tic-tac-toe game tree (noughts and crosses). MAX may make nine different maneuvers from his starting position. The game alternates between MAXs setting an X and MINs placing an O until we reach leaf nodes corresponding to terminal states, such as one player having three in a row or all of the squares being filled. The utility value of the terminal state from the perspective of MAX is shown by the number on each leaf node; high values are thought to be beneficial for MAX and bad for MIN

The game tree for tic-tac-toe is relatively short, with just $9! = 362,880$ terminal nodes. However, because there are over 1040 nodes in chess, the game tree is better viewed as a theoretical construct that cannot be realized in the actual world. But, no matter how big the game tree is, MAX's goal is to find a solid move. A tree that is superimposed on the whole game tree and examines enough nodes to allow a player to identify what move to make is referred to as a search tree.



A sequence of actions leading to a goal state—a terminal state that is a win—would be the best solution in a typical search problem. MIN has something to say about it in an adversarial search. MAX must therefore devise a contingent strategy that specifies M A X's initial state move, then MAX's movements in the states resulting from every conceivable MIN response, then MAX's moves in the states resulting from every possible MIN reaction to those moves, and so on. This is quite similar to the AND-OR search method, with MAX acting as OR and MIN acting as AND. When playing an infallible opponent, an optimal strategy produces results that are as least as excellent as any other plan. We'll start by demonstrating how to find the best plan.

We'll move to the trivial game in the **figure below** since even a simple game like tic-tac-toe is too complex for us to draw the full game tree on one page. MAX's root node moves are designated by the letters a1, a2, and a3. MIN's probable answers to a1 are b1, b2, b3, and so on. This game is over after MAX and MIN each make one move. (In game terms, this tree consists of two half-moves and is one move deep, each of which is referred to as a ply.) The terminal states in this game have utility values ranging from 2 to 14.

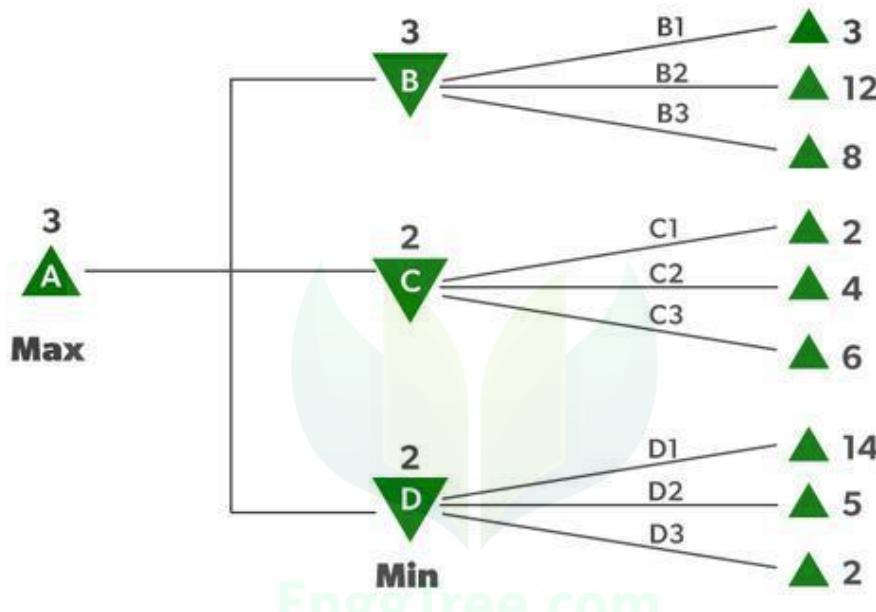
Game's Utility Function

The optimal strategy can be found from the minimax value of each node, which we express as MINIMAX, given a game tree (n). Assuming that both players play optimally from there

through the finish of the game, the utility (for MAX) of being in the corresponding state is the node's minimax value. The usefulness of a terminal state is obviously its minimax value. Furthermore, if given the option, MAX prefers to shift to a maximum value state, whereas MIN wants to move to a minimum value state. So here's what we've got:

$\text{MINIMAX}(s) =$

$$\begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MIN} \end{cases}$$



Let's use these definitions to analyze the game tree shown in **the figure above**. The game's UTILITY function provides utility values to the terminal nodes on the bottom level. Because the first MIN node, B, has three successor states with values of 3, 12, and 8, its minimax value is 3. Minimax value 2 is also used by the other two MIN nodes. The root node is a MAX node, with minimax values of 3, 2, and 2, resulting in a minimax value of 3. We can also find the root of the minimax decision: action a1 is the best option for MAX since it leads to the highest minimax value.

This concept of optimal MAX play requires that MIN plays optimally as well—it maximizes MAX's worst-case outcome. What happens if MIN isn't performing at its best? Then it's a simple matter of demonstrating that MAX can perform even better. Other strategies may outperform the minimax method against suboptimal opponents, but they will always outperform optimal opponents.

3. Alpha-Beta Pruning Search:

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.

- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called pruning. This involves two threshold parameter Alpha and beta for future expansion, so it is called alpha-beta pruning. It is also called as Alpha-Beta Algorithm.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.
- The two-parameter can be defined as:
 - a. Alpha: The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is $-\infty$.
 - b. Beta: The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is $+\infty$.

The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

Condition for Alpha-beta pruning:

The main condition which required for alpha-beta pruning

$$\alpha >= \beta$$

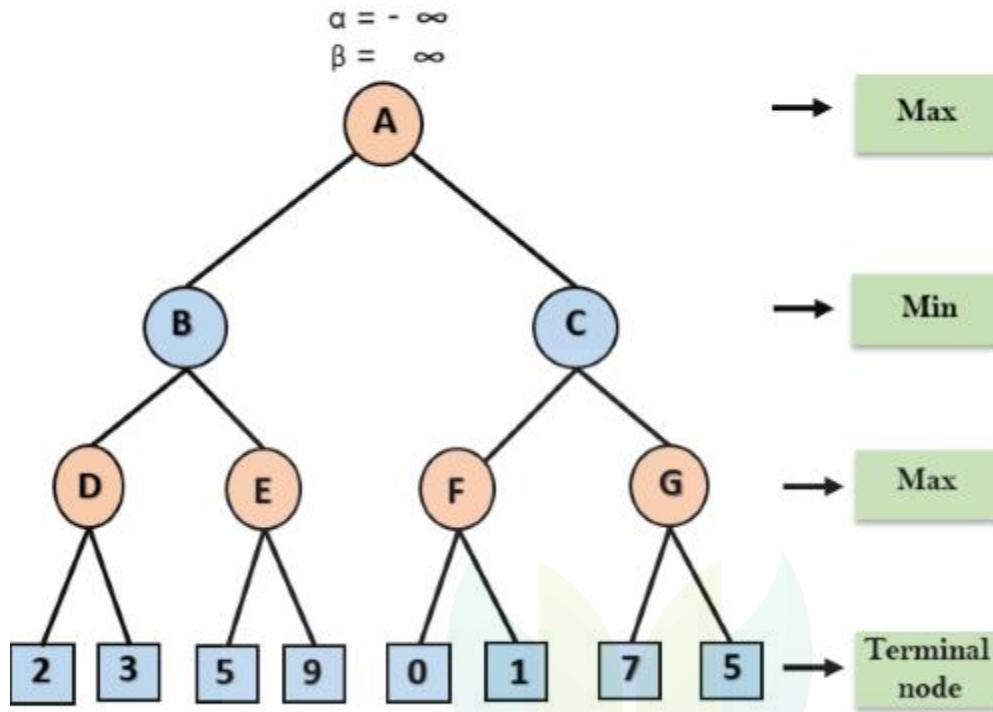
Key points about alpha-beta pruning:

- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

Working of Alpha-Beta Pruning:

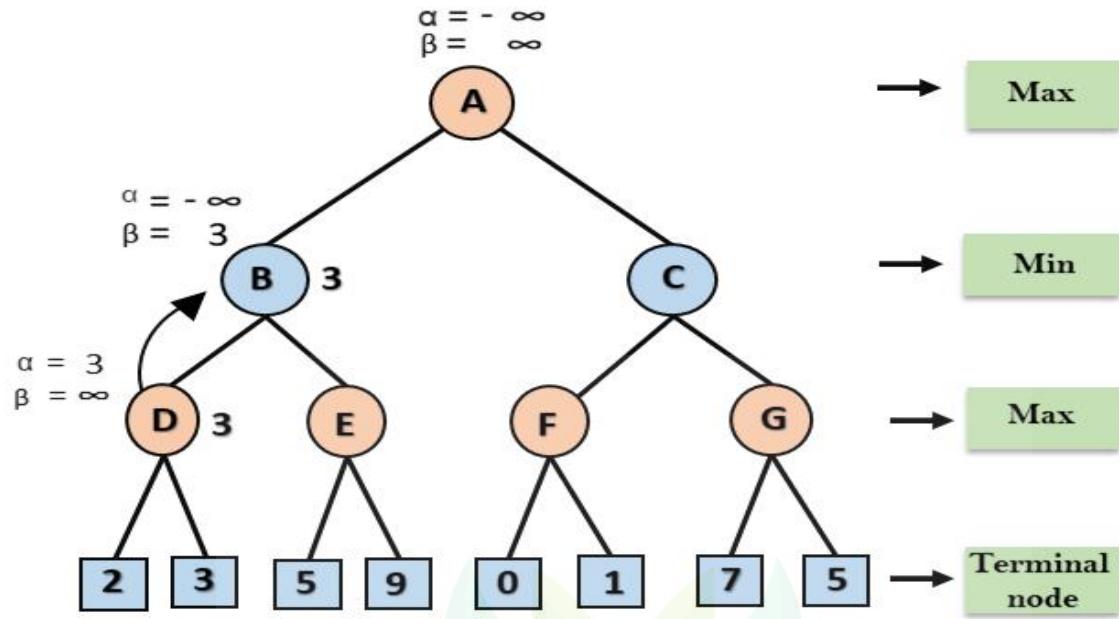
Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

Step 1: At the first step the, Max player will start first move from node A where $\alpha = -\infty$ and $\beta = +\infty$, these value of alpha and beta passed down to node B where again $\alpha = -\infty$ and $\beta = +\infty$, and Node B passes the same value to its child D.



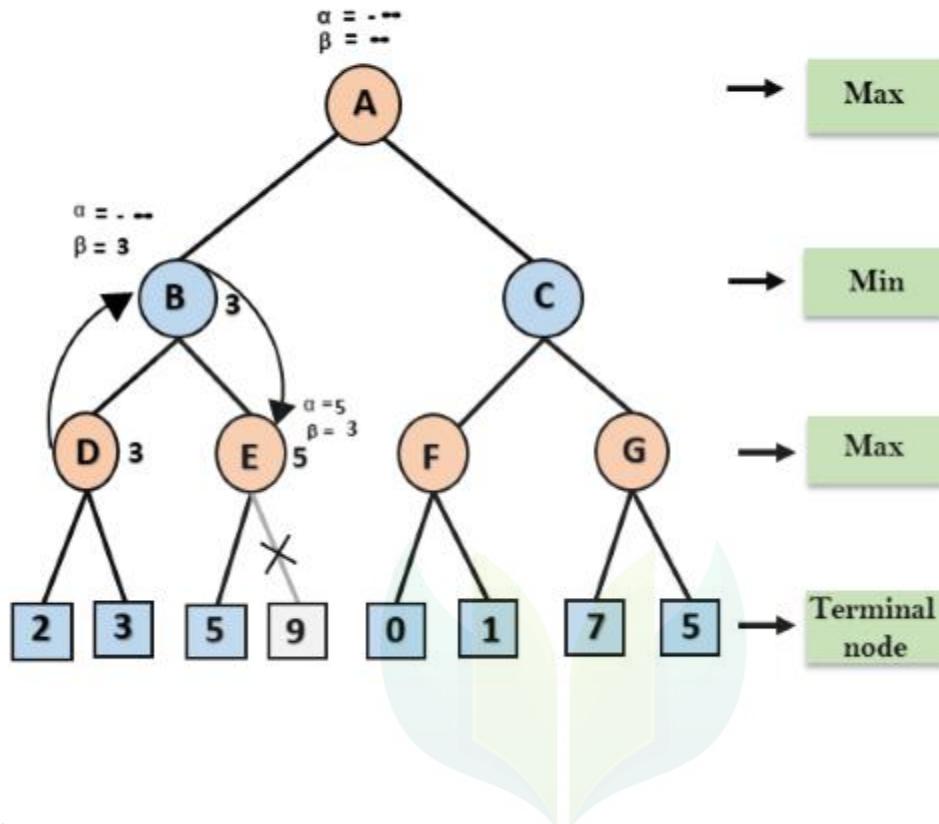
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

Step 3: Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now $\beta = +\infty$, will compare with the available subsequent nodes value, i.e. min ($\infty, 3$) = 3, hence at node B now $\alpha = -\infty$, and $\beta = 3$.



In the next step, algorithm traverse the next successor of Node B which is node E, and the values of $\alpha = -\infty$, and $\beta = 3$ will also be passed.

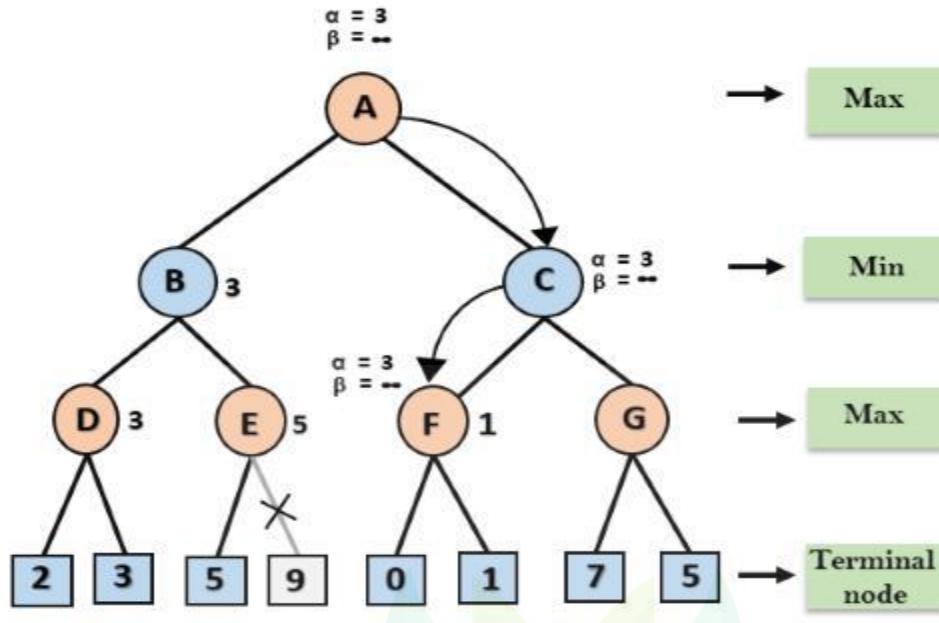
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so $\max(-\infty, 5) = 5$, hence at node E $\alpha = 5$ and $\beta = 3$, where $\alpha >= \beta$, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.



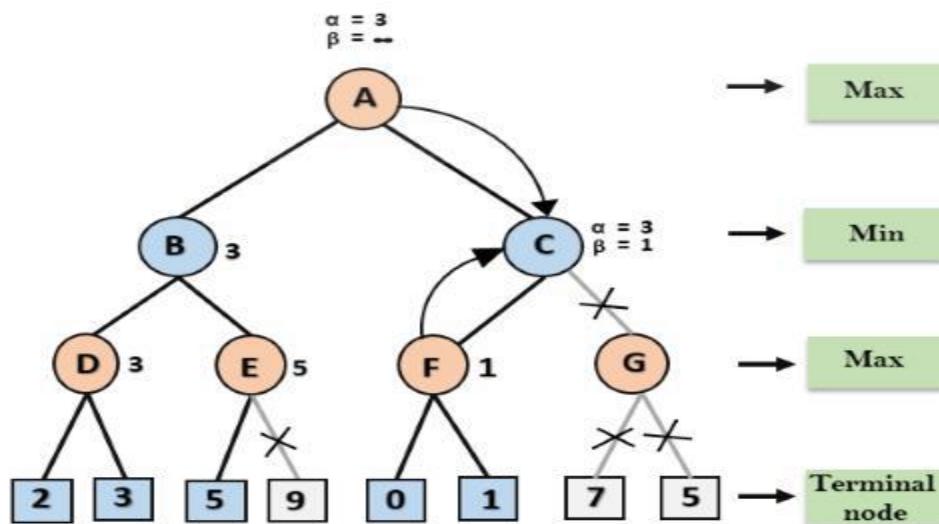
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (- ∞ , 3)= 3, and $\beta= +\infty$, these two values now passes to right successor of A which is Node C.

At node C, $\alpha=3$ and $\beta= +\infty$, and the same values will be passed on to node F.

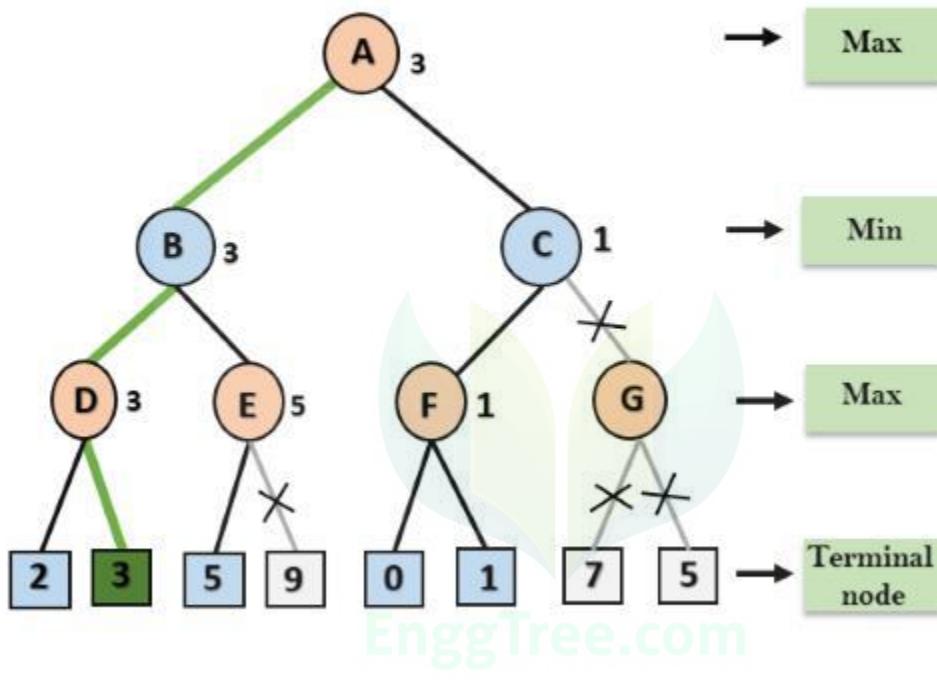
Step 6: At node F, again the value of α will be compared with left child which is 0, and $\max(3,0)= 3$, and then compared with right child which is 1, and $\max(3,1)= 3$ still α remains 3, but the node value of F will become 1.



Step 7: Node F returns the node value 1 to node C, at C $\alpha=3$ and $\beta=+\infty$, here the value of beta will be changed, it will compare with 1 so $\min(\infty, 1) = 1$. Now at C, $\alpha=3$ and $\beta=1$, and again it satisfies the condition $\alpha \geq \beta$, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.



Step 8: C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is showing the nodes which are computed and nodes which have never been computed. Hence the optimal value for the maximizer is 3 for this example.



Move Ordering in Alpha-Beta pruning:

The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning.

It can be of two types:

Worst ordering: In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is $O(bm)$.

Ideal ordering: The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it

first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is $O(bm/2)$.

Rules to find good ordering:

Following are some rules to find good ordering in alpha-beta pruning:

- Occur the best move from the shallowest node.
- Order the nodes in the tree such that the best nodes are checked first.
- Use domain knowledge while finding the best move. Ex: for Chess, try order: captures first, then threats, then forward moves, backward moves.
- We can bookkeep the states, as there is a possibility that states may repeat.

4. Monte Carlo Tree Search (MCTS):

Monte Carlo Tree Search (MCTS) is a search technique in the field of Artificial Intelligence (AI). It is a probabilistic and heuristic driven search algorithm that combines the classic tree search implementations alongside machine learning principles of reinforcement learning. In tree search, there's always the possibility that the current best action is actually not the most optimal action. In such cases, MCTS algorithm becomes useful as it continues to evaluate other alternatives periodically during the learning phase by executing them, instead of the current perceived optimal strategy. This is known as the "exploration-exploitation trade-off". It exploits the actions and strategies that is found to be the best till now but also must continue to explore the local space of alternative decisions and find out if they could replace the current best.

Exploration helps in exploring and discovering the unexplored parts of the tree, which could result in finding a more optimal path. In other words, we can say that exploration expands the tree's breadth more than its depth. Exploration can be useful to ensure that MCTS is not overlooking any potentially better paths. But it quickly becomes inefficient in situations with large number of steps or repetitions. In order to avoid that, it is balanced out by exploitation. Exploitation sticks to a single path that has the greatest estimated value. This is a greedy approach and this will extend the tree's depth more than its breadth. In simple words, UCB formula applied to trees helps to balance the exploration-exploitation trade-off by periodically exploring relatively unexplored nodes of the tree and discovering potentially more optimal paths than the one it is currently exploiting. For this characteristic, MCTS becomes particularly useful in making optimal decisions in Artificial Intelligence (AI) problems.

Monte Carlo Tree Search (MCTS) algorithm:

In MCTS, nodes are the building blocks of the search tree. These nodes are formed based on the outcome of a number of simulations. The process of Monte Carlo Tree Search can be broken down into four distinct steps, viz., selection, expansion, simulation and

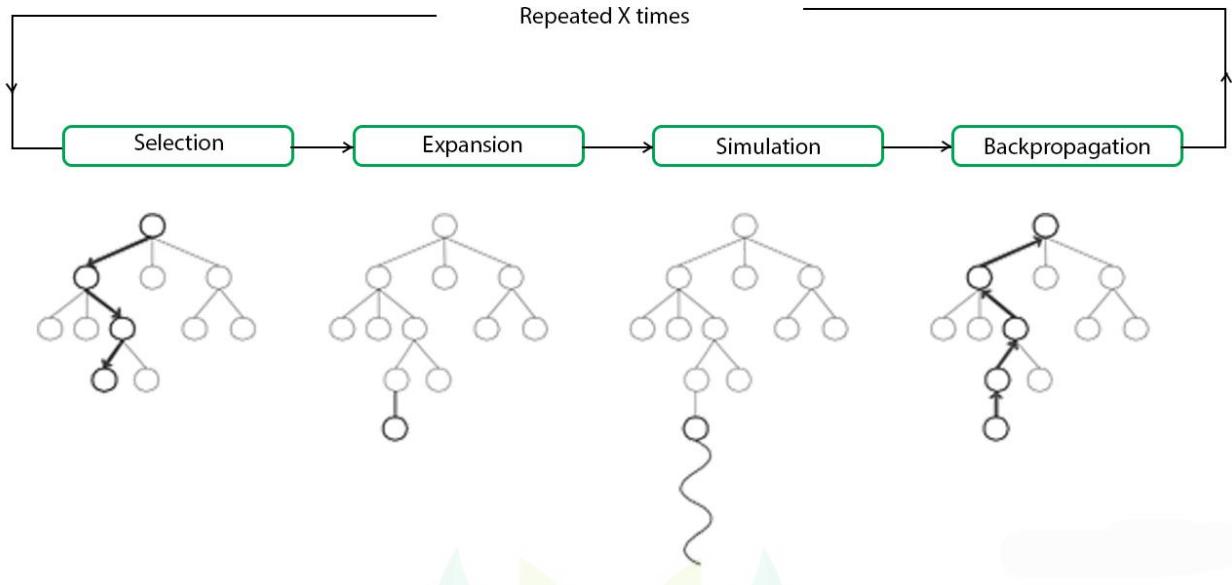
backpropagation. Each of these steps is explained in details below:

- **Selection:** In this process, the MCTS algorithm traverses the current tree from the root node using a specific strategy. The strategy uses an evaluation function to optimally select nodes with the highest estimated value. MCTS uses the Upper Confidence Bound (UCB) formula applied to trees as the strategy in the selection process to traverse the tree. It balances the exploration-exploitation trade-off. During tree traversal, a node is selected based on some parameters that return the maximum value. The parameters are characterized by the formula that is typically used for this purpose is given below.

$$S_i = x_i + C \sqrt{\frac{\ln(t)}{n_i}}$$

- where;
 - S_i = value of a node i
 - x_i = empirical mean of a node i
 - C = a constant
 - t = total number of simulations
- When traversing a tree during the selection process, the child node that returns the greatest value from the above equation will be one that will get selected. During traversal, once a child node is found which is also a leaf node, the MCTS jumps into the expansion step.
- **Expansion:** In this process, a new child node is added to the tree to that node which was optimally reached during the selection process.
 - **Simulation:** In this process, a simulation is performed by choosing moves or strategies until a result or predefined state is achieved.
 - **Backpropagation:** After determining the value of the newly added node, the remaining tree must be updated. So, the backpropagation process is performed, where it backpropagates from the new node to the root node. During the process, the number of simulation stored in each node is incremented. Also, if the new node's simulation results in a win, then the number of wins is also incremented.

The above steps can be visually understood by the diagram given below:



These types of algorithms are particularly useful in turn based games where there is no element of chance in the game mechanics, such as Tic Tac Toe, Connect 4, Checkers, Chess, Go, etc. This has recently been used by Artificial Intelligence Programs like AlphaGo, to play against the world's top Go players. But, its application is not limited to games only. It can be used in any situation which is described by state-action pairs and simulations used to forecast outcomes.

As we can see, the MCTS algorithm reduces to a very few set of functions which we can use any choice of games or in any optimizing strategy.

Advantages of Monte Carlo Tree Search:

1. MCTS is a simple algorithm to implement.
2. Monte Carlo Tree Search is a heuristic algorithm. MCTS can operate effectively without any knowledge in the particular domain, apart from the rules and end conditions, and can find its own moves and learn from them by playing random playouts.
3. The MCTS can be saved in any intermediate state and that state can be used in future use cases whenever required.
4. MCTS supports asymmetric expansion of the search tree based on the circumstances in which it is operating.

Disadvantages of Monte Carlo Tree Search:

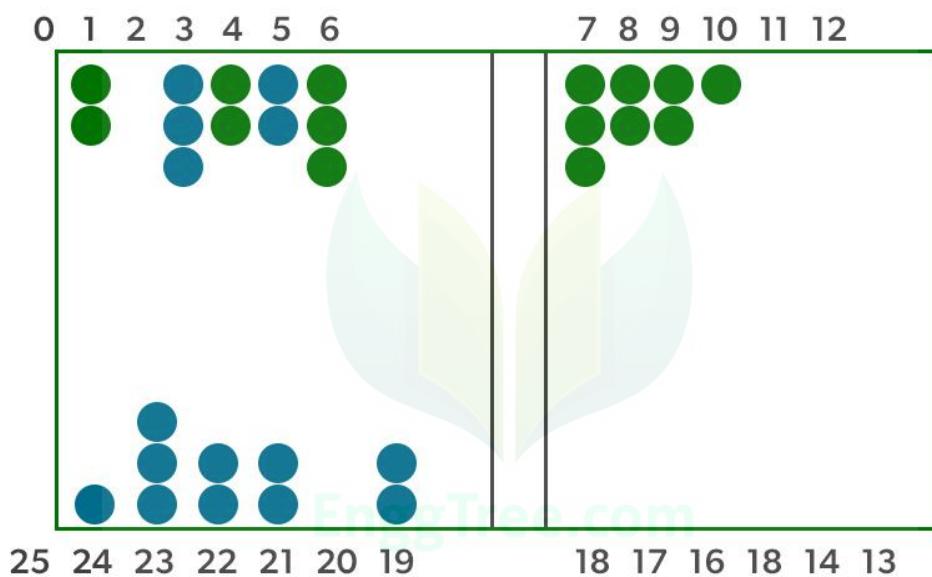
1. As the tree growth becomes rapid after a few iterations, it requires a huge amount of memory.
2. There is a bit of a reliability issue with Monte Carlo Tree Search. In certain scenarios, there might be a single branch or path, that might lead to loss against the opposition when implemented for those turn-based games. This is mainly due to the vast amount of

combinations and each of the nodes might not be visited enough number of times to understand its result or outcome in the long run.

3. MCTS algorithm needs a huge number of iterations to be able to effectively decide the most efficient path. So, there is a bit of a speed issue there.

5. Stochastic games:

Many unforeseeable external occurrences can place us in unforeseen circumstances in real life. Many games, such as dice tossing, have a random element to reflect this unpredictability. These are known as stochastic games. Backgammon is a classic game that mixes skill and luck. The legal moves are determined by rolling dice at the start of each player's turn white, for example, has rolled a 6–5 and has four alternative moves in the backgammon scenario shown in the figure below.

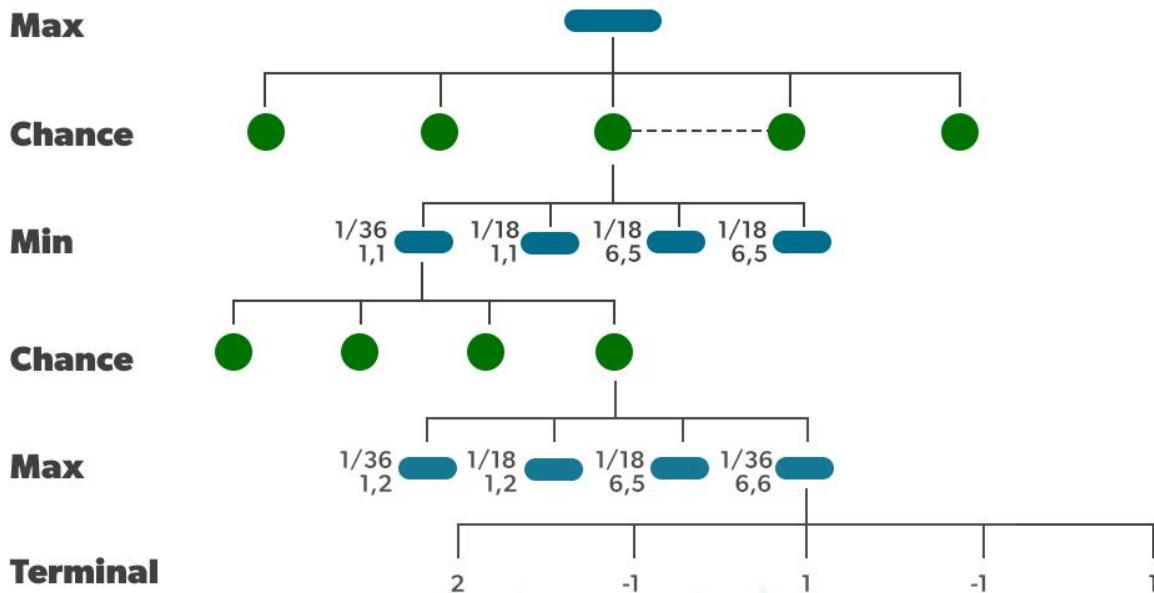


This is a standard backgammon position. The object of the game is to get all of one's pieces off the board as quickly as possible. White moves in a clockwise direction toward 25, while Black moves in a counterclockwise direction toward 0. Unless there are many opponent pieces, a piece can advance to any position; if there is only one opponent, it is caught and must start over. White has rolled a 6–5 and must pick between four valid moves: (5–10,5–11), (5–11,19–24), (5–10,10–16), and (5–11,11–16), where the notation (5–11,11–16) denotes moving one piece from position 5 to 11 and then another from 11 to 16.

Stochastic game tree for a backgammon position

White knows his or her own legal moves, but he or she has no idea how Black will roll, and thus has no idea what Black's legal moves will be. That means White won't be able to build a normal game tree-like in chess or tic-tac-toe. In backgammon, in addition to M A X and M I N nodes, a game tree must include chance nodes. The figure below depicts chance nodes as circles. The possible dice rolls are indicated by the branches leading from each chance node; each branch is labelled with the roll and its probability. There are 36 different ways to roll two dice, each equally likely, yet there are only 21 distinct rolls because a 6–5 is the same as a 5–6.

$P(1-1) = 1/36$ because each of the six doubles (1–1 through 6–6) has a probability of $1/36$. Each of the other 15 rolls has a $1/18$ chance of happening.



The following phase is to learn how to make good decisions. Obviously, we want to choose the move that will put us in the best position. Positions, on the other hand, do not have specific minimum and maximum values. Instead, we can only compute a position's anticipated value, which is the average of all potential outcomes of the chance nodes.

As a result, we can generalize the deterministic minimax value to an expected-minimax value for games with chance nodes. Terminal nodes, MAX and MIN nodes (for which the dice roll is known), and MAX and MIN nodes (for which the dice roll is unknown) all function as before. We compute the expected value for chance nodes, which is the sum of all outcomes, weighted by the probability of each chance action.

$$\text{EXPECTIMINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_a \text{EXPECTIMINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MAX} \\ \min_a \text{EXPECTIMINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MIN} \\ \sum_r P(r) \text{EXPECTIMINIMAX}(\text{RESULT}(s, r)) & \text{if } \text{PLAYER}(s) = \text{CHANCE} \end{cases}$$

where r is a possible dice roll (or other random events) and $\text{RESULT}(s,r)$ denotes the same state as s , but with the addition that the dice roll's result is r .

6. Partially observable games:

A partially observable system is one in which the entire state of the system is not fully visible to an external sensor. In a partially observable system the observer may utilise a memory system in order to add information to the observer's understanding of the system.

An example of a partially observable system would be a card game in which some of the cards are discarded into a pile face down. In this case the observer is only able to view their own cards and potentially those of the dealer. They are not able to view the face-down (used) cards,

nor the cards that will be dealt at some stage in the future. A memory system can be used to remember the previously dealt cards that are now on the used pile. This adds to the total sum of knowledge that the observer can use to make decisions.

In contrast, a fully observable system would be that of chess. In chess (apart from the 'who is moving next' state, and minor subtleties such as whether a side has castled, which may not be clear) the full state of the system is observable at any point in time.

Partially observable is a term used in a variety of mathematical settings, including that of artificial intelligence and partially observable Markov decision processes.

7. Constraint satisfaction problems:

We have seen so many techniques like Local search, Adversarial search to solve different problems. The objective of every problem-solving technique is one, i.e., to find a solution to reach the goal. Although, in adversarial search and local search, there were no constraints on the agents while solving the problems and reaching to its solutions.

Constraint satisfaction technique. By the name, it is understood that constraint satisfaction means solving a problem under certain constraints or rules.

Constraint satisfaction is a technique where a problem is solved when its values satisfy certain constraints or rules of the problem. Such type of technique leads to a deeper understanding of the problem structure as well as its complexity.

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.

In constraint satisfaction, domains are the spaces where the variables reside, following the problem specific constraints. These are the three main elements of a constraint satisfaction technique. 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.

Solving Constraint Satisfaction Problems

The requirements to solve a constraint satisfaction problem (CSP) is:

- A state-space
- The notion of the solution.

A state in state-space is defined by assigning values to some or all variables such as {X1=v1, X2=v2, and so on...}.

An assignment of values to a variable can be done in three ways:

- Consistent or Legal Assignment: An assignment which does not violate any constraint or rule is called Consistent or legal assignment.
- Complete Assignment: An assignment where every variable is assigned with a value, and the solution to the CSP remains consistent. Such assignment is known as Complete assignment.
- Partial Assignment: An assignment which assigns values to some of the variables only. Such type of assignments are called Partial assignments.

Types of Domains in CSP

There are following two types of domains which are used by the variables :

- Discrete Domain: It is an infinite domain which can have one state for multiple variables. For example, a start state can be allocated infinite times for each variable.
- Finite Domain: It is a finite domain which can have continuous states describing one domain for one specific variable. It is also called a continuous domain.

Constraint Types in CSP

With respect to the variables, basically there are following types of constraints:

- Unary Constraints: It is the simplest type of constraints that restricts the value of a single variable.
- Binary Constraints: It is the constraint type which relates two variables. A value x_2 will contain a value which lies between x_1 and x_3 .
- Global Constraints: It is the constraint type which involves an arbitrary number of variables.
- Some special types of solution algorithms are used to solve the following types of constraints:
- Linear Constraints: These type of constraints are commonly used in linear programming where each variable containing an integer value exists in linear form only.
- Non-linear Constraints: These type of constraints are used in non-linear programming where each variable (an integer value) exists in a non-linear form.

Note: A special constraint which works in real-world is known as Preference constraint.

Constraint Propagation

In local state-spaces, the choice is only one, i.e., to search for a solution. But in CSP, we have two choices either:

- We can search for a solution or
- We can perform a special type of inference called constraint propagation.

Constraint propagation is a special type of inference which helps in reducing the legal number of values for the variables. The idea behind constraint propagation is local consistency.

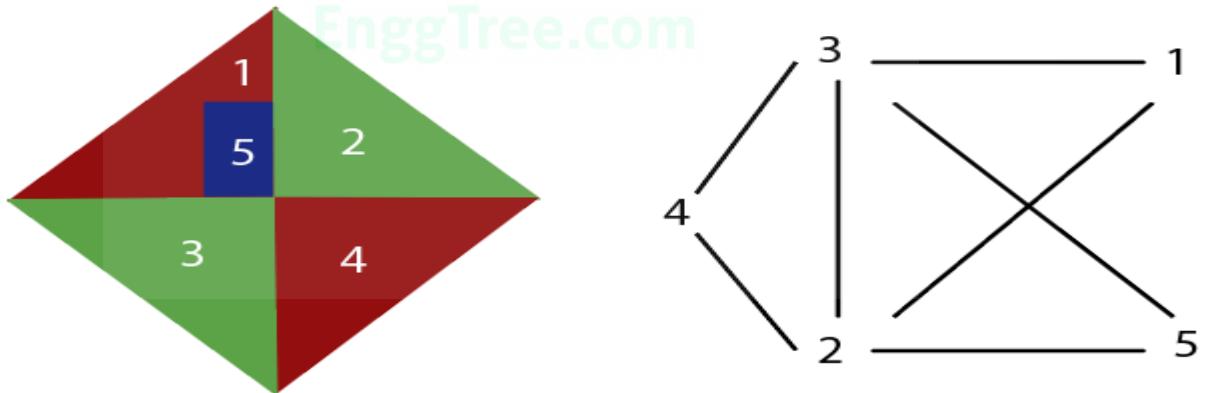
In local consistency, variables are treated as nodes, and each binary constraint is treated as an arc in the given problem. There are following local consistencies which are discussed below:

- Node Consistency: A single variable is said to be node consistent if all the values in the variable's domain satisfy the unary constraints on the variables.
- Arc Consistency: A variable is arc consistent if every value in its domain satisfies the binary constraints of the variables.
- Path Consistency: When the evaluation of a set of two variable with respect to a third variable can be extended over another variable, satisfying all the binary constraints. It is similar to arc consistency.
- k-consistency: This type of consistency is used to define the notion of stronger forms of propagation. Here, we examine the k-consistency of the variables.

CSP Problems

Constraint satisfaction includes those problems which contains some constraints while solving the problem. CSP includes the following problems:

- Graph Coloring: The problem where the constraint is that no adjacent sides can have the same color.



Graph Coloring

- Sudoku Playing: The gameplay where the constraint is that no number from 0-9 can be repeated in the same row or column.

SUDOKU

4							5	9
2	6		5				3	
			9	2				
		2	6			1		
	3	8	1	9	7			
7			3		5			
		3	4					
3				6		2	7	
5	9						6	

Puzzle

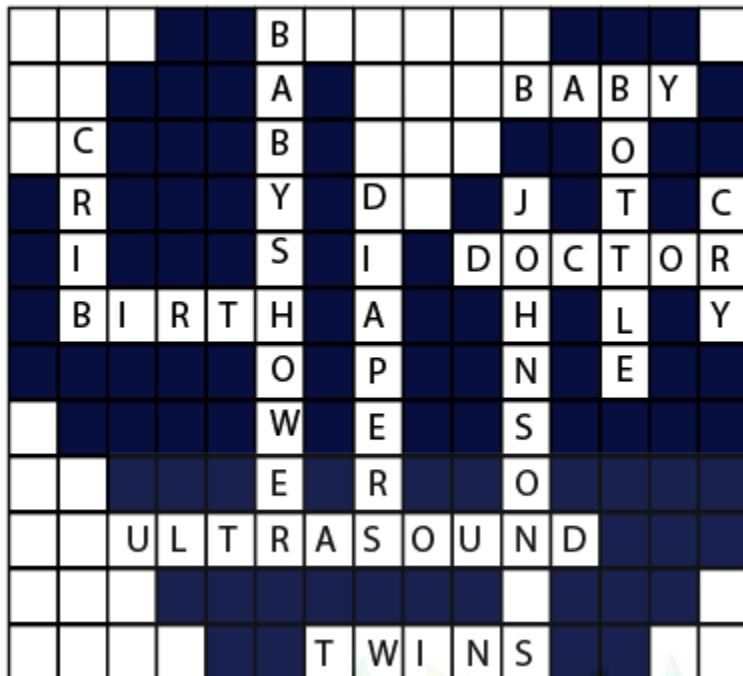
4	1	7	6	8	3	2	5	9
2	6	9	5	7	1	8	3	4
3	8	5	4	9	2	6	7	1
8	4	2	7	6	5	9	1	3
6	5	3	8	1	9	7	4	2
9	7	1	2	3	4	5	6	8
7	2	6	3	4	8	1	9	5
1	3	8	9	5	6	4	2	7
5	9	4	1	2	7	3	8	6

Solution

- n-queen problem: In n-queen problem, the constraint is that no queen should be placed either diagonally, in the same row or column.

Note: The n-queen problem is already discussed in Problem-solving in AI section.

- Crossword: In crossword problem, the constraint is that there should be the correct formation of the words, and it should be meaningful.



- Latin square Problem: In this game, the task is to search the pattern which is occurring several times in the game. They may be shuffled but will contain the same digits.

	1	1	1		1	1	1	1
1	2	3	4		1	2	3	4
1	3	4	2		1	5	4	3
1	4	2	3		1	4	3	5
1	2	4	3		1	3	2	5

Latin Squence Problem

- Cryptarithmetic Problem: This problem has one most important constraint that is, we cannot assign a different digit to the same character. All digits should contain a unique alphabet.

Cryptarithmetic Problem

Cryptarithmetic Problem is a type of constraint satisfaction problem where the game is about digits and its unique replacement either with alphabets or other symbols. In **cryptarithmetic problem**, the digits (0-9) get substituted by some possible alphabets or symbols. The task in cryptarithmetic problem is to substitute each digit with an alphabet to get the result arithmetically correct.

We can perform all the arithmetic operations on a given cryptarithmetic problem. The rules or constraints on a cryptarithmetic problem are as follows:

- There should be a unique digit to be replaced with a unique alphabet.
- The result should satisfy the predefined arithmetic rules, i.e., $2+2=4$, nothing else.
- Digits should be from **0-9** only.
- There should be only one carry forward, while performing the addition operation on a problem.
- The problem can be solved from both sides, i.e., **lefthand side (L.H.S), or righthand side (R.H.S)**

Let's understand the cryptarithmetic problem as well its constraints better with the help of an example:

- Given a cryptarithmetic problem, i.e., $\begin{array}{r} \text{SEND} \\ + \text{MORE} \\ \hline \text{MONEY} \end{array}$

$$\begin{array}{r} \text{SEND} \\ + \text{MORE} \\ \hline \text{MONEY} \end{array}$$

- In this example, add both terms **SEND** and **MORE** to bring **MONEY** as a result.

Follow the below steps to understand the given problem by breaking it into its subparts:

- Starting from the left hand side (L.H.S) , the terms are S and M. Assign a digit which could give a satisfactory result. Let's assign S->9 and M->1.

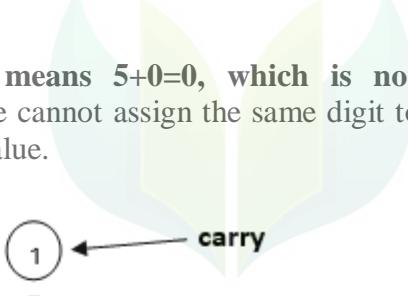
$$\begin{array}{r}
 S \\
 + M \\
 \hline
 \underline{M O}
 \end{array}
 \quad
 \begin{array}{r}
 9 \\
 + 1 \\
 \hline
 \underline{1 0}
 \end{array}$$

Hence, we get a satisfactory result by adding up the terms and got an assignment for O as O > 0 as well.

- Now, move ahead to the next terms E and O to get N as its output.

$$\begin{array}{r}
 E \\
 + O \\
 \hline
 \underline{N}
 \end{array}
 \quad
 \begin{array}{r}
 5 \\
 + 0 \\
 \hline
 \underline{5}
 \end{array}$$

Adding E and O, which means $5+0=0$, which is not possible because according to cryptarithmic constraints, we cannot assign the same digit to two letters. So, we need to think more and assign some other value.



The diagram shows a carry operation. A small circle contains the digit '1' with an arrow pointing to it labeled 'carry'. This indicates that a carry of 1 is being added to the sum of E and O.

$$\begin{array}{r}
 E \\
 + O \\
 \hline
 \underline{N}
 \end{array}
 \quad
 \begin{array}{r}
 5 \\
 + 0 \\
 \hline
 \underline{6}
 \end{array}$$

Note: When we will solve further, we will get one carry, so after applying it, the answer will be satisfied.

- Further, adding the next two terms N and R we get,

$$\begin{array}{r}
 N \\
 + R \\
 \hline
 \underline{E}
 \end{array}
 \quad
 \begin{array}{r}
 6 \\
 + 8 \\
 \hline
 \underline{1 4}
 \end{array}$$

But, we have already assigned E->5. Thus, the above result does not satisfy the values because we are getting a different value for E. So, we need to think more.

Again, after solving the whole problem, we will get a carryover on this term, so our answer will be satisfied.

$$\begin{array}{r}
 & \textcircled{1} \\
 & 6 \\
 \text{N} & \xrightarrow{\quad} \\
 + R & \xrightarrow{\quad} \\
 \hline
 \text{E} & \xrightarrow{\quad} \\
 & 15
 \end{array}$$

carry

where 1 will be carry forward to the above term

Let's move ahead.

- Again, on adding the last two terms, i.e., the rightmost terms D and E, we get Y as its result.

$$\begin{array}{r}
 \text{D} \\
 + \text{E} \\
 \hline
 \text{Y}
 \end{array}$$

7
+ 5
12

where 1 will be carry forward to the above term

- Keeping all the constraints in mind, the final resultant is as follows:

$$\begin{array}{r}
 \text{S E N D} \\
 + \text{M O R E} \\
 \hline
 \text{M O N E Y}
 \end{array}$$

- Below is the representation of the assignment of the digits to the alphabets.

S	9
E	5
N	6
D	7
M	1
O	0
R	8
Y	2

More examples of cryptarithmic problems can be:

1.

$$\begin{array}{r}
 \text{B A S E} \\
 + \text{B A L L} \\
 \hline
 \text{G A M E S}
 \end{array}$$

B	7
A	4
S	8
E	3
L	5
G	1
M	9

2.

$$\begin{array}{r}
 \text{Y O U R} \\
 + \text{Y O U} \\
 \hline
 \text{H E A R T}
 \end{array}$$

Y	9
O	4
U	2
R	6
H	1
E	0
A	3
T	8

8. Backtracking search for CSP:

Backtracking search, a form of depth-first search, is commonly used for solving CSPs. Inference can be interwoven with search.

Commutativity: CSPs are all commutative. A problem is commutative if the order of application of any given set of actions has no effect on the outcome.

Backtracking search: A depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.

Backtracking algorithm repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value.

There is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, action function, transition model, or goal test.

BACkTRACKING-SARCH keeps only a single representation of a state and alters that representation rather than creating a new ones.

```

function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)
  
function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var  $\leftarrow$  SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences  $\leftarrow$  INFERENCE(csp, var, value)
      if inferences  $\neq$  failure then
        add inferences to assignment
        result  $\leftarrow$  BACKTRACK(assignment, csp)
        if result  $\neq$  failure then
          return result
        remove {var = value} and inferences from assignment
  return failure

```

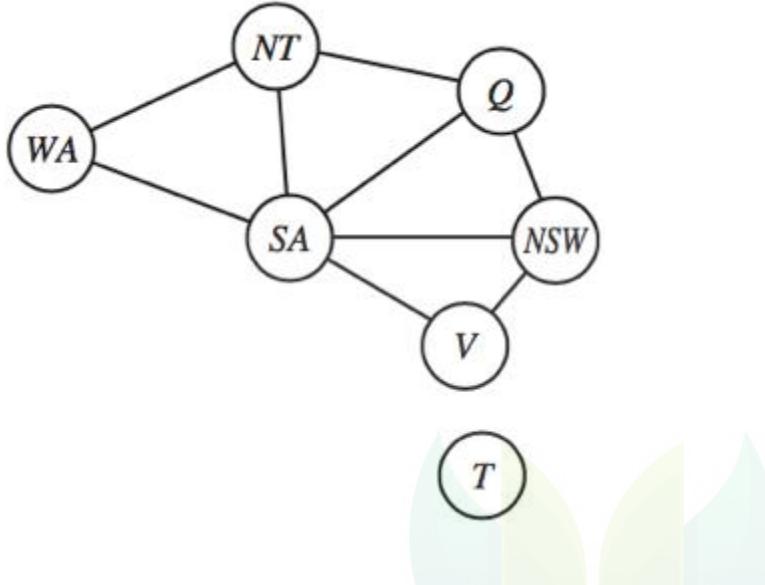
Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or *k*-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

To solve CSPs efficiently without domain-specific knowledge, address following questions:

1)function **SELECT-UNASSIGNED-VARIABLE**: which variable should be assigned next?

- function **ORDER-DOMAIN-VALUES**: in what order should its values be tried?
 2) function **INFERENCE**: what inferences should be performed at each step in the search?
 3) When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

1. Variable and value ordering



SELECT-UNASSIGNED-VARIABLE

Variable selection—fail-first

Minimum-remaining-values (MRV) heuristic: The idea of choosing the variable with the fewest “legal” value. A.k.a. “most constrained variable” or “fail-first” heuristic, it picks a variable that is most likely to cause a failure soon thereby pruning the search tree. If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables.

E.g. After the assignment for WA=red and NT=green, there is only one possible value for SA, so it makes sense to assign SA=blue next rather than assigning Q.

[Powerful guide]

Degree heuristic: The degree heuristic attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables. **[useful tie-breaker]**

e.g. SA is the variable with highest degree 5; the other variables have degree 2 or 3; T has degree 0.

ORDER-DOMAIN-VALUES

Value selection—fail-last

If we are trying to find all the solution to a problem (not just the first one), then the ordering does not matter.

Least-constraining-value heuristic: prefers the value that rules out the fewest choice for the neighboring variables in the constraint graph. (**Try to leave the maximum flexibility for subsequent variable assignments.**)

e.g. We have generated the partial assignment with WA=red and NT=green and that our next choice is for Q. Blue would be a bad choice because it eliminates the last legal value left for Q's neighbor, SA, therefore prefers red to blue.

2. Interleaving search and inference

INFERENCE

forward checking: [One of the simplest forms of inference.] Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for X.

There is no reason to do forward checking if we have already done arc consistency as a preprocessing step.

	WA	NT	Q	NSW	V	SA	T
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After WA=red	(R)	G B	R G B	R G B	R G B	G B	R G B
After Q=green	(R)	B	(G)	R	B	R G B	B
After V=blue	(R)	B	(G)	R		(B)	R G B

Figure 6.7 The progress of a map-coloring search with forward checking. WA = red is assigned first; then forward checking deletes red from the domains of the neighboring variables NT and SA. After Q = green is assigned, green is deleted from the domains of NT, SA, and NSW. After V = blue is assigned, blue is deleted from the domains of NSW and SA, leaving SA with no legal values.

Advantage: For many problems the search will be more effective if we combine the MRV heuristic with forward checking.

Disadvantage: Forward checking only makes the current variable arc-consistent, but doesn't look ahead and make all the other variables arc-consistent.

MAC (Maintaining Arc Consistency) algorithm: [More powerful than forward checking, detect this inconsistency.] After a variable X_i is assigned a value, the INFERENCE procedure calls AC-3, but instead of a queue of all arcs in the CSP, we start with only the arcs(X_j, X_i) for all X_j that are unassigned variables that are neighbors of X_i . From there, AC-3 does constraint propagation in the usual way, and if any variable has its domain reduced to the empty set, the call to AC-3 fails and we know to backtrack immediately.

Intelligent backtracking

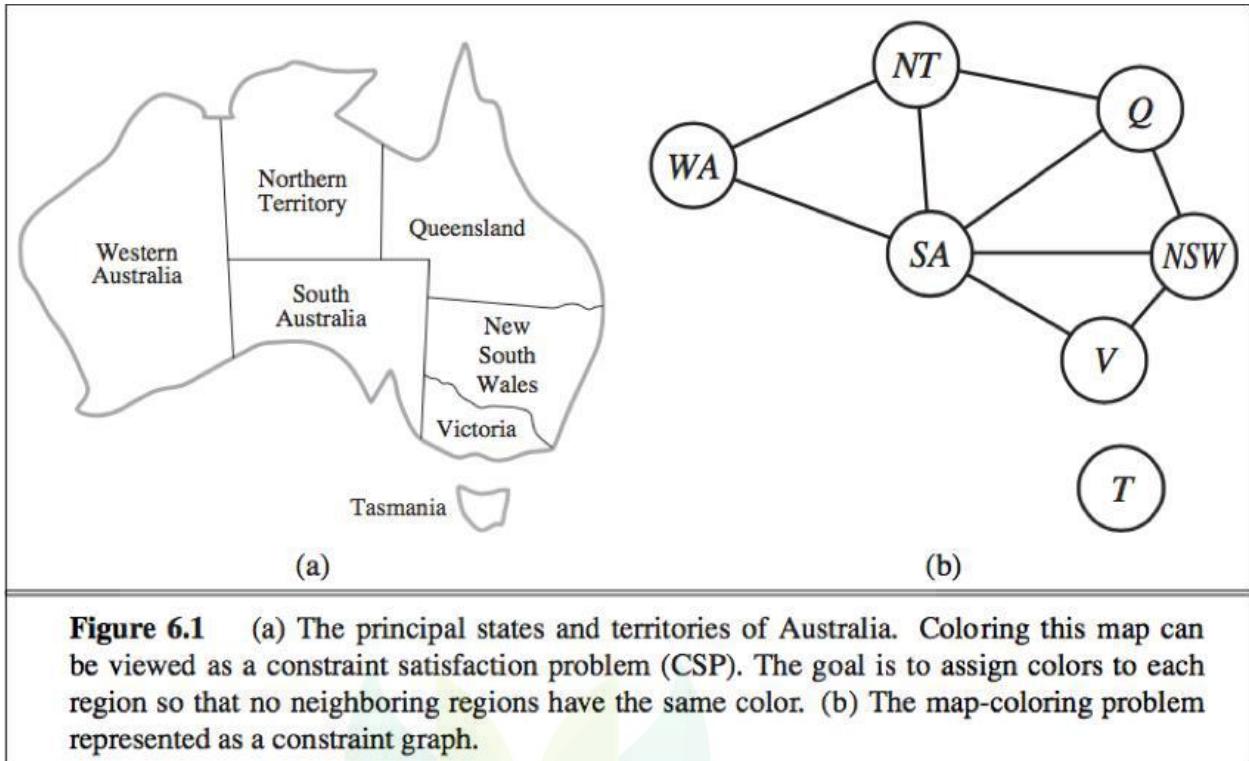


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

chronological backtracking: The BACKTRACKING-SEARCH in Fig 6.5. When a branch of the search fails, back up to the preceding variable and try a different value for it. (The most recent decision point is revisited.)

e.g.

Suppose we have generated the partial assignment {Q=red, NSW=green, V=blue, T=red}.

When we try the next variable SA, we see every value violates a constraint.

We back up to T and try a new color, it cannot resolve the problem.

Intelligent backtracking: Backtrack to a variable that was responsible for making one of the possible values of the next variable (e.g. SA) impossible.

Conflict set for a variable: A set of assignments that are in conflict with some value for that variable.

(e.g. The set {Q=red, NSW=green, V=blue} is the conflict set for SA.)

backjumping method: Backtracks to the most recent assignment in the conflict set.

(e.g. backjumping would jump over T and try a new value for V.)

Forward checking can supply the conflict set with no extra work.

Whenever forward checking based on an assignment $X=x$ deletes a value from Y 's domain, add $X=x$ to Y 's conflict set;

If the last value is deleted from Y 's domain, the assignment in the conflict set of Y are added to the conflict set of X .

In fact, every branch pruned by backjumping is also pruned by forward checking. Hence simple backjumping is redundant in a forward-checking search or in a search that uses stronger consistency checking (such as MAC).

Conflict-directed backjumping:

e.g.

consider the partial assignment which is proved to be inconsistent: {WA=red, NSW=red}.

We try T=red next and then assign NT, Q, V, SA, no assignment can work for these last 4 variables.

Eventually we run out of value to try at NT, but simple backjumping cannot work because NT doesn't have a complete conflict set of preceding variables that caused to fail.

The set {WA, NSW} is a deeper notion of the conflict set for NT, caused NT together with any subsequent variables to have no consistent solution. So the algorithm should backtrack to NSW and skip over T.

A backjumping algorithm that uses conflict sets defined in this way is called conflict-direct backjumping.

How to Compute:

When a variable's domain becomes empty, the "terminal" failure occurs, that variable has a standard conflict set.

Let X_j be the current variable, let $\text{conf}(X_j)$ be its conflict set. If every possible value for X_j fails, backjump to the most recent variable X_i in $\text{conf}(X_j)$, and set

$$\text{conf}(X_i) \leftarrow \text{conf}(X_i) \cup \text{conf}(X_j) - \{X_i\}.$$

The conflict set for an variable means, there is no solution from that variable onward, given the preceding assignment to the conflict set.

e.g.

assign WA, NSW, T, NT, Q, V, SA.

SA fails, and its conflict set is {WA, NT, Q}. (standard conflict set)

Backjump to Q, its conflict set is {NT, NSW} \cup {WA, NT, Q} - {Q} = {WA, NT, NSW}.

Backtrack to NT, its conflict set is {WA} \cup {WA, NT, NSW} - {NT} = {WA, NSW}.

Hence the algorithm backjump to NSW. (over T)

After backjumping from a contradiction, how to avoid running into the same problem again:

Constraint learning: The idea of finding a minimum set of variables from the conflict set that causes the problem. This set of variables, along with their corresponding values, is called a **no-good**. We then record the no-good, either by adding a new constraint to the CSP or by keeping a separate cache of no-goods.

Backtracking occurs when no legal assignment can be found for a variable. **Conflict-directed backjumping** backtracks directly to the source of the problem.

9. Local search for CSP:

Local search algorithms for CSPs use a complete-state formulation: the initial state assigns a value to every variable, and the search change the value of one variable at a time.

The min-conflicts heuristic: In choosing a new value for a variable, select the value that results in the minimum number of conflicts with other variables.

```

function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
           max_steps, the number of steps allowed before giving up

  current  $\leftarrow$  an initial complete assignment for csp
  for i = 1 to max_steps do
    if current is a solution for csp then return current
    var  $\leftarrow$  a randomly chosen conflicted variable from csp.VARIABLES
    value  $\leftarrow$  the value v for var that minimizes CONFLICTS(var, v, current, csp)
    set var = value in current
  return failure

```

Figure 6.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

Local search techniques in Section 4.1 can be used in local search for CSPs.

The landscape of a CSP under the mini-conflicts heuristic usually has a series of plateau.

Simulated annealing and Plateau search (i.e. allowing sideways moves to another state with the same score) can help local search find its way off the plateau. This wandering on the plateau can be directed with **tabu search**: keeping a small list of recently visited states and forbidding the algorithm to return to those states.

Constraint weighting: a technique that can help concentrate the search on the important constraints.

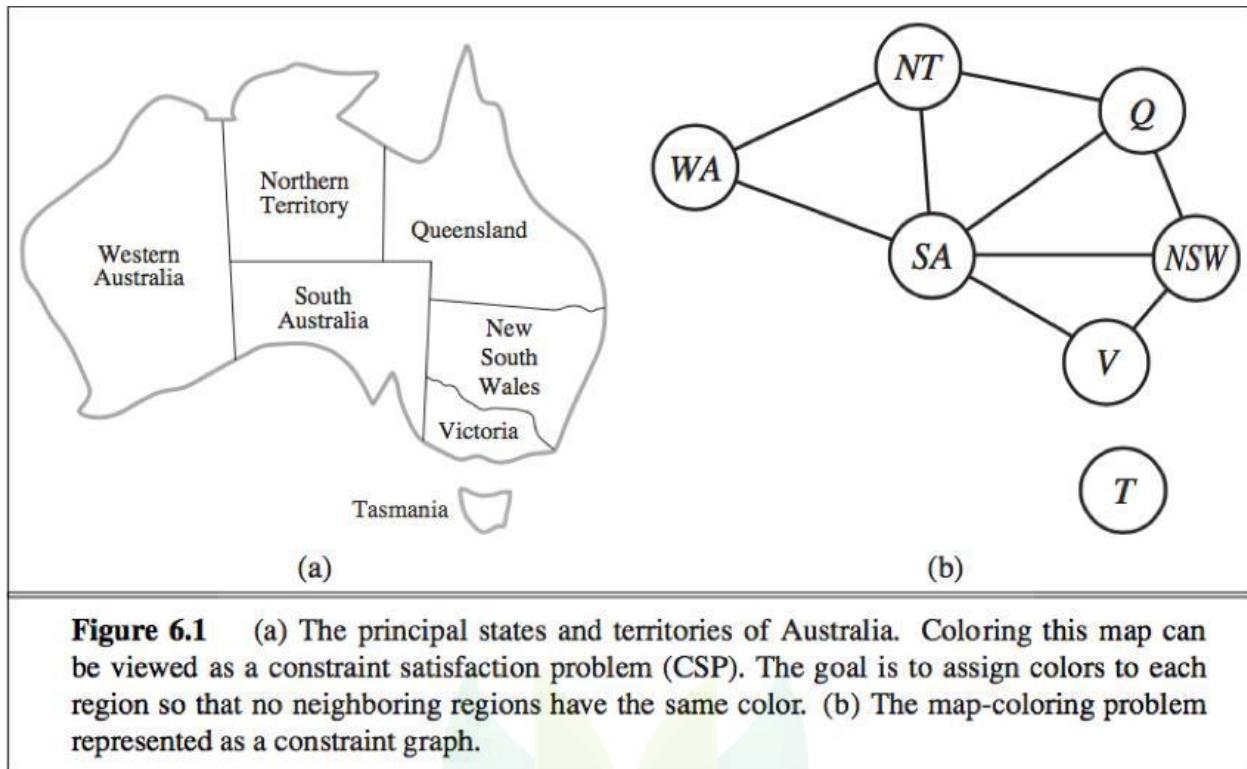
Each constraint is given a numeric weight W_i , initially all 1.

At each step, the algorithm chooses a variable/value pair to change that will result in the lowest total weight of all violated constraints.

The weights are then adjusted by incrementing the weight of each constraint that is violated by the current assignment.

Local search can be used in an online setting when the problem changes, this is particularly important in scheduling problems.

The structure of problem



1. The structure of constraint graph

The structure of the problem as represented by the constraint graph can be used to find solution quickly.

e.g. The problem can be decomposed into 2 **independent subproblems**: Coloring T and coloring the mainland.

Tree: A constraint graph is a tree when any two variables are connected by only one path.

Directed arc consistency (DAC): A CSP is defined to be directed arc-consistent under an ordering of variables X_1, X_2, \dots, X_n if and only if every X_i is arc-consistent with each X_j for $j > i$. By using DAC, any tree-structured CSP can be solved in time linear in the number of variables.

How to solve a tree-structure CSP:

Pick any variable to be the root of the tree;

Choose an ordering of the variable such that each variable appears after its parent in the tree. (topological sort)

Any tree with n nodes has $n-1$ arcs, so we can make this graph directed arc-consistent in $O(n)$ steps, each of which must compare up to d possible domain values for 2 variables, for a total time of $O(nd^2)$.

Once we have a directed arc-consistent graph, we can just march down the list of variables and choose any remaining value.

Since each link from a parent to its child is arc consistent, we won't have to backtrack, and can move linearly through the variables.

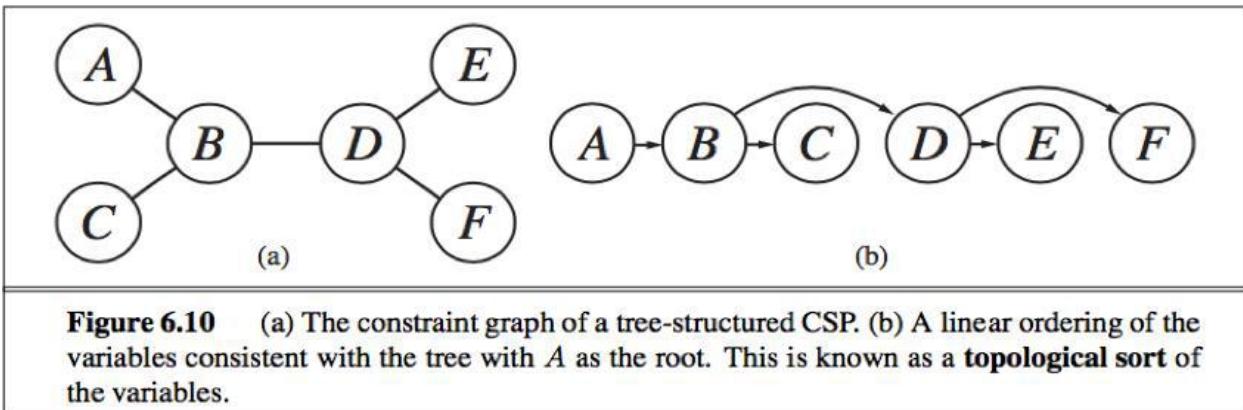


Figure 6.10 (a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with A as the root. This is known as a **topological sort** of the variables.

```

function TREE-CSP-SOLVER(csp) returns a solution, or failure
  inputs: csp, a CSP with components  $X$ ,  $D$ ,  $C$ 

   $n \leftarrow$  number of variables in  $X$ 
  assignment  $\leftarrow$  an empty assignment
  root  $\leftarrow$  any variable in  $X$ 
   $X \leftarrow \text{TOPOLOGICALSORT}(X, \text{root})$ 
  for  $j = n$  down to 2 do
    MAKE-ARC-CONSISTENT(PARENT( $X_j$ ),  $X_j$ )
    if it cannot be made consistent then return failure
  for  $i = 1$  to  $n$  do
    assignment[ $X_i$ ]  $\leftarrow$  any consistent value from  $D_i$ 
    if there is no consistent value then return failure
  return assignment

```

Figure 6.11 The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.

There are 2 primary ways to reduce more general constraint graphs to trees:

1. Based on removing nodes;

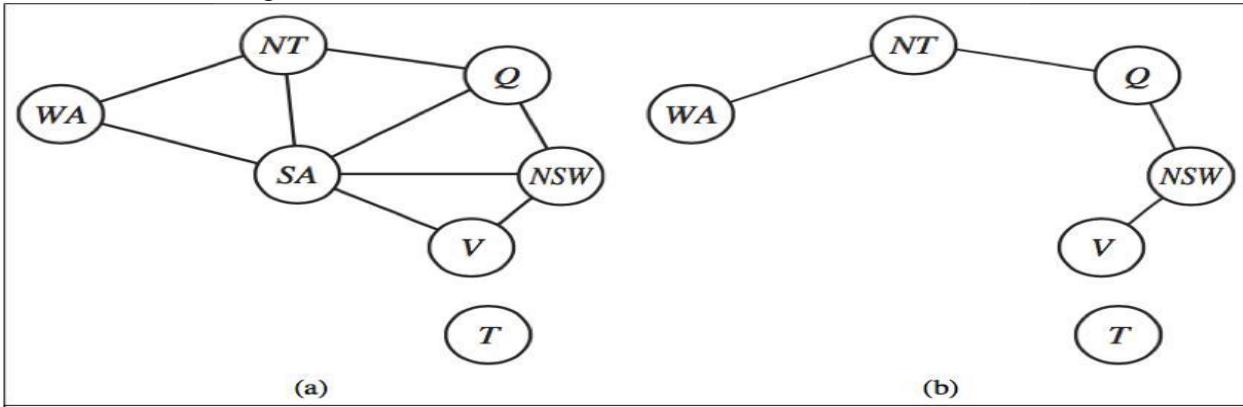


Figure 6.12 (a) The original constraint graph from Figure 6.1. (b) The constraint graph after the removal of SA .

e.g. We can delete SA from the graph by fixing a value for SA and deleting from the domains of other variables any values that are inconsistent with the value chosen for SA.

The general algorithm:

Choose a subset S of the CSP's variables such that the constraint graph becomes a tree after removal of S. S is called a **cycle cutset**.

For each possible assignment to the variables in S that satisfies all constraints on S,

(a) remove from the domain of the remaining variables any values that are inconsistent with the assignment for S, and

(b) If the remaining CSP has a solution, return it together with the assignment for S.

Time complexity: $O(d_c \cdot (n - c)d^2)$, c is the size of the cycle cut set.

Cutset conditioning: The overall algorithmic approach of efficient approximation algorithms to find the smallest cycle cutset.

2. Based on collapsing nodes together

Tree decomposition: construct a **tree decomposition** of the constraint graph into a set of connected subproblems, each subproblem is solved independently, and the resulting solutions are then combined.

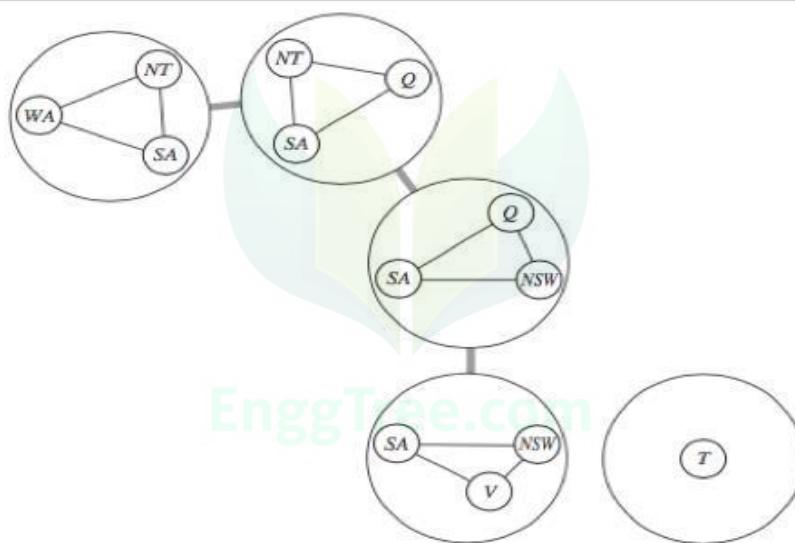


Figure 6.13 A tree decomposition of the constraint graph in Figure 6.12(a).

A tree decomposition must satisfy 3 requirements:

- Every variable in the original problem appears in at least one of the subproblems.
- If 2 variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems.
- If a variable appears in 2 subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems.

We solve each subproblem independently.

If any one has no solution, the entire problem has no solution.

If we can solve all the subproblems, then construct a global solution as follows:

First, view each subproblem as a “mega-variable” whose domain is the set of all solutions for the subproblem.

Then, solve the constraints connecting the subproblems using the efficient algorithm for trees.

A given constraint graph admits many tree decomposition;

In choosing a decomposition, the aim is to make the subproblems as small as possible.

Tree width:

The tree width of a tree decomposition of a graph is one less than the size of the largest subproblems.

The tree width of the graph itself is the minimum tree width among all its tree decompositions.

Time complexity: $O(ndw+1)$, w is the tree width of the graph.

2. The structure in the values of variables

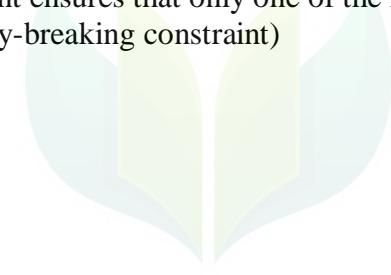
By introducing a **symmetry-breaking constraint**, we can break the **value symmetry** and reduce the search space by a factor of $n!$.

e.g.

Consider the map-coloring problems with n colors, for every consistent solution, there is actually a set of $n!$ solutions formed by permuting the color names.(value symmetry)

On the Australia map, WA, NT and SA must all have different colors, so there are $3! = 6$ ways to assign.

We can impose an arbitrary ordering constraint $NT < SA < WA$ that requires the 3 values to be in alphabetical order. This constraint ensures that only one of the $n!$ solution is possible: {NT=blue, SA=green, WA=red}. (symmetry-breaking constraint)



AL3391 ARTIFICIAL INTELLIGENCE
UNIT IV

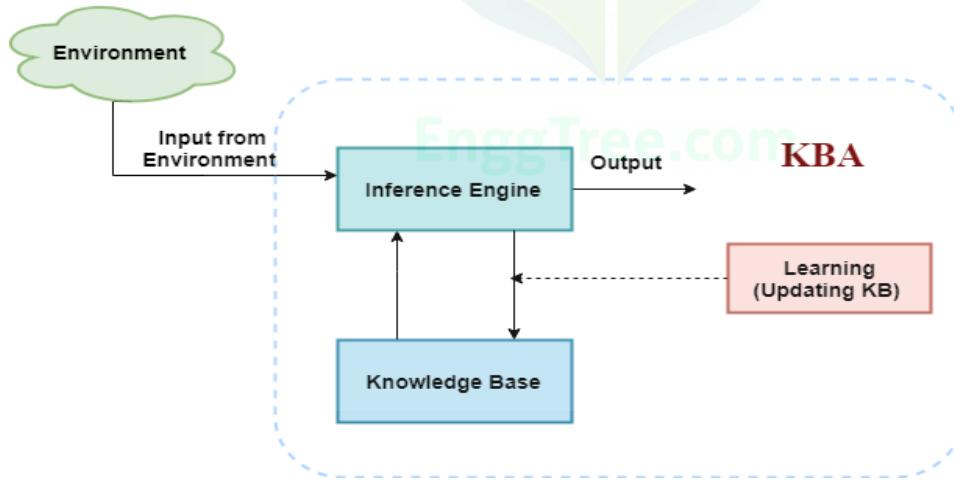
1. Knowledge-based agent in Artificial intelligence

- An intelligent agent needs **knowledge** about the real world for taking decisions and **reasoning** to act efficiently.
- Knowledge-based agents are those agents who have the capability of **maintaining an internal state of knowledge, reason over that knowledge, update their knowledge after observations and take actions. These agents can represent the world with some formal representation and act intelligently.**
- Knowledge-based agents are composed of two main parts:
 - **Knowledge-base and**
 - **Inference system.**

A knowledge-based agent must able to do the following:

- An agent should be able to represent states, actions, etc.
- An agent Should be able to incorporate new percepts
- An agent can update the internal representation of the world
- An agent can deduce the internal representation of the world
- An agent can deduce appropriate actions.

The architecture of knowledge-based agent:



The above diagram is representing a generalized architecture for a knowledge-based agent. The knowledge-based agent (KBA) take input from the environment by perceiving the environment. The input is taken by the inference engine of the agent and which also communicate with KB to decide as per the knowledge store in KB. The learning element of KBA regularly updates the KB by learning new knowledge.

Knowledge base: Knowledge-base is a central component of a knowledge-based agent, it is also known as KB. It is a collection of sentences (here 'sentence' is a technical term and it is not identical to sentence in English). These sentences are expressed in a language which is called a knowledge representation language. The Knowledge-base of KBA stores fact about the world.

Why use a knowledge base?

Knowledge-base is required for updating knowledge for an agent to learn with experiences and take action as per the knowledge.

Inference system

Inference means deriving new sentences from old. Inference system allows us to add a new sentence to the knowledge base. A sentence is a proposition about the world. Inference system applies logical rules to the KB to deduce new information.

Inference system generates new facts so that an agent can update the KB. An inference system works mainly in two rules which are given as:

- **Forward chaining**
- **Backward chaining**

Operations Performed by KBA

Following are three operations which are performed by KBA in order to show the intelligent behavior:

1. **TELL:** This operation tells the knowledge base what it perceives from the environment.
2. **ASK:** This operation asks the knowledge base what action it should perform.
3. **Perform:** It performs the selected action.

A generic knowledge-based agent:

Following is the structure outline of a generic knowledge-based agents program:

```
function KB-AGENT(percept):
    persistent: KB, a knowledge base
        t, a counter, initially 0, indicating time
    TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
    Action = ASK(KB, MAKE-ACTION-QUERY(t))
    TELL(KB, MAKE-ACTION-SENTENCE(action, t))
    t = t + 1
    return action
```

The knowledge-based agent takes percept as input and returns an action as output. The agent maintains the knowledge base, KB, and it initially has some background knowledge of the real world. It also has a counter to indicate the time for the whole process, and this counter is initialized with zero.

Each time when the function is called, it performs its three operations:

- Firstly it TELLS the KB what it perceives.
- Secondly, it asks KB what action it should take
- Third agent program TELLS the KB that which action was chosen.

The MAKE-PERCEPT-SENTENCE generates a sentence as setting that the agent perceived the given percept at the given time.

The MAKE-ACTION-QUERY generates a sentence to ask which action should be done at the current time.

MAKE-ACTION-SENTENCE generates a sentence which asserts that the chosen action was executed.

Various levels of knowledge-based agent:

A knowledge-based agent can be viewed at different levels which are given below:

1. Knowledge level

Knowledge level is the first level of knowledge-based agent, and in this level, we need to specify what the agent knows, and what the agent goals are. With these specifications, we can fix its behavior. For example, suppose an automated taxi agent needs to go from a station A to station B, and he knows the way from A to B, so this comes at the knowledge level.

2. Logical level:

are encoded into different logics. At the logical level, an encoding of knowledge into logical sentences occurs. At the logical level we can expect to the automated taxi agent to reach to the destination B.

3. Implementation level:

This is the physical representation of logic and knowledge. At the implementation level agent perform actions as per logical and knowledge level. At this level, an automated taxi agent actually implement his knowledge and logic so that he can reach to the destination.

Approaches to designing a knowledge-based agent:

There are mainly two approaches to build a knowledge-based agent:

1. **1. Declarative approach:** We can create a knowledge-based agent by initializing with an empty knowledge base and telling the agent all the sentences with which we want to start with. This approach is called Declarative approach.
2. **2. Procedural approach:** In the procedural approach, we directly encode desired behavior as a program code. Which means we just need to write a program that already encodes the desired behavior or agent.

However, in the real world, a successful agent can be built by combining both declarative and procedural approaches, and declarative knowledge can often be compiled into more efficient procedural code.

2. Propositional logic

Propositional logic (PL) is the simplest form of logic where all the statements are made by propositions. A proposition is a declarative statement which is either true or false. It is a technique of knowledge representation in logical and mathematical form.

Example:

- a) It is Sunday.
- b) The Sun rises from West (False proposition)
- c) $3+3=7$ (False proposition)
- d) 5 is a prime number.

Following are some basic facts about propositional logic:

- o Propositional logic is also called Boolean logic as it works on 0 and 1.
- o In propositional logic, we use symbolic variables to represent the logic, and we can use any symbol for a representing a proposition, such A, B, C, P, Q, R, etc.
- o Propositions can be either true or false, but it cannot be both.
- o Propositional logic consists of an object, relations or function, and **logical connectives**.
- o These connectives are also called logical operators.
- o The propositions and connectives are the basic elements of the propositional logic.
- o Connectives can be said as a logical operator which connects two sentences.
- o A proposition formula which is always true is called **tautology**, and it is also called a valid sentence.
- o A proposition formula which is always false is called **Contradiction**.
- o A proposition formula which has both true and false values is called
- o Statements which are questions, commands, or opinions are not propositions such as "**Where is Rohini**", "**How are you**", "**What is your name**", are not propositions.

Syntax of propositional logic:

The syntax of propositional logic defines the allowable sentences for the knowledge representation. There are two types of Propositions:

- a. **Atomic Propositions**
- b. **Compound propositions**

EnggTree.com

- o **Atomic Proposition:** Atomic propositions are the simple propositions. It consists of a single proposition symbol. These are the sentences which must be either true or false.

Example:

- a) $2+2$ is **4**, it is an atomic proposition as it is a **true** fact.
 - b) "The Sun is **cold**" is also a proposition as it is a **false** fact.
- o **Compound proposition:** Compound propositions are constructed by combining simpler or atomic propositions, using parenthesis and logical connectives.

Example:

- a) "It is raining today, and street is wet."
- b) "Ankit is a doctor, and his clinic is in Mumbai."

Logical Connectives:

Logical connectives are used to connect two simpler propositions or representing a sentence logically. We can create compound propositions with the help of logical connectives. There are mainly five connectives, which are given as follows:

1. **Negation:** A sentence such as $\neg P$ is called negation of P. A literal can be either Positive literal or negative literal.
2. **Conjunction:** A sentence which has \wedge connective such as, $P \wedge Q$ is called a conjunction.
Example: Rohan is intelligent and hardworking. It can be written as,
P= Rohan is intelligent,
Q= Rohan is hardworking. $\rightarrow P \wedge Q$.
3. **Disjunction:** A sentence which has \vee connective, such as $P \vee Q$. is called disjunction, where P and Q are the propositions.
Example: "Ritika is a doctor or Engineer",
Here P= Ritika is Doctor. Q= Ritika is Doctor, so we can write it as $P \vee Q$.
4. **Implication:** A sentence such as $P \rightarrow Q$, is called an implication. Implications are also known as if-then rules. It can be represented as
If it is raining, then the street is wet.
Let P= It is raining, and Q= Street is wet, so it is represented as $P \rightarrow Q$
5. **Biconditional:** A sentence such as $P \Leftrightarrow Q$ is a Biconditional sentence, example **If I am breathing, then I am alive**
P= I am breathing, Q= I am alive, it can be represented as $P \Leftrightarrow Q$.

Following is the summarized table for Propositional Logic Connectives:

Connective symbols	Word	Technical term	Example
\wedge	AND	Conjunction	$A \wedge B$
\vee	OR	Disjunction	$A \vee B$
\rightarrow	Implies	Implication	$A \rightarrow B$
\Leftrightarrow	If and only if	Biconditional	$A \Leftrightarrow B$
\neg or \sim	Not	Negation	$\neg A$ or $\sim B$

Truth Table:

In propositional logic, we need to know the truth values of propositions in all possible scenarios. We can combine all the possible combination with logical connectives, and the representation of these combinations in a tabular format is

called **Truth table**. Following are the truth table for all logical connectives:

For Negation:

P	$\neg P$
True	False
False	True

For Conjunction:

P	Q	$P \wedge Q$
True	True	True
True	False	False
False	True	False
False	False	False

For disjunction:

P	Q	$P \vee Q$
True	True	True
False	True	True
True	False	True
False	False	False

For Implication:

P	Q	$P \rightarrow Q$
True	True	True
True	False	False
False	True	True
False	False	True

For Biconditional:

P	Q	$P \leftrightarrow Q$
True	True	True
True	False	False
False	True	False
False	False	True

Truth table with three propositions:

We can build a proposition composing three propositions P, Q, and R. This truth table is made-up of 8^n Tuples as we have taken three proposition symbols.

P	Q	R	$\neg R$	$P \vee Q$	$P \vee Q \rightarrow \neg R$
True	True	True	False	True	False
True	True	False	True	True	True
True	False	True	False	True	False
True	False	False	True	True	True
False	True	True	False	True	False
False	True	False	True	True	True
False	False	True	False	False	True
False	False	False	True	False	True

Precedence of connectives:

Just like arithmetic operators, there is a precedence order for propositional connectors or logical operators. This order should be followed while evaluating a propositional problem. Following is the list of the precedence order for operators:

Precedence	Operators
First Precedence	Parenthesis
Second Precedence	Negation
Third Precedence	Conjunction(AND)
Fourth Precedence	Disjunction(OR)
Fifth Precedence	Implication
Six Precedence	Biconditional

Note: For better understanding use parenthesis to make sure of the correct interpretations. Such as $\neg R \vee Q$, It can be interpreted as $(\neg R) \vee Q$

Logical equivalence:

Logical equivalence is one of the features of propositional logic. Two propositions are said to be logically equivalent if and only if the columns in the truth table are identical to each other.

Let's take two propositions A and B, so for logical equivalence, we can write it as $A \Leftrightarrow B$. In below truth table we can see that column for $\neg A \vee B$ and $A \rightarrow B$, are identical hence A is Equivalent to B

A	B	$\neg A$	$\neg A \vee B$	$A \rightarrow B$
T	T	F	T	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

Properties of Operators:

- **Commutativity:**
 - $P \wedge Q = Q \wedge P$, or
 - $P \vee Q = Q \vee P$.
- **Associativity:**
 - $(P \wedge Q) \wedge R = P \wedge (Q \wedge R)$,
 - $(P \vee Q) \vee R = P \vee (Q \vee R)$
- **Identity element:**
 - $P \wedge \text{True} = P$,
 - $P \vee \text{True} = \text{True}$.
- **Distributive:**
 - $P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$.
 - $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$.
- **DE Morgan's Law:**
 - $\neg(P \wedge Q) = (\neg P) \vee (\neg Q)$
 - $\neg(P \vee Q) = (\neg P) \wedge (\neg Q)$.
- **Double-negation elimination:**
 - $\neg(\neg P) = P$.

Limitations of Propositional logic:

- We cannot represent relations like ALL, some, or none with propositional logic. Example:
 - a. **All the girls are intelligent.**
 - b. **Some apples are sweet.**

Propositional logic has limited expressive power.

In propositional logic, we cannot describe statements in terms of their properties or logical relationships.

3. Propositional theorem proving

Theorem proving: Applying rules of inference directly to the sentences in our knowledge base to construct a proof of the desired sentence without consulting models.

Inference rules are patterns of sound inference that can be used to find proofs. The **resolution** rule yields a complete inference algorithm for knowledge bases that are expressed in **conjunctive normal form**. **Forward chaining** and **backward chaining** are very natural reasoning algorithms for knowledge bases in **Horn form**.

Logical equivalence:

Two sentences α and β are logically equivalent if they are true in the same set of models. (write as $\alpha \equiv \beta$). Also: $\alpha \equiv \beta$ if and only if $\alpha \models \beta$ and $\beta \models \alpha$.

$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$	commutativity of \wedge
$(\alpha \vee \beta) \equiv (\beta \vee \alpha)$	commutativity of \vee
$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$	associativity of \wedge
$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$	associativity of \vee
$\neg(\neg\alpha) \equiv \alpha$	double-negation elimination
$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha)$	contraposition
$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta)$	implication elimination
$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$	biconditional elimination
$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$	De Morgan
$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$	De Morgan
$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	distributivity of \wedge over \vee
$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	distributivity of \vee over \wedge

Figure 7.11 Standard logical equivalences. The symbols α , β , and γ stand for arbitrary sentences of propositional logic.

Validity: A sentence is valid if it is true in all models.

Valid sentences are also known as **tautologies**—they are necessarily true. Every valid sentence is logically equivalent to *True*.

The **deduction theorem**: For any sentence α and β , $\alpha \vDash \beta$ if and only if the sentence $(\alpha \Rightarrow \beta)$ is valid.

Satisfiability: A sentence is satisfiable if it is true in, or satisfied by, some model. Satisfiability can be checked by enumerating the possible models until one is found that satisfies the sentence.

The **SAT** problem: The problem of determining the satisfiability of sentences in propositional logic.

Validity and satisfiability are connected:

α is valid iff $\neg\alpha$ is unsatisfiable;

α is satisfiable iff $\neg\alpha$ is not valid;

$\alpha \vDash \beta$ if and only if the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable.

Proving β from α by checking the unsatisfiability of $(\alpha \wedge \neg\beta)$ corresponds to **proof by refutation / proof by contradiction**.

Inference and proofs

Inferences rules (such as Modus Ponens and And-Elimination) can be applied to derive to a **proof**.

• **Modus Ponens:**

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}.$$

Whenever any sentences of the form $\alpha \Rightarrow \beta$ and α are given, then the sentence β can be inferred.

• **And-Elimination:**

$$\frac{\alpha \wedge \beta}{\alpha}.$$

From a conjunction, any of the conjuncts can be inferred.

• All of **logical equivalence** (in Figure 7.11) can be used as inference rules.

$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$	commutativity of \wedge
$(\alpha \vee \beta) \equiv (\beta \vee \alpha)$	commutativity of \vee
$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$	associativity of \wedge
$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$	associativity of \vee
$\neg(\neg\alpha) \equiv \alpha$	double-negation elimination
$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha)$	contraposition
$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta)$	implication elimination
$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$	biconditional elimination
$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$	De Morgan
$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$	De Morgan
$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	distributivity of \wedge over \vee
$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	distributivity of \vee over \wedge

Figure 7.11 Standard logical equivalences. The symbols α , β , and γ stand for arbitrary sentences of propositional logic.

e.g. The equivalence for biconditional elimination yields 2 inference rules:

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad \text{and} \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}.$$

• De Morgan's rule

We can apply any of the search algorithms in Chapter 3 to find a sequence of steps that constitutes a proof. We just need to define a proof problem as follows:

- INITIAL STATE: the initial knowledge base;
- ACTION: the set of actions consists of all the inference rules applied to all the sentences that match the top half of the inference rule.
- RESULT: the result of an action is to add the sentence in the bottom half of the inference rule.
- GOAL: the goal is a state that contains the sentence we are trying to prove.

In many practical cases, finding a proof can be more efficient than enumerating models, because the proof can ignore irrelevant propositions, no matter how many of them they are.

Monotonicity: A property of logical system, says that the set of entailed sentences can only increase as information is added to the knowledge base.

For any sentences α and β ,

If $KB \models \alpha$ then $KB \wedge \beta \models \alpha$.

Monotonicity means that inference rules can be applied whenever suitable premises are found in the knowledge base, what else in the knowledge base cannot invalidate any conclusion already inferred.

Proof by resolution

Resolution: An inference rule that yields a complete inference algorithm when coupled with any complete search algorithm.

Clause: A disjunction of literals. (e.g. $A \vee B$). A single literal can be viewed as a **unit clause** (a disjunction of one literal).

Unit resolution inference rule: Takes a clause and a literal and produces a new clause.

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \quad m}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k}$$

where each ℓ is a literal, ℓ_i and m are complementary literals (one is the negation of the other).

Full resolution rule: Takes 2 clauses and produces a new clause.

$$\frac{l_1 \vee \dots \vee l_k, \quad m_1 \vee \dots \vee m_n}{l_1 \vee \dots \vee l_{i-1} \vee l_{i+1} \vee \dots \vee l_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee v m_n}$$

where l_i and m_j are complementary literals.

Notice: The resulting clause should contain only one copy of each literal. The removal of multiple copies of literal is called **factoring**.

e.g. resolve($A \vee B$) with ($A \vee \neg B$), obtain($A \vee A$) and reduce it to just A .

The resolution rule is sound and complete.

Conjunctive normal form

Conjunctive normal form (CNF): A sentence expressed as a conjunction of clauses is said to be in CNF. Every sentence of propositional logic is logically equivalent to a conjunction of clauses, after converting a sentence into CNF, it can be used as input to a resolution procedure.

A resolution algorithm

```

function PL-RESOLUTION( $KB, \alpha$ ) returns true or false
  inputs:  $KB$ , the knowledge base, a sentence in propositional logic
            $\alpha$ , the query, a sentence in propositional logic

   $clauses \leftarrow$  the set of clauses in the CNF representation of  $KB \wedge \neg\alpha$ 
   $new \leftarrow \{ \}$ 
  loop do
    for each pair of clauses  $C_i, C_j$  in  $clauses$  do
       $resolvents \leftarrow$  PL-RESOLVE( $C_i, C_j$ )
      if  $resolvents$  contains the empty clause then return true
       $new \leftarrow new \cup resolvents$ 
    if  $new \subseteq clauses$  then return false
     $clauses \leftarrow clauses \cup new$ 
  
```

Figure 7.12 A simple resolution algorithm for propositional logic. The function PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

e.g.

$$KB = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$$

$$\alpha = \neg P_{1,2}$$

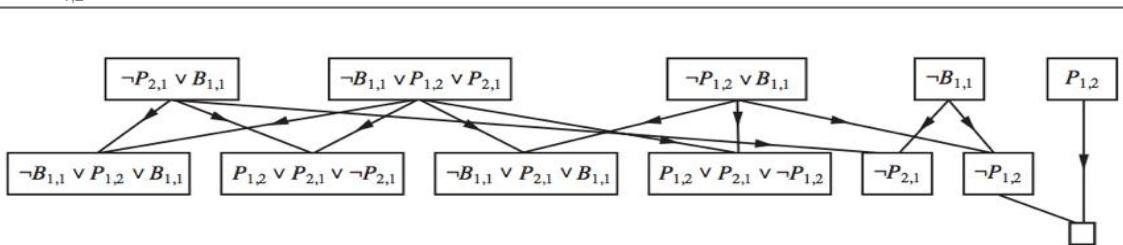


Figure 7.13 Partial application of PL-RESOLUTION to a simple inference in the wumpus world. $\neg P_{1,2}$ is shown to follow from the first four clauses in the top row.

Notice: Any clause in which two complementary literals appear can be discarded, because it is always equivalent to *True*.

e.g. $B_{1,1} \vee \neg B_{1,1} \vee P_{1,2} = \text{True}$ $\vee P_{1,2} = \text{True}$.

PL-RESOLUTION is complete.

Horn clauses and definite clauses

<i>CNF Sentence</i>	\rightarrow	<i>Clause₁</i> $\wedge \dots \wedge$ <i>Clause_n</i>
<i>Clause</i>	\rightarrow	<i>Literal₁</i> $\vee \dots \vee$ <i>Literal_m</i>
<i>Literal</i>	\rightarrow	<i>Symbol</i> $ \neg \text{Symbol}$
<i>Symbol</i>	\rightarrow	<i>P</i> $ \text{Q}$ $ \text{R}$ $ \dots$
<i>Horn Clause Form</i>	\rightarrow	<i>Definite Clause Form</i> $ \text{Goal Clause Form}$
<i>Definite Clause Form</i>	\rightarrow	$(\text{Symbol}_1 \wedge \dots \wedge \text{Symbol}_l) \Rightarrow \text{Symbol}$
<i>Goal Clause Form</i>	\rightarrow	$(\text{Symbol}_1 \wedge \dots \wedge \text{Symbol}_l) \Rightarrow \text{False}$

Figure 7.14 A grammar for conjunctive normal form, Horn clauses, and definite clauses. A clause such as $A \wedge B \Rightarrow C$ is still a definite clause when it is written as $\neg A \vee \neg B \vee C$, but only the former is considered the canonical form for definite clauses. One more class is the k -CNF sentence, which is a CNF sentence where each clause has at most k literals.

Definite clause: A disjunction of literals of which exactly one is positive. (e.g. $\neg L_{1,1} \vee \neg \text{Breeze} \vee B_{1,1}$)

Every definite clause can be written as an implication, whose premise is a conjunction of positive literals and whose conclusion is a single positive literal.

Horn clause: A disjunction of literals of which at most one is positive. (All definite clauses are Horn clauses.)

In Horn form, the premise is called the **body** and the conclusion is called the **head**.

A sentence consisting of a single positive literal is called a **fact**, it too can be written in implication form.

Horn clause are closed under resolution: if you resolve 2 horn clauses, you get back a horn clause.

Inference with horn clauses can be done through the **forward-chaining** and **backward-chaining** algorithms.

Deciding entailment with Horn clauses can be done in time that is linear in the size of the knowledge base.

Goal clause: A clause with no positive literals.

Forward and backward chaining

forward-chaining algorithm: PL-FC-ENTAILS?(KB, q) (runs in linear time)

Forward chaining is **sound** and **complete**.

```

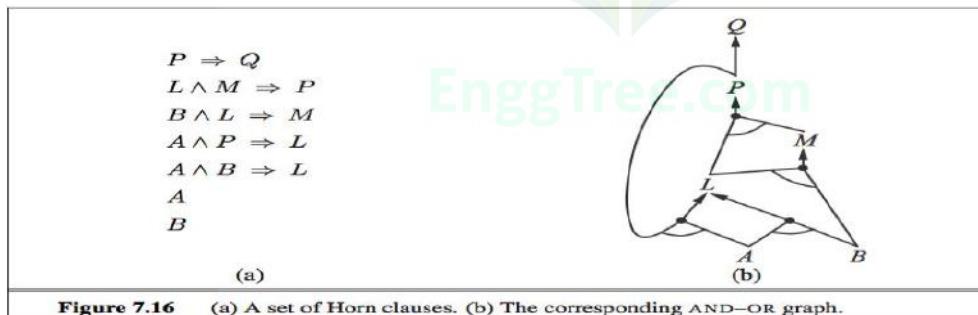
function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional definite clauses
            q, the query, a proposition symbol
  count  $\leftarrow$  a table, where count[c] is the number of symbols in c's premise
  inferred  $\leftarrow$  a table, where inferred[s] is initially false for all symbols
  agenda  $\leftarrow$  a queue of symbols, initially symbols known to be true in KB

  while agenda is not empty do
    p  $\leftarrow$  POP(agenda)
    if p = q then return true
    if inferred[p] = false then
      inferred[p]  $\leftarrow$  true
      for each clause c in KB where p is in c.PREMISE do
        decrement count[c]
        if count[c] = 0 then add c.CONCLUSION to agenda
  return false

```

Figure 7.15 The forward-chaining algorithm for propositional logic. The *agenda* keeps track of symbols known to be true but not yet “processed.” The *count* table keeps track of how many premises of each implication are as yet unknown. Whenever a new symbol *p* from the agenda is processed, the count is reduced by one for each implication in whose premise *p* appears (easily identified in constant time with appropriate indexing.) If a count reaches zero, all the premises of the implication are known, so its conclusion can be added to the agenda. Finally, we need to keep track of which symbols have been processed; a symbol that is already in the set of inferred symbols need not be added to the agenda again. This avoids redundant work and prevents loops caused by implications such as $P \Rightarrow Q$ and $Q \Rightarrow P$.

e.g. A knowledge base of horn clauses with A and B as known facts.



Fixed point: The algorithm reaches a fixed point where no new inferences are possible.

Data-driven reasoning: Reasoning in which the focus of attention starts with the known data. It can be used within an agent to derive conclusions from incoming percept, often without a specific query in mind. (forward chaining is an example)

Backward-chaining algorithm: works backward from the query.

If the query *q* is known to be true, no work is needed;

Otherwise the algorithm finds those implications in the KB whose conclusion is *q*. If all the premises of one of those implications can be proved true (by backward chaining), then *q* is true. (runs in linear time)

in the corresponding AND-OR graph: it works back down the graph until it reaches a set of known facts.
(Backward-chaining algorithm is essentially identical to the AND-OR-GRAFH-SEARCH algorithm.)

Backward-chaining is a form of **goal-directed reasoning**.

4. Propositional model checking

The set of possible models, given a fixed propositional vocabulary, is finite, so entailment can be checked by enumerating models. Efficient **model-checking** inference algorithms for propositional logic include backtracking and local search methods and can often solve large problems quickly.

2 families of algorithms for the SAT problem based on model checking:

- a. based on backtracking
- b. based on local hill-climbing search

1. A complete backtracking algorithm

David-Putnam algorithm (DPLL):

```

function DPLL-SATISFIABLE?(s) returns true or false
  inputs: s, a sentence in propositional logic

  clauses  $\leftarrow$  the set of clauses in the CNF representation of s
  symbols  $\leftarrow$  a list of the proposition symbols in s
  return DPLL(clauses, symbols, { })

function DPLL(clauses, symbols, model) returns true or false
  if every clause in clauses is true in model then return true
  if some clause in clauses is false in model then return false
  P, value  $\leftarrow$  FIND-PURE-SYMBOL(symbols, clauses, model)
  if P is non-null then return DPLL(clauses, symbols - P, model  $\cup$  {P=value})
  P, value  $\leftarrow$  FIND-UNIT-CLAUSE(clauses, model)
  if P is non-null then return DPLL(clauses, symbols - P, model  $\cup$  {P=value})
  P  $\leftarrow$  FIRST(symbols); rest  $\leftarrow$  REST(symbols)
  return DPLL(clauses, rest, model  $\cup$  {P=true}) or
         DPLL(clauses, rest, model  $\cup$  {P=false}))
```

Figure 7.17 The DPLL algorithm for checking satisfiability of a sentence in propositional logic. The ideas behind FIND-PURE-SYMBOL and FIND-UNIT-CLAUSE are described in the text; each returns a symbol (or null) and the truth value to assign to that symbol. Like TT-ENTAILS?, DPLL operates over partial models.

DPLL embodies 3 improvements over the scheme of TT-ENTAILS?: Early termination, pure symbol heuristic, unit clause heuristic.

Tricks that enable SAT solvers to scale up to large problems: Component analysis, variable and value ordering, intelligent backtracking, random restarts, clever indexing.

Local search algorithms

Local search algorithms can be applied directly to the SAT problem, provided that choose the right evaluation function. (We can choose an evaluation function that counts the number of unsatisfied clauses.)

These algorithms take steps in the space of complete assignments, flipping the truth value of one symbol at a time. The space usually contains many local minima, to escape from which various forms of randomness are required.

Local search methods such as WALKSAT can be used to find solutions. Such algorithm are sound but not complete. **WALKSAT:** one of the simplest and most effective algorithms.

```

function WALKSAT(clauses, p, max_flips) returns a satisfying model or failure
  inputs: clauses, a set of clauses in propositional logic
    p, the probability of choosing to do a “random walk” move, typically around 0.5
    max_flips, number of flips allowed before giving up

  model  $\leftarrow$  a random assignment of true/false to the symbols in clauses
  for i = 1 to max_flips do
    if model satisfies clauses then return model
    clause  $\leftarrow$  a randomly selected clause from clauses that is false in model
    with probability p flip the value in model of a randomly selected symbol from clause
    else flip whichever symbol in clause maximizes the number of satisfied clauses
  return failure

```

Figure 7.18 The WALKSAT algorithm for checking satisfiability by randomly flipping the values of variables. Many versions of the algorithm exist.

On every iteration, the algorithm picks an unsatisfied clause, and chooses randomly between 2 ways to pick a symbol to flip:

Either a. a “min-conflicts” step that minimizes the number of unsatisfied clauses in the new state;
Or b. a “random walk” step that picks the symbol randomly.

When the algorithm returns a model, the input sentence is indeed satisfiable;

When the algorithm returns failure, there are 2 possible causes:

Either a. The sentence is unsatisfiable;

Or b. We need to give the algorithm more time.

If we set *max_flips*= ∞ , *p*>0, the algorithm will:

Either a. eventually return a model if one exists

Or b. never terminate if the sentence is unsatisfiable.

Thus WALKSAT is useful when we expect a solution to exist, but cannot always detect unsatisfiability.

The landscape of random SAT problems

Underconstrained problem: When we look at satisfiability problems in CNF, an underconstrained problem is one with relatively few clauses constraining the variables.

An **overconstrained problem** has many clauses relative to the number of variables and is likely to have no solutions.

The notation $CNF_k(m, n)$ denotes a *k*-CNF sentence with *m* clauses and *n* symbols. (with *n* variables and *k* literals per clause).

Given a source of random sentences, where the clauses are chosen uniformly, independently and without replacement from among all clauses with *k* different literals, which are positive or negative at random.

Hardness: problems right at the threshold > overconstrained problems > underconstrained problems

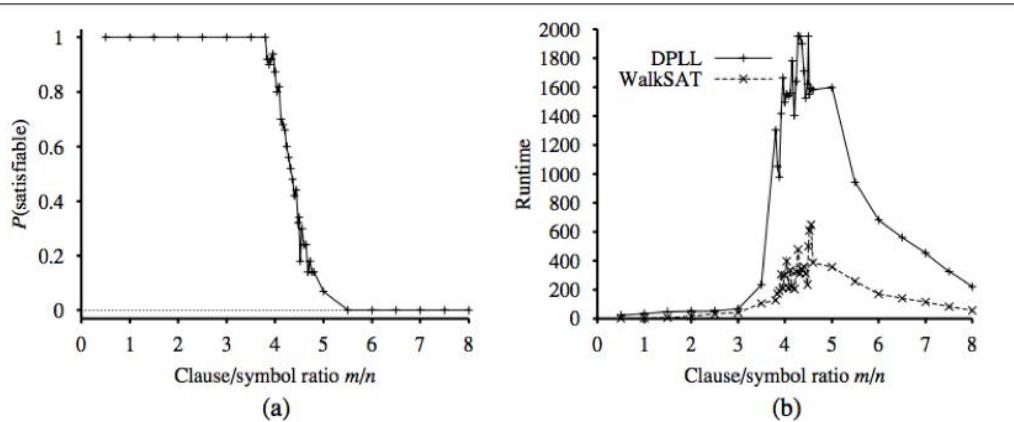


Figure 7.19 (a) Graph showing the probability that a random 3-CNF sentence with $n = 50$ symbols is satisfiable, as a function of the clause/symbol ratio m/n . (b) Graph of the median run time (measured in number of recursive calls to DPLL, a good proxy) on random 3-CNF sentences. The most difficult problems have a clause/symbol ratio of about 4.3.

Satisfiability threshold conjecture: A theory says that for every $k \geq 3$, there is a threshold ratio r_k , such that as n goes to infinity, the probability that $\text{CNF}_k(n, rn)$ is satisfiable becomes 1 for all values of r below the threshold, and 0 for all values above. (remains unproven)

5. Agents based on propositional logic

1. The current state of the world

We can associate proposition with timestamp to avoid contradiction.

e.g. $\neg \text{Stench}^3$, Stench^4

fluent: refer an aspect of the world that changes. (E.g. $L_{x,y}^t$)

atemporal variables: Symbols associated with permanent aspects of the world do not need a time superscript.
Effect axioms: specify the outcome of an action at the next time step.

Frame problem: some information lost because the effect axioms fails to state what remains unchanged as the result of an action.

Solution: add frame axioms explicitly asserting all the propositions that remain the same.

Representation frame problem: The proliferation of frame axioms is inefficient, the set of frame axioms will be $O(mn)$ in a world with m different actions and n fluents.

Solution: because the world exhibits **locality** (for humans each action typically changes no more than some number k of those fluents.) Define the transition model with a set of axioms of size $O(mk)$ rather than size $O(mn)$.

Inferential frame problem: The problem of projecting forward the results of a t step plan of action in time $O(kt)$ rather than $O(nt)$.

Solution: change one's focus from writing axioms about actions to writing axioms about fluents.

For each fluent F , we will have an axiom that defines the truth value of F^{t+1} in terms of fluents at time t and the action that may have occurred at time t .

The truth value of F^{t+1} can be set in one of 2 ways:

Either a. The action at time t cause F to be true at $t+1$

Or b. F was already true at time t and the action at time t does not cause it to be false.

An axiom of this form is called a **successor-state axiom** and has this schema:

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t) .$$

Qualification problem: specifying all unusual exceptions that could cause the action to fail.

2. A hybrid agent

Hybrid agent: combines the ability to deduce various aspect of the state of the world with condition-action rules, and with problem-solving algorithms.

The agent maintains and update KB as a current plan.

The initial KB contains the atemporal axioms. (don't depend on t)

At each time step, the new percept sentence is added along with all the axioms that depend on t (such as the successor-state axioms).

Then the agent use logical inference by ASKING questions of the KB (to work out which squares are safe and which have yet to be visited).

The main body of the agent program constructs a plan based on a decreasing priority of goals:

1. If there is a glitter, construct a plan to grab the gold, follow a route back to the initial location and climb out of the cave;
2. Otherwise if there is no current plan, plan a route (with A* search) to the closest safe square unvisited yet, making sure the route goes through only safe squares;
3. If there are no safe squares to explore, if still has an arrow, try to make a safe square by shooting at one of the possible wumpus locations.
4. If this fails, look for a square to explore that is not provably unsafe.
5. If there is no such square, the mission is impossible, then retreat to the initial location and climb out of the cave.

```

function HYBRID-WUMPUS-AGENT(percept) returns an action
  inputs: percept, a list, [stench,breeze,glitter,bump,scream]
  persistent: KB, a knowledge base, initially the atemporal “wumpus physics”
    t, a counter, initially 0, indicating time
    plan, an action sequence, initially empty

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  TELL the KB the temporal “physics” sentences for time t
  safe  $\leftarrow \{[x, y] : \text{ASK}(KB, OK_{x,y}^t) = \text{true}\}$ 
  if ASK(KB, Glittert) = true then
    plan  $\leftarrow [\text{Grab}] + \text{PLAN-ROUTE}(\text{current}, \{[1,1]\}, \text{safe}) + [\text{Climb}]$ 
  if plan is empty then
    unvisited  $\leftarrow \{[x, y] : \text{ASK}(KB, L_{x,y}^{t'}) = \text{false} \text{ for all } t' \leq t\}$ 
    plan  $\leftarrow \text{PLAN-ROUTE}(\text{current}, \text{unvisited} \cap \text{safe}, \text{safe})$ 
  if plan is empty and ASK(KB, HaveArrowt) = true then
    possible_wumpus  $\leftarrow \{[x, y] : \text{ASK}(KB, \neg W_{x,y}) = \text{false}\}$ 
    plan  $\leftarrow \text{PLAN-SHOT}(\text{current}, \text{possible\_wumpus}, \text{safe})$ 
  if plan is empty then // no choice but to take a risk
    not_unsafe  $\leftarrow \{[x, y] : \text{ASK}(KB, \neg OK_{x,y}^t) = \text{false}\}$ 
    plan  $\leftarrow \text{PLAN-ROUTE}(\text{current}, \text{unvisited} \cap \text{not\_unsafe}, \text{safe})$ 
  if plan is empty then
    plan  $\leftarrow \text{PLAN-ROUTE}(\text{current}, \{[1, 1]\}, \text{safe}) + [\text{Climb}]$ 
    action  $\leftarrow \text{POP}(\text{plan})$ 
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t  $\leftarrow t + 1$ 
  return action

function PLAN-ROUTE(current,goals,allowed) returns an action sequence
  inputs: current, the agent's current position
    goals, a set of squares; try to plan a route to one of them
    allowed, a set of squares that can form part of the route

  problem  $\leftarrow \text{ROUTE-PROBLEM}(\text{current}, \text{goals}, \text{allowed})$ 
  return A*-GRAPH-SEARCH(problem)

```

Figure 7.20 A hybrid agent program for the wumpus world. It uses a propositional knowledge base to infer the state of the world, and a combination of problem-solving search and domain-specific code to decide what actions to take.

Weakness: The computational expense goes up as time goes by.

3. Logical state estimation

To get a constant update time, we need to **cache** the result of inference.

Belief state: Some representation of the set of all possible current state of the world. (used to replace the past history of percepts and all their ramifications)

$$WumpusAlive^1 \wedge L_{2,1}^1 \wedge B_{2,1} \wedge (P_{3,1} \vee P_{2,2})$$

We use a logical sentence involving the proposition symbols associated with the current time step and the temporal symbols.

Logical state estimation involves maintaining a logical sentence that describes the set of possible states consistent with the observation history. Each update step requires inference using the transition model of the environment, which is built from **successor-state axioms** that specify how each **fluent** changes.

State estimation: The process of updating the belief state as new percepts arrive.

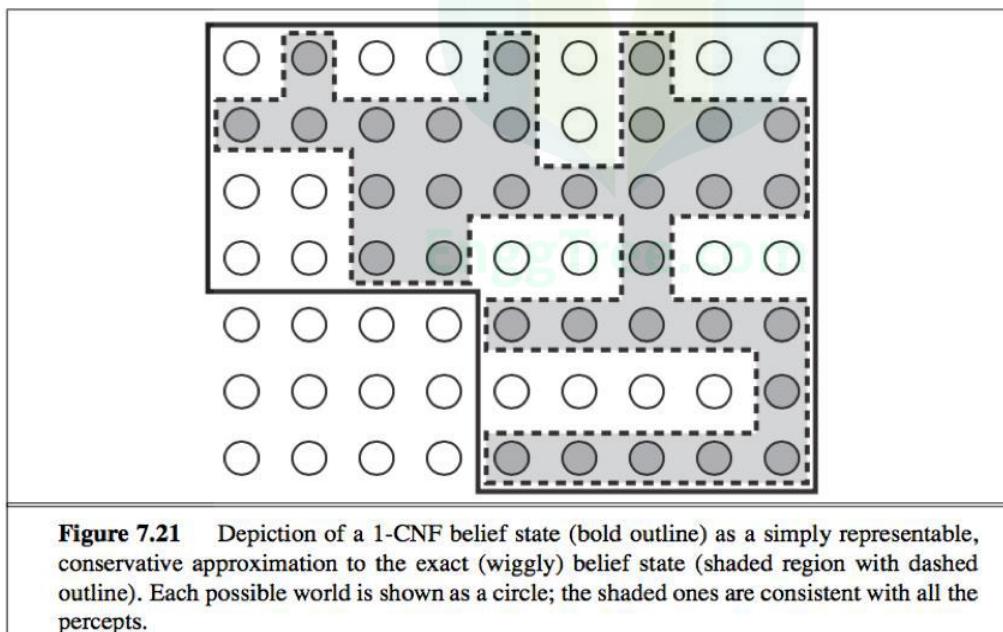
Exact state estimation may require logical formulas whose size is exponential in the number of symbols.

One common scheme for approximate state estimation: to represent belief state as conjunctions of literals (1-CNF formulas).

The agent simply tries to prove X^t and $\neg X^t$ for each symbol X^t , given the belief state at $t-1$.

The conjunction of provable literals becomes the new belief state, and the previous belief state is discarded. (This scheme may lose some information as time goes along.)

The set of possible states represented by the 1-CNF belief state includes all states that are in fact possible given the full percept history. The 1-CNF belief state acts as a simple outer envelope, or **conservative approximation**.



4. Making plans by propositional inference

We can make plans by logical inference instead of A* search in Figure 7.20.

Basic idea:

1. Construct a sentence that includes:
 - a) $Init^0$: a collection of assertions about the initial state;
 - b) $Transition^1, \dots, Transition^t$: The successor-state axioms for all possible actions at each time up to some maximum time t ;
 - c) $HaveGold^t \wedge ClimbedOut^t$: The assertion that the goal is achieved at time t .
2. Present the whole sentence to a SAT solver. If the solver finds a satisfying model, the goal is achievable; else the planning is impossible.
3. Assuming a model is found, extract from the model those variables that represent actions and are assigned true.

Together they represent a plan to achieve the goals.

Decisions within a logical agent can be made by SAT solving: finding possible models specifying future action sequences that reach the goal. This approach works only for fully observable or sensorless environment.

SATPLAN: A propositional planning. (Cannot be used in a partially observable environment)

SATPLAN finds models for a sentence containing the initial state, the goal, the successor-state axioms, and the action exclusion axioms.

(Because the agent does not know how many steps it will take to reach the goal, the algorithm tries each possible number of steps t up to some maximum conceivable plan length T_{\max} .)

```

function SATPLAN(init, transition, goal,  $T_{\max}$ ) returns solution or failure
  inputs: init, transition, goal, constitute a description of the problem
           $T_{\max}$ , an upper limit for plan length

  for  $t = 0$  to  $T_{\max}$  do
    cnf  $\leftarrow$  TRANSLATE-TO-SAT(init, transition, goal,  $t$ )
    model  $\leftarrow$  SAT-SOLVER(cnf)
    if model is not null then
      return EXTRACT-SOLUTION(model)
  return failure

```

Figure 7.22 The SATPLAN algorithm. The planning problem is translated into a CNF sentence in which the goal is asserted to hold at a fixed time step t and axioms are included for each time step up to t . If the satisfiability algorithm finds a model, then a plan is extracted by looking at those proposition symbols that refer to actions and are assigned *true* in the model. If no model exists, then the process is repeated with the goal moved one step later.

Precondition axioms: stating that an action occurrence requires the preconditions to be satisfied, added to avoid generating plans with illegal actions.

Action exclusion axioms: added to avoid the creation of plans with multiple simultaneous actions that interfere with each other.

Propositional logic does not scale to environments of unbounded size because it lacks the expressive power to deal concisely with time, space and universal patterns of relationships among objects.

6. First-order logic

First-Order Logic in Artificial intelligence

In the topic of Propositional logic, we have seen that how to represent statements using propositional logic. But unfortunately, in propositional logic, we can only represent the facts, which are either true or false. PL is not sufficient to represent the complex sentences or natural language statements. The propositional logic has very limited expressive power. Consider the following sentence, which we cannot represent using PL logic.

- "Some humans are intelligent", or
- "Sachin likes cricket."

To represent the above statements, PL logic is not sufficient, so we required some more powerful logic, such as first-order logic.

First-Order logic:

- First-order logic is another way of knowledge representation in artificial intelligence. It is an extension to propositional logic.

- FOL is sufficiently expressive to represent the natural language statements in a concise way.
- First-order logic is also known as **Predicate logic or First-order predicate logic**. First-order logic is a powerful language that develops information about the objects in a more easy way and can also express the relationship between those objects.
- First-order logic (like natural language) does not only assume that the world contains facts like propositional logic but also assumes the following things in the world:
 - **Objects:** A, B, people, numbers, colors, wars, theories, squares, pits, wumpus,
 - **Relations: It can be unary relation such as:** red, round, is adjacent, **or n-any relation such as:** the sister of, brother of, has color, comes between
 - **Function:** Father of, best friend, third inning of, end of,
- As a natural language, first-order logic also has two main parts:
 - a. **Syntax**
 - b. **Semantics**

Syntax of First-Order logic:

The syntax of FOL determines which collection of symbols is a logical expression in first-order logic. The basic syntactic elements of first-order logic are symbols. We write statements in short-hand notation in FOL.

Basic Elements of First-order logic:

Following are the basic elements of FOL syntax:

Constant	1, 2, A, John, Mumbai, cat,....
Variables	x, y, z, a, b,....
Predicates	Brother, Father, >,....
Function	sqrt, LeftLegOf,
Connectives	\wedge , \vee , \neg , \Rightarrow , \Leftrightarrow
Equality	$=$
Quantifier	\forall , \exists

Atomic sentences:

- Atomic sentences are the most basic sentences of first-order logic. These sentences are formed from a predicate symbol followed by a parenthesis with a sequence of terms.
- We can represent atomic sentences as **Predicate (term1, term2, , term n)**.

Example: Ravi and Ajay are brothers: \Rightarrow Brothers(Ravi, Ajay).
Chinky is a cat: \Rightarrow cat(Chinky).

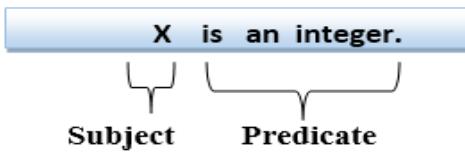
Complex Sentences:

- Complex sentences are made by combining atomic sentences using connectives.

First-order logic statements can be divided into two parts:

- **Subject:** Subject is the main part of the statement.
- **Predicate:** A predicate can be defined as a relation, which binds two atoms together in a statement.

Consider the statement: "x is an integer.", it consists of two parts, the first part x is the subject of the statement and second part "is an integer," is known as a predicate.



Quantifiers in First-order logic:

- A quantifier is a language element which generates quantification, and quantification specifies the quantity of specimen in the universe of discourse.
- These are the symbols that permit to determine or identify the range and scope of the variable in the logical expression. There are two types of quantifier:
 - a. **Universal Quantifier, (for all, everyone, everything)**
 - b. **Existential quantifier, (for some, at least one).**

Universal Quantifier:

Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing.

The Universal quantifier is represented by a symbol \forall , which resembles an inverted A.

Note: In universal quantifier we use implication " \rightarrow ".

If x is a variable, then $\forall x$ is read as:

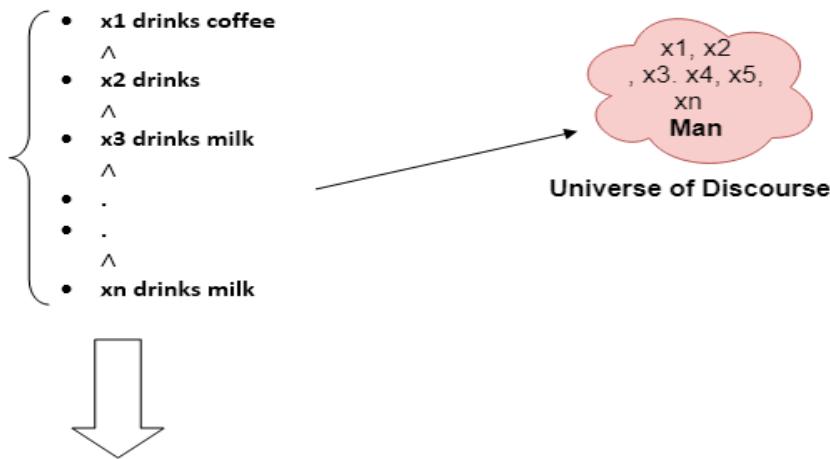
- **For all x**
- **For each x**
- **For every x.**

EnggTree.com

Example:

All man drink coffee.

Let a variable x which refers to a cat so all x can be represented in UOD as below:



So in shorthand notation, we can write it as :

$\forall x \text{ man}(x) \rightarrow \text{drink}(x, \text{coffee})$.

It will be read as: There are all x where x is a man who drink coffee.

Existential Quantifier:

Existential quantifiers are the type of quantifiers, which express that the statement within its scope is true for at least one instance of something.

It is denoted by the logical operator \exists , which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.

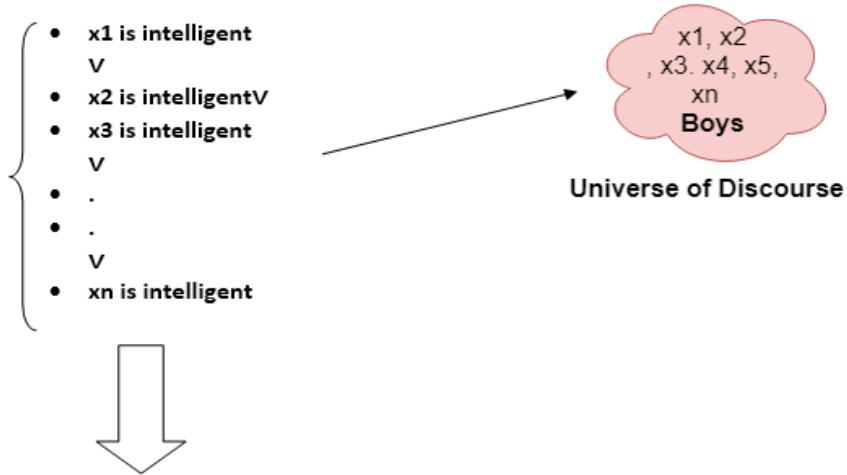
Note: In Existential quantifier we always use AND or Conjunction symbol (\wedge).

If x is a variable, then existential quantifier will be $\exists x$ or $\exists(x)$. And it will be read as:

- **There exists a 'x.'**
- **For some 'x.'**
- **For at least one 'x.'**

Example:

Some boys are intelligent.



So in short-hand notation, we can write it as:

$\exists x: \text{boys}(x) \wedge \text{intelligent}(x)$

It will be read as: There are some x where x is a boy who is intelligent.

Points to remember:

- The main connective for universal quantifier \forall is implication \rightarrow .
- The main connective for existential quantifier \exists is and \wedge .

Properties of Quantifiers:

- In universal quantifier, $\forall x \forall y$ is similar to $\forall y \forall x$.
- In Existential quantifier, $\exists x \exists y$ is similar to $\exists y \exists x$.
- $\exists x \forall y$ is not similar to $\forall y \exists x$.

Some Examples of FOL using quantifier:

1. All birds fly.

In this question the predicate is "**fly(bird)**."

And since there are all birds who fly so it will be represented as follows.

$$\forall x \text{ bird}(x) \rightarrow \text{fly}(x).$$

2. Every man respects his parent.

In this question, the predicate is "**respect(x, y)**," where $x = \text{man}$, and $y = \text{parent}$.

Since there is every man so will use \forall , and it will be represented as follows:

$$\forall x \text{ man}(x) \rightarrow \text{respects}(x, \text{parent}).$$

3. Some boys play cricket.

In this question, the predicate is "**play(x, y)**," where $x = \text{boys}$, and $y = \text{game}$. Since there are some boys so we will use \exists , and it will be represented as:

$$\exists x \text{ boys}(x) \rightarrow \text{play}(x, \text{cricket}).$$

4. Not all students like both Mathematics and Science.

In this question, the predicate is "**like(x, y)**," where $x = \text{student}$, and $y = \text{subject}$.

Since there are not all students, so we will use **V with negation**, so following representation for this:

$$\neg\forall (x) [\text{student}(x) \rightarrow \text{like}(x, \text{Mathematics}) \wedge \text{like}(x, \text{Science})].$$

5. Only one student failed in Mathematics.

In this question, the predicate is "**failed(x, y)**," where **x= student**, and **y= subject**.

Since there is only one student who failed in Mathematics, so we will use following representation for this:

$$\exists(x) [\text{student}(x) \rightarrow \text{failed}(x, \text{Mathematics}) \wedge \forall(y) [\neg(x=y) \wedge \text{student}(y) \rightarrow \neg\text{failed}(x, \text{Mathematics})]].$$

Free and Bound Variables:

The quantifiers interact with variables which appear in a suitable way. There are two types of variables in First-order logic which are given below:

Free Variable: A variable is said to be a free variable in a formula if it occurs outside the scope of the quantifier.

Example: $\forall x \exists(y)[P(x, y, z)]$, where **z** is a free variable.

Bound Variable: A variable is said to be a bound variable in a formula if it occurs within the scope of the quantifier.

Example: $\forall x [A(x) B(y)]$, here **x** and **y** are the bound variables.

7. Knowledge representation and engineering

Knowledge Engineering in First-order logic

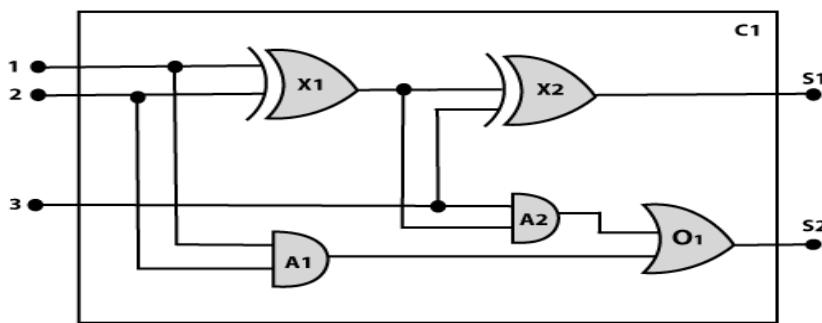
What is knowledge-engineering?

The process of constructing a knowledge-base in first-order logic is called as knowledge- engineering. In **knowledge-engineering**, someone who investigates a particular domain, learns important concept of that domain, and generates a formal representation of the objects, is known as **knowledge engineer**.

In this topic, we will understand the Knowledge engineering process in an electronic circuit domain, which is already familiar. This approach is mainly suitable for creating **special-purpose knowledge base**.

The knowledge-engineering process:

Following are some main steps of the knowledge-engineering process. Using these steps, we will develop a knowledge base which will allow us to reason about digital circuit (**One-bit full adder**) which is given below



1. Identify the task:

The first step of the process is to identify the task, and for the digital circuit, there are various reasoning tasks

At the first level or highest level, we will examine the functionality of the circuit:

- **Does the circuit add properly?**
- **What will be the output of gate A2, if all the inputs are high?**

At the second level, we will examine the circuit structure details such as:

- **Which gate is connected to the first input terminal?**
- **Does the circuit have feedback loops?**

2. Assemble the relevant knowledge:

In the second step, we will assemble the relevant knowledge which is required for digital circuits. So for digital circuits, we have the following required knowledge:

- Logic circuits are made up of wires and gates.
- Signal flows through wires to the input terminal of the gate, and each gate produces the corresponding output which flows further.
- In this logic circuit, there are four types of gates used: **AND, OR, XOR, and NOT**.
- All these gates have one output terminal and two input terminals (except NOT gate, it has one input terminal).

3. Decide on vocabulary:

The next step of the process is to select functions, predicate, and constants to represent the circuits, terminals, signals, and gates. Firstly we will distinguish the gates from each other and from other objects. Each gate is represented as an object which is named by a constant, such as, **Gate(X1)**. The functionality of each gate is determined by its type, which is taken as constants such as **AND, OR, XOR, or NOT**. Circuits will be identified by a predicate: **Circuit (C1)**.

For the terminal, we will use predicate: **Terminal(x)**.

For gate input, we will use the function **In(1, X1)** for denoting the first input terminal of the gate, and for output terminal we will use **Out (1, X1)**.

The function **Arity(c, i, j)** is used to denote that circuit c has i input, j output.

The connectivity between gates can be represented by predicate **Connect(Out(1, X1), In(1, X1))**.

We use a unary predicate **On (t)**, which is true if the signal at a terminal is on.

4. Encode general knowledge about the domain:

To encode the general knowledge about the logic circuit, we need some following rules:

- If two terminals are connected then they have the same input signal, it can be represented as:

$\forall t_1, t_2 \text{ Terminal } (t_1) \wedge \text{Terminal } (t_2) \wedge \text{Connect } (t_1, t_2) \rightarrow \text{Signal } (t_1) = \text{Signal } (2).$

- Signal at every terminal will have either value 0 or 1, it will be represented as:

$\forall t \text{ Terminal } (t) \rightarrow \text{Signal } (t) = 1 \vee \text{Signal } (t) = 0.$

- Connect predicates are commutative:

$\forall t_1, t_2 \text{ Connect}(t_1, t_2) \rightarrow \text{Connect } (t_2, t_1).$

- Representation of types of gates:

$\forall g \text{ Gate}(g) \wedge r = \text{Type}(g) \rightarrow r = \text{OR} \vee r = \text{AND} \vee r = \text{XOR} \vee r = \text{NOT}.$

- Output of AND gate will be zero if and only if any of its input is zero.

$\forall g \text{ Gate}(g) \wedge \text{Type}(g) = \text{AND} \rightarrow \text{Signal } (\text{Out}(1, g)) = 0 \Leftrightarrow \exists n \text{ Signal } (\text{In}(n, g)) = 0.$

- Output of OR gate is 1 if and only if any of its input is 1:

$\forall g \text{ Gate}(g) \wedge \text{Type}(g) = \text{OR} \rightarrow \text{Signal } (\text{Out}(1, g)) = 1 \Leftrightarrow \exists n \text{ Signal } (\text{In}(n, g)) = 1$

- Output of XOR gate is 1 if and only if its inputs are different:

$\forall g \text{ Gate}(g) \wedge \text{Type}(g) = \text{XOR} \rightarrow \text{Signal } (\text{Out}(1, g)) = 1 \Leftrightarrow \text{Signal } (\text{In}(1, g)) \neq \text{Signal } (\text{In}(2, g)).$

- Output of NOT gate is invert of its input:

$\forall g \text{ Gate}(g) \wedge \text{Type}(g) = \text{NOT} \rightarrow \text{Signal } (\text{In}(1, g)) \neq \text{Signal } (\text{Out}(1, g)).$

- All the gates in the above circuit have two inputs and one output (except NOT gate).

$\forall g \text{ Gate}(g) \wedge \text{Type}(g) = \text{NOT} \rightarrow \text{Arity}(g, 1, 1)$

$\forall g \text{ Gate}(g) \wedge r = \text{Type}(g) \wedge (r = \text{AND} \vee r = \text{OR} \vee r = \text{XOR}) \rightarrow \text{Arity}(g, 2, 1).$

- All gates are logic circuits:

$\forall g \text{ Gate}(g) \rightarrow \text{Circuit } (g).$

5. Encode a description of the problem instance:

Now we encode problem of circuit C1, firstly we categorize the circuit and its gate components. This step is easy if ontology about the problem is already thought. This step involves the writing simple atomics sentences of instances of concepts, which is known as ontology.

For the given circuit C1, we can encode the problem instance in atomic sentences as below:

Since in the circuit there are two XOR, two AND, and one OR gate so atomic sentences for these gates will be:

For XOR gate: Type(x1)=XOR, Type(X2) = XOR

For AND gate: Type(A1) = AND, Type(A2)= AND

For OR gate: Type (O1) = OR.

And then represent the connections between all the gates.

Note: Ontology defines a particular theory of the nature of existence.

6. Pose queries to the inference procedure and get answers:

In this step, we will find all the possible set of values of all the terminal for the adder circuit. The first query will be:

What should be the combination of input which would generate the first output of circuit C1, as 0 and a second output to be 1?

$\exists i_1, i_2, i_3 \text{ Signal (In}(1, C1))=i_1 \wedge \text{ Signal (In}(2, C1))=i_2 \wedge \text{ Signal (In}(3, C1))=i_3 \\ \wedge \text{ Signal (Out}(1, C1)) =0 \wedge \text{ Signal (Out}(2, C1))=1$

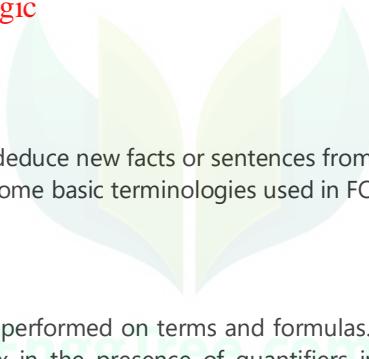
7. Debug the knowledge base:

Now we will debug the knowledge base, and this is the last step of the complete process. In this step, we will try to debug the issues of knowledge base.

In the knowledge base, we may have omitted assertions like $1 \neq 0$.

8. Inferences in first-order logic

Inference in First-Order Logic



Inference in First-Order Logic is used to deduce new facts or sentences from existing sentences. Before understanding the FOL inference rule, let's understand some basic terminologies used in FOL.

Substitution:

Substitution is a fundamental operation performed on terms and formulas. It occurs in all inference systems in first-order logic. The substitution is complex in the presence of quantifiers in FOL. If we write $F[a/x]$, so it refers to substitute a constant "a" in place of variable "x".

Note: First-order logic is capable of expressing facts about some or all objects in the universe.

Equality:

First-Order logic does not only use predicate and terms for making atomic sentences but also uses another way, which is equality in FOL. For this, we can use **equality symbols** which specify that the two terms refer to the same object.

Example: Brother (John) = Smith.

As in the above example, the object referred by the **Brother (John)** is similar to the object referred by **Smith**. The equality symbol can also be used with negation to represent that two terms are not the same objects.

Example: $\neg(x=y)$ which is equivalent to $x \neq y$.

FOL inference rules for quantifier:

As propositional logic we also have inference rules in first-order logic, so following are some basic inference rules in FOL:

- **Universal Generalization**
- **Universal Instantiation**
- **Existential Instantiation**
- **Existential introduction**

1. Universal Generalization:

- Universal generalization is a valid inference rule which states that if premise $P(c)$ is true for any arbitrary element c in the universe of discourse, then we can have a conclusion as $\forall x P(x)$.

$$\frac{P(c)}{\forall x P(x)}$$

- It can be represented as: $\frac{P(c)}{\forall x P(x)}$.
- This rule can be used if we want to show that every element has a similar property.
- In this rule, x must not appear as a free variable.

Example: Let's represent, $P(c)$: "A byte contains 8 bits", so for $\forall x P(x)$ "All bytes contain 8 bits.", it will also be true.

2. Universal Instantiation:

- Universal instantiation is also called as universal elimination or UI is a valid inference rule. It can be applied multiple times to add new sentences.
- The new KB is logically equivalent to the previous KB.
- As per UI, **we can infer any sentence obtained by substituting a ground term for the variable.**
- The UI rule state that we can infer any sentence $P(c)$ by substituting a ground term c (a constant within domain x) from $\forall x P(x)$ **for any object in the universe of discourse.**

$$\frac{\forall x P(x)}{P(c)}$$

- It can be represented as: $\frac{\forall x P(x)}{P(c)}$.
- **Example:1.**
IF "Every person like ice-cream" $\Rightarrow \forall x P(x)$ so we can infer that "John likes ice-cream" $\Rightarrow P(c)$
- **Example: 2.**
Let's take a famous example,
"All kings who are greedy are Evil." So let our knowledge base contains this detail as in the form of FOL:

$$\forall x \text{king}(x) \wedge \text{greedy}(x) \rightarrow \text{Evil}(x),$$

So from this information, we can infer any of the following statements using Universal Instantiation:

- **King(John) \wedge Greedy (John) \rightarrow Evil (John),**
- **King(Richard) \wedge Greedy (Richard) \rightarrow Evil (Richard),**
- **King(Father(John)) \wedge Greedy (Father(John)) \rightarrow Evil (Father(John)),**

3. Existential Instantiation:

Existential instantiation is also called as Existential Elimination, which is a valid inference rule in first-order logic.

- It can be applied only once to replace the existential sentence.
- The new KB is not logically equivalent to old KB, but it will be satisfiable if old KB was satisfiable.
- This rule states that one can infer $P(c)$ from the formula given in the form of $\exists x P(x)$ for a new constant symbol c .
- The restriction with this rule is that c used in the rule must be a new term for which $P(c)$ is true.

$$\frac{\exists x P(x)}{P(c)}$$

- It can be represented as:

Example:

From the given sentence: $\exists x \text{Crown}(x) \wedge \text{OnHead}(x, \text{John})$,

So we can infer: $\text{Crown}(K) \wedge \text{OnHead}(K, \text{John})$, as long as K does not appear in the knowledge base.

- The above used K is a constant symbol, which is called **Skolem constant**.
- The Existential instantiation is a special case of **Skolemization process**.

4. Existential introduction

- An existential introduction is also known as an existential generalization, which is a valid inference rule in first-order logic.
- This rule states that if there is some element c in the universe of discourse which has a property P , then we can infer that there exists something in the universe which has the property P .

$$\frac{P(c)}{\exists x P(x)}$$

- It can be represented as:
- **Example: Let's say that,**
 "Priyanka got good marks in English."
 "Therefore, someone got good marks in English."

Generalized Modus Ponens Rule:

For the inference process in FOL, we have a single inference rule which is called Generalized Modus Ponens. It is lifted version of Modus ponens.

Generalized Modus Ponens can be summarized as, " P implies Q and P is asserted to be true, therefore Q must be True."

According to Modus Ponens, for atomic sentences \mathbf{pi} , \mathbf{pi}' , \mathbf{q} . Where there is a substitution θ such that $\text{SUBST}(\theta, \mathbf{pi}') = \text{SUBST}(\theta, \mathbf{pi})$, it can be represented as:

$$\frac{\mathbf{p1}', \mathbf{p2}', \dots, \mathbf{pn}', (\mathbf{p1} \wedge \mathbf{p2} \wedge \dots \wedge \mathbf{pn} \Rightarrow \mathbf{q})}{\text{SUBST}(\theta, \mathbf{q})}$$

Example:

We will use this rule for Kings are evil, so we will find some x such that x is king, and x is greedy so we can infer that x is evil.

Here let say, p_1' is $\text{king}(\text{John})$ p_1 is $\text{king}(x)$
 p_2' is $\text{Greedy}(y)$ p_2 is $\text{Greedy}(x)$
 θ is $\{x/\text{John}, y/\text{John}\}$ q is $\text{evil}(x)$
 $\text{SUBST}(\theta, q)$.

9. Forward chaining and Backward chaining

Forward Chaining and backward chaining in AI

In artificial intelligence, forward and backward chaining is one of the important topics, but before understanding forward and backward chaining lets first understand that from where these two terms came.

Inference engine:

The inference engine is the component of the intelligent system in artificial intelligence, which applies logical rules to the knowledge base to infer new information from known facts. The first inference engine was part of the expert system. Inference engine commonly proceeds in two modes, which are:

- A. **Forward chaining**
- B. **Backward chaining**

Horn Clause and Definite clause:

Horn clause and definite clause are the forms of sentences, which enables knowledge base to use a more restricted and efficient inference algorithm. Logical inference algorithms use forward and backward chaining approaches, which require KB in the form of the **first-order definite clause**.

Definite clause: A clause which is a disjunction of literals with **exactly one positive literal** is known as a definite clause or strict horn clause.

Horn clause: A clause which is a disjunction of literals with **at most one positive literal** is known as horn clause. Hence all the definite clauses are horn clauses.

Example: $(\neg p \vee \neg q \vee k)$. It has only one positive literal k .

It is equivalent to $p \wedge q \rightarrow k$.

A. Forward Chaining

Forward chaining is also known as a forward deduction or forward reasoning method when using an inference engine. Forward chaining is a form of reasoning which start with atomic sentences in the knowledge base and applies inference rules (Modus Ponens) in the forward direction to extract more data until a goal is reached.

The Forward-chaining algorithm starts from known facts, triggers all rules whose premises are satisfied, and add their conclusion to the known facts. This process repeats until the problem is solved.

Properties of Forward-Chaining:

- It is a down-up approach, as it moves from bottom to top.
- It is a process of making a conclusion based on known facts or data, by starting from the initial state and reaches the goal state.
- Forward-chaining approach is also called as data-driven as we reach to the goal using available data.
- Forward -chaining approach is commonly used in the expert system, such as CLIPS, business, and production rule systems.

Consider the following famous example which we will use in both approaches:

Example:

"As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an American citizen."

Prove that "**Robert is criminal.**"

To solve the above problem, first, we will convert all the above facts into first-order definite clauses, and then we will use a forward-chaining algorithm to reach the goal.

Facts Conversion into FOL:

- It is a crime for an American to sell weapons to hostile nations. (Let's say p, q, and r are variables)
American (p) \wedge weapon(q) \wedge sells (p, q, r) \wedge hostile(r) \rightarrow Criminal(p)(1)
- Country A has some missiles. **?p Owns(A, p) \wedge Missile(p).** It can be written in two definite clauses by using Existential Instantiation, introducing new Constant T1.
Owns(A, T1)(2)
Missile(T1)(3)
- All of the missiles were sold to country A by Robert.
?p Missiles(p) \wedge Owns (A, p) \rightarrow Sells (Robert, p, A)(4)
- Missiles are weapons.
Missile(p) \rightarrow Weapons (p)(5)
- Enemy of America is known as hostile.
Enemy(p, America) \rightarrow Hostile(p)(6)
- Country A is an enemy of America.
Enemy (A, America)(7)
- Robert is American
American(Robert).(8)

Forward chaining proof:

Step-1:

In the first step we will start with the known facts and will choose the sentences which do not have implications, such as: **American(Robert), Enemy(A, America), Owns(A, T1), and Missile(T1)**. All these facts will be represented as below.

American (Robert)	Missile (T1)	Owns (A,T1)	Enemy (A, America)
-------------------	--------------	-------------	--------------------

Step-2:

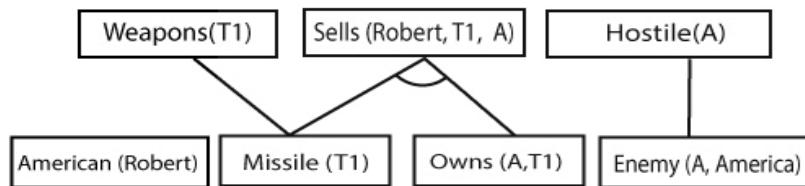
At the second step, we will see those facts which infer from available facts and with satisfied premises.

Rule-(1) does not satisfy premises, so it will not be added in the first iteration.

Rule-(2) and (3) are already added.

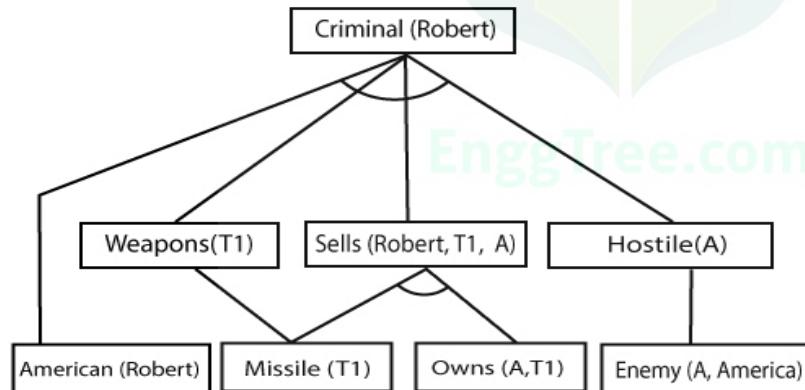
Rule-(4) satisfy with the substitution {p/T1}, **so Sells (Robert, T1, A)** is added, which infers from the conjunction of Rule (2) and (3).

Rule-(6) is satisfied with the substitution(p/A), so Hostile(A) is added and which infers from Rule-(7).



Step-3:

At step-3, as we can check Rule-(1) is satisfied with the substitution {p/Robert, q/T1, r/A}, **so we can add Criminal(Robert)** which infers all the available facts. And hence we reached our goal statement.



Hence it is proved that Robert is Criminal using forward chaining approach.

B. Backward Chaining:

Backward-chaining is also known as a backward deduction or backward reasoning method when using an inference engine. A backward chaining algorithm is a form of reasoning, which starts with the goal and works backward, chaining through rules to find known facts that support the goal.

Properties of backward chaining:

- o It is known as a top-down approach.
- o Backward-chaining is based on modus ponens inference rule.

- In backward chaining, the goal is broken into sub-goal or sub-goals to prove the facts true.
- It is called a goal-driven approach, as a list of goals decides which rules are selected and used.
- Backward -chaining algorithm is used in game theory, automated theorem proving tools, inference engines, proof assistants, and various AI applications.
- The backward-chaining method mostly used a **depth-first search** strategy for proof.

Example:

In backward-chaining, we will use the same above example, and will rewrite all the rules.

- **American (p) \wedge weapon(q) \wedge sells (p, q, r) \wedge hostile(r) \rightarrow Criminal(p) ... (1)**
Owns(A, T1)(2)
- **Missile(T1)**
- **?p Missiles(p) \wedge Owns (A, p) \rightarrow Sells (Robert, p, A)(4)**
- **Missile(p) \rightarrow Weapons (p)(5)**
- **Enemy(p, America) \rightarrow Hostile(p)(6)**
- **Enemy (A, America)(7)**
- **American(Robert).(8)**

Backward-Chaining proof:

In Backward chaining, we will start with our goal predicate, which is **Criminal(Robert)**, and then infer further rules.

Step-1:

At the first step, we will take the goal fact. And from the goal fact, we will infer other facts, and at last, we will prove those facts true. So our goal fact is "Robert is Criminal," so following is the predicate of it.

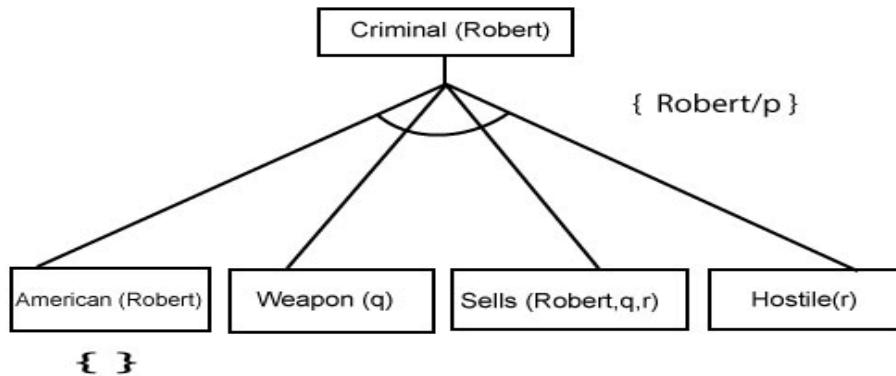
Criminal (Robert)

EnggTree.com

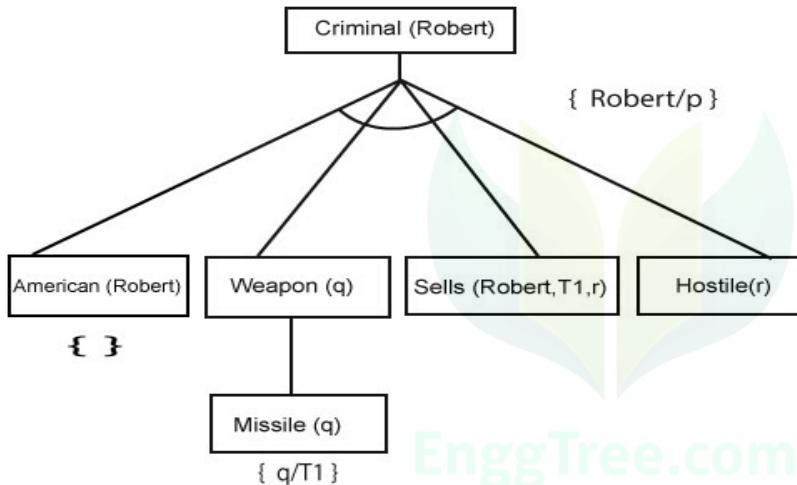
Step-2:

At the second step, we will infer other facts form goal fact which satisfies the rules. So as we can see in Rule-1, the goal predicate Criminal (Robert) is present with substitution {Robert/P}. So we will add all the conjunctive facts below the first level and will replace p with Robert.

Here we can see American (Robert) is a fact, so it is proved here.

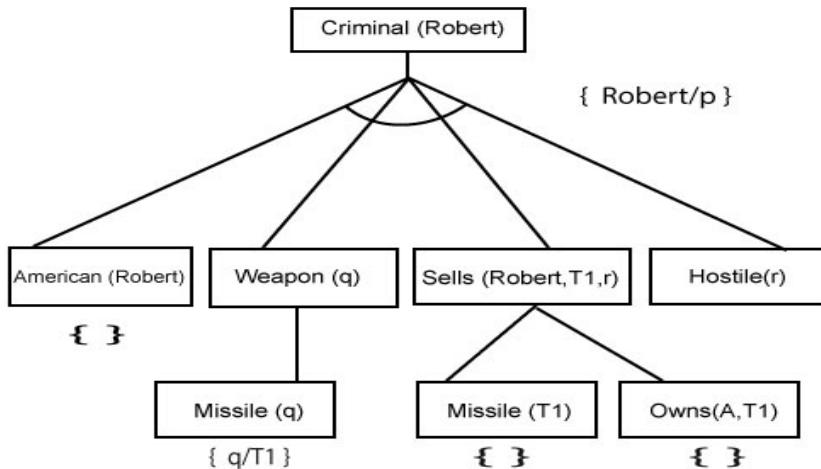


Step-3:t At step-3, we will extract further fact Missle(q) which infer from Weapon(q), as it satisfies Rule-(5). Weapon(q) is also true with the substitution of a constant T1 at q.

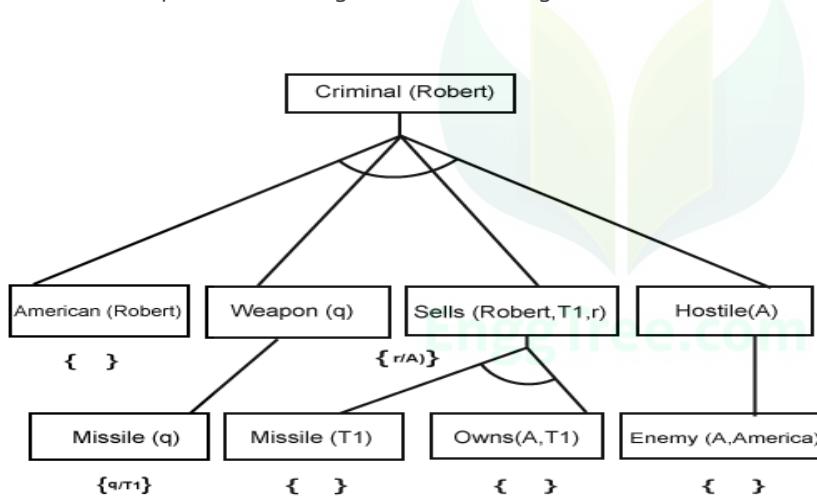


Step-4:

At step-4, we can infer facts Missle(T1) and Owns(A, T1) form Sells(Robert, T1, r) which satisfies the **Rule- 4**, with the substitution of A in place of r. So these two statements are proved here.

**Step-5:**

At step-5, we can infer the fact **Enemy(A, America)** from **Hostile(A)** which satisfies Rule- 6. And hence all the statements are proved true using backward chaining.



10. Difference between backward chaining and forward chaining

Difference between backward chaining and forward chaining

Following is the difference between the forward chaining and backward chaining:

- Forward chaining as the name suggests, start from the known facts and move forward by applying inference rules to extract more data, and it continues until it reaches to the goal, whereas backward chaining starts from the goal, move backward by using inference rules to determine the facts that satisfy the goal.
- Forward chaining is called a **data-driven** inference technique, whereas backward chaining is called a **goal-driven** inference technique.
- Forward chaining is known as the **down-up** approach, whereas backward chaining is known as a **top-down** approach.

- Forward chaining uses **breadth-first search** strategy, whereas backward chaining uses **depth-first search** strategy.
- Forward and backward chaining both applies **Modus ponens** inference rule.
- Forward chaining can be used for tasks such as **planning, design process monitoring, diagnosis, and classification**, whereas backward chaining can be used for **classification and diagnosis tasks**.
- Forward chaining can be like an exhaustive search, whereas backward chaining tries to avoid the unnecessary path of reasoning.
- In forward-chaining there can be various ASK questions from the knowledge base, whereas in backward chaining there can be fewer ASK questions.
- Forward chaining is slow as it checks for all the rules, whereas backward chaining is fast as it checks few required rules only.

S. No.	Forward Chaining	Backward Chaining
1.	Forward chaining starts from known facts and applies inference rule to extract more data until it reaches to the goal.	Backward chaining starts from the goal and works backward through inference rules to find the required facts that support the goal.
2.	It is a bottom-up approach	It is a top-down approach
3.	Forward chaining is known as data-driven inference technique as we reach to the goal using the available data.	Backward chaining is known as goal-driven technique as we start from the goal and divide into sub-goal to extract the facts.
4.	Forward chaining reasoning applies a breadth-first search strategy.	Backward chaining reasoning applies a depth-first search strategy.
5.	Forward chaining tests for all the available rules	Backward chaining only tests for few required rules.
6.	Forward chaining is suitable for the planning, monitoring, control, and interpretation application.	Backward chaining is suitable for diagnostic, prescription, and debugging application.
7.	Forward chaining can generate an infinite number of possible conclusions.	Backward chaining generates a finite number of possible conclusions.
8.	It operates in the forward direction.	It operates in the backward direction.
9.	Forward chaining is aimed for any conclusion.	Backward chaining is only aimed for the required data.

11. Resolution

Resolution in FOL

Resolution

Resolution is a theorem proving technique that proceeds by building refutation proofs, i.e., proofs by contradictions. It was invented by a Mathematician John Alan Robinson in the year 1965.

Resolution is used, if there are various statements are given, and we need to prove a conclusion of those statements. Unification is a key concept in proofs by resolutions. Resolution is a single inference rule which can efficiently operate on the **conjunctive normal form or clausal form**.

Clause: Disjunction of literals (an atomic sentence) is called a **clause**. It is also known as a unit clause.

Conjunctive Normal Form: A sentence represented as a conjunction of clauses is said to be **conjunctive normal form** or **CNF**.

The resolution inference rule:

The resolution rule for first-order logic is simply a lifted version of the propositional rule. Resolution can resolve two clauses if they contain complementary literals, which are assumed to be standardized apart so that they share no variables.

$$\frac{l_1 \vee \dots \vee l_k, \quad m_1 \vee \dots \vee m_n}{\text{SUBST}(\theta, l_1 \vee \dots \vee l_{i-1} \vee l_{i+1} \vee \dots \vee l_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n)}$$

Where l_i and m_j are complementary literals.

This rule is also called the **binary resolution rule** because it only resolves exactly two literals.

Example:

We can resolve two clauses which are given below:

$$[\text{Animal}(g(x)) \vee \text{Loves}(f(x), x)] \quad \text{and} \quad [\neg \text{Loves}(a, b) \vee \neg \text{Kills}(a, b)]$$

Where two complimentary literals are: **Loves(f(x), x)** and **\neg Loves(a, b)**

These literals can be unified with unifier $\theta = [a/f(x), \text{and } b/x]$, and it will generate a resolvent clause:

$$[\text{Animal}(g(x)) \vee \neg \text{Kills}(f(x), x)].$$

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).

To better understand all the above steps, we will take an example in which we will apply resolution.

Example:

- a. John likes all kind of food.
- b. Apple and vegetable are food
- c. Anything anyone eats and not killed is food.
- d. Anil eats peanuts and still alive

- e. Harry eats everything that Anil eats.
Prove by resolution that:
- f. John likes peanuts.

Step-1: Conversion of Facts into FOL

In the first step we will convert all the given statements into its first order logic.

- a. $\forall x: \text{food}(x) \rightarrow \text{likes}(\text{John}, x)$
- b. $\text{food}(\text{Apple}) \wedge \text{food}(\text{vegetables})$
- c. $\forall x \forall y: \text{eats}(x, y) \wedge \neg \text{killed}(x) \rightarrow \text{food}(y)$
- d. $\text{eats}(\text{Anil}, \text{Peanuts}) \wedge \text{alive}(\text{Anil})$.
- e. $\forall x: \text{eats}(\text{Anil}, x) \rightarrow \text{eats}(\text{Harry}, x)$
- f. $\forall x: \neg \text{killed}(x) \rightarrow \text{alive}(x)$ } added predicates.
- g. $\forall x: \text{alive}(x) \rightarrow \neg \text{killed}(x)$
- h. $\text{likes}(\text{John}, \text{Peanuts})$

Step-2: Conversion of FOL into CNF

In First order logic resolution, it is required to convert the FOL into CNF as CNF form makes easier for resolution proofs.

- o **Eliminate all implication (\rightarrow) and rewrite**
 - a. $\forall x \neg \text{food}(x) \vee \text{likes}(\text{John}, x)$
 - b. $\text{food}(\text{Apple}) \wedge \text{food}(\text{vegetables})$
 - c. $\forall x \forall y \neg [\text{eats}(x, y) \wedge \neg \text{killed}(x)] \vee \text{food}(y)$
 - d. $\text{eats}(\text{Anil}, \text{Peanuts}) \wedge \text{alive}(\text{Anil})$
 - e. $\forall x \neg \text{eats}(\text{Anil}, x) \vee \text{eats}(\text{Harry}, x)$
 - f. $\forall x \neg [\neg \text{killed}(x)] \vee \text{alive}(x)$
 - g. $\forall x \neg \text{alive}(x) \vee \neg \text{killed}(x)$
 - h. $\text{likes}(\text{John}, \text{Peanuts})$.
- o **Move negation (\neg)inwards and rewrite**
 - . $\forall x \neg \text{food}(x) \vee \text{likes}(\text{John}, x)$
 - a. $\text{food}(\text{Apple}) \wedge \text{food}(\text{vegetables})$
 - b. $\forall x \forall y \neg \text{eats}(x, y) \vee \text{killed}(x) \vee \text{food}(y)$
 - c. $\text{eats}(\text{Anil}, \text{Peanuts}) \wedge \text{alive}(\text{Anil})$
 - d. $\forall x \neg \text{eats}(\text{Anil}, x) \vee \text{eats}(\text{Harry}, x)$
 - e. $\forall x \neg \text{killed}(x) \vee \text{alive}(x)$
 - f. $\forall x \neg \text{alive}(x) \vee \neg \text{killed}(x)$
 - g. $\text{likes}(\text{John}, \text{Peanuts})$.
- o **Rename variables or standardize variables**
 - . $\forall x \neg \text{food}(x) \vee \text{likes}(\text{John}, x)$
 - a. $\text{food}(\text{Apple}) \wedge \text{food}(\text{vegetables})$
 - b. $\forall y \forall z \neg \text{eats}(y, z) \vee \text{killed}(y) \vee \text{food}(z)$
 - c. $\text{eats}(\text{Anil}, \text{Peanuts}) \wedge \text{alive}(\text{Anil})$

- d. $\forall w \neg \text{eats}(\text{Anil}, w) \vee \text{eats}(\text{Harry}, w)$
 - e. $\forall g \neg \text{killed}(g) \vee \text{alive}(g)$
 - f. $\forall k \neg \text{alive}(k) \vee \neg \text{killed}(k)$
 - g. likes(John, Peanuts).
- o **Eliminate existential instantiation quantifier by elimination.**
In this step, we will eliminate existential quantifier \exists , and this process is known as **Skolemization**. But in this example problem since there is no existential quantifier so all the statements will remain same in this step.
 - o **Drop Universal quantifiers.**
In this step we will drop all universal quantifier since all the statements are not implicitly quantified so we don't need it.
 - a. $\neg \text{food}(x) \vee \text{likes}(\text{John}, x)$
 - b. food(Apple)
 - c. food(vegetables)
 - d. $\neg \text{eats}(y, z) \vee \text{killed}(y) \vee \text{food}(z)$
 - e. eats (Anil, Peanuts)
 - f. alive(Anil)
 - g. $\neg \text{eats}(\text{Anil}, w) \vee \text{eats}(\text{Harry}, w)$
 - h. killed(g) V alive(g)
 - i. $\neg \text{alive}(k) \vee \neg \text{killed}(k)$
 - j. likes(John, Peanuts).

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

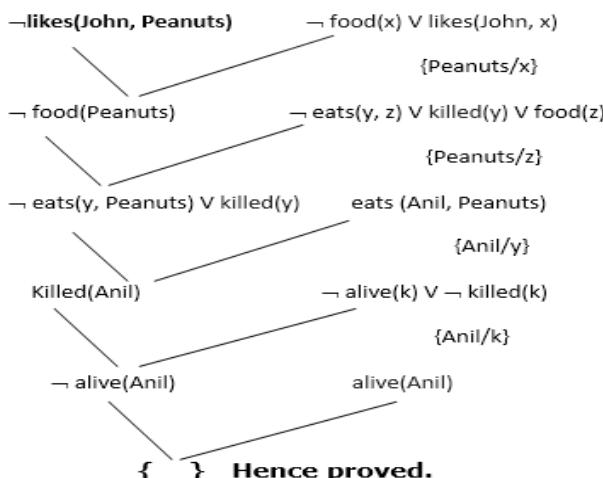
- o **Distribute conjunction over disjunction** \neg .
- This step will not make any change in this problem.

Step-3: Negate the statement to be proved

In this statement, we will apply negation to the conclusion statements, which will be written as $\neg \text{likes}(\text{John}, \text{Peanuts})$

Step-4: Draw Resolution graph:

Now in this step, we will solve the problem by resolution tree using substitution. For the above problem, it will be given as follows:



Hence the negation of the conclusion has been proved as a complete contradiction with the given set of statements.

Explanation of Resolution graph:

- In the first step of resolution graph, $\neg \text{likes}(\text{John}, \text{Peanuts})$, and $\text{likes}(\text{John}, x)$ get resolved(canceled) by substitution of $\{\text{Peanuts}/x\}$, and we are left with $\neg \text{food}(\text{Peanuts})$
- In the second step of the resolution graph, $\neg \text{food}(\text{Peanuts})$, and $\text{food}(z)$ get resolved (canceled) by substitution of $\{\text{Peanuts}/z\}$, and we are left with $\neg \text{eats}(y, \text{Peanuts}) \vee \text{killed}(y)$.
- In the third step of the resolution graph, $\neg \text{eats}(y, \text{Peanuts})$ and $\text{eats}(\text{Anil}, \text{Peanuts})$ get resolved by substitution $\{\text{Anil}/y\}$, and we are left with $\text{Killed}(\text{Anil})$.
- In the fourth step of the resolution graph, $\text{Killed}(\text{Anil})$ and $\neg \text{killed}(k)$ get resolve by substitution $\{\text{Anil}/k\}$, and we are left with $\neg \text{alive}(\text{Anil})$.
- In the last step of the resolution graph $\neg \text{alive}(\text{Anil})$ and $\text{alive}(\text{Anil})$ get resolved.



EnggTree.com

AL3391 ARTIFICIAL INTELLIGENCE**UNIT V****PROBABILISTIC REASONING**

Acting under uncertainty – Bayesian inference – naïve Bayes models. Probabilistic reasoning – Bayesian networks – exact inference in BN – approximate inference in BN – causal networks.

1. Acting under uncertainty

Uncertainty:

Till now, we have learned knowledge representation using first-order logic and propositional logic with certainty, which means we were sure about the predicates. With this knowledge representation, we might write $A \rightarrow B$, which means if A is true then B is true, but consider a situation where we are not sure about whether A is true or not then we cannot express this statement, this situation is called uncertainty.

So to represent uncertain knowledge, where we are not sure about the predicates, we need uncertain reasoning or probabilistic reasoning.

Causes of uncertainty:

Following are some leading causes of uncertainty to occur in the real world.

1. Information occurred from unreliable sources.
2. Experimental Errors
3. Equipment fault
4. Temperature variation
5. Climate change.

2. Bayesian inference

Bayes' theorem in Artificial intelligence

Bayes' theorem:

Bayes' theorem is also known as **Bayes' rule**, **Bayes' law**, or **Bayesian reasoning**, which determines the probability of an event with uncertain knowledge.

In probability theory, it relates the conditional probability and marginal probabilities of two random events.

Bayes' theorem was named after the British mathematician **Thomas Bayes**. The **Bayesian inference** is an application of Bayes' theorem, which is fundamental to Bayesian statistics.

It is a way to calculate the value of $P(B|A)$ with the knowledge of $P(A|B)$.

Bayes' theorem allows updating the probability prediction of an event by observing new information of the real world.

Example: If cancer corresponds to one's age then by using Bayes' theorem, we can determine the probability of cancer more accurately with the help of age.

Bayes' theorem can be derived using product rule and conditional probability of event A with known event B:

As from product rule we can write:

$$P(A \wedge B) = P(A|B) P(B) \text{ or}$$

Similarly, the probability of event B with known event A:

$$P(A \wedge B) = P(B|A) P(A)$$

Equating right hand side of both the equations, we will get:

$$P(A|B) = \frac{P(B|A) P(A)}{P(B)} \quad \dots(a)$$

The above equation (a) is called as **Bayes' rule** or **Bayes' theorem**. This equation is basic of most modern AI systems for **probabilistic inference**.

It shows the simple relationship between joint and conditional probabilities. Here,

$P(A|B)$ is known as **posterior**, which we need to calculate, and it will be read as Probability of hypothesis A when we have occurred an evidence B.

$P(B|A)$ is called the likelihood, in which we consider that hypothesis is true, then we calculate the probability of evidence.

$P(A)$ is called the **prior probability**, probability of hypothesis before considering the evidence

$P(B)$ is called **marginal probability**, pure probability of an evidence.

In the equation (a), in general, we can write $P(B) = P(A_i) * P(B|A_i)$, hence the Bayes' rule can be written as:

$$P(A_i|B) = \frac{P(A_i) * P(B|A_i)}{\sum_{i=1}^k P(A_i) * P(B|A_i)}$$

Where $A_1, A_2, A_3, \dots, A_n$ is a set of mutually exclusive and exhaustive events.

Applying Bayes' rule:

Bayes' rule allows us to compute the single term $P(B|A)$ in terms of $P(A|B)$, $P(B)$, and $P(A)$. This is very useful in cases where we have a good probability of these three terms and want to determine the fourth one. Suppose we want to perceive the effect of some unknown cause, and want to compute that cause, then the Bayes' rule becomes:

$$P(\text{cause}|\text{effect}) = \frac{P(\text{effect}|\text{cause}) P(\text{cause})}{P(\text{effect})}$$

Example-1:

Question: what is the probability that a patient has diseases meningitis with a stiff neck?

Given Data:

A doctor is aware that disease meningitis causes a patient to have a stiff neck, and it occurs 80% of the time. He is also aware of some more facts, which are given as follows:

- The Known probability that a patient has meningitis disease is 1/30,000.
- The Known probability that a patient has a stiff neck is 2%.

Let a be the proposition that patient has stiff neck and b be the proposition that patient has meningitis. , so we can calculate the following as:

$$P(a|b) = 0.8$$

$$P(b) = 1/30000$$

$$P(a) = .02$$

$$P(b|a) = \frac{P(a|b)P(b)}{P(a)} = \frac{0.8 \times (\frac{1}{30000})}{0.02} = 0.001333333.$$

Hence, we can assume that 1 patient out of 750 patients has meningitis disease with a stiff neck.

Example-2:

Question: From a standard deck of playing cards, a single card is drawn. The probability that the card is king is 4/52, then calculate posterior probability $P(\text{King}|\text{Face})$, which means the drawn face card is a king card.

Solution:

$$P(\text{king}|\text{face}) = \frac{P(\text{Face}|\text{king}) \times P(\text{King})}{P(\text{Face})} \quad \dots\dots(i)$$

$P(\text{king})$: probability that the card is King= 4/52= 1/13

$P(\text{face})$: probability that a card is a face card= 3/13

$P(\text{Face}|\text{King})$: probability of face card when we assume it is a king = 1

Putting all values in equation (i) we will get:

$$P(\text{king}|\text{face}) = \frac{1 * (\frac{1}{13})}{(\frac{3}{13})} = 1/3, \text{ it is a probability that a face card is a king card.}$$

Application of Bayes' theorem in Artificial intelligence:

Following are some applications of Bayes' theorem:

- It is used to calculate the next step of the robot when the already executed step is given.
- Bayes' theorem is helpful in weather forecasting.
- It can solve the Monty Hall problem.

3. Probabilistic reasoning

Probabilistic reasoning:

Probabilistic reasoning is a way of knowledge representation where we apply the concept of probability to indicate the uncertainty in knowledge. In probabilistic reasoning, we combine probability theory with logic to handle the uncertainty.

We use probability in probabilistic reasoning because it provides a way to handle the uncertainty that is the result of someone's laziness and ignorance.

In the real world, there are lots of scenarios, where the certainty of something is not confirmed, such as "It will rain today," "behavior of someone for some situations," "A match between two teams or two players." These are probable sentences for which we can assume that it will happen but not sure about it, so here we use probabilistic reasoning.

Need of probabilistic reasoning in AI:

- When there are unpredictable outcomes.
- When specifications or possibilities of predicates becomes too large to handle.
- When an unknown error occurs during an experiment.

In probabilistic reasoning, there are two ways to solve problems with uncertain knowledge:

- **Bayes' rule**
- **Bayesian Statistics**

As probabilistic reasoning uses probability and related terms, so before understanding probabilistic reasoning, let's understand some common terms:

Probability: Probability can be defined as a chance that an uncertain event will occur. It is the numerical measure of the likelihood that an event will occur. The value of probability always remains between 0 and 1 that represent ideal uncertainties.

0 ≤ P(A) ≤ 1, where P(A) is the probability of an event A.

P(A) = 0, indicates total uncertainty in an event A.

P(A) = 1, indicates total certainty in an event A.

We can find the probability of an uncertain event by using the below formula.

$$\text{Probability of occurrence} = \frac{\text{Number of desired outcomes}}{\text{Total number of outcomes}}$$

- $P(\neg A)$ = probability of a not happening event.
- $P(\neg A) + P(A) = 1$.

Event: Each possible outcome of a variable is called an event.

Sample space: The collection of all possible events is called sample space.

Random variables: Random variables are used to represent the events and objects in the real world.

Prior probability: The prior probability of an event is probability computed before observing new information.

Posterior Probability: The probability that is calculated after all evidence or information has taken into account. It is a combination of prior probability and new information.

Conditional probability:

Conditional probability is a probability of occurring an event when another event has already happened.

Let's suppose, we want to calculate the event A when event B has already occurred, "the probability of A under the conditions of B", it can be written as:

$$P(A|B) = \frac{P(A \wedge B)}{P(B)}$$

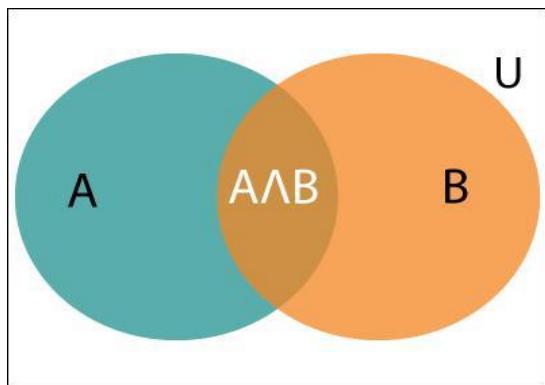
Where $P(A \wedge B)$ = Joint probability of A and B

$P(B)$ = Marginal probability of B.

If the probability of A is given and we need to find the probability of B, then it will be given as:

$$P(B|A) = \frac{P(A \wedge B)}{P(A)}$$

It can be explained by using the below Venn diagram, where B is occurred event, so sample space will be reduced to set B, and now we can only calculate event A when event B is already occurred by dividing the probability of **P(A \wedge B)** by **P(B)**.

**Example:**

In a class, there are 70% of the students who like English and 40% of the students who like English and mathematics, and then what is the percent of students those who like English also like mathematics?

Solution:

Let, A is an event that a student likes Mathematics

B is an event that a student likes English.

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{0.4}{0.7} = 57\%$$

Hence, 57% are the students who like English also like Mathematics.

 EnggTree.com

4. Bayesian networks or Belief networks

Bayesian Belief Network in artificial intelligence

Bayesian belief network is key computer technology for dealing with probabilistic events and to solve a problem which has uncertainty. We can define a Bayesian network as:

"A Bayesian network is a probabilistic graphical model which represents a set of variables and their conditional dependencies using a directed acyclic graph."

It is also called a **Bayes network**, **belief network**, **decision network**, or **Bayesian model**.

Bayesian networks are probabilistic, because these networks are built from a **probability distribution**, and also use probability theory for prediction and anomaly detection.

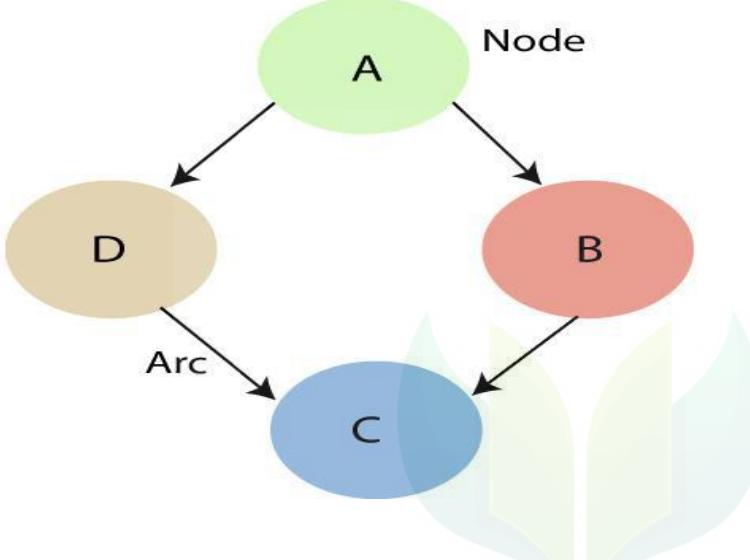
Real world applications are probabilistic in nature, and to represent the relationship between multiple events, we need a Bayesian network. It can also be used in various tasks including **prediction**, **anomaly detection**, **diagnostics**, **automated insight**, **reasoning**, **time series prediction**, and **decision making under uncertainty**.

Bayesian Network can be used for building models from data and experts opinions, and it consists of two parts:

- **Directed Acyclic Graph**
- **Table of conditional probabilities.**

The generalized form of Bayesian network that represents and solve decision problems under uncertain knowledge is known as an **Influence diagram**.

A Bayesian network graph is made up of nodes and Arcs (directed links), where:



- Each **node** corresponds to the random variables, and a variable can be **continuous** or **discrete**.
- **Arc or directed arrows** represent the causal relationship or conditional probabilities between random variables. These directed links or arrows connect the pair of nodes in the graph. These links represent that one node directly influence the other node, and if there is no directed link that means that nodes are independent with each other
 - In the above diagram, A, B, C, and D are random variables represented by the nodes of the network graph.
 - If we are considering node B, which is connected with node A by a directed arrow, then node A is called the parent of Node B.
 - Node C is independent of node A.

Note: The Bayesian network graph does not contain any cyclic graph. Hence, it is known as a directed acyclic graph or DAG

The Bayesian network has mainly two components:

- **Causal Component**
- **Actual numbers**

Each node in the Bayesian network has condition probability distribution $P(X_i | \text{Parent}(X_i))$, which determines the effect of the parent on that node.

Bayesian network is based on Joint probability distribution and conditional probability. So let's first understand the joint probability distribution:

Joint probability distribution:

If we have variables $x_1, x_2, x_3, \dots, x_n$, then the probabilities of a different combination of $x_1, x_2, x_3, \dots, x_n$, are known as Joint probability distribution.

$P[x_1, x_2, x_3, \dots, x_n]$, it can be written as the following way in terms of the joint probability distribution.

$$= P[x_1 | x_2, x_3, \dots, x_n] P[x_2, x_3, \dots, x_n]$$

$$= P[x_1 | x_2, x_3, \dots, x_n] P[x_2 | x_3, \dots, x_n] \dots P[x_{n-1} | x_n] P[x_n].$$

In general for each variable X_i , we can write the equation as:

$$P(X_i | X_{i-1}, \dots, X_1) = P(X_i | \text{Parents}(X_i))$$

Explanation of Bayesian network:

Let's understand the Bayesian network through an example by creating a directed acyclic graph:

Example: Harry installed a new burglar alarm at his home to detect burglary. The alarm reliably responds at detecting a burglary but also responds for minor earthquakes. Harry has two neighbors David and Sophia, who have taken a responsibility to inform Harry at work when they hear the alarm. David always calls Harry when he hears the alarm, but sometimes he got confused with the phone ringing and calls at that time too. On the other hand, Sophia likes to listen to high music, so sometimes she misses to hear the alarm. Here we would like to compute the probability of Burglary Alarm.

Problem:

Calculate the probability that alarm has sounded, but there is neither a burglary, nor an earthquake occurred, and David and Sophia both called the Harry.

Solution:

- The Bayesian network for the above problem is given below. The network structure is showing that burglary and earthquake is the parent node of the alarm and directly affecting the probability of alarm's going off, but David and Sophia's calls depend on alarm probability.
- The network is representing that our assumptions do not directly perceive the burglary and also do not notice the minor earthquake, and they also not confer before calling.
- The conditional distributions for each node are given as conditional probabilities table or CPT.
- Each row in the CPT must be sum to 1 because all the entries in the table represent an exhaustive set of cases for the variable.
- In CPT, a boolean variable with k boolean parents contains 2^k probabilities. Hence, if there are two parents, then CPT will contain 4 probability values

List of all events occurring in this network:

- **Burglary (B)**
- **Earthquake(E)**
- **Alarm(A)**
- **David Calls(D)**

- **Sophia calls(S)**

We can write the events of problem statement in the form of probability: $P[D, S, A, B, E]$, can rewrite the above probability statement using joint probability distribution:

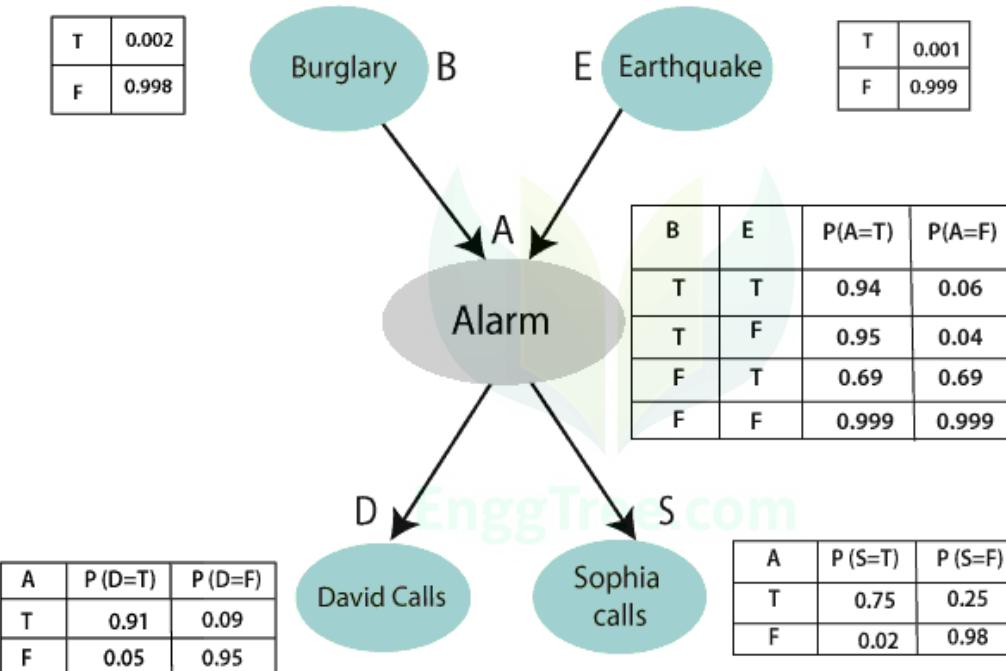
$$P[D, S, A, B, E] = P[D | S, A, B, E] \cdot P[S, A, B, E]$$

$$= P[D | S, A, B, E] \cdot P[S | A, B, E] \cdot P[A, B, E]$$

$$= P[D | A] \cdot P[S | A, B, E] \cdot P[A, B, E]$$

$$= P[D | A] \cdot P[S | A] \cdot P[A | B, E] \cdot P[B, E]$$

$$= P[D | A] \cdot P[S | A] \cdot P[A | B, E] \cdot P[B | E] \cdot P[E]$$



Let's take the observed probability for the Burglary and earthquake component:

$P(B = \text{True}) = 0.002$, which is the probability of burglary.

$P(B = \text{False}) = 0.998$, which is the probability of no burglary.

$P(E = \text{True}) = 0.001$, which is the probability of a minor earthquake

$P(E = \text{False}) = 0.999$, Which is the probability that an earthquake not occurred.

We can provide the conditional probabilities as per the below tables:

Conditional probability table for Alarm A:

The Conditional probability of Alarm A depends on Burglar and earthquake:

B	E	P(A= True)	P(A= False)
True	True	0.94	0.06
True	False	0.95	0.04
False	True	0.31	0.69
False	False	0.001	0.999

Conditional probability table for David Calls:

The Conditional probability of David that he will call depends on the probability of Alarm.

A	P(D= True)	P(D= False)
True	0.91	0.09
False	0.05	0.95

Conditional probability table for Sophia Calls:

The Conditional probability of Sophia that she calls is depending on its Parent Node "Alarm."

A	P(S= True)	P(S= False)
True	0.75	0.25
False	0.02	0.98

From the formula of joint distribution, we can write the problem statement in the form of probability distribution:

$$\begin{aligned}
 P(S, D, A, \neg B, \neg E) &= P(S|A) * P(D|A) * P(A|\neg B \wedge \neg E) * P(\neg B) * P(\neg E) \\
 &= 0.75 * 0.91 * 0.001 * 0.998 * 0.999 \\
 &= \mathbf{0.00068045}.
 \end{aligned}$$

Hence, a Bayesian network can answer any query about the domain by using Joint distribution.

The semantics of Bayesian Network:

There are two ways to understand the semantics of the Bayesian network, which is given below:

1. To understand the network as the representation of the Joint probability distribution.

It is helpful to understand how to construct the network.

2. To understand the network as an encoding of a collection of conditional independence statements.

It is helpful in designing inference procedure.

5. Inference in Bayesian Networks

1. Exact inference
2. Approximate inference

1. Exact inference:

In exact inference, we analytically compute the conditional probability distribution over the variables of interest.

But sometimes, that's too hard to do, in which case we can use approximation techniques based on statistical sampling

Given a Bayesian network, what questions might we want to ask?

- Conditional probability query: $P(x | e)$
- Maximum a posteriori probability: What value of x maximizes $P(x|e)$?

General question: What's the whole probability distribution over variable X given evidence e , $P(X | e)$?

In our discrete probability situation, the only way to answer a MAP query is to compute the probability of x given e for all possible values of x and see which one is greatest

So, in general, we'd like to be able to compute a whole probability distribution over some variable or variables X , given instantiations of a set of variables e

Using the joint distribution

EnggTree.com

To answer any query involving a conjunction of variables, sum over the variables not involved in the query

Given the joint distribution over the variables, we can easily answer any question about the value of a single variable by summing (or marginalizing) over the other variables.

$$\begin{aligned} \Pr(d) &= \sum_{ABC} \Pr(a,b,c,d) \\ &= \sum_{a \in \text{dom}(A)} \sum_{b \in \text{dom}(B)} \sum_{c \in \text{dom}(C)} \Pr(A = a \wedge B = b \wedge C = c) \end{aligned}$$

So, in a domain with four variables, A, B, C, and D, the probability that variable D has value d is the sum over all possible combinations of values of the other three variables of the joint probability of all four values. This is exactly the same as the procedure we went through in the last lecture, where to compute the probability of cavity, we added up the probability of cavity and toothache and the probability of cavity and not toothache.

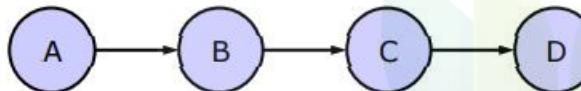
In general, we'll use the first notation, with a single summation indexed by a list of variable names, and a joint probability expression that mentions values of those variables. But here we can see the completely written-out definition, just so we all know what the shorthand is supposed to mean.

$$\Pr(d | b) = \frac{\Pr(b, d)}{\Pr(b)} = \frac{\sum_{AC} \Pr(a, b, c, d)}{\sum_{ACD} \Pr(a, b, c, d)}$$

To compute a conditional probability, we reduce it to a ratio of conjunctive queries using the definition of conditional probability, and then answer each of those queries by marginalizing out the variables not mentioned.

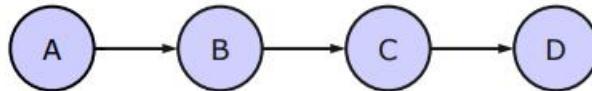
In the numerator, here, you can see that we're only summing over variables A and C, because b and d are instantiated in the query.

Simple Case



We're going to learn a general purpose algorithm for answering these joint queries fairly efficiently. We'll start by looking at a very simple case to build up our intuitions, then we'll write down the algorithm, then we'll apply it to a more complex case.

Here's our very simple case. It's a bayes net with four nodes, arranged in a chain.



$$\begin{aligned}
 \Pr(d) &= \sum_{ABC} \Pr(a, b, c, d) \\
 &= \sum_{ABC} \Pr(d | c) \Pr(c | b) \Pr(b | a) \Pr(a) \\
 &= \sum_C \sum_B \sum_A \Pr(d | c) \Pr(c | b) \Pr(b | a) \Pr(a) \\
 &= \sum_C \Pr(d | c) \sum_B \Pr(c | b) \sum_A \Pr(b | a) \Pr(a)
 \end{aligned}$$

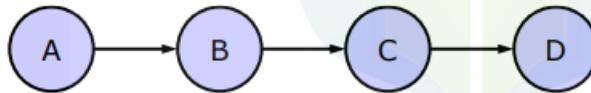
So, we know from before that the probability that variable D has some value little d is the sum over A, B, and C of the joint distribution, with d fixed.

Now, using the chain rule of Bayesian networks, we can write down the joint probability as a product over the nodes of the probability of each node's value given the values of its parents. So, in this case, we get $P(d|c)$ times $P(c|b)$ times $P(b|a)$ times $P(a)$.

This expression gives us a method for answering the query, given the conditional probabilities that are stored in the net. And this method can be applied directly to any other bayes net. But there's a problem with it: it requires enumerating all possible combinations of assignments to A, B, and C, and then, for each one, multiplying the factors for each node. That's an enormous amount of work and we'd like to avoid it if at all possible.

So, we'll try rewriting the expression into something that might be more efficient to evaluate. First, we can make our summation into three separate summations, one over each variable.

Then, by distributivity of addition over multiplication, we can push the summations in, so that the sum over A includes all the terms that mention A, but no others, and so on. It's pretty clear that this expression is the same as the previous one in value, but it can be evaluated more efficiently. We're still, eventually, enumerating all assignments to the three variables, but we're doing somewhat fewer multiplications than before. So this is still not completely satisfactory.



$$\Pr(d) = \sum_c \Pr(d | c) \sum_b \Pr(c | b) \underbrace{\sum_a \Pr(b | a) \Pr(a)}_{\begin{bmatrix} \Pr(b_1 | a_1) \Pr(a_1) & \Pr(b_1 | a_2) \Pr(a_2) \\ \Pr(b_2 | a_1) \Pr(a_1) & \Pr(b_2 | a_2) \Pr(a_2) \end{bmatrix}}$$

If you look, for a minute, at the terms inside the summation over A, you'll see that we're doing these multiplications over for each value of C, which isn't necessary, because they're independent of C. Our idea, here, is to do the multiplications once and store them for later use. So, first, for each value of A and B, we can compute the product, generating a two dimensional matrix.

$$\Pr(d) = \sum_C \Pr(d | c) \sum_B \Pr(c | b) \sum_A \Pr(b | a) \Pr(a)$$

A
 ↓
 $\left[\begin{array}{l} \sum_A \Pr(b_1 | a) \Pr(a) \\ \sum_A \Pr(b_2 | a) \Pr(a) \end{array} \right]$

Then, we can sum over the rows of the matrix, yielding one value of the sum for each possible value of b.

$$\Pr(d) = \sum_C \Pr(d | c) \sum_B \Pr(c | b) \sum_A \Pr(b | a) \Pr(a)$$

A
 ↓
 $f_1(b)$

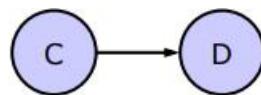
We'll call this set of values, which depends on b, f1 of b.

$$\Pr(d) = \sum_C \Pr(d | c) \sum_B \Pr(c | b) f_1(b)$$

B
 ↓
 $f_2(c)$

Now, we can substitute f1 of b in for the sum over A in our previous expression. And, effectively, we can remove node A from our diagram. Now, we express the contribution of b, which takes the contribution of a into account, as f_1 of b.

We can continue the process in basically the same way. We can look at the summation over b and see that the only other variable it involves is c. We can summarize those products as a set of factors, one for each value of c. We'll call those factors f_2 of c.



$$\Pr(d) = \sum_C \Pr(d | c) f_2(c)$$

We substitute f_2 of c into the formula, remove node b from the diagram, and now we're down to a simple expression in which d is known and we have to sum over values of c.

Variable Elimination Algorithm

Given a Bayesian network, and an elimination order for the non-query variables , compute

$$\sum_{X_1} \sum_{X_2} \dots \sum_{X_m} \prod_j \Pr(x_j | Pa(x_j))$$

For i = m downto 1

- remove all the factors that mention X_i
- multiply those factors, getting a value for each combination of mentioned variables
- sum over X_i
- put this new factor into the factor set

That was a simple special case. Now we can look at the algorithm in the general case. Let's assume that we're given a Bayesian network and an ordering on the variables that aren't fixed in the query. We'll come back later to the question of the influence of the order, and how we might find a good one.

We can express the probability of the query variables as a sum over each value of each of the non-query variables of a product over each node in the network, of the probability that that variable has the given value given the values of its parents.

So, we'll eliminate the variables from the inside out. Starting with variable X_m and finishing with variable X_1 .

To eliminate variable X_i , we start by gathering up all of the factors that mention X_i , and removing them from our set of factors. Let's say there are k such factors.

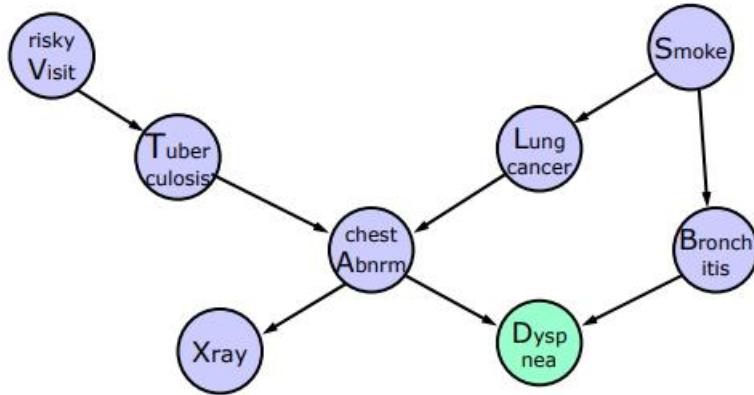
Now, we make a $k+1$ dimensional table, indexed by X_i as well as each of the other variables that is mentioned in our set of factors.

We then sum the table over the X_i dimension, resulting in a k -dimensional table.

This table is our new factor, and we put a term for it back into our set of factors.

Once we've eliminated all the summations, we have the desired value.

One more example



$$\Pr(d) = \sum_{A,B,L,T,S,X,V} \frac{\Pr(d | a,b) \Pr(a | t,l) \Pr(b | s) \Pr(l | s) \Pr(s)}{\Pr(x | a) \Pr(t | v) \Pr(v)}$$

$$\Pr(d) = \sum_{A,B,L,T,S,X} \frac{\Pr(d | a,b) \Pr(a | t,l) \Pr(b | s) \Pr(l | s) \Pr(s)}{\Pr(x | a) \underbrace{\sum_{V} \Pr(t | v) \Pr(v)}_{f_1(t)}}$$

$$\Pr(d) = \sum_{A,B,L,T,S,X} \frac{\Pr(d | a,b) \Pr(a | t,l) \Pr(b | s) \Pr(l | s) \Pr(s)}{\Pr(x | a) f_1(t)}$$

$$\Pr(d) = \sum_{A,B,L,T,S} \underbrace{\Pr(x | a)}_{1} \frac{\Pr(d | a,b) \Pr(a | t,l) \Pr(b | s) \Pr(l | s) \Pr(s) f_1(t)}{\Pr(s)}$$

$$\Pr(d) = \sum_{A,B,L,T,S} \Pr(d | a,b) \Pr(a | t,l) \Pr(b | s) \Pr(l | s) \Pr(s) f_1(t)$$

$$\Pr(d) = \sum_{A,B,L,T} \Pr(d | a,b) \Pr(a | t,l) f_1(t) \underbrace{\sum_S \Pr(b | s) \Pr(l | s) \Pr(s)}_{f_2(b,l)}$$

$$\Pr(d) = \sum_{A,B,L,T} \Pr(d | a,b) \Pr(a | t,l) f_1(t) \underbrace{\sum_S \Pr(b | s) \Pr(l | s) \Pr(s)}_{f_2(b,l)}$$

$$\Pr(d) = \sum_{A,B,L,T} \Pr(d | a,b) \Pr(a | t,l) f_1(t) f_2(b,l)$$

$$\Pr(d) = \sum_{A,B,L} \Pr(d | a,b) f_1(b,l) \underbrace{\sum_T \Pr(a | t,l) f_1(t)}_{f_3(a,l)}$$

$$\Pr(d) = \sum_{A,B,L} \Pr(d | a,b) f_2(b,l) f_3(a,l)$$

Here's a more complicated example, to illustrate the variable elimination algorithm in a more general case. We have this big network that encodes a domain for diagnosing lung disease. (Dyspnea, as I understand it, is shortness of breath).

We'll do variable elimination on this graph using elimination order A, B, L, T, S, X, V.

So, we start by eliminating V. We gather the two terms that mention V and see that they also involve variable T. So, we compute the product for each value of T, and summarize those in the factor f_1 of T.

Now we can substitute that factor in for the summation, and remove the node from the network.

The next variable to be eliminated is X. There is actually only one term involving X, and it also involves variable A. So, for each value of A, we compute the sum over X of $P(x|a)$. But wait! We know what this value is! If we fix a and sum over x, these probabilities have to add up to 1.

So, rather than adding another factor to our expression, we can just remove the whole sum. In general, the only nodes that will have an influence on the probability of D are its ancestors.

Now, it's time to eliminate S. We find that there are three terms involving S, and we gather them into the sum. These three terms involve two other variables, B and L. So we have to make a factor that specifies, for each value of B and L, the value of the sum of products.

We'll call that factor f_2 of b and l.

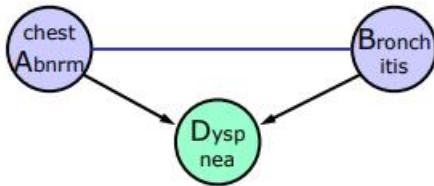
Now we can substitute that factor back into our expression. We can also eliminate node S. But in eliminating S, we've added a direct dependency between L and B (they used to be dependent via S, but now the dependency is encoded explicitly in $f_2(b)$). We'll show that in the graph by drawing a line between the two nodes. It's not exactly a standard directed conditional dependence, but it's still useful to show that they're coupled.

Now we eliminate T. It involves two terms, which themselves involve variables A and L. So we make a new factor f_3 of A and L.

We can substitute in that factor and eliminate T. We're getting close!

$$\Pr(d) = \sum_{A,B} \Pr(d | a,b) \underbrace{\sum_L f_2(b,l) f_3(a,l)}_{f_4(a,b)}$$

Next we eliminate L. It involves these two factors, which depend on variables A and B. So we make a new factor, f4 of A and B, and substitute it in. We remove node L, but couple A and B.



$$\Pr(d) = \sum_{A,B} \Pr(d | a,b) f_4(a,b)$$

At this point, we could just do the summations over A and B and be done. But to finish out the algorithm the way a computer would, it's time to eliminate variable B.

$$\Pr(d) = \sum_A \underbrace{\sum_B \Pr(d | a,b) f_4(a,b)}_{f_5(a)}$$

It involves both of our remaining terms, and it seems to depend on variables A and D. However, in this case, we're interested in the probability of a particular value, little d of D, and so the variable d is instantiated. Thus, we can treat it as a constant in this expression, and we only need to generate a factor over a, which we'll call f5 of a. And we can now, in some sense, remove D from our network as well (because we've already factored it into our answer).



EnggTree.com

$$\Pr(d) = \sum_A f_5(a)$$

Finally, to get the probability that variable D has value little d, we simply sum factor f5 over all values of a. Yay! We did it.

Properties of Variable Elimination

Let's see how the variable elimination algorithm performs, both in theory and in practice.

- Time is exponential in size of largest factor
- Bad elimination order can generate huge factors
- NP Hard to find the best elimination order

- Even the best elimination order may generate large factors
- There are reasonable heuristics for picking an elimination order (such as choosing the variable that results in the smallest next factor)
- Inference in polytrees (nets with no cycles) is linear in size of the network (the largest CPT)
- Many problems with very large nets have only small factors, and thus efficient inference

First of all, it's pretty easy to see that it runs in time exponential in the number of variables involved in the largest factor. Creating a factor with k variables involves making a $k+1$ dimensional table. If you have b values per variable, that's a table of size b^{k+1} . To make each entry, you have to multiply at most n numbers, where n is the number of nodes. We have to do this for each variable to be eliminated (which is usually close to n). So we have something like time = $O(n^2 b^k)$.

How big the factors are depends on the elimination order. You'll see in one of the recitation exercises just how dramatic the difference in factor sizes can be. A bad elimination order can generate huge factors.

So, we'd like to use the elimination order that generates the smallest factors. Unfortunately, it turns out to be NP hard to find the best elimination order.

At least, there are some fairly reasonable heuristics for choosing an elimination order. It's usually done dynamically. So, rather than fixing the elimination order in advance, as we suggested in the algorithm description, you can pick the next variable to be eliminated depending on the situation. In particular, one reasonable heuristic is to pick the variable to eliminate next that will result in the smallest factor. This greedy approach won't always be optimal, but it's not usually too bad.

There is one case where Bayes net inference in general, and the variable elimination algorithm in particular is fairly efficient, and that's when the network is a polytree. A polytree is a network with no cycles. That is, a network in which, for any two nodes, there is only one path between them. In a polytree, inference is linear in the size of the network, where the size of the network is defined to be the size of the largest conditional probability table (or exponential in the maximum number of parents of any node). In a polytree, the optimal elimination order is to start at the root nodes, and work downwards, always eliminating a variable that no longer has any parents. In doing so, we never introduce additional connections into the network.

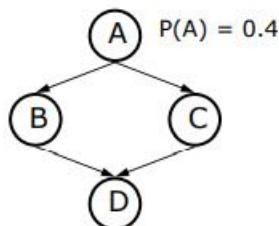
So, inference in polytrees is efficient, and even in many large non-polytree networks, it's possible to keep the factors small, and therefore to do inference relatively efficiently.

When the network is such that the factors are, of necessity, large, we'll have to turn to a different class of methods.

2. Approximate inference:

Sampling

To get approximate answer we can do stochastic simulation (sampling).



A	B	C	D
T	F	T	T
...			

- Flip a coin where $P(T)=0.4$, assume we get T, use that value for A
- Given $A=T$, lookup $P(B|A=T)$ and flip a coin with that prob., assume we get F
- Similarly for C and D
- We get one sample from joint distribution of these four vars

Another strategy, which is a theme that comes up also more and more in AI actually, is to say, well, we didn't really want the right answer anyway. Let's try to do an approximation. And you can also show that it's computationally hard to get an approximation that's within epsilon of the answer that you want, but again that doesn't keep us from trying.

So, the other thing that we can do is the stochastic simulation or sampling. In sampling, what we do is we look at the root nodes of our graph, and attached to this root node is some probability that A is going to be true, right? Maybe it's .4. So we flip a coin that comes up heads with probability .4 and see if we get true or false.

We flip our coin, let's say, and we get true for A -- this time. And now, given the assignment of true to A, we look in the conditional probability table for B given A = true, and that gives us a probability for B.

Now, we flip a coin with that probability. Say we get False. We enter that into the table.

We do the same thing for C, and let's say we get True.

Now, we look in the CPT for D given B and C, for the case where B is false and C is true, and we flip a coin with that probability, in order to get a value for D.

So, there's one sample from the joint distribution of these four variables. And you can just keep doing this, all day and all night, and generate a big pile of samples, using that algorithm. And now you can ask various questions.

Estimate:

$$P^*(D|A) = \#D,A / \#A$$

Let's say you want to know the probability of D given A. How would you answer -- given all the examples -- what would you do to compute the probability of D given A? You would just count. You'd count the number of cases in which A and D were true, and you'd divide that by the number of cases in

which A was true, and that would give you an unbiased estimate of the probability of D given A. The more samples, the more confidence you'd have that the estimated probability is close to the true one.

Estimation

- Some probabilities are easier than others to estimate
- In generating the table, the rare events will not be well represented
- $P(\text{Disease} \mid \text{spots-on-your-tongue, sore toe})$
- If spots-on-your-tongue and sore toe are not root nodes, you would generate a huge table but the cases of interest would be very sparse in the table
- Importance sampling lets you focus on the set of cases that are important to answering your question

It's going to turn out that some probabilities are easier than other ones to estimate.

Exactly because of the process we're using to generate the samples, the majority of them will be the typical cases. Oh, it's someone with a cold, someone with malaria, someone with a cold, someone with a cold. So the rare results are not going to come up very often. And so doing this sampling naively can make it really hard to estimate the probability of a rare event. If it's something that happens one in ten thousand times, well, you know for sure you're going to need, some number of tens of thousands of samples to get even a reasonable estimate of that probability.

Imagine that you want to estimate the probability of some disease given -- oh, I don't know -- spots on your tongue and a sore toe. Somebody walks in and they have a really peculiar set of symptoms, and you want to know what's the probability that they have some disease.

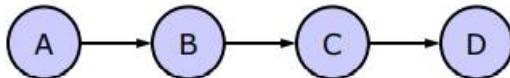
Well, if the symptoms are root nodes, it's easy. If the symptoms were root nodes, you could just assign the root nodes to have their observed values and then simulate the rest of the network as before.

But if the symptoms aren't root nodes then if you do naïve sampling, you would generate a giant table of samples, and you'd have to go and look and say, gosh, how many cases do I have where somebody has spots on their tongue and a sore toe; and the answer would be, well, maybe zero or not very many.

There's a technique called importance sampling, which allows you to draw examples from a distribution that's going to be more helpful and then reweight them so that you can still get an unbiased estimate of the desired conditional probability. It's a bit beyond the scope of this class to get into the details, but it's an important and effective idea.

Recitation Problem

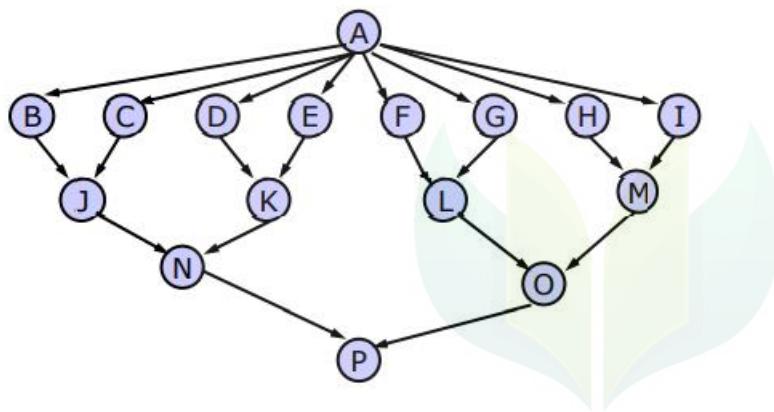
- Do the variable elimination algorithm on the net below using the elimination order A,B,C (that is, eliminate node C first). In computing $P(D=d)$, what factors do you get?
- What if you wanted to compute the whole marginal distribution $P(D)$?



Here's the network we started with. We used elimination order C, B, A (we eliminated A first). Now we're going to explore what happens when we eliminate the variables in the opposite order. First, work on the case we did, where we're trying to calculate the probability that node D takes on a particular value, little d. Remember that little d is a constant in this case. Now, do the case where we're trying to find the whole distribution over D, so we don't know a particular value for little d.

Another Recitation Problem

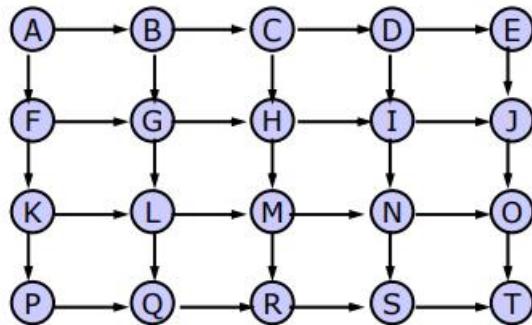
Find an elimination order that keeps the factors small for the net below, or show that there is no such order.



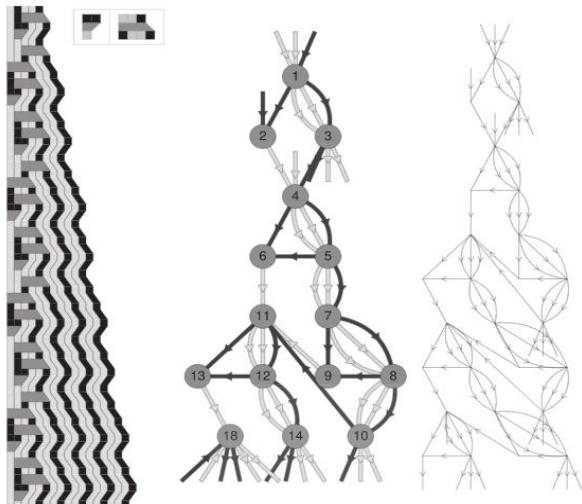
Here's a pretty complicated graph. But notice that no node has more than 2 parents, so none of the CPTs are huge. The question is, is this graph hard for variable elimination? More concretely, can you find an elimination order that results only in fairly small factors? Is there an elimination order that generates a huge factor?

The Last Recitation Problem

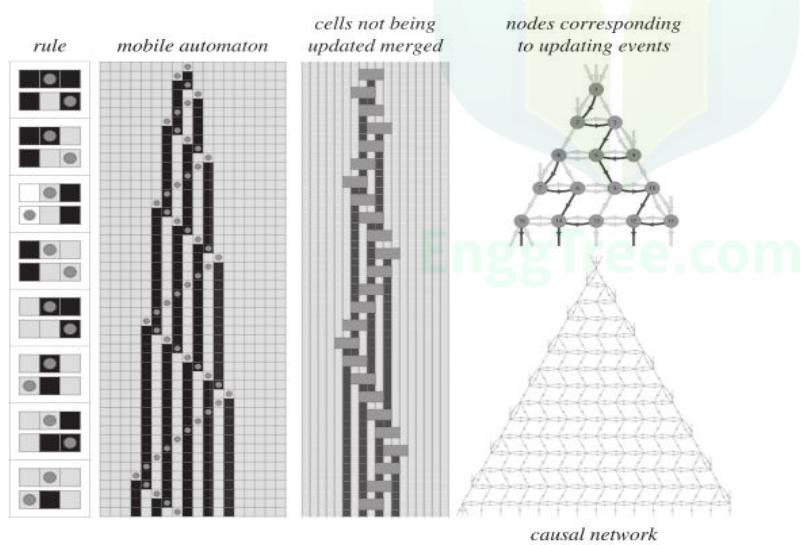
Bayesian networks (or related models) are often used in computer vision, but they almost always require sampling. What happens when you try to do variable elimination on a model like the grid below?



6. Casual Networks:



A causal network is an acyclic digraph arising from an evolution of a substitution system, and representing its history. The illustration above shows a causal network corresponding to the rules $\{B B \rightarrow A, A A B \rightarrow B A A B\}$ (applied in a left-to-right scan) and initial condition $A B A A B$.



The figure above shows the procedure for diagrammatically creating a causal network from a [mobile automaton](#).

In an evolution of a [multiway system](#), each substitution event is a vertex in a causal network. Two events which are related by causal dependence, meaning one occurs just before the other, have an edge between the corresponding vertices in the causal network. More precisely, the edge is a directed edge leading from the past event to the future event.

Some causal networks are independent of the choice of evolution, and these are called causally invariant.