Course Material

ARTIFICIAL INTELLIGENCE

Prepared by

D.Muthukumaran,
Assistant Professor/ECE.

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SRI CHANDRASEKHARENDRA SARASWATHI VISWA
MAHAVIDYALAYA
(Deemed to be University U/S 3 of UGC Act 1956)
Accredited with "A" Grade by NAAC
Enathur, Kanchipuram – 631 561

Syllabus (Open Elective -III)-VII Semester)

UNIT I

Introduction – Foundations of AI, the History of AI –Intelligent Agent –Agent and Environment, Good Behaviour: The Concept of Rationality, Nature of Environments, Structure of Agents- Problem Solving Agents -Example Problems.

UNIT II

Uninformed Searching strategies-Breadth First Search, Depth First search, Depth limited search, Iterative deepening search, Bidirectional Search - Avoiding repeated States - Searching with Partial information –Informed search strategies – Greedy Best First Search-A* Search-Heuristic Functions- Local Search Algorithms for Optimization Problems-Local search in Continuous Spaces.

UNIT III

Online Search Agents and Unknown Environments-Online Search Problems, Online Search Agents- Online Local search, learning in Online Search – Constraint Satisfaction Problems- Backtracking CSP, The Structure of Problems-Adversarial Search-Games, Optimal Decisions in Games, Alpha- Beta Pruning.

UNIT IV

Logical agents – Knowledge Based Agents, The Wumpus World, Propositional Logic-A very simple Logic –First Order logic– inferences in first order logic – forward chaining – backward chaining – Unification – Resolution.

UNIT V

Planning with state space search – Partial-order planning – Planning graphs – Planning and acting in the real world.

TEXT BOOKS:

1. S. Russel and P. Norvig, “Artificial Intelligence –A Modern Approach”, Second Edition, Pearson Education 2003.

REFERENCES:

1. David Poole, Alan Mackworth, Randy Goebel, “Computational Intelligence: a Logical Approach”, Oxford University Press, 2004.

2. G. Luger, “Artificial Intelligence: Structures and Strategies for Complex Problem Solving”, Fourth Edition, Pearson Education, 2002.

UNIT-1

Introduction – Foundations of AI, the History of AI –Intelligent Agent – Agent and Environment, Good Behaviour: The Concept of Rationality, Nature of Environments, Structure of Agents- Problem Solving Agents -Example Problems.

AIM & OBJECTIVES

To understand some fundamentals of AI and algorithms required to produce AI systems.

PRE- REQUISITE: Basic knowledge of Computer Architecture.

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 future. AI holds a tendency to cause a machine to work as a human.

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:

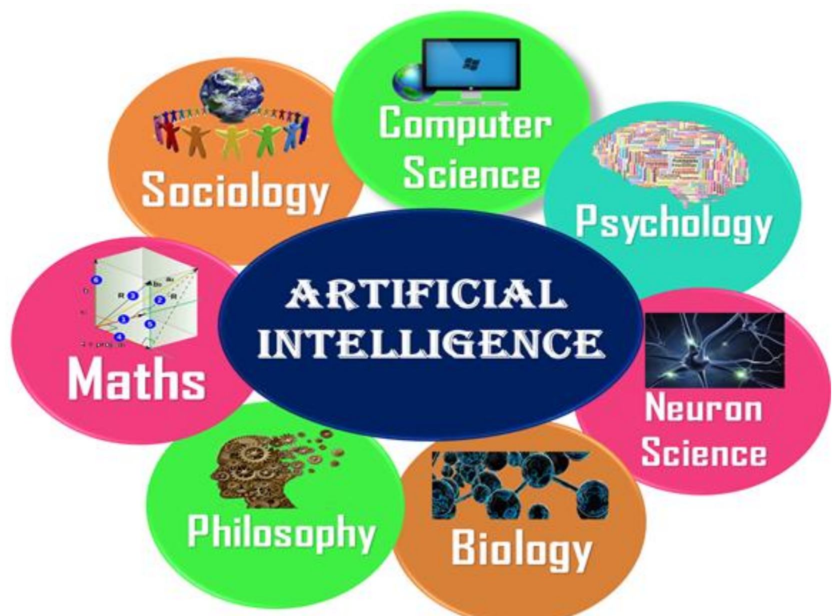
1. Replicate human intelligence
2. Solve Knowledge-intensive tasks
3. An intelligent connection of perception and action
4. Building a machine which can perform tasks that requires human intelligence such as:
    - Proving a theorem
    - Playing chess
    - Plan some surgical operation
    - Driving 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 error: 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.

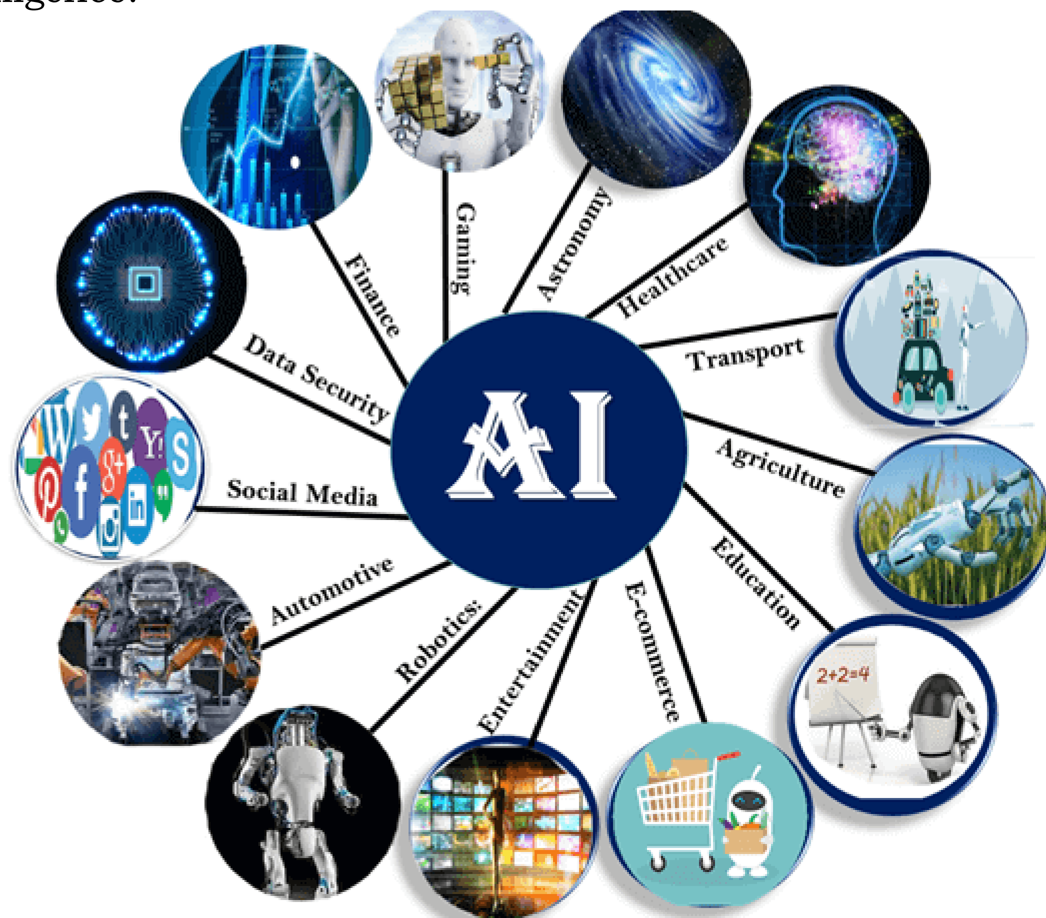
- Increase dependency on machines: With the increment of technology, people are getting more dependent on devices and hence they are losing their mental capabilities.
- No Original Creativity: As humans are so creative and can imagine some new ideas but still AI machines cannot beat this power of human intelligence and cannot be creative and imaginative.

Application of AI

Artificial Intelligence has various applications in today's society.

It is becoming essential for today's time because it can solve complex problems with an efficient way in multiple industries, such as Healthcare, entertainment, finance, education, etc.
AI is making our daily life more comfortable and fast.

Following are some sectors which have the application of Artificial Intelligence:



1. AI in Astronomy
    - Artificial Intelligence can be very useful to solve complex universe problems. AI technology can be helpful for understanding the universe such as how it works, origin, etc.

2. AI in Healthcare
    - In the last, five to ten years, AI becoming more advantageous for the healthcare industry and going to have a significant impact on this industry.

    - Healthcare Industries are applying AI to make a better and faster diagnosis than humans. AI can help doctors with diagnoses and can inform when patients are worsening so that medical help can reach to the patient before hospitalization.

3. AI in Gaming
    - AI can be used for gaming purpose. The AI machines can play strategic games like chess, where the machine needs to think of a large number of possible places.

4. AI in Finance
    - AI and finance industries are the best matches for each other. The finance industry is implementing automation, chatbot, adaptive intelligence, algorithm trading, and machine learning into financial processes.

5. AI in Data Security
    - The security of data is crucial for every company and cyber-attacks are growing very rapidly in the digital world. AI can be used to make your data more safe and secure. Some examples such as AEG bot, AI2 Platform, are used to determine software bug and cyber-attacks in a better way.

6. AI in Social Media
    - Social Media sites such as Facebook, Twitter, and Snapchat contain billions of user profiles, which need to be stored and managed in a very efficient way. AI can organize and manage massive amounts of data. AI can analyze lots of data to identify the latest trends, hashtag, and requirement of different users.

7. AI in Travel & Transport
   - AI is becoming highly demanding for travel industries. AI is capable of doing various travel related works such as from making travel arrangement to suggesting the hotels, flights, and best routes to the customers. Travel industries are using AI-powered chatbots which can make human-like interaction with customers for better and fast response.

8. AI in Automotive Industry
   - Some Automotive industries are using AI to provide virtual assistant to their user for better performance. Such as Tesla has introduced TeslaBot, an intelligent virtual assistant.

   - Various Industries are currently working for developing self-driven cars which can make your journey more safe and secure.

9. AI in Robotics:
   - Artificial Intelligence has a remarkable role in Robotics. Usually, general robots are programmed such that they can perform some repetitive task, but with the help of AI, we can create intelligent robots which can perform tasks with their own experiences without pre-programmed.

   - Humanoid Robots are best examples for AI in robotics, recently the intelligent Humanoid robot named as Erica and Sophia has been developed which can talk and behave like humans.

10. AI in Entertainment
    - We are currently using some AI based applications in our daily life with some entertainment services such as Netflix or Amazon. With the help of ML/AI algorithms, these services show the recommendations for programs or shows.

11. AI in Agriculture
    - Agriculture is an area which requires various resources, labor, money, and time for best result. Now a day's agriculture is becoming digital, and AI is emerging in this field. Agriculture is applying AI as agriculture robotics, solid and crop monitoring, predictive analysis. AI in agriculture can be very helpful for farmers.

12. AI in E-commerce

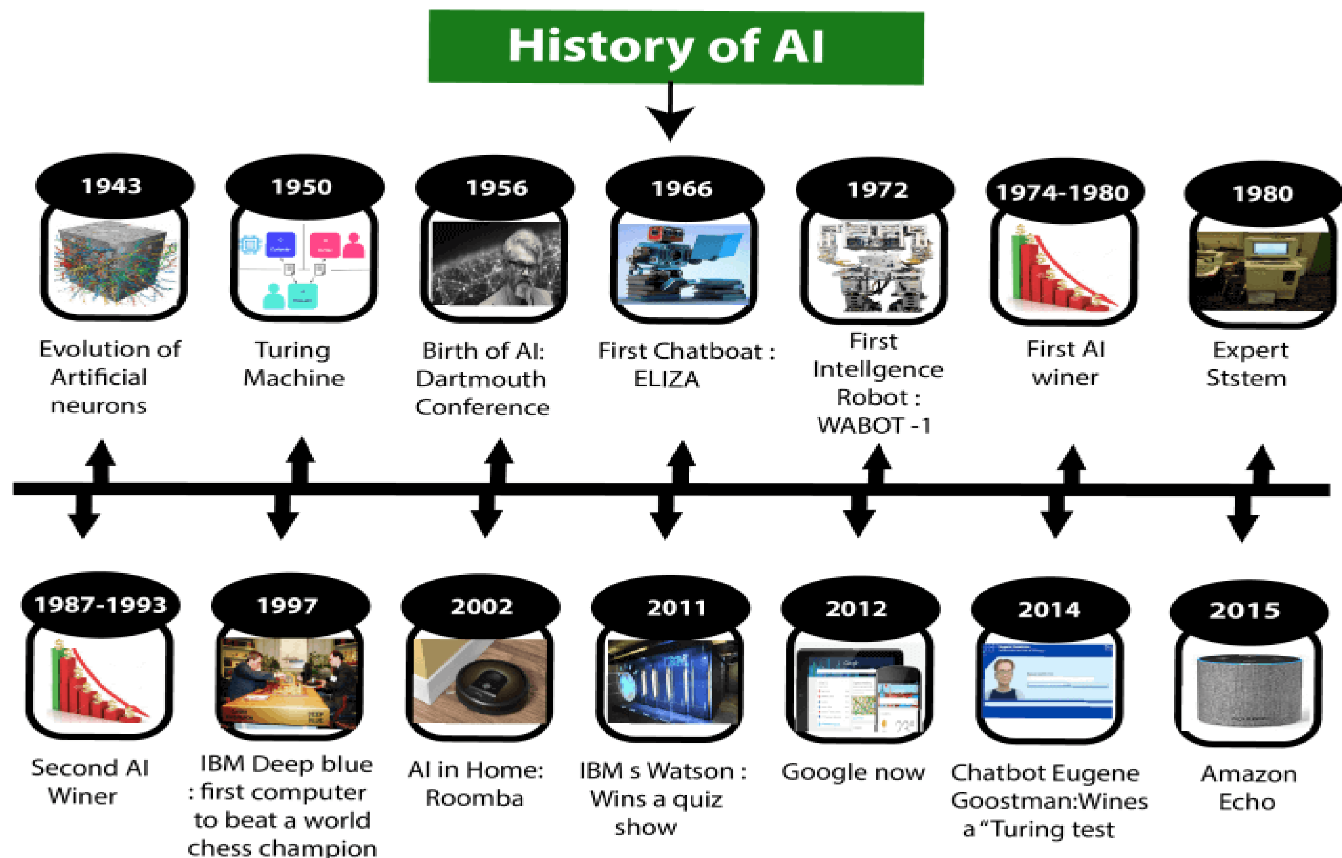
- AI is providing a competitive edge to the e-commerce industry, and it is becoming more demanding in the e-commerce business. AI is helping shoppers to discover associated products with recommended size, color, or even brand.

13. AI in education:

- AI can automate grading so that the tutor can have more time to teach. AI chatbot can communicate with students as a teaching assistant.
- AI in the future can be work as a personal virtual tutor for students, which will be accessible easily at any time and any place.

History of Artificial Intelligence

Artificial Intelligence is not a new word and not a new technology for researchers. This technology is much older than you would imagine. Even there are the myths of Mechanical men in Ancient Greek and Egyptian Myths. Following are some milestones in the history of AI which defines the journey from the AI generation to till date development.



Maturation of Artificial Intelligence (1943-1952)

- Year 1943: The first work which is now recognized as AI was done by Warren McCulloch and Walter pits in 1943. They proposed a model of artificial neurons.
- Year 1949: Donald Hebb demonstrated an updating rule for modifying the connection strength between neurons. His rule is now called Hebbian learning.
- Year 1950: The Alan Turing who was an English mathematician and pioneered Machine learning in 1950. Alan Turing publishes "Computing Machinery and Intelligence" in which he proposed a test. The test can check the machine's ability to exhibit intelligent behavior equivalent to human intelligence, called a Turing test.

The birth of Artificial Intelligence (1952-1956)

- Year 1955: An Allen Newell and Herbert A. Simon created the "first artificial intelligence program “Which was named as "Logic Theorist". This program had proved 38 of 52 Mathematics theorems, and find new and more elegant proofs for some theorems.
- Year 1956: The word "Artificial Intelligence" first adopted by American Computer scientist John McCarthy at the Dartmouth Conference. For the first time, AI coined as an academic field.

At that time high-level computer languages such as FORTRAN, LISP, or COBOL were invented. And the enthusiasm for AI was very high at that time.
The golden years-Early enthusiasm (1956-1974)

- Year 1966: The researchers emphasized developing algorithms which can solve mathematical problems. Joseph Weizenbaum created the first chatbot in 1966, which was named as ELIZA.

- Year 1972: The first intelligent humanoid robot was built in Japan which was named as WABOT-1.

The first AI winter (1974-1980)

- The duration between years 1974 to 1980 was the first AI winter duration. AI winter refers to the time period where computer scientist dealt with a severe shortage of funding from government for AI researches.

- During AI winters, an interest of publicity on artificial intelligence was decreased.

A boom of AI (1980-1987)

- Year 1980: After AI winter duration, AI came back with "Expert System". Expert systems were programmed that emulate the decision-making ability of a human expert.

- In the Year 1980, the first national conference of the American Association of Artificial Intelligence was held at Stanford University.

The second AI winter (1987-1993)

- The duration between the years 1987 to 1993 was the second AI Winter duration.

- Again Investors and government stopped in funding for AI research as due to high cost but not efficient result. The expert system such as XCON was very cost effective.

The emergence of intelligent agents (1993-2011)

- Year 1997: In the year 1997, IBM Deep Blue beats world chess champion, Gary Kasparov, and became the first computer to beat a world chess champion.
- Year 2002: for the first time, AI entered the home in the form of Roomba, a vacuum cleaner.
- Year 2006: AI came in the Business world till the year 2006. Companies like Facebook, Twitter, and Netflix also started using AI.

Deep learning, big data and artificial general intelligence (2011-present)

- Year 2011: In the year 2011, IBM's Watson won jeopardy, a quiz show, where it had to solve the complex questions as well as riddles. Watson had proved that it could understand natural language and can solve tricky questions quickly.

- Year 2012: Google has launched an Android app feature "Google now", which was able to provide information to the user as a prediction.

- Year 2014: In the year 2014, Chatbot "Eugene Goostman" won a competition in the infamous "Turing test."

- Year 2018: The "Project Debater" from IBM debated on complex topics with two master debaters and also performed extremely well.

- Google has demonstrated an AI program "Duplex" which was a virtual assistant and which had taken hairdresser appointment on call and lady on other side didn't notice that she was talking with the machine.

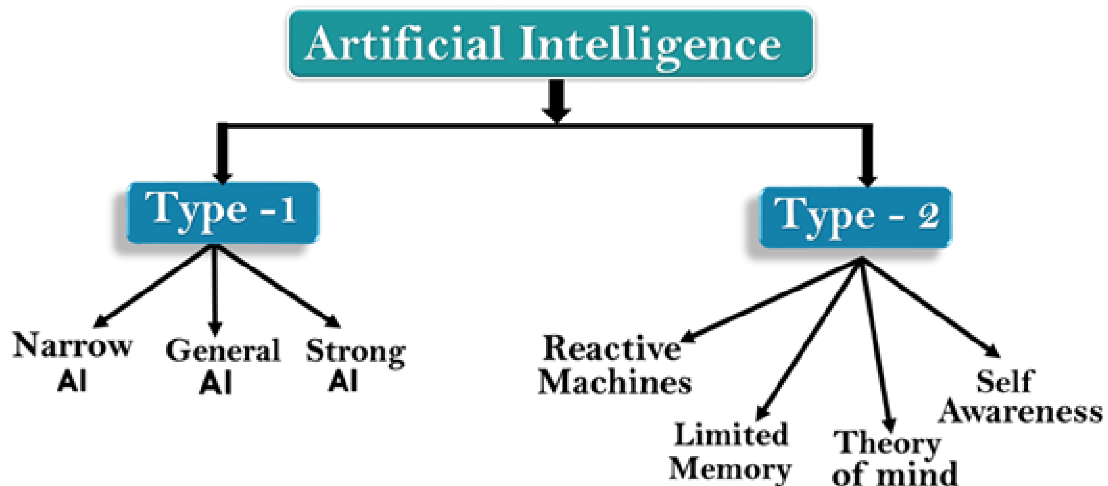
Now AI has developed to a remarkable level. The concept of Deep learning, big data, and data science are now trending like a boom. Nowadays companies like Google, Facebook, IBM, and Amazon are working with AI and creating amazing devices.

The future of Artificial Intelligence is inspiring and will come with high intelligence.

Types of Artificial Intelligence:

Artificial Intelligence can be divided in various types, there are mainly two types of main categorization which are based on capabilities and based on functionally of AI.

Following is flow diagram which explains the types of AI.



AI type-1: Based on Capabilities

1. Weak AI or Narrow AI:

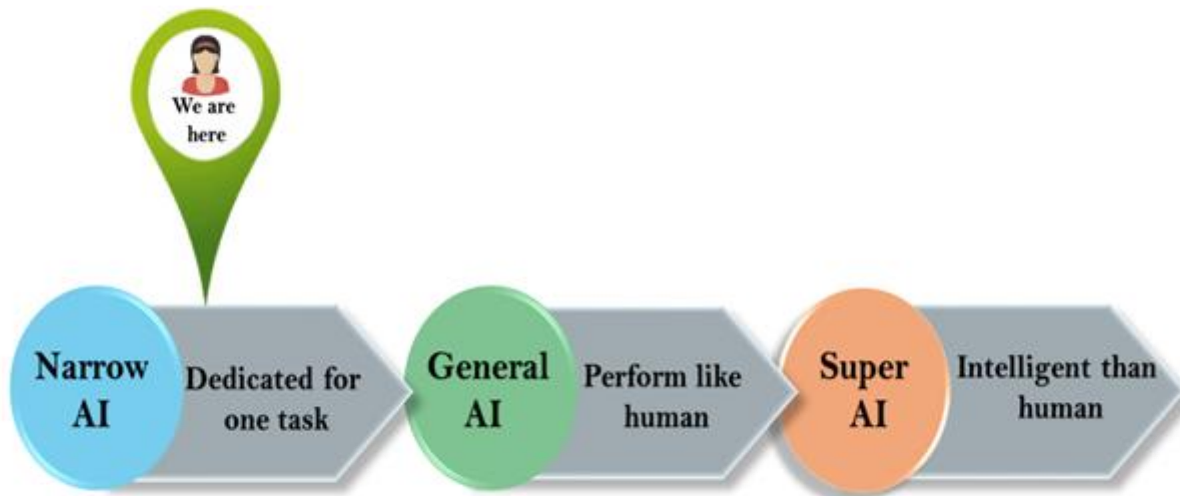
- Narrow AI is a type of AI which is able to perform a dedicated task with intelligence. The most common and currently available AI is Narrow AI in the world of Artificial Intelligence.
- Narrow AI cannot perform beyond its field or limitations, as it is only trained for one specific task. Hence it is also termed as weak AI. Narrow AI can fail in unpredictable ways if it goes beyond its limits.
- Apple Siriis a good example of Narrow AI, but it operates with a limited pre-defined range of functions.
- IBM's Watson supercomputer also comes under Narrow AI, as it uses an Expert system approach combined with Machine learning and natural language processing.
- Some Examples of Narrow AI are playing chess, purchasing suggestions on e-commerce site, self-driving cars, speech recognition, and image recognition.

2. General AI:

- General AI is a type of intelligence which could perform any intellectual task with efficiency like a human.
- The idea behind the general AI to make such a system which could be smarter and think like a human by its own.
- Currently, there is no such system exist which could come under general AI and can perform any task as perfect as a human.

- The worldwide researchers are now focused on developing machines with General AI.
- As systems with general AI are still under research, and it will take lots of efforts and time to develop such systems.

3. Super AI:
   - Super AI is a level of Intelligence of Systems at which machines could surpass human intelligence, and can perform any task better than human with cognitive properties. It is an outcome of general AI.
   - Some key characteristics of strong AI include capability include the ability to think, to reason, solve the puzzle, make judgments, plan, learn, and communicate by its own.
   - Super AI is still a hypothetical concept of Artificial Intelligence. Development of such systems in real is still world changing task.



Artificial Intelligence type-2:

Based on functionality

1. Reactive Machines
   - Purely reactive machines are the most basic types of Artificial Intelligence.
   - Such AI systems do not store memories or past experiences for future actions.
   - These machines only focus on current scenarios and react on it as per possible best action.
   - IBM's Deep Blue system is an example of reactive machines.
   - Google's AlphaGo is also an example of reactive machines.

2. Limited Memory
   - Limited memory machines can store past experiences or some data for a short period of time.
   - These machines can use stored data for a limited time period only.
   - Self-driving cars are one of the best examples of Limited Memory systems. These cars can store recent speed of nearby cars, the distance of other cars, speed limit, and other information to navigate the road.

3. Theory of Mind
   - Theory of Mind AI should understand the human emotions, people, beliefs, and be able to interact socially like humans.
   - This type of AI machines is still not developed, but researchers are making lots of efforts and improvement for developing such AI machines.

4. Self-Awareness
   - Self-awareness AI is the future of Artificial Intelligence. These machines will be super intelligent, and will have their own consciousness, sentiments, and self-awareness.
   - These machines will be smarter than human mind.
   - Self-Awareness AI does not exist in reality still and it is a hypothetical concept.

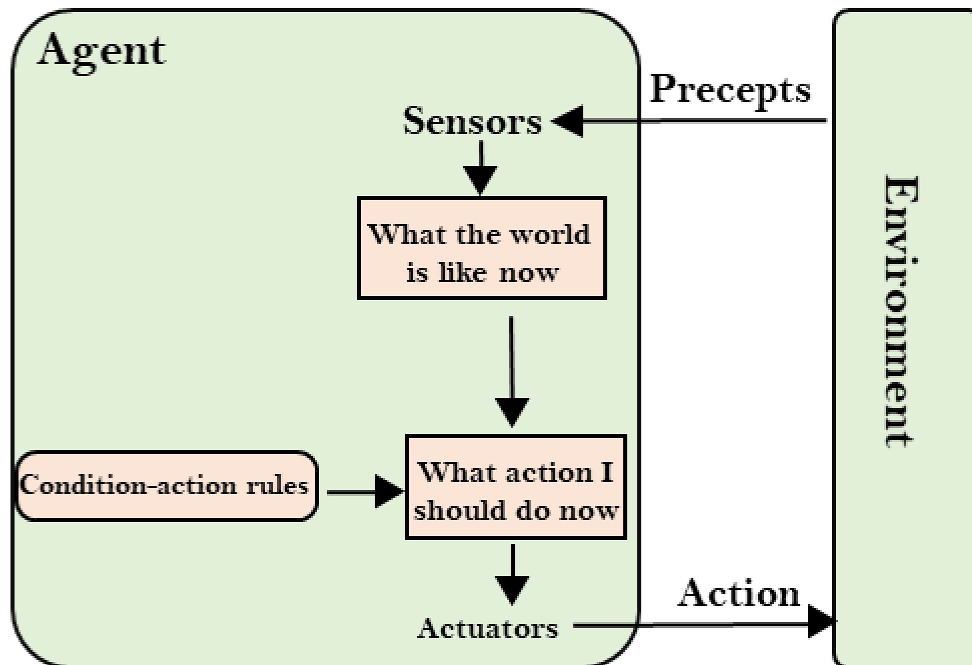
Types of AI Agents

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.

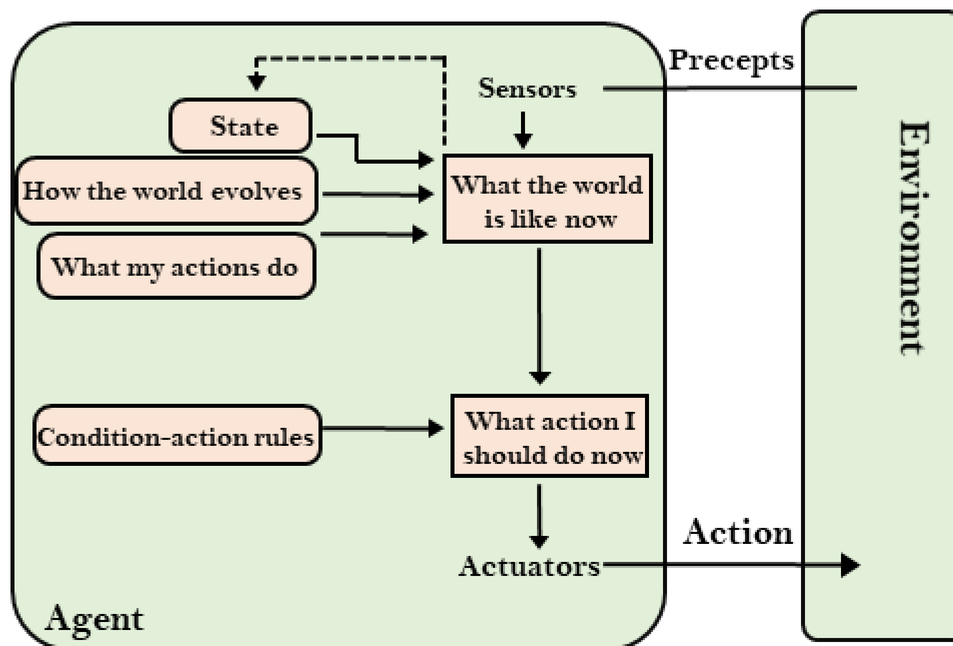
2. Model-based reflex agent

- The Model-based agent can work in a partially observable environment, and track the situation.

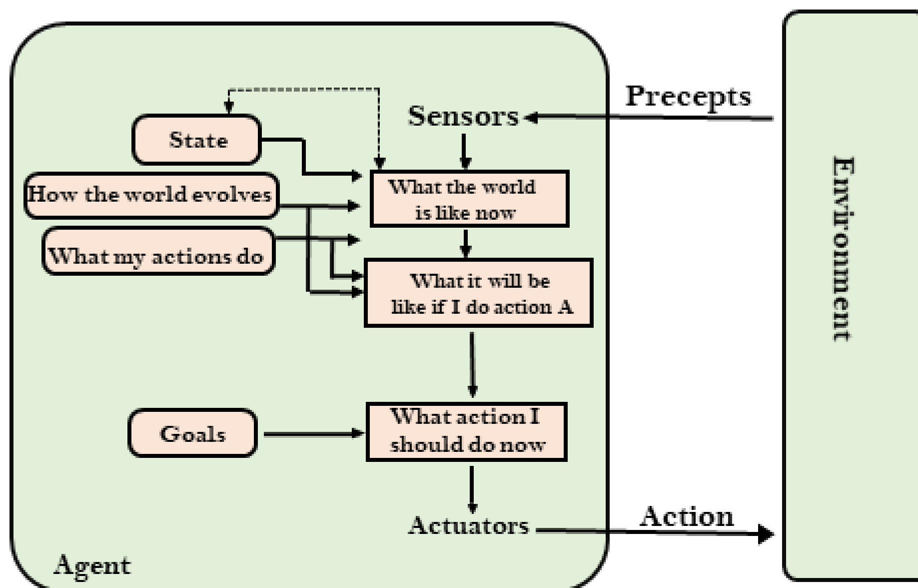
- A model-based agent has two important factors:
    - Model: It is knowledge about "how things happen in the world," so it is called a Model-based agent.
    - Internal State: It is a representation of the current state based on percept history.

- 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.

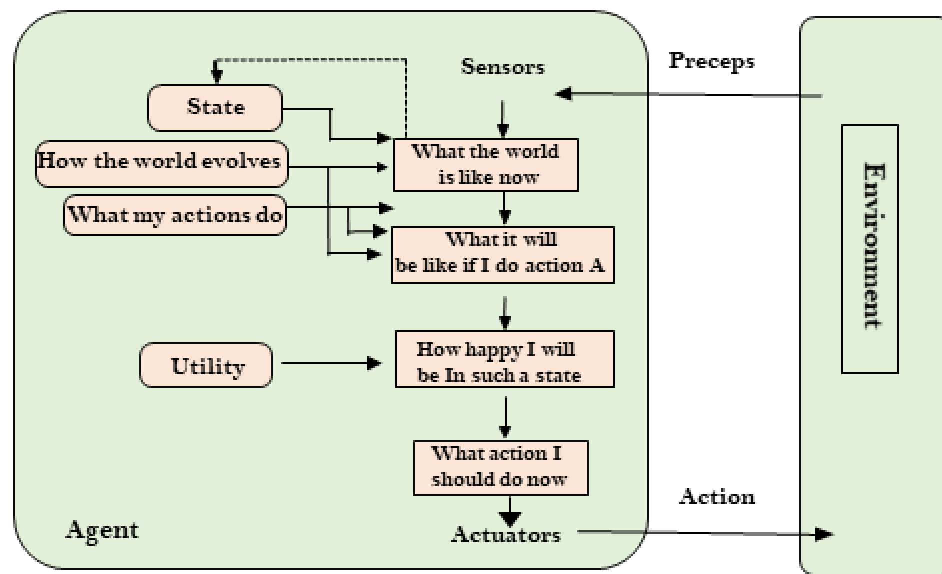


3. Goal-based agents



- 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.

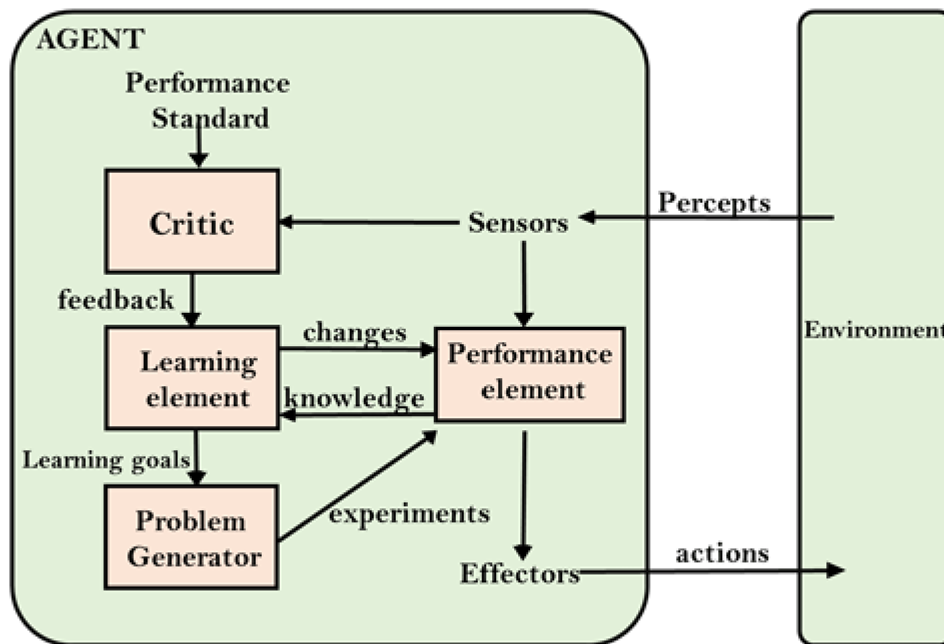
4. Utility-based agents
- These agents are similar to the goal-based agent but provide an extra component of utility measurement which makes them different by providing a measure of success at a given state.
- Utility-based agent act based not only goals but also the best way to achieve the goal.
- The 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.



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 perceive its environment through sensors and act upon that environment through actuators.

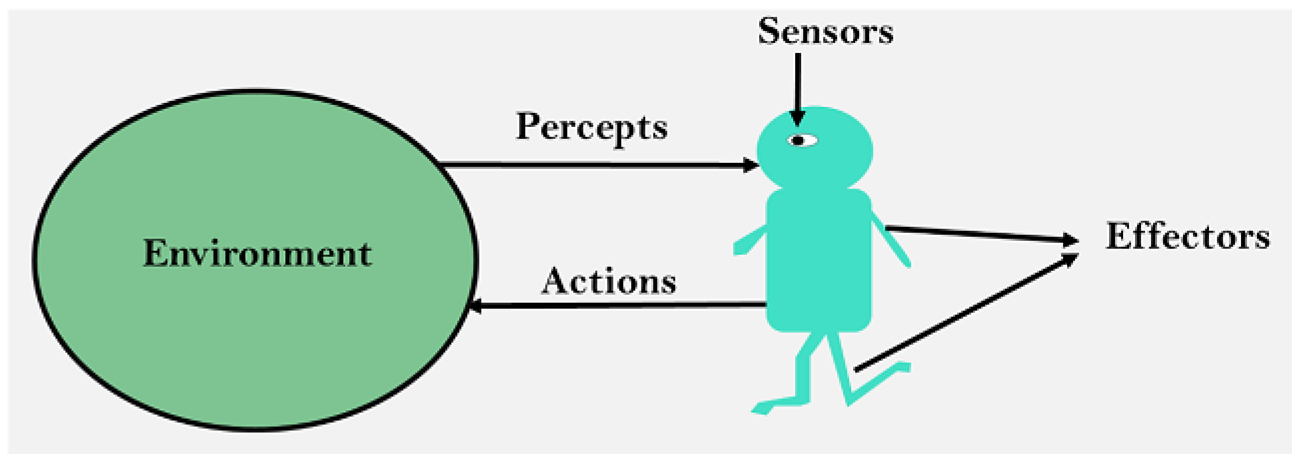
An Agent runs in the cycle of perceiving, thinking, and acting. An agent can be:

- 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 autonomous entities 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.

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.

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:

Agent = Architecture + 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

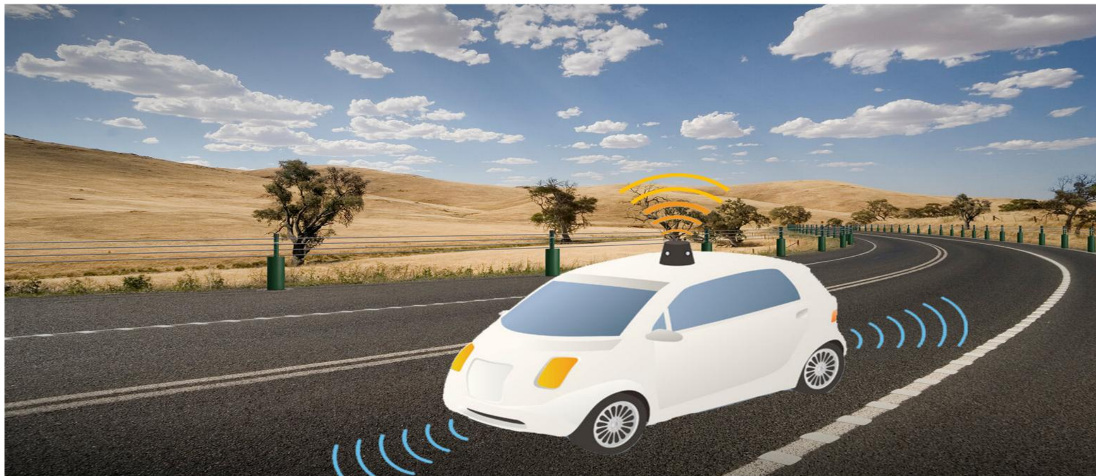
PEAS is a type of model on which an AI agent works upon. When we define an AI agent or rational agent, then we can group its properties under PEAS representation model.

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
Medical Diagnose	Healthy patient Minimized cost	Patient Hospital Staff	Tests Treatments	Keyboard (Entry of symptoms)
Vacuum Cleaner	Cleanness Efficiency Battery life Security	Room Table Wood floor Carpet Various obstacles	Wheels Brushes Vacuum Extractor	Camera Dirt detection sensor Cliff sensor Bump Sensor Infrared Wall Sensor
Part picking Robot	Percentage of parts in correct bins.	Conveyor belt with parts, Bins	Jointed Arms Hand	Camera 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

As per Russell and Norvig, an 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 gamecomes 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.

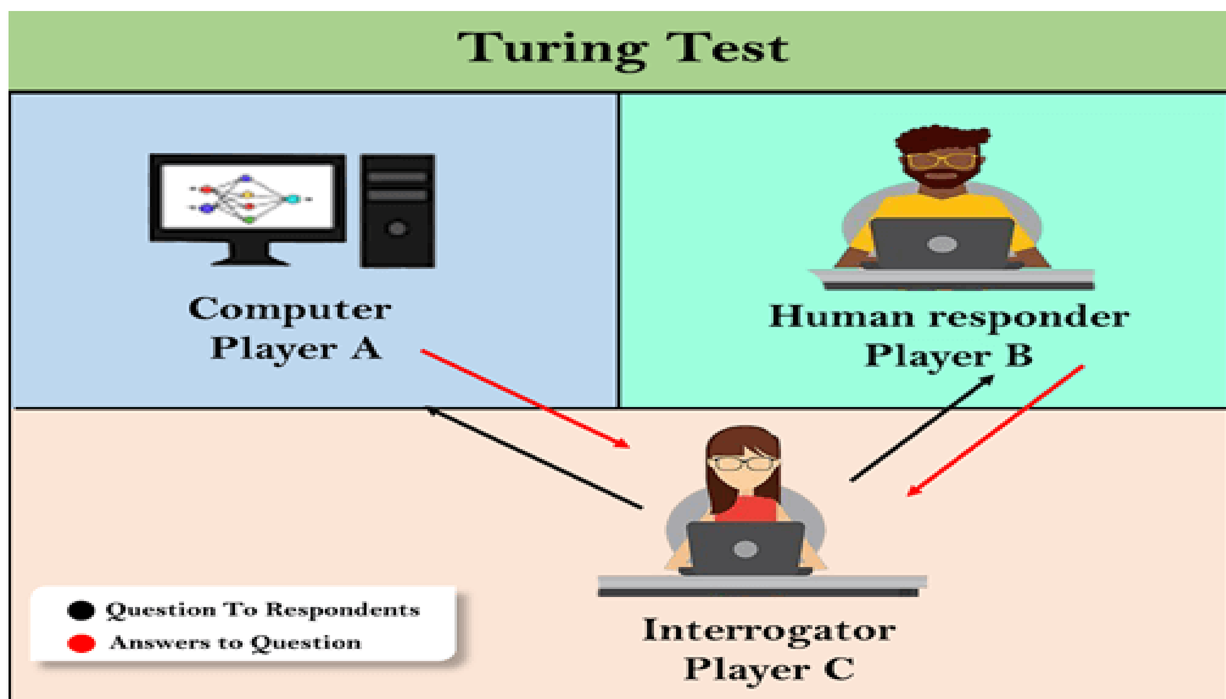
8. Accessible vs Inaccessible

- 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.

Turing Test in AI

In 1950, Alan Turing introduced a test to check whether a machine can think like a human or not, this test is known as the Turing Test. In this test, Turing proposed that the computer can be said to be an intelligent if it can mimic human response under specific conditions.

Turing Test was introduced by Turing in his 1950 paper, "Computing Machinery and Intelligence," which considered the question, "Can Machine think?"



The Turing test is based on a party game "Imitation game," with some modifications. This game involves three players in which one player is Computer, another player is human responder, and the third player is a human Interrogator, who is isolated from other two

players and his job is to find that which player is machine among two of them.

Consider, Player A is a computer, Player B is human, and Player C is an interrogator. Interrogator is aware that one of them is machine, but he needs to identify this on the basis of questions and their responses.

The conversation between all players is via keyboard and screen so the result would not depend on the machine's ability to convert words as speech.

The test result does not depend on each correct answer, but only how closely its responses like a human answer. The computer is permitted to do everything possible to force a wrong identification by the interrogator.
The questions and answers can be like:

Interrogator: Are you a computer?

PlayerA (Computer): No

Interrogator: Multiply two large numbers such as (256896489*456725896)

Player A: Long pause and give the wrong answer.

In this game, if an interrogator would not be able to identify which is a machine and which is human, then the computer passes the test successfully, and the machine is said to be intelligent and can think like a human.

"In 1991, the New York businessman Hugh Loebner announces the prize competition, offering a $100,000 prize for the first computer to pass the Turing test. However, no AI program to till date, come close to passing an undiluted Turing test".

Chatbots to attempt the Turing test:

ELIZA: ELIZA was a Natural language processing computer program created by Joseph Weizenbaum. It was created to demonstrate the ability of communication between machine and humans. It was one of the first chatterbots, which has attempted the Turing Test.

Parry: Parry was a chatterbot created by Kenneth Colby in 1972. Parry was designed to simulate a person with Paranoid schizophrenia (most common chronic mental disorder). Parry was described as "ELIZA with attitude." Parry was tested using a variation of the Turing Test in the early 1970s.

Eugene Goostman: Eugene Goostman was a chatbot developed in Saint Petersburg in 2001. This bot has competed in the various number of Turing Test. In June 2012, at an event, Goostman won the competition promoted as largest-ever Turing test content, in which it has convinced 29% of judges that it was a human. Goostman resembled as a 13-year old virtual boy.

The Chinese Room Argument:

There were many philosophers who really disagreed with the complete concept of Artificial Intelligence. The most famous argument in this list was "Chinese Room."

In the year 1980, John Searle presented "Chinese Room" thought experiment, in his paper "Mind, Brains, and Program," which was against the validity of Turing's Test. According to his argument, "Programming a computer may make it to understand a language, but it will not produce a real understanding of language or consciousness in a computer."
He argued that Machine such as ELIZA and Parry could easily pass the Turing test by manipulating keywords and symbol, but they had no real understanding of language.

So it cannot be described as "thinking" capability of a machine such as a human.

Features required for a machine to pass the Turing test:

- Natural language processing: NLP is required to communicate with Interrogator in general human language like English.
- Knowledge representation: To store and retrieve information during the test.
- Automated reasoning: To use the previously stored information for answering the questions.
- Machine learning: To adapt new changes and can detect generalized patterns.

- Vision (For total Turing test): To recognize the interrogator actions and other objects during a test.
- Motor Control (For total Turing test): To act upon objects if requested.

MCQ

1. Who is known as the -Father of AI"?
    a. Fisher Ada
    b. Alan Turing
    c. John McCarthy
    d. Allen Newell

2. The state-space of the problem includes
    a. Initial state
    b. Action
    c. Transition model
    d. All the above

3. An AI system is composed of
    a. Agent
    b. Environment
    c. Agent and Environment
    d. None of the above

4. Agents can be grouped into classes based on their degree of perceived

    a. Intelligence
    b. Capability
    c. Intelligence and capability
    d. Performance

5. Which agent can work in a partially observable environment, and track the situation?

    a) Simple Reflex Agent
    b) Model-based reflex agent
    c) Goal-based agents
    d) Utility-based agent

6. Which type of agent acts not only for goals but also for the best way to achieve the goal?

    a. Simple Reflex Agent
    b. Model-based reflex agent
    c. Goal-based agents
    d. Utility-based agent

7. Which agent is useful when there are multiple possible alternatives?

    a. Simple Reflex Agent
    b. Model-based reflex agent
    c. Goal-based agents
    d. Utility-based agent

8. Which type of agent works on Condition-action rule?

    a. Simple Reflex Agent
    b. Model-based reflex agent
    c. Goal-based agents
    d. Utility-based agent

9. Rationality can be judged on the basis of

    a. Performance measure which defines the success criterion.
    b. Agent prior knowledge of its environment.
    c. The sequence of percepts.
    d. All the above

10. Which device detects the change in the environment and sends the information to other electronic devices?

    a. Sensors
    b. Actuators
    c. Effectors
    d. All the above

CONCLUSION:

Upon completion of this, Students should be able to

To understand the some fundamentals of AI and AI systems

REFERENCES

1. David Poole, Alan Mackworth, Randy Goebel, "Computational Intelligence: a Logical Approach", Oxford University Press, 2004.

2. G. Luger, "Artificial Intelligence: Structures and Strategies for Complex Problem Solving", Fourth Edition, Pearson Education, 2002.

ASSIGNMENT

1. Define intelligent agent.
2. Explain about the foundations of AI.
3. Explain the types of Agent and its environment
4. Explain the structure of agents.
5. Explain about the problem solving agent.

UNIT-II

Uninformed Searching strategies-Breadth First Search, Depth First search, Depth limited search, Iterative deepening search, Bidirectional Search - Avoiding repeated States - Searching with Partial information –Informed search strategies – Greedy Best First Search-A* Search-Heuristic Functions Local Search Algorithms for Optimization Problems-Local search in Continuous Spaces

AIM & OBJECTIVES

- To understand the fundamental concepts of Propagation.
- To understand fading techniques and its types.
- To understand about Antenna Diversity

PRE- REQUISITE: Basic knowledge of Wireless Communication

Search Algorithms in Artificial Intelligence

Search algorithms are one of the most important areas of Artificial Intelligence. This topic will explain all about the search algorithms in AI.

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.

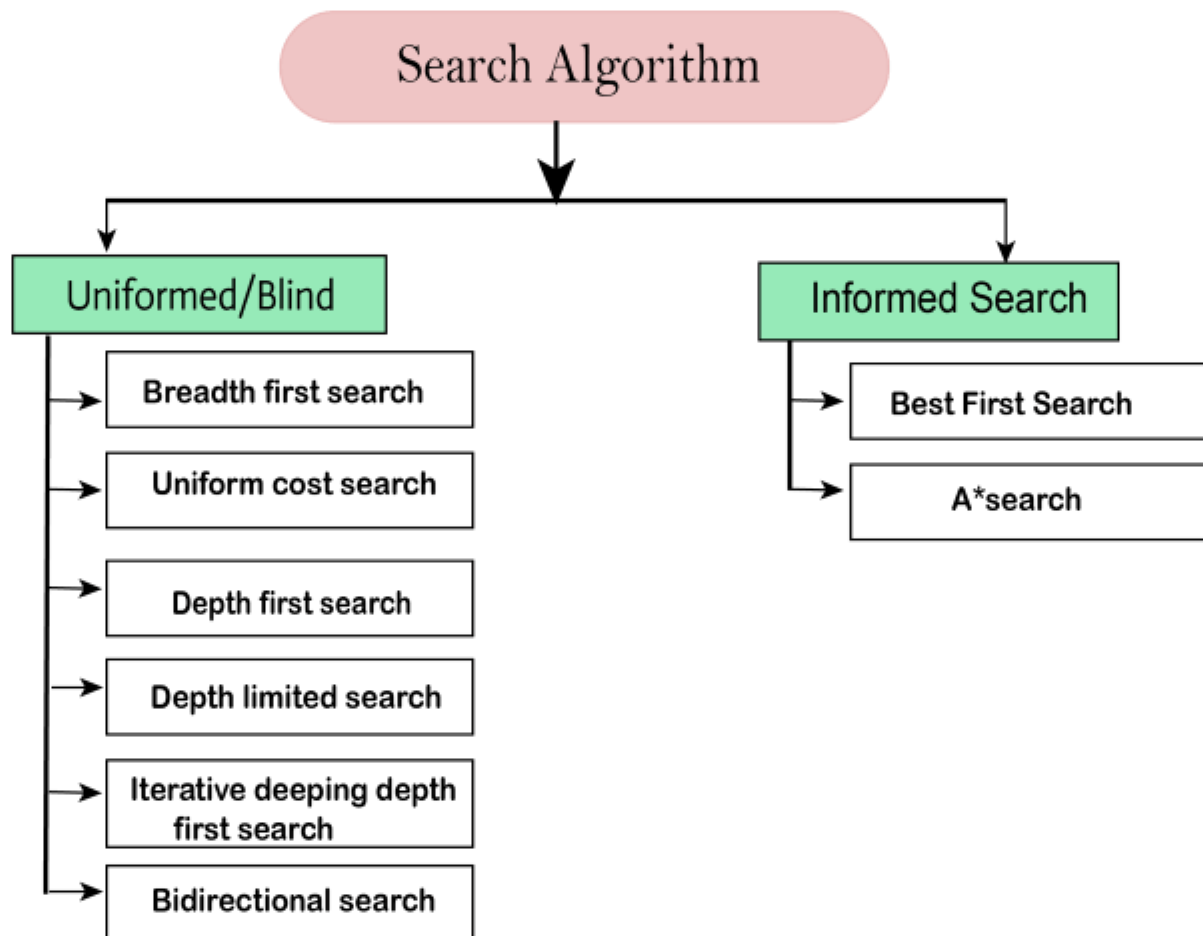
Optimality: If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.

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/Blind Search:

The uninformed search does not contain any domain knowledge such as closeness, the location of the goal. It operates in a brute-force way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes. Uninformed search applies a way in which search tree is searched without any information about the search space like initial state operators and test for the goal, so it is also called blind search. It examines each node of the tree until it achieves the goal node.

It can be divided into five main types:

- Breadth-first search
- Uniform cost search
- Depth-first search
- Iterative deepening depth-first search
- Bidirectional Search

Informed Search

Informed search algorithms use domain knowledge. In an informed search, problem information is available which can guide the search. Informed search strategies can find a solution more efficiently than an uninformed search strategy. Informed search is also called a Heuristic search.

A heuristic is a way which might not always be guaranteed for best solutions but guaranteed to find a good solution in reasonable time.

Informed search can solve much complex problem which could not be solved in another way.

An example of informed search algorithms is a traveling salesman problem.

1. Greedy Search
2. A* Search

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.

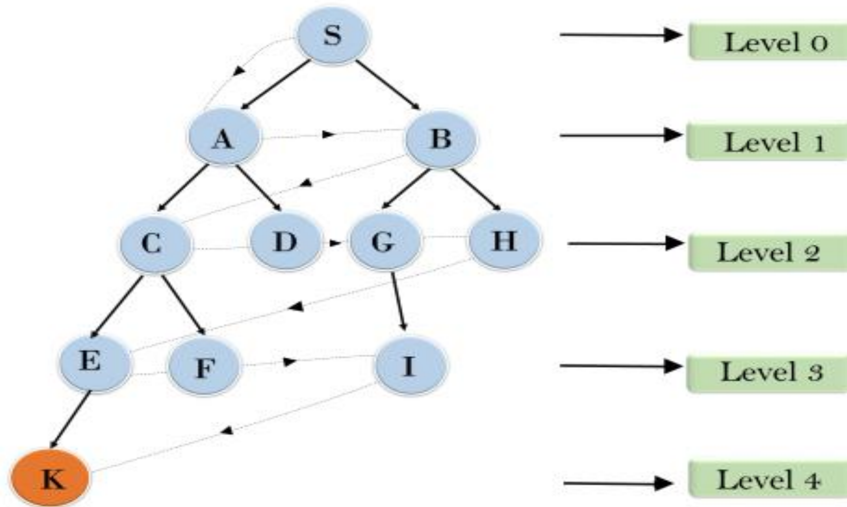
Disadvantages:

- 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:

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

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 isa 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.

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:

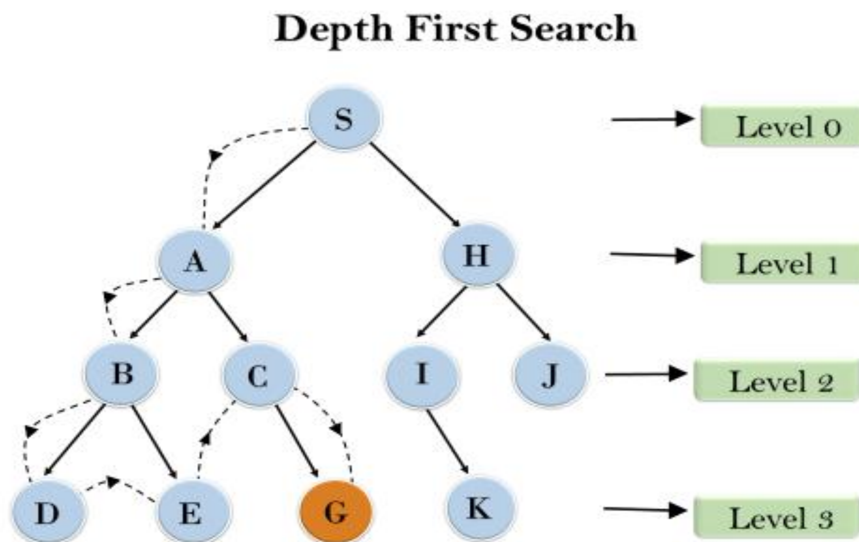
- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

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:

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

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.

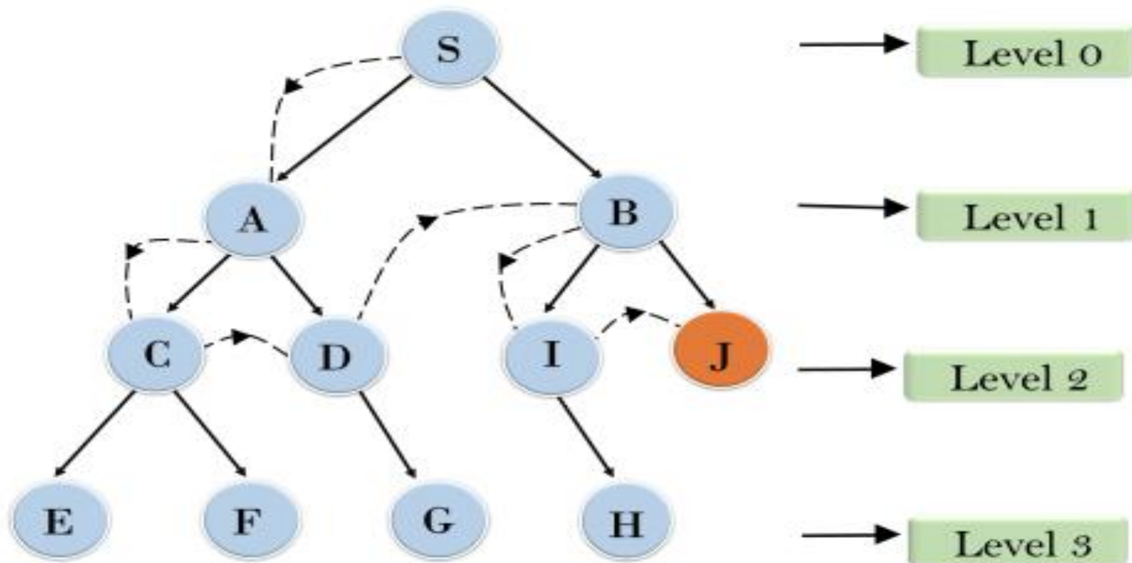
Advantages:

Depth-limited search is Memory efficient.

Disadvantages:

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

Example:



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

Time Complexity: Time complexity of DLS algorithm is $O(b^{\ell})$.

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

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

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 form 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.

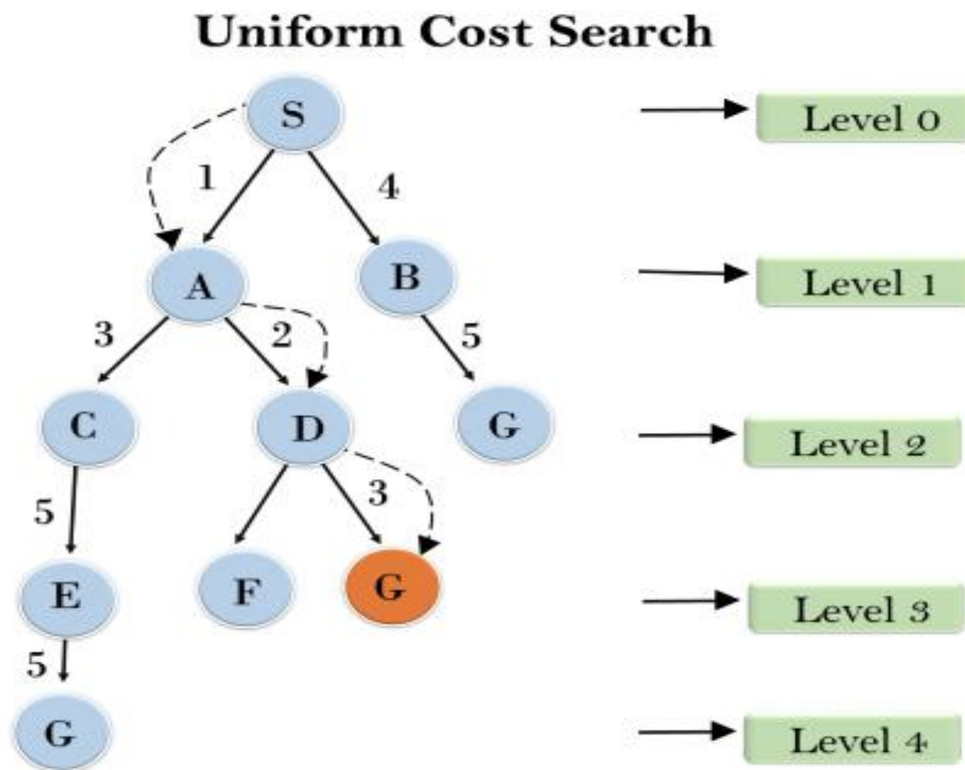
Advantages:

- 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 involve 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*/ε+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 $O(b^{1 + [C^*/\varepsilon]})$/.

Space Complexity:

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

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.

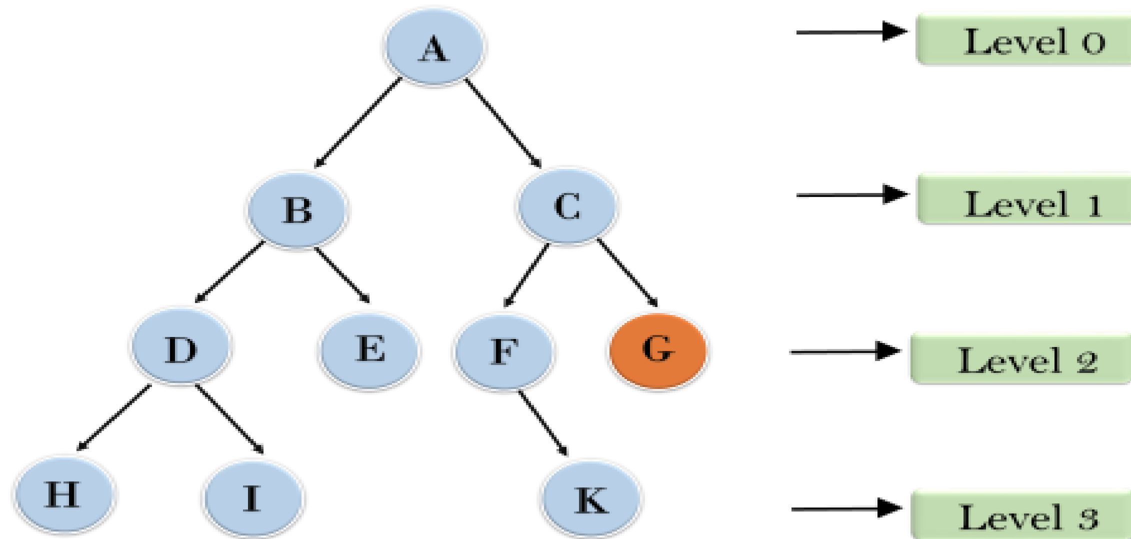
Disadvantages:

- 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'stIteration----->A
2'ndIteration---->A,B,C
3'rdIteration------>A,B,D,E,C,F,G
4'thIteration------>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 if the 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).

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:

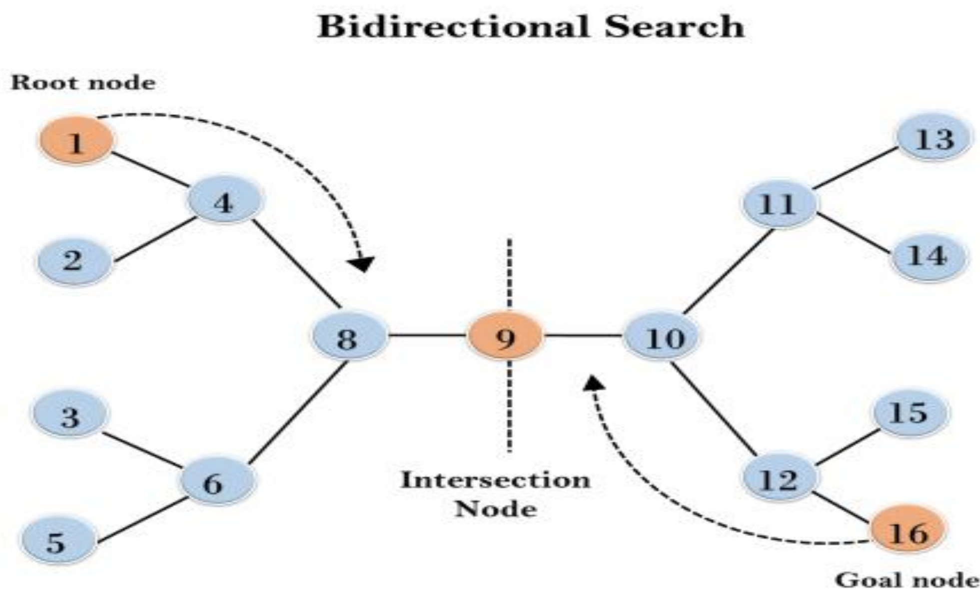
- Bidirectional search is fast.
- Bidirectional search requires less memory

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 helps 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: $h(n) <= h^*(n)$

Here $h(n)$ is heuristic cost, and $h^*(n)$ is the estimated cost.

Hence heuristic cost should be less than or equal to the estimated cost.

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 unit 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.

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

Were, 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:

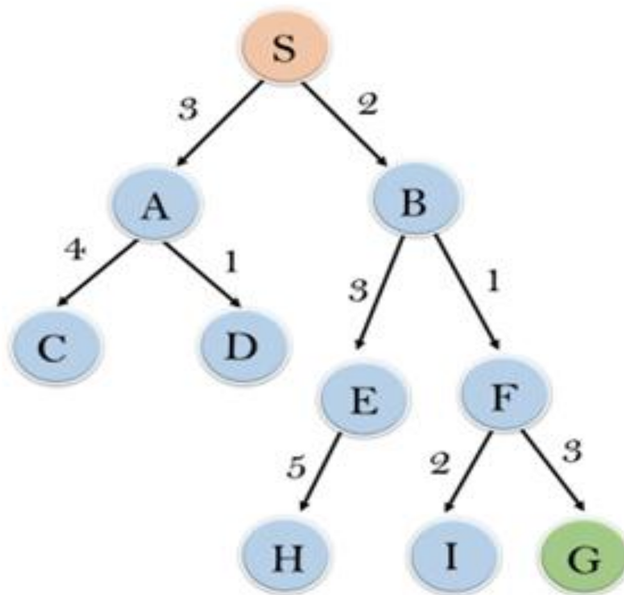
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

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.

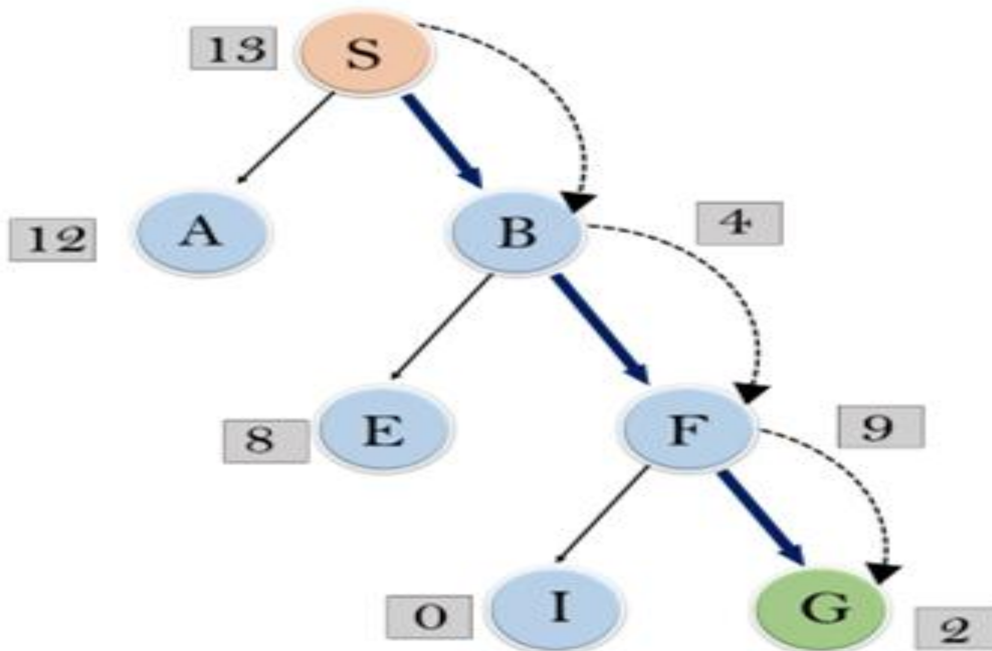
Example:

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.



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

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]

Iteration2: Open[E,F,A],Closed[S,B]
: Open [E, A], Closed [S, B, F]

Iteration3: 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.

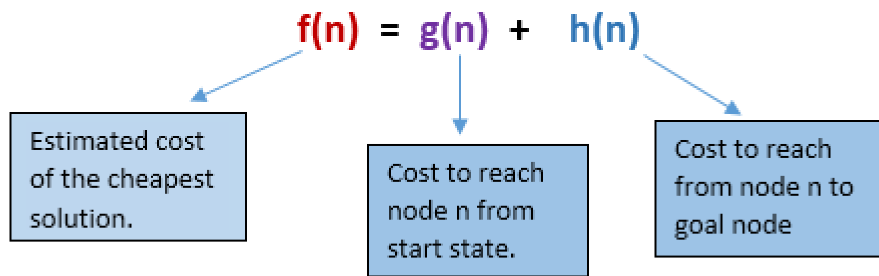
Optimal: Greedy best first search algorithm is not optimal.

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:

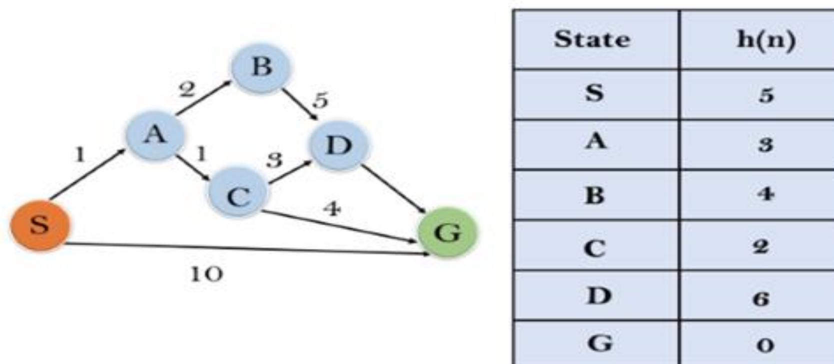
- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.

- 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:

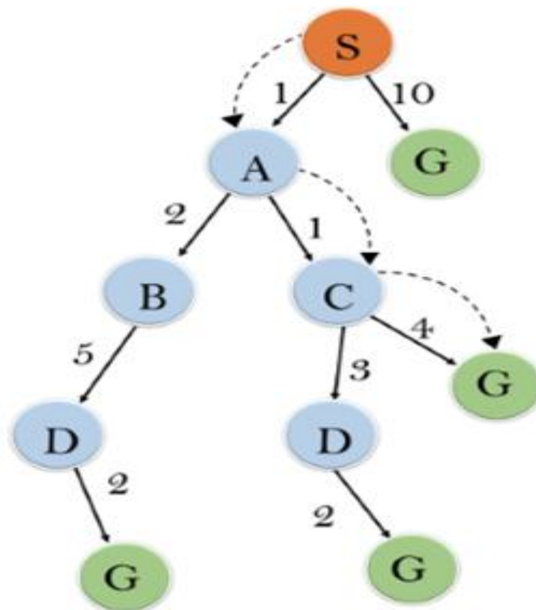
In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.

Here we will use OPEN and CLOSED list.



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

Solution :



Initialization: {(S, 5)}

Iteration1: {(S--> A, 4), (S-->G, 10)}

Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->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)<="" li="">

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)

Hill Climbing Algorithm in Artificial Intelligence

- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.

- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem 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.

- A node of hill climbing algorithm has two components which are state and value.

- Hill Climbing is mostly used when a good heuristic is available.

- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

Features of Hill Climbing:

Following are some main features of Hill Climbing Algorithm:

- Generate and Test variant: Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.

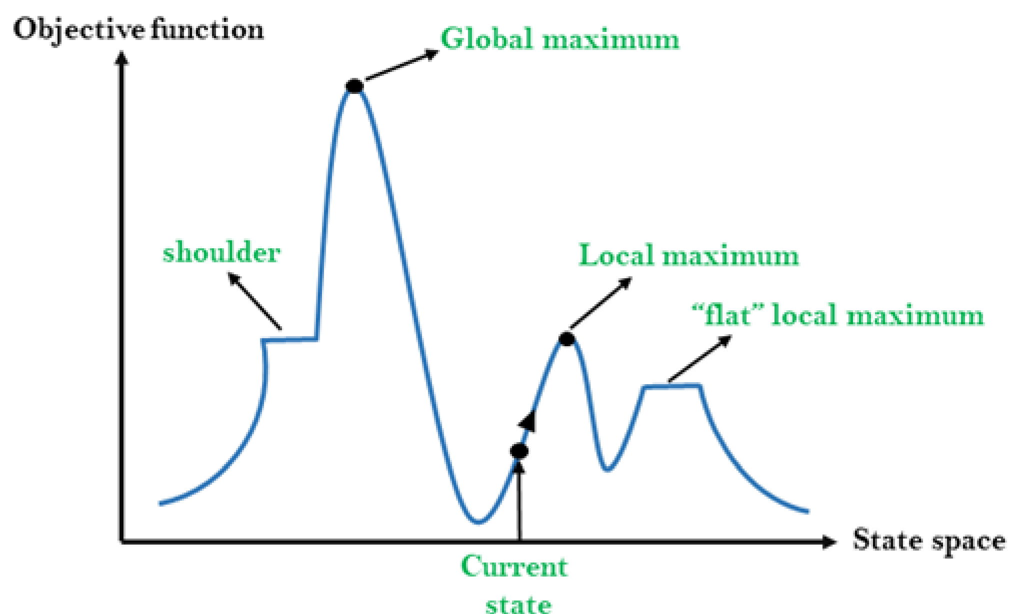
- Greedy approach: Hill-climbing algorithm search moves in the direction which optimizes the cost.
- No backtracking: It does not backtrack the search space, as it does not remember the previous states.

State-space Diagram for Hill Climbing:

The state-space landscape is a graphical representation of the hill-climbing algorithm which is showing a graph between various states of algorithm and Objective function/Cost.

On Y-axis we have taken the function which can be an objective function or cost function, and state-space on the x-axis. If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum.

If the function of Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.



Different regions in the state space landscape:

Local Maximum: Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.
C++ vs Java

Global Maximum: Global maximum is the best possible state of state space landscape. It has the highest value of objective function.

Current state: It is a state in a landscape diagram where an agent is currently present.

Flat local maximum: It is a flat space in the landscape where all the neighbor states of current states have the same value.

Shoulder: It is a plateau region which has an uphill edge.

Types of Hill Climbing Algorithm:

- Simple hill Climbing:
- Steepest-Ascent hill-climbing:
- Stochastic hill Climbing:

1. Simple Hill Climbing:

Simple hill climbing is the simplest way to implement a hill climbing algorithm. It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as a current state. It only checks it's one successor state, and if it finds better than the current state, then move else be in the same state.

This algorithm has the following features:

- Less time consuming
- Less optimal solution and the solution is not guaranteed

Algorithm for Simple Hill Climbing:

- Step 1: Evaluate the initial state, if it is 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:
    a. If it is goal state, then return success and quit.
    b. Else if it is better than the current state then assign new state as a current state.
    c. Else if not better than the current state, then return to step2.
- Step 5: Exit.

2. Steepest-Ascent hill climbing:

The steepest-Ascent algorithm is a variation of simple hill climbing algorithm. This algorithm examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state. This algorithm consumes more time as it searches for multiple neighbors

Algorithm for Steepest-Ascent hill climbing:

- Step 1: Evaluate the initial state, if it is goal state then return success and stop, else make current state as initial state.
- Step 2: Loop until a solution is found or the current state does not change.
    a. Let SUCC be a state such that any successor of the current state will be better than it.
    b. For each operator that applies to the current state:
        a. Apply the new operator and generate a new state.
        b. Evaluate the new state.
        c. If it is goal state, then return it and quit, else compare it to the SUCC.
        d. If it is better than SUCC, then set new state as SUCC.
        e. If the SUCC is better than the current state, then set current state to SUCC.
- Step 5: Exit.

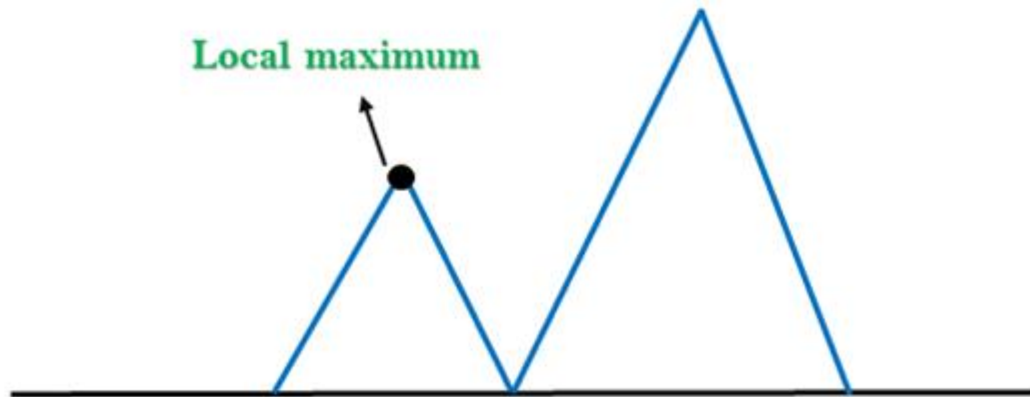
3. Stochastic hill climbing:

Stochastic hill climbing does not examine for all its neighbor before moving. Rather, this search algorithm selects one neighbor node at random and decides whether to choose it as a current state or examine another state.

Problems in Hill Climbing Algorithm:

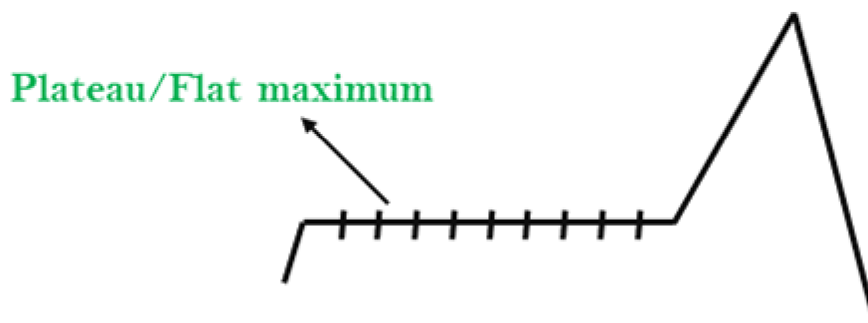
1. Local Maximum: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

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



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

Solution: The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.



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

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

Simulated Annealing:

A hill-climbing algorithm which never makes a move towards a lower value guaranteed to be incomplete because it can get stuck on a local maximum. And if algorithm applies a random walk, by moving a successor, then it may complete but not efficient.

Simulated Annealing is an algorithm which yields both efficiency and completeness.

In mechanical term Annealing is a process of hardening a metal or glass to a high temperature then cooling gradually, so this allows the metal to reach a low-energy crystalline state. The same process is used in simulated annealing in which the algorithm picks a random move, instead of picking the best move. If the random move improves the state, then it follows the same path.
Otherwise, the algorithm follows the path which has a probability of less than 1 or it moves downhill and chooses another path.

MCQ

1. Problem solving agents are also called as

    a. Simple agent
    b. Reflex agent
    c. Rational agent
    d. Goal based agent

2. Which represents a set of possible solutions, which a system may have?

    a) Search space
    b) Start state
    c) Search tree
    d) Goal test

3. If a solution has the lowest cost among all solutions, then it is called as

    a) Optimal solution
    b) Path cost
    c) Transition model
    d) None of the above

4. Which search does not contain any domain knowledge such as closeness, the location of the goal?

    a) Uninformed search
    b) Informed search
    c) Blind search
    d) Both A and C

5. Which algorithm is a combination of DFS and BFS algorithms?

    a) Iterative deepening depth-first Search
    b) Simple Search
    c) Complex search
    d) Bidirectional search

6. If the environment is not fully observable or deterministic, then which type of problems will occur?

   a) Contingency problem
   b) Conformant problem
   c) Sensorless problems
   d) All the above

7. The Estimated cost of cheapest solution f(n) =

   a) h(n)
   b) g(n)
   c) h(n) * g(n)
   d) h(n) + g(n)

8. Which is defined by the value of the objective function or heuristic cost function?

   a) Location
   b) Elevation
   c) Both
   d) None of the Above

9. Which type of Search Algorithm requires less computation?

   a) Informed search
   b) Uninformed search
   c) Both
   d) None of the above

10. A node of hill climbing algorithm has

a) State components
b) Value components
c) Both
d) None of the above

CONCLUSION:

Upon completion of this, Students should be able to

- ❖ Understand the AI systems able to exhibit limited human-like abilities, particularly in the form of problem solving by search

REFERENCES

1. David Poole, Alan Mackworth, Randy Goebel, “Computational Intelligence: a Logical Approach”, Oxford University Press, 2004.
2. G. Luger, “Artificial Intelligence: Structures and Strategies for Complex Problem Solving”, Fourth Edition, Pearson Education, 2002.

ASSIGNMENT

1. Explain about the Informed Search Algorithm.
2. Explain about the Uninformed Search Algorithm.
3. Explain about Local search Algorithm.
4. Explain about Local search in continuous spaces.
5. Explain about optimization problems.

UNIT-3

Online Search Agents and Unknown Environments-Online Search Problems, Online Search Agents- Online Local search, learning in Online Search – Constraint Satisfaction Problems- Backtracking CSP, The Structure of Problems-Adversarial Search-Games, Optimal Decisions in Games, AlphaBeta Pruning.

AIM & OBJECTIVES

- To understand the Online Search Agents.
- To understand Constraint Satisfaction Problems.
- To understand Adversarial Search.

PRE- REQUISITE: Basic knowledge of Computer Architecture.

Online Search Agents and Unknown Environments

An online search agent operates by interleaving computation and action: first it takes an action and then it observes the environment and computes the next action. Online search is a good idea in dynamic or semi dynamic domains-domains where there is a penalty for sitting around and computing too long. Online search is an even better idea for stochastic domains.
(The term "online" is commonly used in computer science to refer to algorithms that must process input data as they are received, rather than waiting for the entire input data set to become available.)

In general, an offline search would have to come up with an exponentially large contingency plan that considers all possible happenings, while an online search need only consider what actually does happen.

For example,
A chess playing agent is well-advised to make its first move long before it has figured out the complete course of the game. Online search is a necessary idea for an exploration problem, where the states and actions are unknown to the agent. An agent in this state

of Ignorance must use its actions as experiments to determine what to do next, and hence must interleave computation and action.
The canonical example of online search is a robot that is placed in a new building and must explore it to build a map that it can use for getting from A to B. Methods for escaping from labyrinths-required knowledge for aspiring heroes of antiquity-are also examples of online search algorithms. Spatial exploration is not the only form of exploration, however.

Consider a newborn baby: it has many possible actions, but knows the outcomes of none of them, and it has experienced only a few of the possible states that it can reach. The baby's gradual discovery of how the world works is, in part, an online search process.

Online search problems

An online search problem can be solved only by an agent executing actions, rather than by a purely computational process. We will assume that the agent knows just the following:

ACTIONS(S), which returns a list of actions allowed in state s;

The step-cost function c(s, a, s^l)-note that this cannot be used until the agent knows that s^l is the outcome; and

GOAL-TEST(S).

Note in particular that the agent cannot access the successors of a state except by actually trying all the actions in that state. For example, in the maze problem shown in Figure, the agent does not know that going Up from (1,l) leads to (1,2); nor, having done that, does it know that going Down will take it back to (1,l). This degree of ignorance can be reduced in some applications-for example, a robot explorer might know how its movement actions work and be ignorant only of the locations of obstacles.
We will assume that the agent can always recognize a state that it has visited before, and we will assume that the actions are deterministic. Finally, the agent might have access to an, admissible heuristic function h(s) that estimates the distance from the current state to a goal state. For example, in Figure, the agent might know the location of the goal and be able to use the Manhattan distance heuristic.

Typically, the agent's objective is to reach a goal state while minimizing cost. (Another possible objective is simply to explore the entire environment.) The cost is the total path cost of the path that the agent actually travels. It is common to compare this cost with the path cost of the path the agent would follow if it knew the search space in advance-that is, the actual shortest path (or shortest complete exploration). In the language of online algorithms
this is called the competitive ratio; we would like it to be as small as possible. Although this sounds like a reasonable request, it is easy to see that the best achievable competitive ratio is infinite in some cases. For example, if some actions are irreversible, the online search might accidentally reach a dead-end state from which no goal state is reachable.

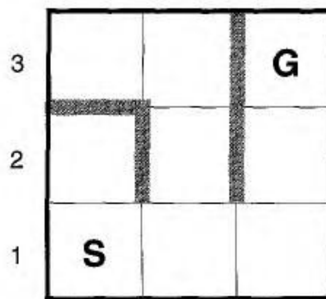
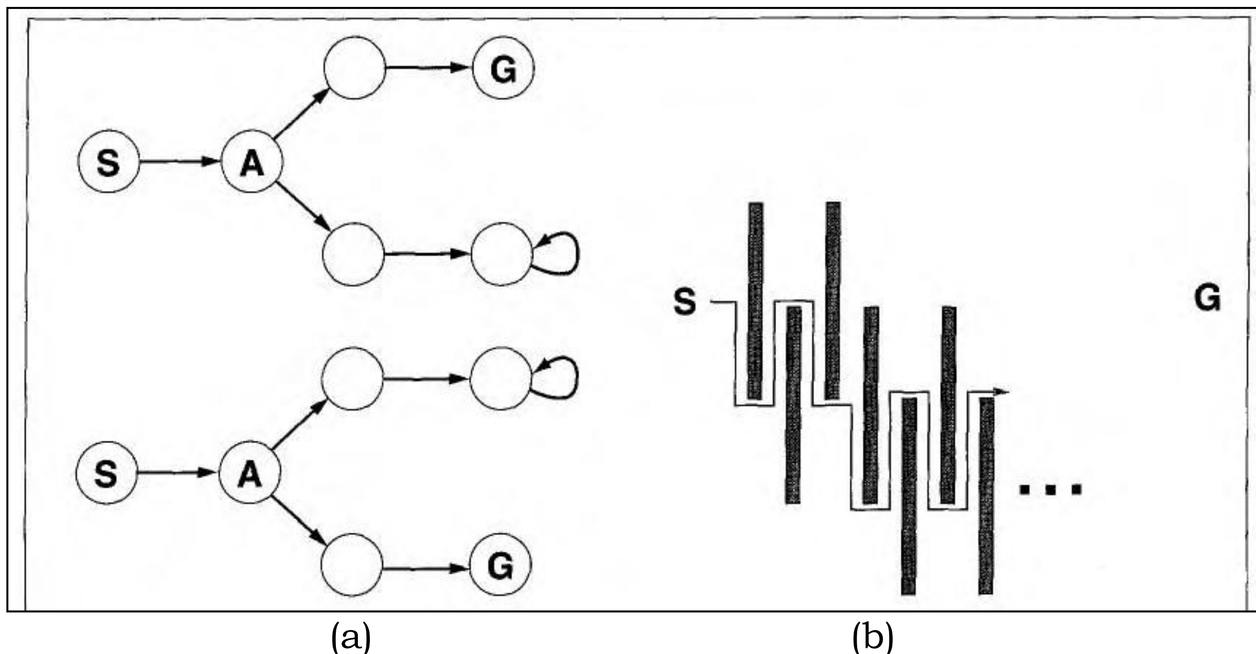


Figure: A simpIe maze problem.

The agent starts at S and must reach G, but knows nothing of the environment.



(a) (b)

(a) Two state spaces that might lead an online search agent into a dead end.Any given agent will fail in at least one of these spaces.

(b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal. Whichever choice the agent makes, the adversary blocks that route with another long, thin wall, so that the path followed is much longer than the best possible path.

Perhaps you find the term "accidentally" unconvincing-after all, there might be an algorithm that happens not to take the dead-end path as it explores. Our claim, to be more precise, is that no algorithm can avoid dead ends in all state spaces.
Consider the two dead-end state spaces in Figure (a). To an online search algorithm that has visited states S and A, the two state spaces look identical, so it must make the same decision in both. Therefore, it will fail in one of them. This is an example of an adversary argument-we can imagine an adversary that constructs the state space while the agent explores it and can put the goals and dead ends wherever it likes.

Dead ends are a real difficulty for robot exploration--staircases, ramps, cliffs, and all kinds of natural terrain present opportunities for irreversible actions. To make progress, we will simply assume that the state space is safely explorable-that is, some goal state is reachable from every reachable state. State spaces with reversible actions, such as mazes and 8-puzzles, can be viewed as undirected graphs and are clearly safely explorable.

Even in safely explorable environments, no bounded competitive ratio can be guaranteed if there are paths of unbounded cost. This is easy to show in environments with irreversible actions, but in fact it remains true for the reversible case as well, as Figure (b) shows. For this reason, it is common to describe the performance of online search algorithms in terms of the size of the entire state space rather than just the depth of the shallowest goal.

Online search agents

After each action, an online agent receives a percept telling it what state it has reached; from this information, it can augment its map of the environment. The current map is used to decide where to go next.

This interleaving of planning and action means that online search algorithms are quite different from the offline search algorithms we have seen previously.

For example, offline algorithms such as A* have the ability to expand a node in one part of the space and then immediately expand a node in another part of the space, because node expansion involves simulated rather than real actions. An online algorithm, on the other hand, can expand only a node that it physically occupies. To avoid traveling all the way across the tree to expand the next node, it seems better to expand nodes in a local order.

Depth-first search has exactly this property, because (except when backtracking) the next node expanded is a child of the previous node expanded.

An online depth-first search agent is shown in Figure. This agent stores its map in a table, result [a, s], that records the state resulting from executing action a in state s. whenever an action from the current state has not been explored, the agent tries that action.

The difficulty comes when the agent has tried all the actions in a state. In offline depth-first search, the state is simply dropped from the queue; in an online search, the agent has to backtrack physically.

In depth-first search, this means going back to the state from which the agent entered the current state most recently. That is achieved by keeping a table that lists, for each state, the predecessor states to which the agent has riot yet backtracked. If the agent has run out of states to which it can backtrack, then its search is complete.

The progress of ONLINE-DFS-AGENT can be traced when applied to the maze given in Figure. It is fairly easy to see that the agent will, in the worst case, end up traversing every link in the state space exactly twice.

For exploration, this is optimal; for finding a goal, on the other hand, the agent's competitive ratio could be arbitrarily bad if it goes off on a long excursion when there is a goal right next to the initial state. An online variant of iterative deepening solves this problem; for an environment that is a uniform tree, the competitive ratio of such an agent is a small constant.

Because of its method of backtracking, ONLINE-DFS-AGENT works only in state spaces where the actions are reversible. There are slightly more complex algorithms that work in general state spaces, but no such algorithm has a bounded competitive ratio.

```
function ONLINE-DFS-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  static: result, a table, indexed by action and state, initially empty
          unexplored, a table that lists, for each visited state, the actions not yet tried
          unbacktracked, a table that lists, for each visited 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 then unexplored[s'] ← ACTIONS(s')
  if s is not null then do
     result[a, s] ← s'
     add s to the front of unbacktracked[s']
  if unexplored[s ]is empty then
     if unbacktracked[s ]is empty then return stop
     else a ← an action b such that result[b, s ] = POP(unbacktracked[s'])
  else a ← POP(unexplored[s'])
  s ← s'
  return a
```

Figure: An online search agent that uses depth-first exploration.

The agent is applicable only in bidirected search spaces.

Online local search

Like depth-first search, **hill-climbing search** has the property of locality in its node expansions. In fact, because it keeps just one current state in memory, hill-climbing search is already an online search algorithm! Unfortunately, it is not very useful in its simplest form because it leaves the agent sitting at local maxima with nowhere to go. Moreover, random restarts cannot be used, because the agent cannot transport itself to a new state.

Instead of random restarts, one might consider using a random walk to explore the environment. A random walk simply selects at random one of the available actions from the current state; preference can be given to actions that have not yet been tried. It is easy to prove that a random walk will eventually find a goal or complete its exploration, provided that the space is finite.15 On the other hand, the process can be very slow. Figure shows an environment in which a random

walk will take exponentially many steps to find the goal, because, at each step, backward progress is twice as likely as forward progress.

The example is contrived, of course, but there are many real-world state spaces whose topology causes these kinds of "traps" for random walks.
Augmenting hill climbing with memory rather than randomness turns out to be a more effective approach. The basic idea is to store a "current best estimate" H(s) of the cost to reach the goal from each state that has been visited. H(s) starts out being just the heuristic

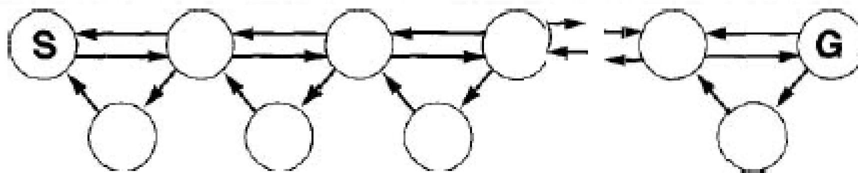


Figure - An environment in which a random walk will take exponentially many steps to find the goal.

estimate h(s) and is updated as the agent gains experience in the state space. Figure shows a simple example in a one-dimensional state space. In (a), the agent seems to be stuck in a flat local minimum at the shaded state. Rather than staying where it is, the agent should follow what seems to be the best path to the goal based on the current cost estimates for its neighbors. The estimated cost to reach the goal through a neighbor s is the cost to get to s plus the estimated cost to get to a goal from there-that is, c(s, a, s) + H(st).

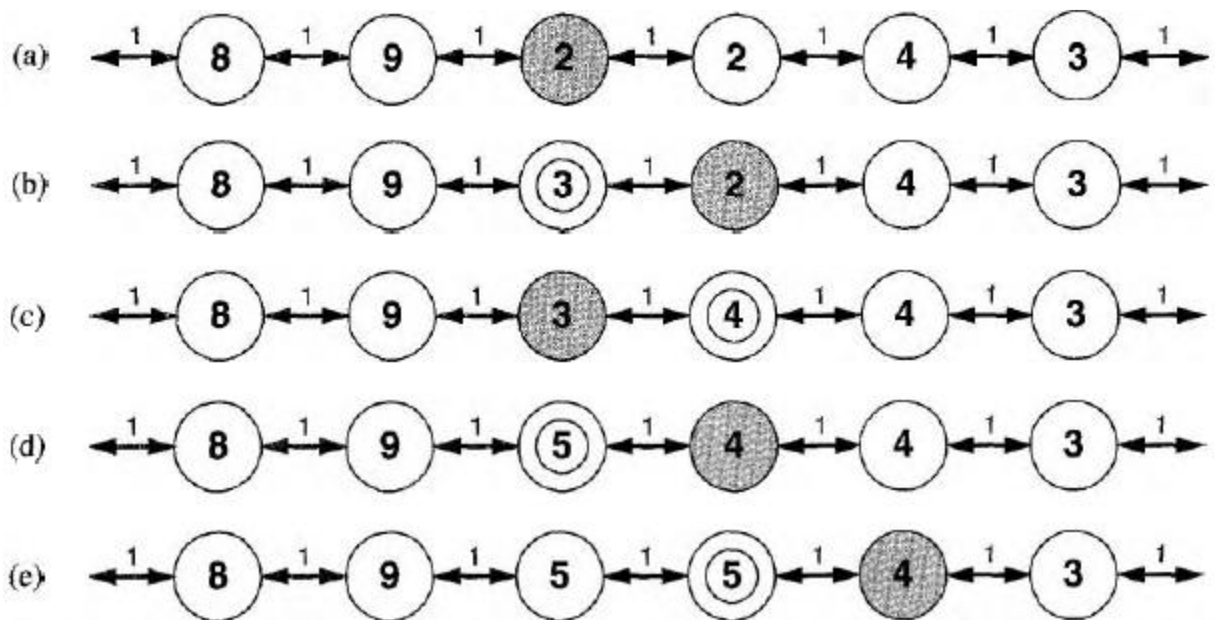
In the example, there are two actions with estimated costs 1 + 9 and 1 + 2, so it seems best to move right. Now, it is clear that the cost estimate of 2 for the shaded state was overly optimistic. Since the best move cost 1 and led to a state that is at least 2 steps from a goal, the shaded state must be at least 3 steps from a goal, so its H should be updated accordingly, as shown in Figure. Continuing this process, the agent will move back and forth twice more, updating H each time and "flattening out" the local minimum until it escapes to the right.

An agent implementing this scheme, which is called learning real-time A* (LRTA*), is shown in Figure. Like ONLINE-DFS-AGENT, it builds a map of the environment using the result table. It updates the cost estimate for the state it has just left and then chooses the

"apparently best" move according to its current cost estimates. One important detail is that actions that have not yet been tried in a state s are always assumed to lead immediately to the goal with the least possible cost, namely h(s). This optimism under uncertainty encourages the agent to explore new, possibly promising paths.

Learning in online search

The initial ignorance of online search agents provides several opportunities for learning. First, the agents learn a "map" of the environment-more precisely, the outcome of each action in each state-simply by recording each of their experiences. (Notice that the assumption of deterministic environments means that one experience is enough for each action.) Second, the local search agents acquire more accurate estimates of the value of each state by using local updating rules.



Five iterations of LRTA* on a one-dimensional state space. Each state is labeled with H(s), the current cost estimate to reach a goal, and each arc is labeled with its step cost. The shaded state marks the location of the agent, and the updated values at each iteration are circled.

```
function LRTA*-AGENT(s') returns an action
   inputs: s', a percept that identifies the current state
   static: result, a table, indexed by action and state, initially empty
           H, a table of cost estimates indexed by state, initially empty
           s, a, the previous state and action, initially null

   if GOAL-TEST(s') then return stop
   if s' is a new state (not in H) then H[s'] ← h(s')
   unless s is null
      result[a, s] ← s'
      H[s] ← min_{b ∈ ACTIONS(s)} LRTA*-COST(s, b, result[b, s], H)
   a ← an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, result[b, s'], H)
   s ← s'
   return a

function LRTA*-COST(s, a, s', H) returns a cost estimate
   if s' is undefined then return h(s)
   else return c(s, a, s') + H[s']
```

LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.

These updates eventually converge to exact values for every state, provided that the agent explores the state space in the right way. Once exact values are known, optimal decisions can be taken simply by moving to the highest-valued successor-that is, pure hill climbing is then an optimal strategy.

If you followed our suggestion to trace the behavior of ONLINE-DFS-AGENT in the environment, you will have noticed that the agent is not very bright. For example, after it has seen that the Up action goes from (1,l) to (1,2), the agent still has no idea that the Down action goes back to (1,1), or that the Up action also goes from (2,l) to (2,2), from (2,2) to (2,3), and so on. In general, we would like the agent to learn that Up increases the y-coordinate unless there is a wall in the way, which Down reduces it, and so on. For this to happen, we need two things. First, we need a formal and explicitly representation for these kinds of general rules; so far, we have hidden the information inside the black box called the successor function.

Constraint satisfaction problem (CSP)

Basically problems can be solved by searching in a space of states. These states can be evaluated by domain-specific heuristics and tested to see whether they are goal states. From the point of view of the search algorithm, however, each state is a black box with no discernible internal structure. It is represented by an arbitrary data structure that can be accessed only by the problem, specific routines-the successor function, heuristic function, and goal test.

Constraint satisfaction problems, whose states and goal test conform to a standard, structured, and very simple representation. Search algorithms can be defined that take advantage of the structure of states and use general-purpose rather than problem-specific heuristics to enable the solution of large problems.

Perhaps most importantly, the standard representation of the goal test reveals the structure of the problem itself. This leads to methods for problem decomposition and to an understanding of the intimate connection between the structure of a problem and the difficulty of solving it.

Formally speaking, a constraint satisfaction problem (or CSP) is defined by a set of variables, XI, X2,. . . , Xn, and a set of constraints, C1, (72,. . . , C,. Each variable Xi has a nonempty domain Di of possible values. Each constraint Ci involves some subset of the variables and specifies the allowable combinations of values for that subset.
A state of the problem is defined by an assignment of values to some or all of the variables, {Xi = vi, Xj =vj, . . .). An assignment that does not violate any constraints is called a consistent or legal assignment. A complete assignment is one in which every variable is mentioned, and a solution to a CSP is a complete assignment that satisfies all the constraints. Some CSPs also require a solution that maximizes an objective function.

So what does all this mean? Suppose we are looking at a map of Australia showing each of its states and territories, as in Figure, and that we are given the task of coloring each region either red, green, or blue in such a way that no neighboring regions have the same color.

To formulate this as a CSP, we define the variables to be the regions: WA, NT, Q, NSW, V, SA, and T. The domain of each variable is the set

{red, green, blue). The constraints require neighboring regions to have distinct colors; for example, the allowable combinations for WA and NT are the pairs {(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)) .
(The constraint can also be represented more succinctly as the inequality WA # NT, provided the constraint satisfaction algorithm has some way to evaluate such expressions.) There are many possible solutions, such as {WA= red, NT = green, Q = red, NSW = green, V= red, SA= blue, T= red).

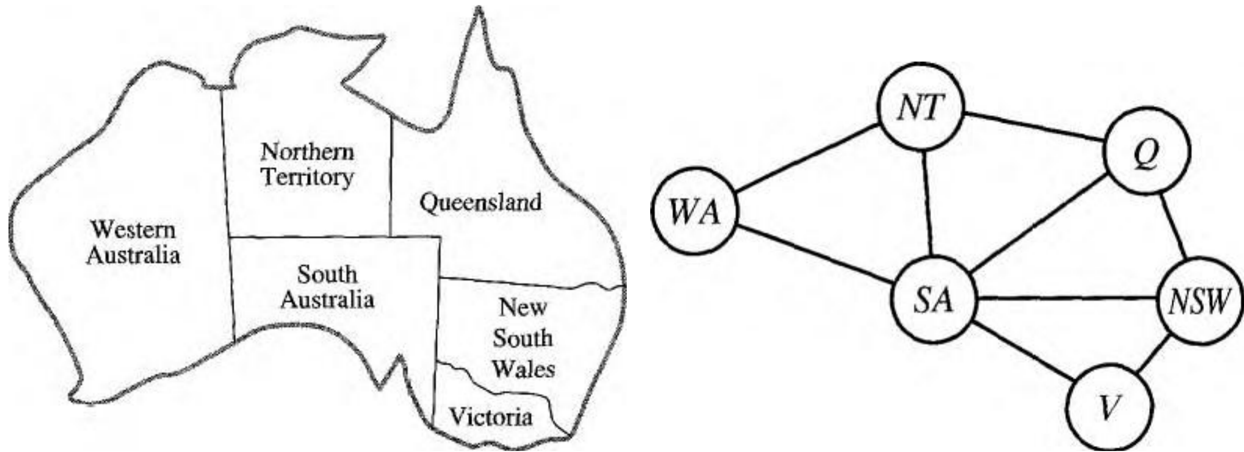


Figure (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color.
(b) The map-coloring problem represented as a constraint graph.

It is helpful to visualize a CSP as a constraint graph, as shown in Figure. The nodes of the graph correspond to variables of the problem and the arcs correspond to constraints. Treating a problem as a CSP confers several important benefits. Because the representation of states in a CSP conforms to a standard pattern-that is, a set of variables with assigned values-the successor function and goal test can be written in a generic way that applies to all CSPs. Furthermore, we can develop effective, generic heuristics that require no additional, domain-specific expertise. Finally, the structure of the constraint graph can be used to simplify the solution process, in some cases giving an exponential reduction in complexity. The CSP representation is the first, and simplest, in a series of representation schemes that will be developed throughout the book.

It is fairly easy to see that a CSP can be given an incremental formulation as a standard search problem as follows:

Initial state: the empty assignment {}, in which all variables are unassigned.

Successor function: a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables.

Goal test: the current assignment is complete.

Path cost: a constant cost (e.g., 1) for every step.

Every solution must be a complete assignment and therefore appears at depth n if there are n variables. Furthermore, the search tree extends only to depth n. For these reasons, depth first search algorithms are popular for CSPs.It is also the case that the path by which a solution is reached is irrelevant. Hence, we can also use a complete-state formulation, in which every state is a complete assignment that might or might not satisfy the constraints. Local search methods work well for this formulation.
The simplest kind of CSP involves variables that are discrete and have finite domains. Map-coloring problems are of this kind.

Finite-domain CSPs include Boolean CSPs, whose variables can be either true or false. Boolean CSPs include as special cases some NP-complete problems. In the worst case, therefore, we cannot expect to solve finite-domain CSPs in less than exponential time. In most practical applications, however, general-purpose CSP algorithms can solve problems orders of magnitude larger than those solvable via the general-purpose search algorithms

Discrete variables can also have infinite domains-for example, the set of integers or the set of strings. For example, when scheduling construction jobs onto a calendar, each job's start date is a variable and the possible values are integer numbers of days from the current date.
With infinite domains, it is no longer possible to1 describe constraints by enumerating all allowed combinations of values.

Special solution algorithms (which we will not discuss here) exist for linear constraints on integer variables-that is, constraints, such as

the one just given, in which each variable appears only in linear form. It can be shown that no algorithm exists for solving general nonlinear constraints on integer variables. In some cases, we can reduce integer constraint problems to finite-domain problems simply by bounding the values of all the variables.

For example, in a scheduling problem, we can set an upper bound equal to the total length of all the jobs to be scheduled. Constraint satisfaction problems with continuous (domains are very common in the real world and are widely studied in the field of operations research. For example, the scheduling of experiments on the Hubble Space Telescope requires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables that must obey a variety of astronomical, precedence, and power constraints.

The best-known category of continuous-domain CSPs is that of linear programming problems, where constraints must be linear inequalities forming a convex region. Linear programming problems can be solved in time polynomial in the number of variables. Problems with different types of constraints and objective functions have also been studied-quadratic programming, second order conic programming, and so on. In addition to examining the types of variables that can appear in CSPs, it is useful to look at the types of constraints.

The simplest type is the unary constraint, which restricts the value of a single variable. For example, it could be the case that South Australians actively dislike the color green. Every unary constraint can be eliminated simply by preprocessing the domain of the corresponding variable to remove any value that violates the constraint. A binary constraint relates two variables. For example, SA # NSW is a binary constraint. A binary CSP is one with only binary constraints; it can be represented as a constraint graph, as in Figure.

BACKTRACKING SEARCH FOR CSP

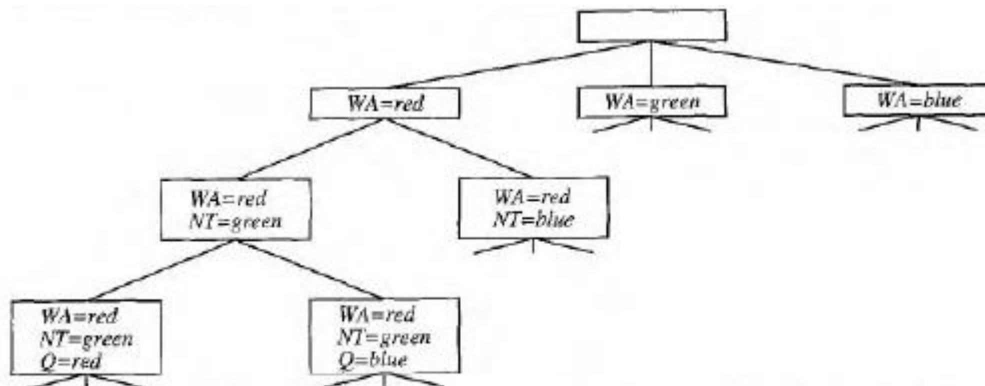
A problem is commutative if the order of application of any given set of actions has no effect on the outcome. This is the case for CSPs because, when assigning values to variables, we reach the same partial assignment, regardless of order. Therefore, all CSP search algorithms generate successors by considering possible assignments

for only a single variable at each node in the search tree. For example, at the root node of a search tree for coloring the map of Australia, we might have a choice between SA = red, SA = green, and SA = blue, but we would never choose between SA = red and WA = blue. With this restriction, the number of leaves is d^n.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
   return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment according to CONSTRAINTS[csp] then
         add {var = value) to assignment
         result ← RECURSIVE-BACKTRACKING(assignment, csp)
         if result ≠ failure then return result
         remove {var = value) from assignment
   return failure
```

A simple backtracking algorithm for constraint satisfaction problems.

The algorithm is modeled on the recursive depth-first search. The functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES can be used to implement the general-purpose heuristics discussed in the text.



Part of the search tree generated by simple backtracking for the map-coloring problem

The term backtracking search is used for 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. The algorithm is shown in Figure. Notice that it uses, in effect, the one-at-a-time method of incremental successor generation described. Also, it extends the current assignment to generate a successor, rather than copying it. Because the representation of CSPs is standardized, there is no need

to supply Backtracking-Search with a domain-specific initial state, successor function, or goal test. Part of the search tree for the Australia problem is shown in Figure, where we have assigned variables in the order WA, NT, Q, ... Plain backtracking is an uninformed algorithm, so we do not expect it to be very effective for large problems. The results for some sample problems are shown in the first column and confirm our expectations. It turns out that we can solve CSPs efficiently without such domain-specific knowledge.

Instead, we find general-purpose methods that address the following questions:

1. Which variable should be assigned next, and in what order should its values be tried?
2. What are the implications of the current variable assignments for the other unassigned variables?
3. When a path fails-that is, a state is reached in which a variable has no legal values can the search avoid repeating this failure in subsequent paths?

The subsections that follow answer each of these questions in turn.
Variable and value ordering

The backtracking algorithm contains the line

Var - SELECT-UNASSIGNED-VARIABLE
(VARIABLES [csp], assignment, csp).

By default, SELECT-UNASSIGNED-VARIABLE simply selects the next unassigned variable in the order given by the list VARIABLES [csp. This static variable ordering seldom results in the most efficient search. For example, after the assignments 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. In fact, after SA is assigned, the choices for Q, NS W, and V are all forced.
This intuitive idea-choosing the variable with the fewest "legal" values-is called the minimum remaining values (MRV) heuristic. It also has been called the "most constrained variable" or "fail-first" heuristic, the latter because it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree. If there is a variable X with zero legal values remaining, the MRV heuristic will select X and failure will be detected immediately-avoiding pointless

searches through other variables which always will fail when X is finally selected.

Propagating information through constraints

So far our search algorithm considers the constraints on a variable only at the time that the variable is chosen by SELECT-UNASSIGNED-VARIABLE. But by looking at some of the constraints earlier in the search, or even before the search has started, we can drastically reduce the search space

Forward checking

One way to make better use of constraints during search is called forward checking. Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y's domain any value that is inconsistent with the value chosen for X. Figure shows the progress of a map-coloring search with forward checking. There are two important points to notice about this example.

	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	R G B
After *V=blue*	(R)	B		R	(B)		R G B

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, green is deleted from the domains of NT, SA, and NS W. After V = blue, blue is deleted from the domains of NSW and SA, leaving SA with no legal values.

First, notice that after assigning WA = red and Q = green, the domains of NT and SA are reduced to a single value; we have eliminated branching on these variables altogether by propagating information from WA and Q. The MRV heuristic, which is an obvious partner for forward checking, would automatically select SA and NT next. (Indeed, we can view forward checking as an efficient way to incrementally compute the information that the MRV heuristic needs to do its job.) A second point to notice is that, after V = blue, the domain of SA is empty. Hence, forward checking has detected that the partial assignment {WA = red, Q = green, V = blue) is inconsistent

with the constraints of the problem, and the algorithm will therefore backtrack immediately.

Constraint propagation

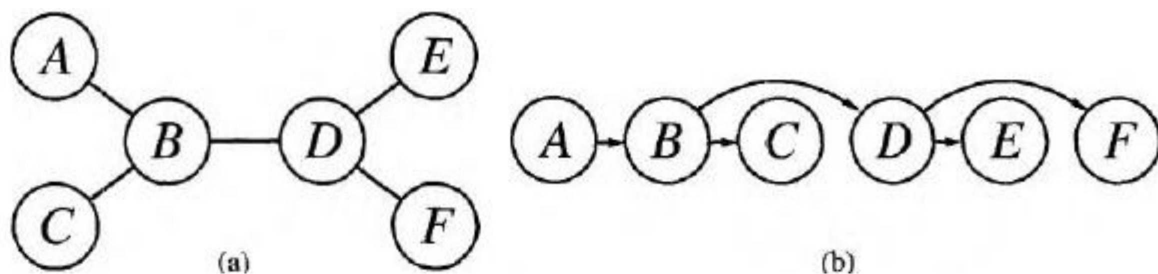
Although forward checking detects many inconsistencies, it does not detect all of them. For example, consider the third row of Figure. It shows that when WA is red and Q is green, both NT and SA are forced to be blue. But they are, adjacent and so cannot have the same value. Forward checking does not detect this as an inconsistency, because it does not look far enough ahead. Constraint propagation is the general term for propagating the implications of a constraint on one variable onto other variables; In this case we need to propagate from WA and Q onto NT and SA, (as was done by forward checking) and then onto the constraint between NT and SA to detect the inconsistency. And we want to do this fast: it is no good reducing the amount of search if we spend more time propagating constraints than we would have spent doing a simple search.

The idea of arc consistency provides a fast method of constraint propagation that is substantially stronger than forward checking. Here, "arc" refers to a directed arc in the constraint graph, such as the arc from SA to NS W. Given the current domains of SA and NS W, the arc is consistent if, for every value x of SA, there is some value y of NS W that is consistent with x. In the third row of Figure, the current domains of SA and NSW are {blue) and {red, blue) respectively. For SA = blue, there is a consistent assignment for NSW, namely, NSW = red; therefore, the arc from SA to I1JS'W is consistent. On the other hand, the reverse arc from NS W to SA is not consistent: for the assignment NS W = blue, there is no consistent assignment for SA. The arc can be made consistent by deleting the value blue from the domain of NS W.

Structure Of Problems

The structure of the problem, as represented by the constraint graph, can be used to find solutions quickly. Most of the approaches here are very general and are applicable to other problems besides CSPs, for example probabilistic reasoning. After all, the only way we can possibly hope to deal with the real world is to decompose it into many sub problems.

Intuitively, it is obvious that coloring Tasmania and coloring the mainland are independent sub problems-any solution for the mainland combined with any solution for Tasmania yields a solution for the whole map. Independence can be ascertained simply by looking for connected components of the constraint graph. Each component corresponds to a sub problem CSP_i. If assignment S, is a solution of CSP,, then U, S, is a solution of U, CSP,. Why is this important? Consider the following: suppose each CSP, has c variables from the total of n variables, where c is a constant. Then there are n/c sub problems, each of which takes at most d^C work to solve. Hence, the total work is $O(d^c n/c)$, which is linear in n; without the decomposition, the total work is $O(d^n)$, which is exponential in n.



(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.

Let's make this more concrete: dividing a Boolean CSP with n = 80 into four sub problems with c = 20 reduce the worst-case solution time from the lifetime of the universe down to less than a second. Completely independent sub problems are delicious, then, but rare. In most cases, the sub problems of a CSP are connected.
The simplest case is when the constraint graph forms a tree: any two variables are connected by at most one path.

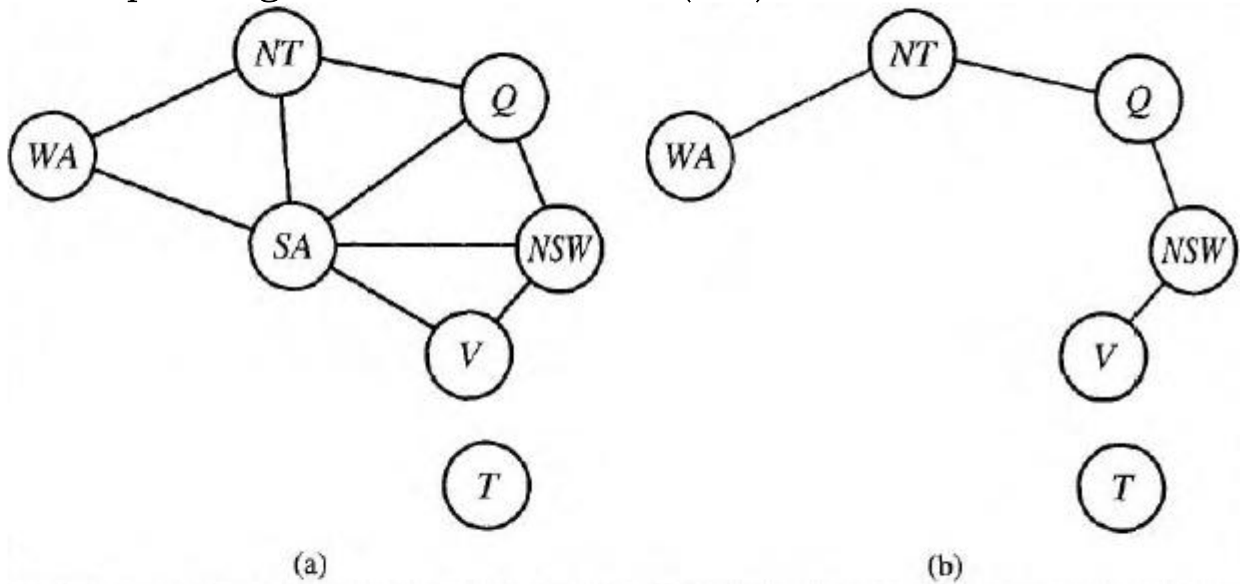
The algorithm has the following steps:

1. Choose any variable as the root of the tree, and order the variables from the root to the leaves in such a way that every node's parent in the tree precedes it in the ordering. Label the variables XI, . . . , X, in order. Now, every variable except the root has exactly one parent variable.
2. For j from n down to 2, apply arc consistency to the arc (Xi, Xj), where Xi is the parent of Xj, removing values from DOMAIN[&] as necessary.

3. For j from 1 to n, assign any value for Xj consistent with the value assigned for Xi, where Xi is the parent of Xj.

There are two key points to note. First, after step 2 the CSP is directionally arc-consistent, so the assignment of values in step 3 requires no backtracking.

Second, by applying the arc-consistency checks in reverse order in step 2, the algorithm ensures that any deleted values cannot endanger the consistency of arcs that have been processed already. The complete algorithm runs in time $O(nd^2)$.



(a) (b)

Now that we have an efficient algorithm for trees, we can consider whether more general constraint graphs can be reduced to trees somehow. There are two primary ways to do this, one based on removing nodes and one based on collapsing nodes together.
The first approach involves assigning values to some variables so that the remaining variables form a tree. Consider the constraint graph for Australia, shown again in Figure. If we could delete South Australia, the graph would become a tree, as in (b). Fortunately, we can do this (in the graph, not the continent) by fixing a value for SA and deleting from the domains of the other variables any values that are inconsistent with the value chosen for SA.

Now, any solution for the CSP after SA and its constraints are removed will be consistent with the value chosen for SA. (This works for binary CSPs; the situation is more complicated with higher-order constraints.) Therefore, we can solve the remaining tree with the

algorithm given above and thus solve the whole problem. Of course, in the general case (as opposed to map coloring) the value chosen for SA could be the wrong one, so we would need to try each of them.
The general algorithm is as follows:

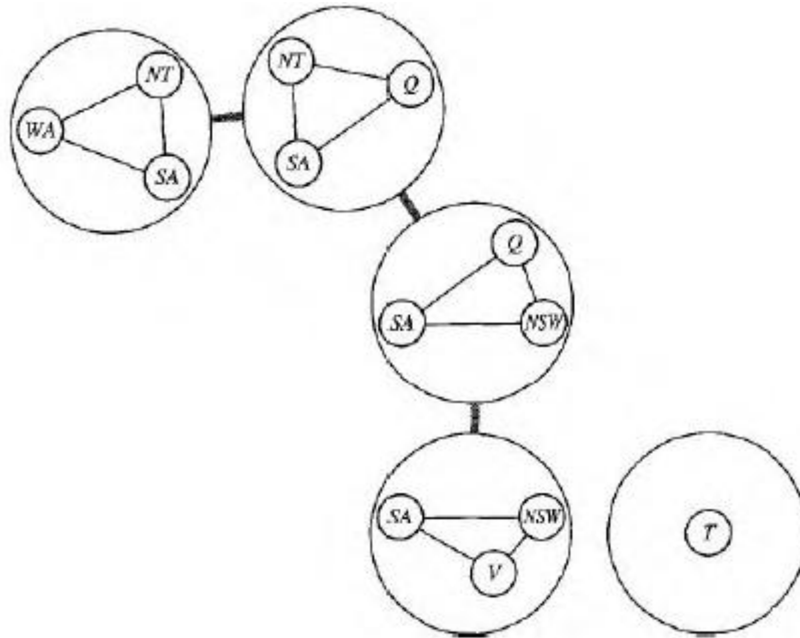
1. Choose a subset S from variables [csp] such that the constraint graph becomes a tree after removal of S. S is called a cycle cutset.
2. For each possible assignment to the variables in S that satisfies all constraints on S,
   (a) remove from the domains of the remaining variables any values that are inconsistent with the assignment for S, and
   (b) If the remaining CSP has a solution, return it together with the assignment for S.

The second approach is based on constructing 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. Like most divide-and-conquer algorithms, this works well if no subproblem is too large. Figure 5.12 shows a tree decomposition of the map coloring problem into five subproblems.

Tree decomposition must satisfy the following three requirements:

Every variable in the original problem appears in at least one of the subproblems. If two 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 two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems.
The first two conditions ensure that all the variables and constraints are represented in the decomposition. The third condition seems rather technical, but simply reflects the constraint that any given variable must have the same value in every subproblem in which it appears; the links joining subproblems in the tree enforce this constraint.



A tree decomposition of the constraint graph

We solve each subproblem independently; if anyone has no solution, we know the entire problem has no solution. If we can solve all the subproblems, then we attempt to construct global solution as follows. First, we view each subproblem as a "mega-variable" whose domain is the set of all solutions for the subproblem. For example, the leftmost subproblems in Figure is a map-coloring problem with three variables and hence has six solutions-one is {WA = red, SA = blue, NT = green). Then, we solve the constraints connecting the subproblems using the efficient algorithm for trees given earlier.

The constraints between subproblems simply insist that the subproblem solutions agree on their shared variables. For example, given the solution {WA = red, SA = blue, NT = green) for the first subproblem, the only consistent solution for the next subproblem is {SA I= blue, NT = green, Q = red). A given constraint graph admits many tree decompositions; in choosing a decomposition, the aim is to make the subproblems as small as possible. The tree width of a tree decomposition of a graph is one less than the size of the largest subproblem; the tree width of the graph itself is defined to be the minimum tree width among all its tree decompositions.
If a graph has tree width w, and we are given the corresponding tree decomposition, then the problem can be solved in $O(nd^{W+1})$ time. Hence, CSPs with constraint graphs of bounded tree width are solvable in polynomial time. Unfortunately, finding the decomposition with minimal tree width is 1VP-hard, but there are heuristic methods that work well in practice.

Adversarial Search

Adversarial search is a search, where we examine the problem which arises when we try to plan ahead of the world and other agents are planning against us.

- In previous topics, we have studied the search strategies which are only associated with a single agent that aims to find the solution which often expressed in the form of a sequence of actions.
- But, there might be some situations where more than one agent is searching for the solution in the same search space, and this situation usually occurs in game playing.
- The environment with more than one agent is termed as multi-agent environment, in which each agent is an opponent of other agent and playing against each other. Each agent needs to consider the action of other agent and effect of that action on their performance.
- So, Searches in which two or more players with conflicting goals are trying to explore the same search space for the solution, are called adversarial searches, often known as Games.
- Games are modeled as a Search problem and heuristic evaluation function, and these are the two main factors which help to model and solve games in AI.

Types of Games in AI:

- **Perfect information:** A game with the perfect information is that in which agents can look into the complete board. Agents have all the information about the game, and they can see each other moves also. Examples are Chess, Checkers, Go, etc.
- **Imperfect information:** If in a game agents do not have all information about the game and not aware with what's going on, such type of games are called the game with imperfect information, such as tic-tac-toe, Battleship, blind, Bridge, etc.

- **Deterministic games:** Deterministic games are those games which follow a strict pattern and set of rules for the games, and there is no randomness associated with them. Examples are chess, Checkers, Go, tic-tac-toe, etc.
- **Non-deterministic games:** Non-deterministic are those games which have various unpredictable events and have a factor of chance or luck. This factor of chance or luck is introduced by either dice or cards. These are random, and each action response is not fixed. Such games are also called as stochastic games. Example: Backgammon, Monopoly, Poker, etc.

Zero-Sum Game

- Zero-sum games are adversarial search which involves pure competition.
- In Zero-sum game each agent's gain or loss of utility is exactly balanced by the losses or gains of utility of another agent.
- One player of the game try to maximize one single value, while other player tries to minimize it.
- Each move by one player in the game is called as ply.
- Chess and tic-tac-toe are examples of a Zero-sum game.

Zero-sum game: Embedded thinking

The Zero-sum game involved embedded thinking in which one agent or player is trying to figure out:

- What to do.
- How to decide the move
- Needs to think about his opponent as well
- The opponent also thinks what to do

Each of the players is trying to find out the response of his opponent to their actions. This requires embedded thinking or backward reasoning to solve the game problems in AI.

Optimal decision in games

A game can be defined as a type of search in AI which can be formalized of the following elements:

- Initial state: It specifies how the game is set up at the start.
- Player(s): It specifies which player has moved in the state space.
- Action(s): It returns the set of legal moves in state space.
- Result(s, a): It is the transition model, which specifies the result of moves in the state space.
- Terminal-Test(s): Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called terminal states.
- Utility(s, p): A utility function gives the final numeric value for a game that ends in terminal states s for player p. It is also called payoff function. For Chess, the outcomes are a win, loss, or draw and its payoff values are +1, 0, ½. And for tic-tac-toe, utility values are +1, -1, and 0.

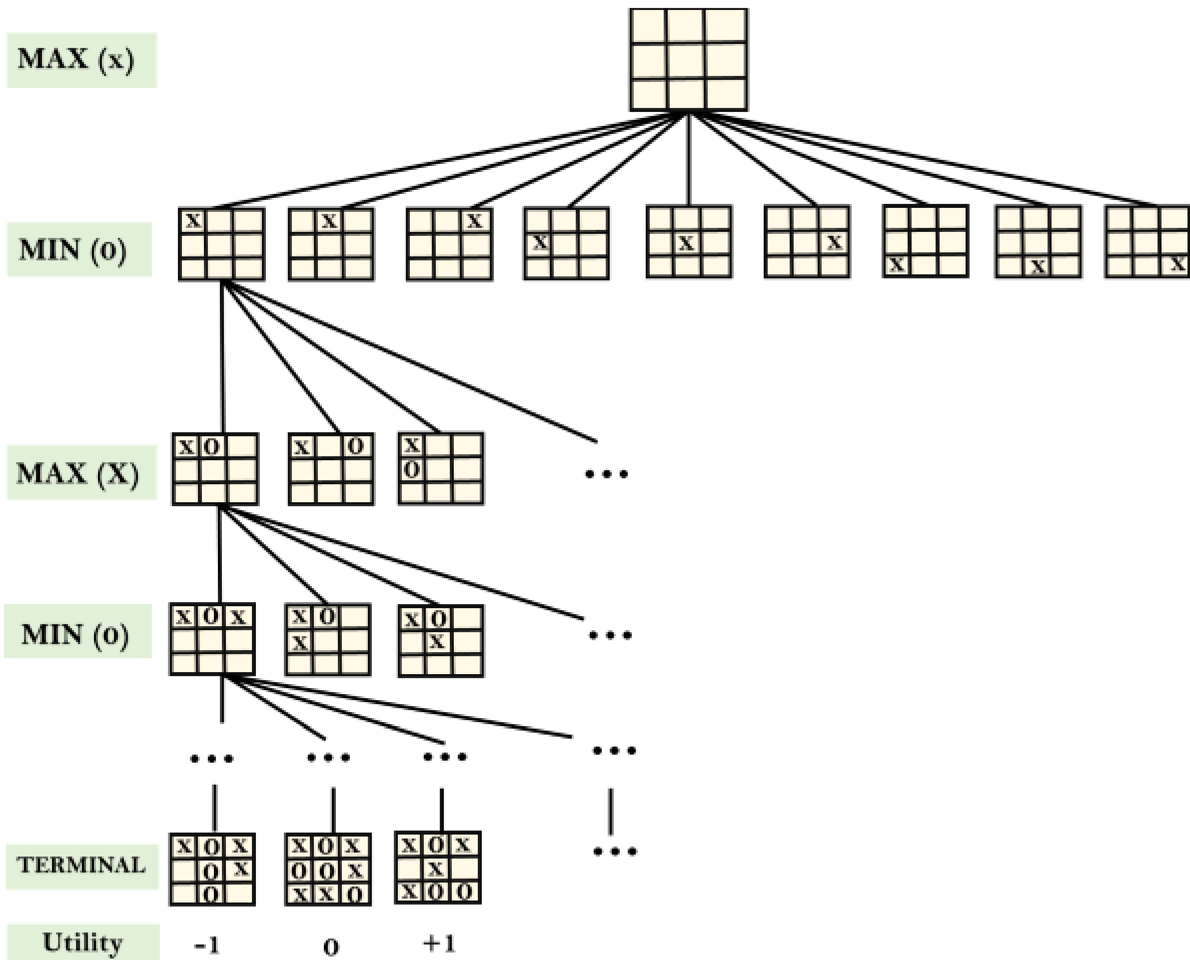
Game tree:

A game tree is a tree where nodes of the tree are the game states and Edges of the tree are the moves by players. Game tree involves initial state, actions function, and result Function.

Example: Tic-Tac-Toe game tree:

The following figure is showing part of the game-tree for tic-tac-toe game. Following are some key points of the game:

- There are two players MAX and MIN.
- Players have an alternate turn and start with MAX.
- MAX maximizes the result of the game tree
- MIN minimizes the result.



Example Explanation:

- From the initial state, MAX has 9 possible moves as he starts first. MAX place x and MIN place o, and both player plays alternatively until we reach a leaf node where one player has three in a row or all squares are filled.
- Both players will compute each node, minimax, the minimax value which is the best achievable utility against an optimal adversary.
- Suppose both the players are well aware of the tic-tac-toe and playing the best play. Each player is doing his best to prevent another one from winning. MIN is acting against Max in the game.
- So in the game tree, we have a layer of Max, a layer of MIN, and each layer is called as Ply. Max place x, then MIN puts o to

prevent Max from winning, and this game continues until the terminal node.

- In this either MIN wins, MAX wins, or it's a draw. This game-tree is the whole search space of possibilities that MIN and MAX are playing tic-tac-toe and taking turns alternately.

Hence adversarial Search for the minimax procedure works as follows:

- It aims to find the optimal strategy for MAX to win the game.
- It follows the approach of Depth-first search.
- In the game tree, optimal leaf node could appear at any depth of the tree.
- Propagate the minimax values up to the tree until the terminal node discovered.

In a given game tree, the optimal strategy can be determined from the minimax value of each node, which can be written as MINIMAX(n). MAX prefer to move to a state of maximum value and MIN prefer to move to a state of minimum value then:

For a state S MINIMAX(s) =

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

Mini-Max Algorithm in Artificial Intelligence

- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
- Mini-Max algorithm uses recursion to search through the game-tree.

- Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state.
- In this algorithm two players play the game, one is called MAX and other is called MIN.
- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

Pseudo-code for MinMax Algorithm:

```
function minimax(node, depth, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
maxEva= -infinity
 for each child of node do
 eva= minimax(child, depth-1, false)
maxEva= max(maxEva,eva)        //gives Maximum of the values
return maxEva

else                         // for Minimizer player
 minEva= +infinity
 for each child of node do
 eva= minimax(child, depth-1, true)
 minEva= min(minEva, eva)         //gives minimum of the values
 return minEva
```

Initial call:

Minimax (node, 3, true)

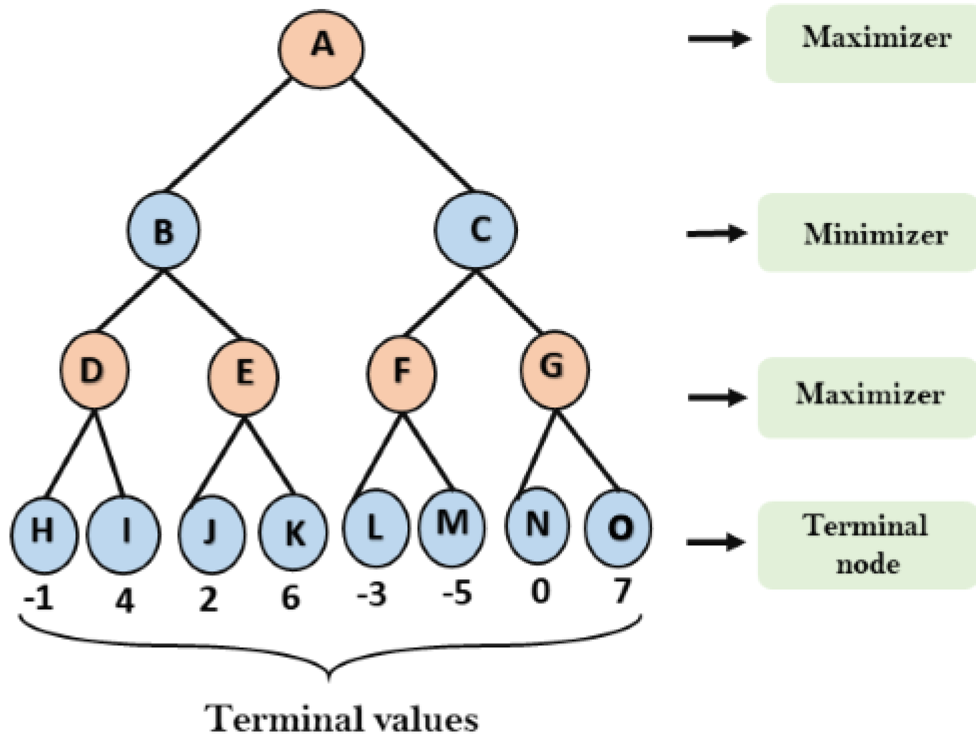
Working of Min-Max Algorithm:

- The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.
- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.
- At the terminal node, the terminal values are given so we will compare those values and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

Step-1:

In the first step, the algorithm generates the entire game-tree and applies the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree.

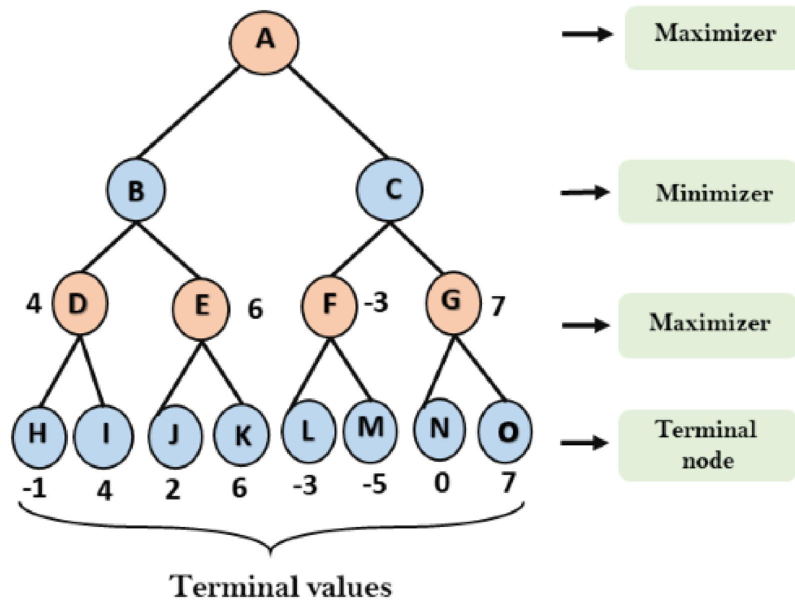
Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimize will take next turn which has worst-case initial value = +infinity.



Step 2:

Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

- For node D max(-1,- -∞) => max(-1,4)= 4
- For Node E max(2, -∞) => max(2, 6)= 6
- For Node F max(-3, -∞) => max(-3,-5) = -3
- For node G max(0, -∞) = max(0, 7) = 7

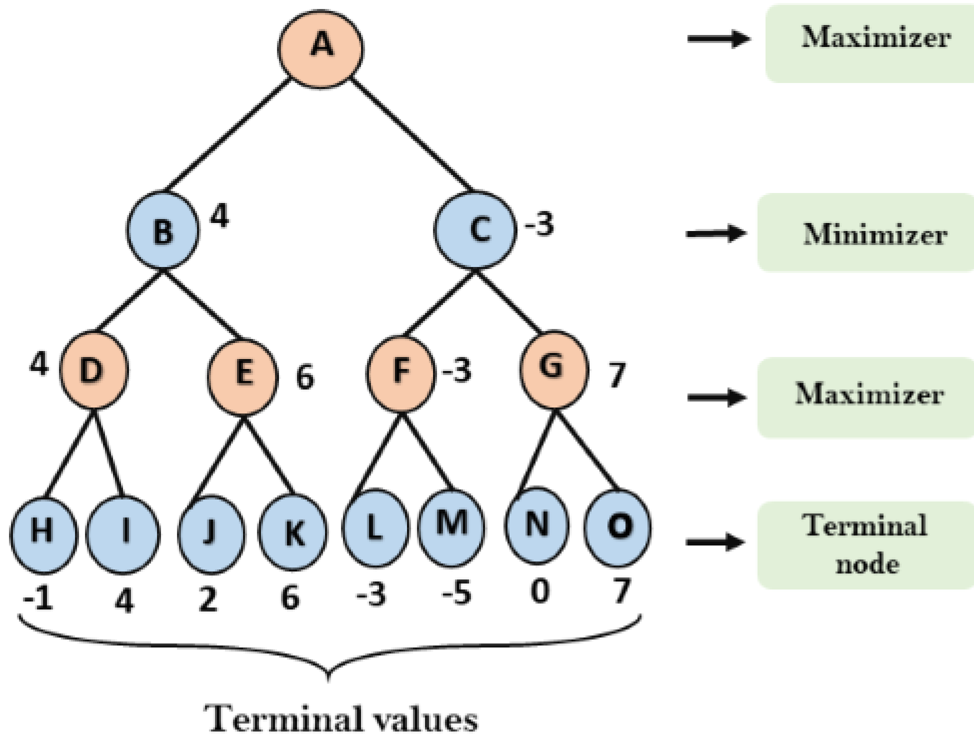


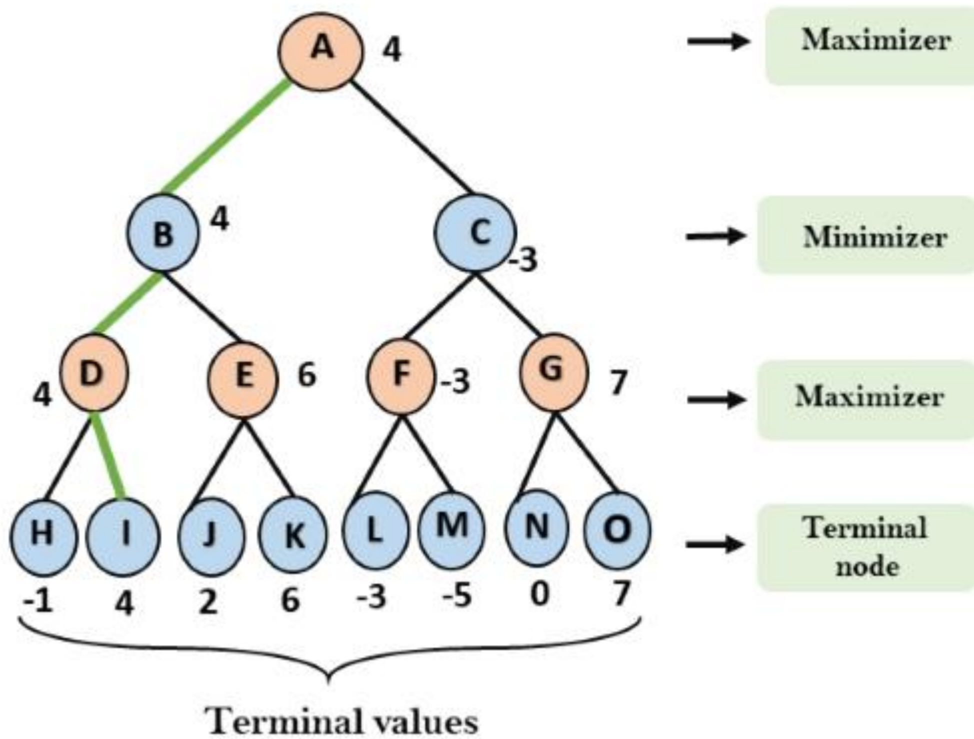
Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

- For node B= min(4,6) = 4
- For node C= min (-3, 7) = -3

Step 4: Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

- For node A max(4, -3)= 4





That was the complete workflow of the minimax two player game.

Properties of Mini-Max algorithm:

- Complete- Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- Optimal- Min-Max algorithm is optimal if both opponents are playing optimally.
- Time complexity- As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is $O(b^m)$, where b is branching factor of the game-tree, and m is the maximum depth of the tree.
- Space Complexity- Space complexity of Mini-max algorithm is also similar to DFS which is O(bm).

Limitation of the minimax Algorithm:

The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games has a huge branching factor, and the player has lots of choices to decide. This limitation of the minimax algorithm can be improved from alpha-beta pruning.

Alpha-Beta Pruning

- 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 -∞.

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 +∞.

- 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 is $\alpha >= \beta$

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.

Pseudo-code for Alpha-beta Pruning:

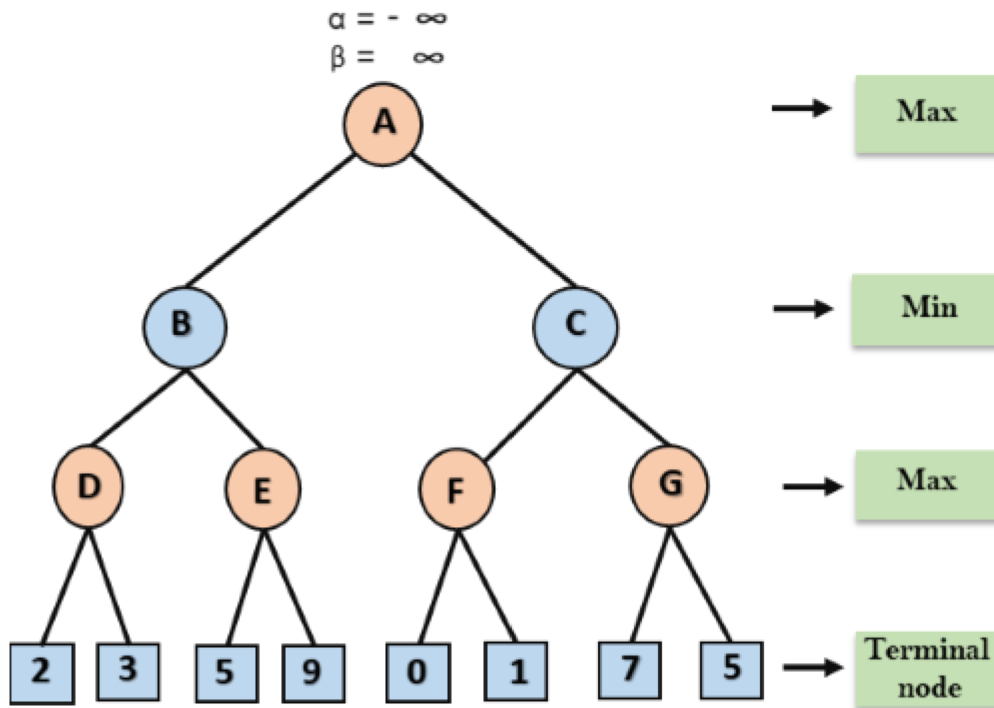
```
function minimax(node, depth, alpha, beta, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
   maxEva= -infinity
   for each child of node do
   eva= minimax(child, depth-1, alpha, beta, False)
  maxEva= max(maxEva, eva)
  alpha= max(alpha, maxEva)
   if beta<=alpha
 break
 return maxEva
```

```
else                          // for Minimizer player
   minEva= +infinity
   for each child of node do
   eva= minimax(child, depth-1, alpha, beta, true)
   minEva= min(minEva, eva)
   beta= min(beta, eva)
    if beta<=alpha
  break
 return minEva
```

Working of Alpha-Beta Pruning:

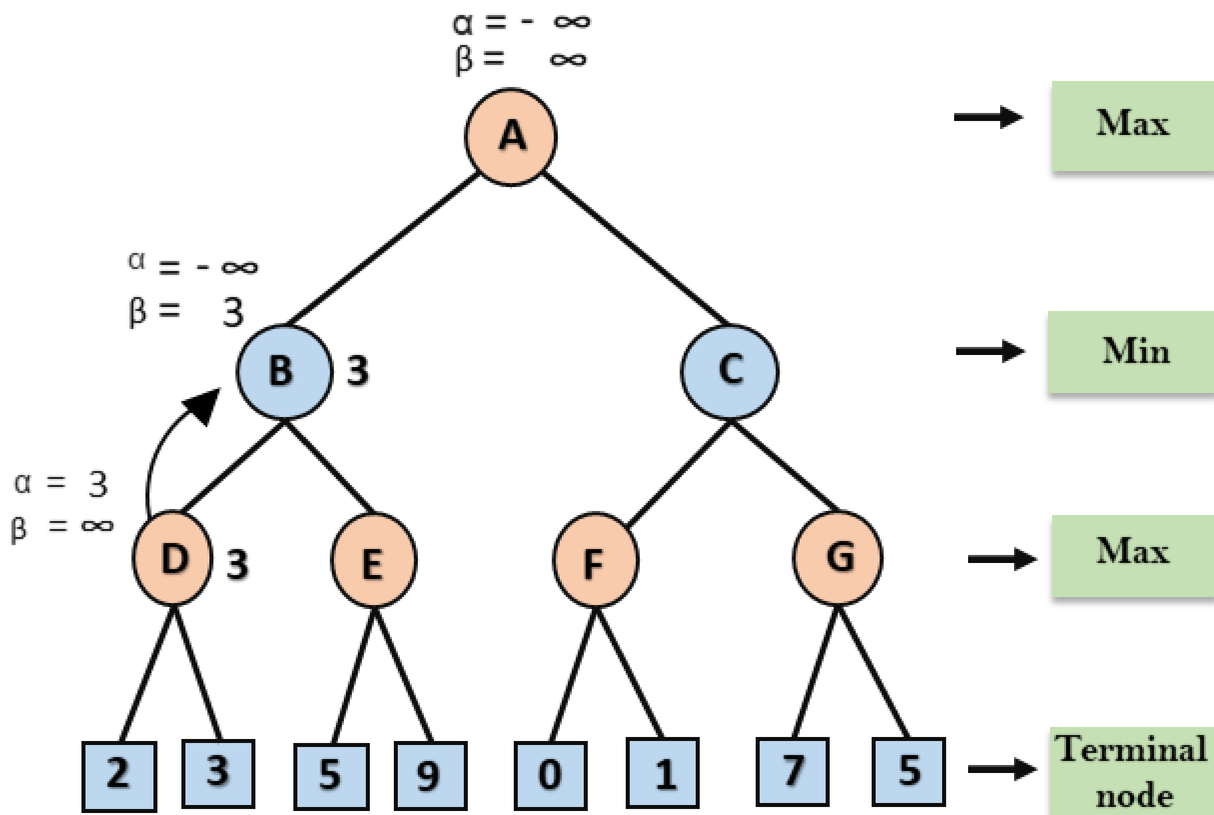
Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

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 backtracks to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.

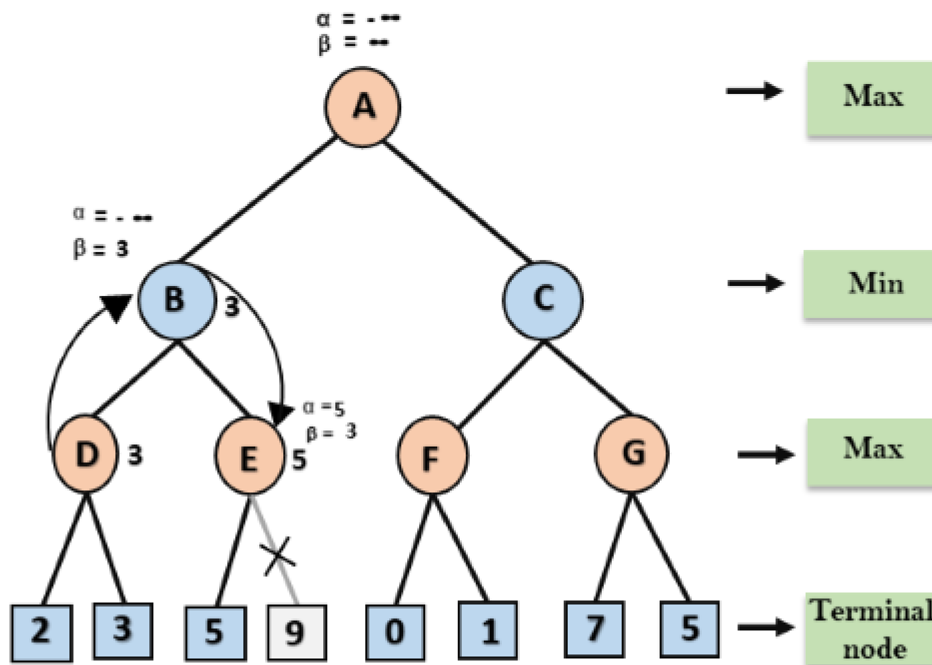


In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

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 (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.

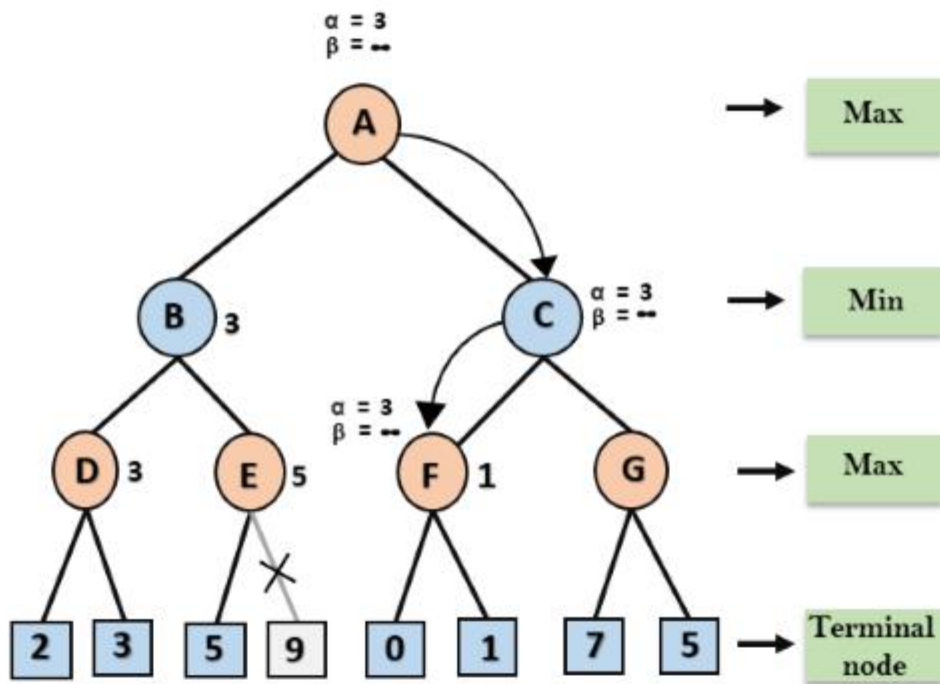
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

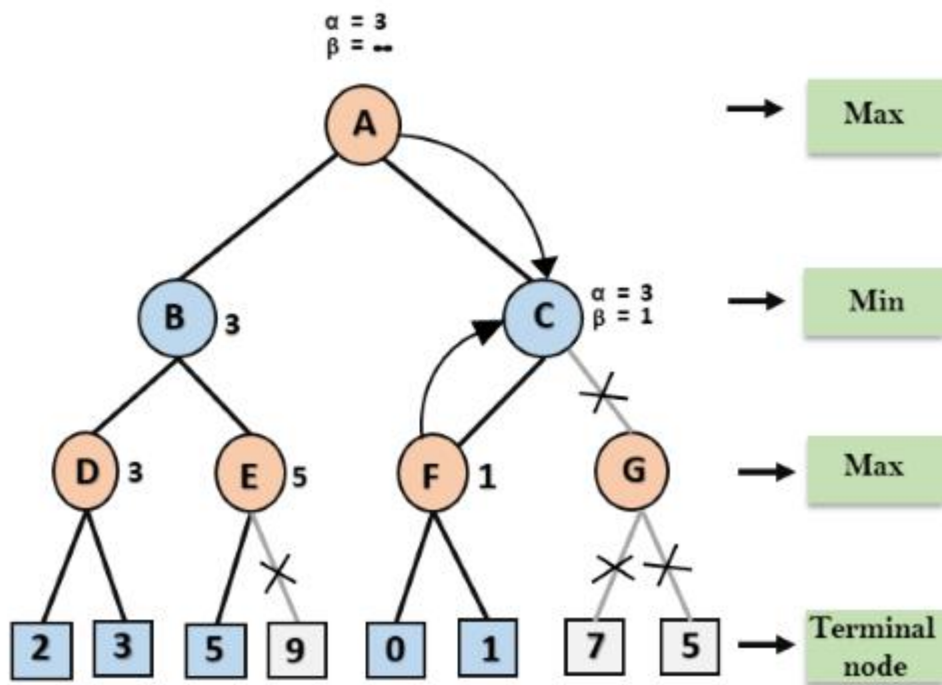
At node C, α=3 and β= +∞, and the same values will be passed on to node F.



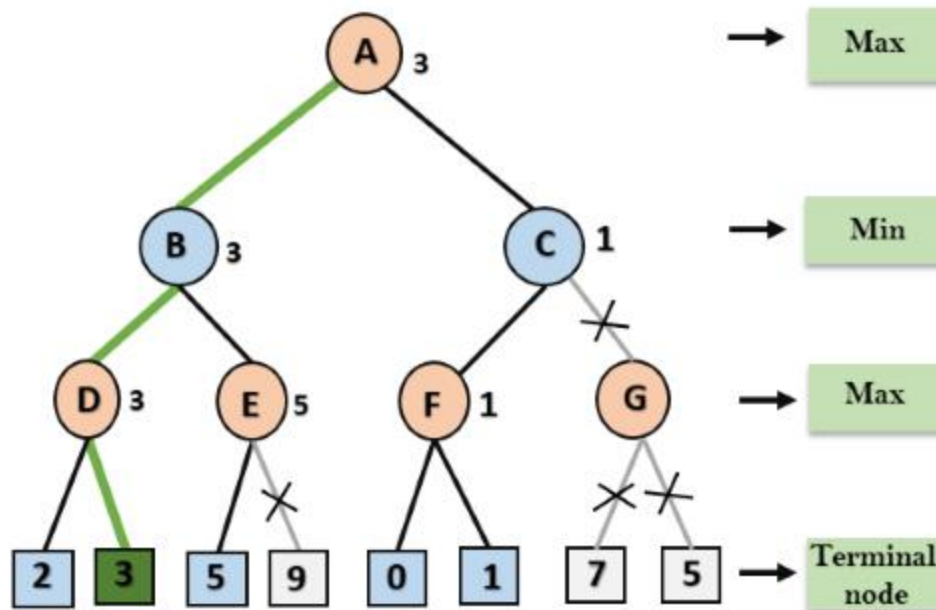
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.

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


α = 3
β = ∞
A
B
3
C
α = 3
β = ∞
α = 3
β = ∞
D
3
E
5
F
1
G
2
3
5
9
0
1
7
5
Max
Min
Max
Terminal node


α = 3
β = ∞
A
B
3
C
α = 3
β = 1
D
3
E
5
F
1
G
2
3
5
9
0
1
7
5
Max
Min
Max
Terminal node

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 the showing the nodes which are computed and nodes which has never 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 $O(b^m)$.

- 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 $O(b^{m/2})$.

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 book keep the states, as there is a possibility that states may repeat

MCQ

1. Online search is a necessary idea for an exploration problem, where the states and actions are

   a. **Unknown to the agent**
   b. Known to the agent
   c. Both
   d. None of the above

2. The step cost function includes
   a. c(s,a,c)
   b. c(s,a,b)
   c. **c(s,a,s)**
   d. c(s,a,d)

3. In some cases the best achievable competitive ratio is
   a. 0
   b. 1
   c. **Infinite**
   d. None of the above

4. The online DFS agent works only in state spaces where the actions are
    a. Irreversible
    b. Reversible
    c. Both
    d. None of the above

5. LRTA* means
    a. Local Ready Time A*
    b. Learning Ready Time A*
    c. Learning Real Time A*
    d. Local Real Time A*

6. CSP means
    a. Constant Satisfaction Problems.
    b. Constraint Statement Problems.
    c. Constraint Satisfaction Problems.
    d. Constraint Salesman Problems.

7. In CSP state is defined by
    a. Variables
    b. Domain
    c. Values
    d. All the above

8. Which search agent operates by interleaving computation and action?
    a) Offline search
    b) Online search
    c) Breadth-first search
    d) Depth-first search

9. Which of the following algorithm is online search algorithm?

    a) Breadth-first search algorithm
    b) Depth-first search algorithm
    c) Hill-climbing search algorithm
    d) None of the mentioned

10. Which of the Following problems can be modeled as CSP?

a) 8-Puzzle problem
b) 8-Queen problem
c) Map coloring problem
d) **All of the mentioned**

CONCLUSION:

Upon completion of this, Students should be able to

- Understand the Online Search Agents.
- Understand Constraint Satisfaction Problems.
- Understand Adversarial Search.

REFERENCES

1. David Poole, Alan Mackworth, Randy Goebel, "Computational Intelligence: a Logical Approach", Oxford University Press, 2004.

2. G. Luger, "Artificial Intelligence: Structures and Strategies for Complex Problem Solving", Fourth Edition, Pearson Education, 2002.

ASSIGNMENT

1. Explain about Online search Agents.
2. Explain about online search problems.
3. Explain about CSP.
4. Explain about optimal decision in games.
5. Explain about AlphaBeta Pruning.

UNIT-4

Logical agents – Knowledge Based Agents, The Wumpus World, Propositional Logic-A very simple Logic –First Order logic– inferences in first order logic – forward chaining – backward chaining – Unification – Resolution.

AIM & OBJECTIVES

- To understand the Logical Agents in AI.
- To understand Wumpus World in AI.
- To understand First Order logic in AI.
- Comparing Forward Chaining – Backward Chaining in AI.

PRE- REQUISITE: Basic knowledge of Computer Architecture.

Logical Agents

The representation of knowledge and the reasoning processes that bring knowledge to life-are central to the entire field of artificial intelligence. Humans, it seems, know things and do reasoning. Knowledge and reasoning are also important for artificial agents because they enable successful behaviors that would be very hard to achieve otherwise. We have seen that knowledge of action outcomes enables problem solving agents to perform well in complex environments. A reflex agent could only find its way from Arad to Bucharest by dumb luck.

The knowledge of problem-solving agents is however, very specific and inflexible. A chess program can calculate the legal moves of its king, but does not know in any useful sense that no piece can be on two different squares at the same time. Knowledge-based agents can benefit from knowledge expressed in very general forms, combining and recombining information to suit myriad purposes. Often, this process can be quite far removed from the needs of the moment-as when a mathematician proves a theorem or an astronomer calculates the earth's life expectancy.

Knowledge and reasoning also play a crucial role in dealing with partially observable environments. A knowledge-based agent can combine general knowledge with current percepts to infer hidden aspects of the current state prior to selecting actions. For example, a physician diagnoses a patient-that is, infers a disease state that is not directly observable prior to choosing a treatment.

Some of the knowledge that the physician uses is in the form of rules learned from textbooks and teachers, and some is in the form of patterns of association that the physician may not be able to consciously describe. If it's inside the physician's head, it counts as knowledge.

Understanding natural language also requires inferring hidden state, namely, the intention of the speaker. When we hear, "John saw the diamond through the window and coveted it," we know "it" refers to the diamond and not the window-we reason, perhaps unconsciously, with our knowledge of relative value. Similarly, when we hear, "John threw the brick through the window and broke it," we know "it" refers to the window.

Reasoning allows us to cope with the virtually infinite variety of utterances using a finite store of commonsense knowledge. Problem-solving agents have difficulty with this kind of ambiguity because their representation of contingency problems is inherently exponential. Our final reason for studying knowledge-based agents is their flexibility. They are able to accept new tasks in the form of explicitly described goals, they can achieve competence quickly by being told or learning new knowledge about the environment, and they can adapt to changes in the environment by updating the relevant knowledge.

Knowledge-Based Agent in Artificial intelligence

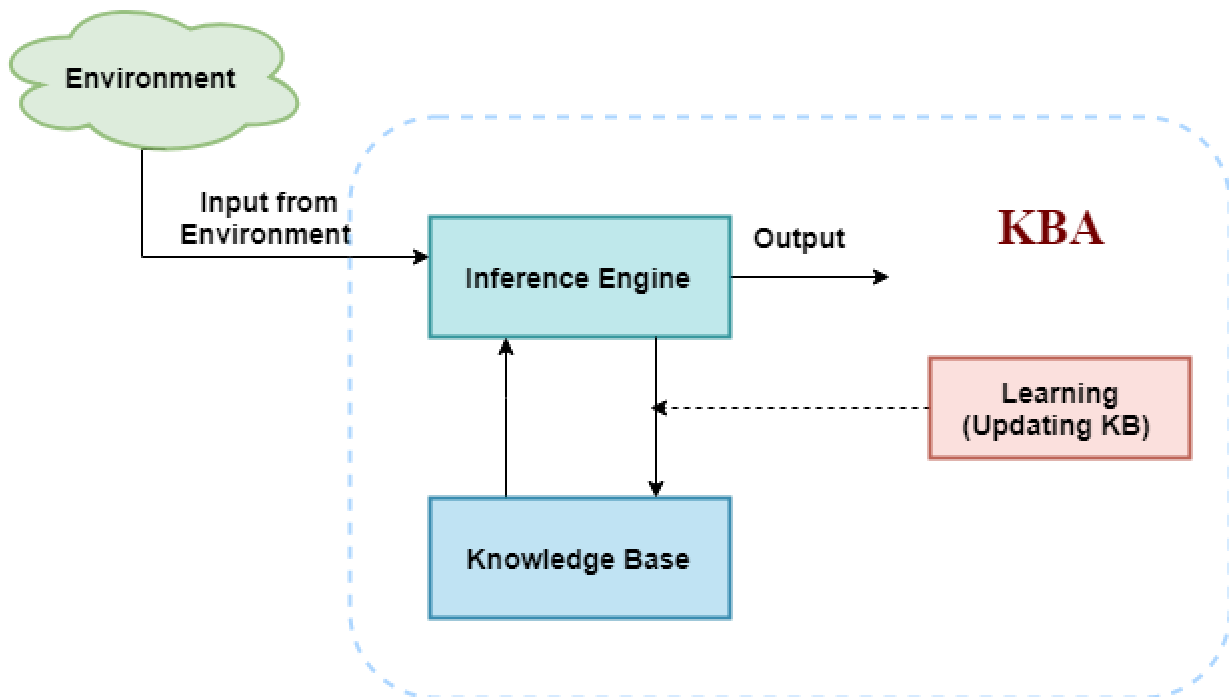
- An intelligent agent needs knowledge about the real world for taking decisions and reasoning to act efficiently.
- Knowledge-based agents are those agents who have the capability of maintaining an internal state of knowledge, reason over that knowledge, update their knowledge after observations and 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) takes 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:

At this level, we understand that how the knowledge representation of knowledge is stored. At this level, sentences 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 implements 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. 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. 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.

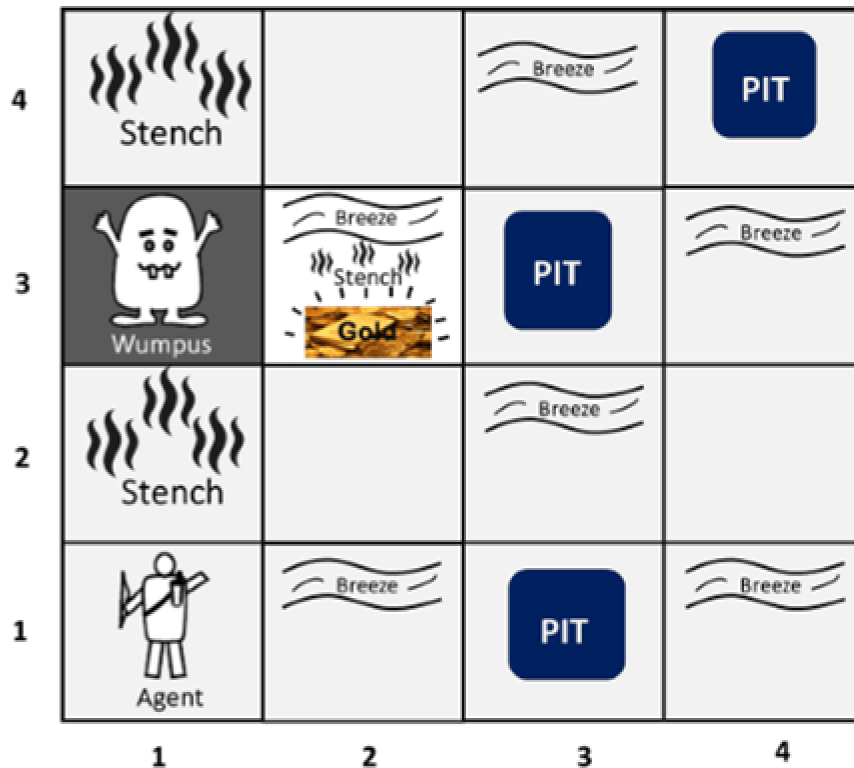
Wumpus world:

The Wumpus world is a simple world example to illustrate the worth of a knowledge-based agent and to represent knowledge representation. It was inspired by a video game Hunt the Wumpus by Gregory Yob in 1973.

The Wumpus world is a cave which has 4/4 rooms connected with passageways. So there are total 16 rooms which are connected with each other. We have a knowledge-based agent who will go forward in this world. The cave has a room with a beast which is called Wumpus, who eats anyone who enters the room. The Wumpus can be shot by the agent, but the agent has a single arrow.

In the Wumpus world, there are some Pits rooms which are bottomless, and if agent falls in Pits, then he will be stuck there forever. The exciting thing with this cave is that in one room there is a possibility of finding a heap of gold. So the agent goal is to find the gold and climb out the cave without fallen into Pits or eaten by Wumpus. The agent will get a reward if he comes out with gold, and he will get a penalty if eaten by Wumpus or falls in the pit.

Following is a sample diagram for representing the Wumpus world. It is showing some rooms with Pits, one room with Wumpus and one agent at (1, 1) square location of the world.



There are also some components which can help the agent to navigate the cave. These components are given as follows:

The rooms adjacent to the Wumpus room are smelly, so that it would have some stench.

a. The room adjacent to PITs has a breeze, so if the agent reaches near to PIT, then he will perceive the breeze.
b. There will be glitter in the room if and only if the room has gold.
c. The Wumpus can be killed by the agent if the agent is facing to it, and Wumpus will emit a horrible scream which can be heard anywhere in the cave.

PEAS description of Wumpus world:

To explain the Wumpus world we have given PEAS description as below:

Performance measure:

- +1000 reward points if the agent comes out of the cave with the gold.
- -1000 points penalty for being eaten by the Wumpus or falling into the pit.

- -1 for each action, and -10 for using an arrow.
- The game ends if either agent dies or came out of the cave.

Environment:

- A 4*4 grid of rooms.
- The agent initially in room square [1, 1], facing toward the right.
- Location of Wumpus and gold are chosen randomly except the first square [1,1].
- Each square of the cave can be a pit with probability 0.2 except the first square.

Actuators:

- Left turn,
- Right turn
- Move forward
- Grab
- Release
- Shoot.

Sensors:

- The agent will perceive the stench if he is in the room adjacent to the Wumpus. (Not diagonally).
- The agent will perceive breeze if he is in the room directly adjacent to the Pit.
- The agent will perceive the glitter in the room where the gold is present.
- The agent will perceive the bump if he walks into a wall.
- When the Wumpus is shot, it emits a horrible scream which can be perceived anywhere in the cave.
- These percepts can be represented as five element list, in which we will have different indicators for each sensor.

- Example if agent perceives stench, breeze, but no glitter, no bump, and no scream then it can be represented as: [Stench, Breeze, None, None, None].

The Wumpus world Properties:

- Partially observable: The Wumpus world is partially observable because the agent can only perceive the close environment such as an adjacent room.
- Deterministic: It is deterministic, as the result and outcome of the world are already known.
- Sequential: The order is important, so it is sequential.
- Static: It is static as Wumpus and Pits are not moving.
- Discrete: The environment is discrete.
- One agent: The environment is a single agent as we have one agent only and Wumpus is not considered as an agent.

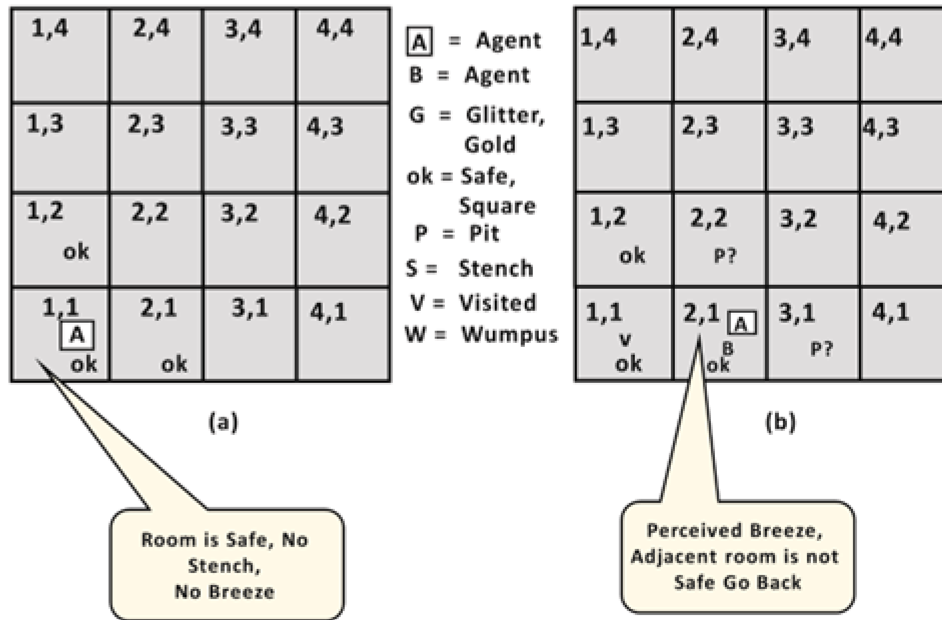
Exploring the Wumpus world:

Now we will explore the Wumpus world and will determine how the agent will find its goal by applying logical reasoning.

Agent's First step:

Initially, the agent is in the first room or on the square [1,1], and we already know that this room is safe for the agent, so to represent on the below diagram (a) that room is safe we will add symbol OK. Symbol A is used to represent agent, symbol B for the breeze, G for Glitter or gold, V for the visited room, P for pits, W for Wumpus.

At Room [1,1] agent does not feel any breeze or any Stench which means the adjacent squares are also OK.



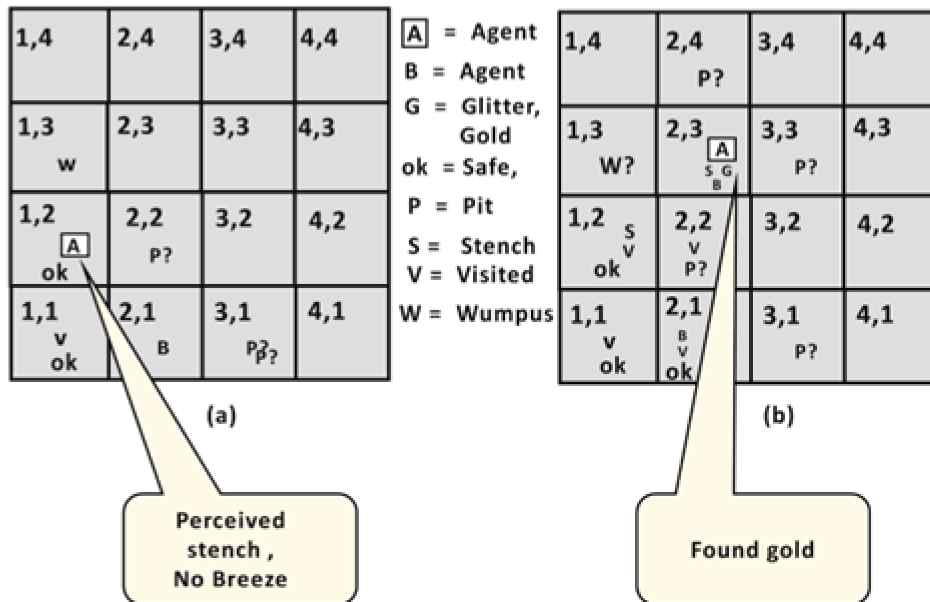
Agent's second Step:

Now agent needs to move forward, so it will either move to [1, 2], or [2,1]. Let's suppose agent moves to the room [2, 1], at this room agent perceives some breeze which means Pit is around this room. The pit can be in [3, 1], or [2,2], so we will add symbol P? to say that, is this Pit room?

Now agent will stop and think and will not make any harmful move. The agent will go back to the [1, 1] room. The room [1,1], and [2,1] are visited by the agent, so we will use symbol V to represent the visited squares.

Agent's third step:

At the third step, now agent will move to the room [1,2] which is OK. In the room [1,2] agent perceives a stench which means there must be a Wumpus nearby. But Wumpus cannot be in the room [1,1] as by rules of the game, and also not in [2,2] (Agent had not detected any stench when he was at [2,1]). Therefore agent infers that Wumpus is in the room [1,3], and in current state, there is no breeze which means in [2,2] there is no Pit and no Wumpus. So it is safe, and we will mark it OK, and the agent moves further in [2,2].



Agent's fourth step:

At room [2,2], here no stench and no breezes present so let's suppose agent decides to move to [2,3]. At room [2,3] agent perceives glitter, so it should grab the gold and climb out of the cave.

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:

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

Following are some basic facts about propositional logic:

- Propositional logic is also called Boolean logic as it works on 0 and 1.
- 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.

- Propositions can be either true or false, but it cannot be both.
- Propositional logic consists of an object, relations or function, and logical connectives.
- These connectives are also called logical operators.
- The propositions and connectives are the basic elements of the propositional logic.
- Connectives can be said as a logical operator which connects two sentences.
- A proposition formula which is always true is called tautology, and it is also called a valid sentence.
- A proposition formula which is always false is called Contradiction.
- A proposition formula which has both true and false values is called
- 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

- 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:

1. a) 2+2 is 4, it is an atomic proposition as it is a true fact.
2. b) "The Sun is cold" is also a proposition as it is a false fact.

- Compound proposition: Compound propositions are constructed by combining simpler or atomic propositions, using parenthesis and logical connectives.

Example:

1. a) "It is raining today, and street is wet."
2. 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⇔ 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 ⇔ Q.

Following is the summarized table for Propositional Logic Connectives:

Connective symbols	Word	Technical term	Example
∧	AND	Conjunction	A ∧ B
∨	OR	Disjunction	A ∨ B
→	Implies	Implication	A → B
⇔	If and only if	Biconditional	A⇔ B
¬ or ~	Not	Negation	¬ A or ¬ 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	¬ P
True	**False**
False	**True**

For Conjunction:

P	Q	P∧ Q
True	True	**True**
True	False	**False**
False	True	**False**
False	False	**False**

For disjunction:

P	Q	P V Q.
True	True	**True**
False	True	**True**
True	False	**True**
False	False	**False**

For Implication:

P	Q	P→ Q
True	True	**True**
True	False	**False**
False	True	**True**
False	False	**True**

For Biconditional:

P	Q	P⇔ 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 8n Tuples as we have taken three proposition symbols.

P	Q	R	¬R	Pv Q	PvQ→¬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

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⇔B. In below truth table we can see that column for ¬A∨ B and A→B, are identical hence A is Equivalent to B

A	B	¬A	¬A∨ B	A→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∧ Q= Q ∧ P, or
  - P ∨ Q = Q ∨ P.
- Associativity:
  - (P ∧ Q) ∧ R= P ∧ (Q ∧ R),
  - (P ∨ Q) ∨ R= P ∨ (Q ∨ R)
- Identity element:
  - P ∧ True = P,
  - P ∨ True= True.
- Distributive:
  - P∧ (Q ∨ R) = (P ∧ Q) ∨ (P ∧ R).
  - P ∨ (Q ∧ R) = (P ∨ Q) ∧ (P ∨ R).
- DE Morgan's Law:
  - ¬ (P ∧ Q) = (¬P) ∨ (¬Q)
  - ¬ (P ∨ Q) = (¬ P) ∧ (¬Q).
- Double-negation elimination:
  - ¬ (¬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.

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:

d. Syntax
e. 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	∧, ∨, ¬, ⇒, ⇔
Equality	==
Quantifier	∀, ∃

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: => Brothers(Ravi, Ajay).
Chinky is a cat: => cat (Chinky).

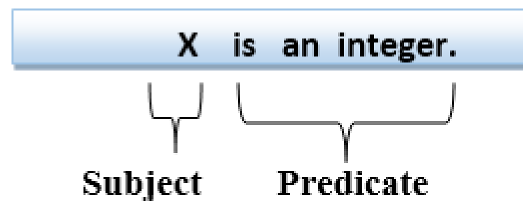
Complex Sentences:

- 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 ∀, which resembles an inverted A.

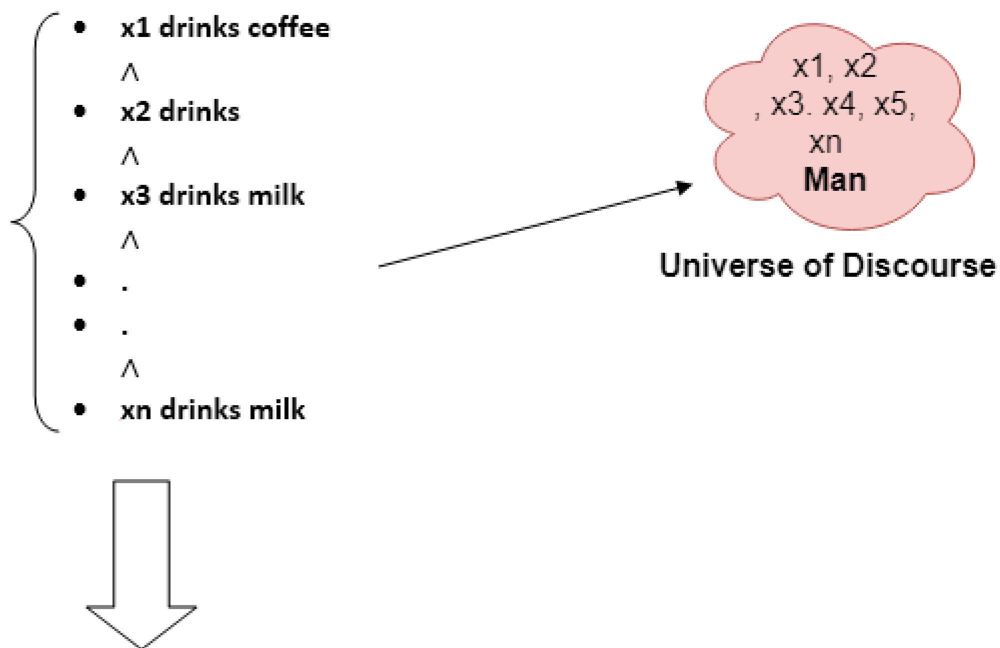
If x is a variable, then ∀x is read as:

- For all x
- For each x
- For every x.

Example:

All man drink coffee.

Let a variable x which refers to a cat so all x can be represented in UOD as below:



∀x man(x) → drink (x, 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.

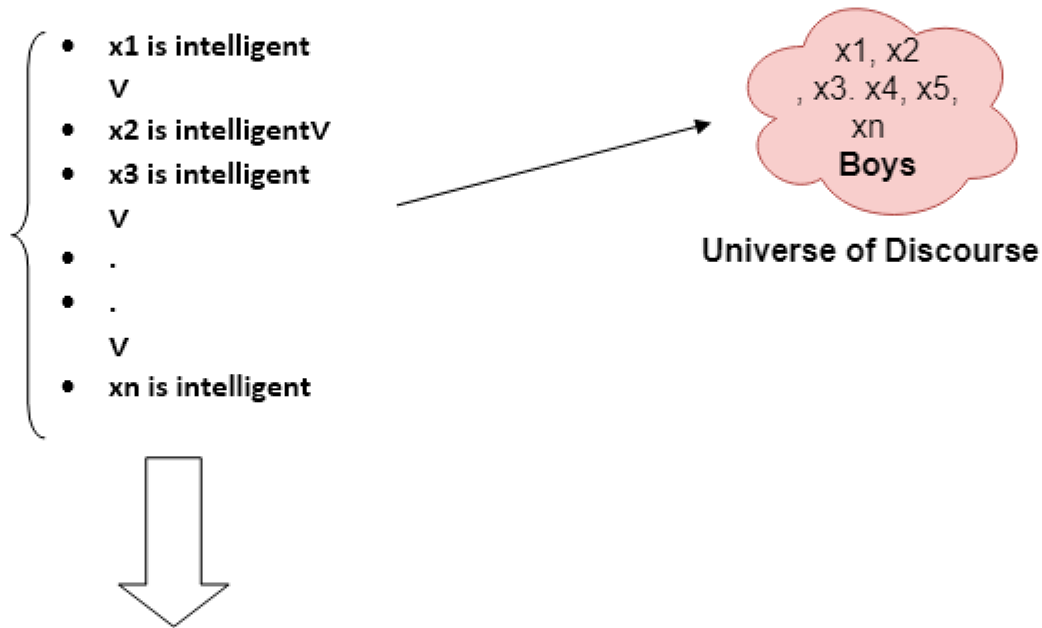
It is denoted by the logical operator ∃, which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.

If x is a variable, then existential quantifier will be ∃x or ∃(x). And it will be read as:

- There exists a 'x.'
- For some 'x.'
- For at least one 'x.'

Example:

Some boys are intelligent.



Ex: boys(x) ∧ intelligent(x)

It will be read as: There are some x where x is a boy who is intelligent.

- The main connective for universal quantifier ∀ is implication →.
- The main connective for existential quantifier ∃ is and ∧.

Properties of Quantifiers:

- In universal quantifier, ∀x∀y is similar to ∀y∀x.
- In Existential quantifier, ∃x∃y is similar to ∃y∃x.
- ∃x∀y is not similar to ∀y∃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. ∀x bird(x) →fly(x).

2. Every man respects his parent.
   In this question, the predicate is "respect(x, y)," where x=man, and y= parent.

   Since there is every man so will use ∀, and it will be represented ∀x man(x) → respects (x, parent).

3. Some boys play cricket.
   In this question, the predicate is "play(x, y)," where x= boys, and y= game. Since there are some boys so we will use ∃, and it will be represented as: ∃x boys(x) → play(x, cricket).

4. Not all students like both Mathematics and Science.
   In this question, the predicate is "like(x, y)," where x= student, and y= subject.
   Since there are not all students, so we will use ∀ with negation, so following representation for this:
   ¬∀ (x) [student(x) → like(x, Mathematics) ∧ like(x, 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:
   ∃(x) [student(x) → failed (x, Mathematics) ∧∀ (y) [¬(x==y) ∧ student(y) → ¬failed (x, 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: ∀x ∃(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: ∀x [A (x) B(y)], here x and y are the bound variables.

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".

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: ¬(x=y) which is equivalent to x ≠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 ∀ 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 ∀ 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 ∀ x P(x) for any object in the universe of discourse.
- It can be represented as: $\frac{\forall x\, P(x)}{P(c)}$.

Example:1.

IF "Every person like ice-cream"=> ∀x P(x) so we can infer that "John likes ice-cream" => 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:

∀x king(x) ∧ greedy (x) → Evil (x),

So from this information, we can infer any of the following statements using Universal Instantiation:

- King(John) ∧ Greedy (John) → Evil (John),
- King(Richard) ∧ Greedy (Richard) → Evil (Richard),
- King(Father(John)) ∧ Greedy (Father(John)) → 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 ∃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.

- It can be represented as: $\frac{\exists x\, P(x)}{P(c)}$

Example:

From the given sentence: ∃x Crown(x) ∧ OnHead(x, John),

So we can infer: Crown(K) ∧ OnHead(K, 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.
- It can be represented as: $\frac{P(c)}{\exists x P(x)}$
- 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 pi, pi', q. Where there is a substitution θ such that SUBST (θ, pi',) = SUBST(θ, pi), it can be represented as:

$$\frac{p1', p2', \ldots, pn', (p1 \land p2 \land \ldots \land pn \Rightarrow q)}{\text{SUBST}(\theta, q)}$$

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, p1' is king(John)   p1 is king(x)

p2' is Greedy(y)   p2 is Greedy(x)

θ is {x/John, y/John}   q is evil(x)

SUBST(θ,q).

Forward Chaining and Backward chaining

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:

- Forward chaining
- 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: $(\neg p \lor \neg q \lor k)$. It has only one positive literal k.

It is equivalent to $p \land q \rightarrow k$.

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.

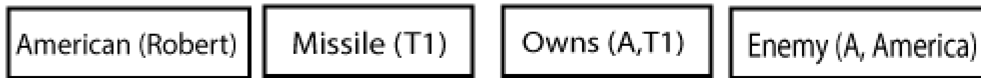
Facts Conversion into FOL:

- It is a crime for an American to sell weapons to hostile nations. (Let's say p, q, and r are variables)
  American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) ...(1)
- Country A has some missiles. ?p Owns(A, p) ∧ 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) ∧ Owns (A, p) → Sells (Robert, p, A) ......(4)
- Missiles are weapons.
  Missile(p) → Weapons (p) ....(5)
- Enemy of America is known as hostile.
  Enemy(p, America) →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.



Step-2:

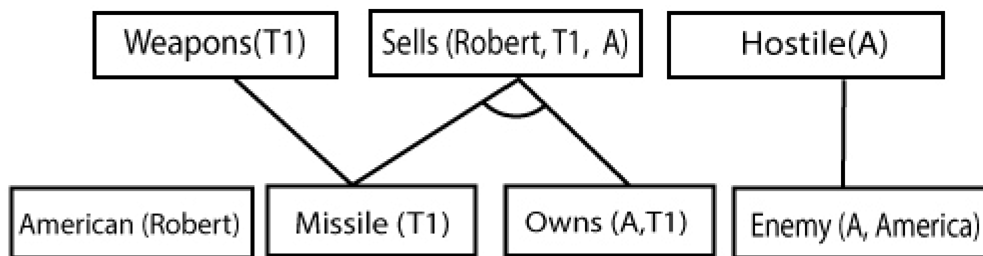
At the second step, we will see those facts which infer from available facts and with satisfied premises.

Rule-(1) does not satisfy premises, so it will not be added in the first iteration.

Rule-(2) and (3) are already added.

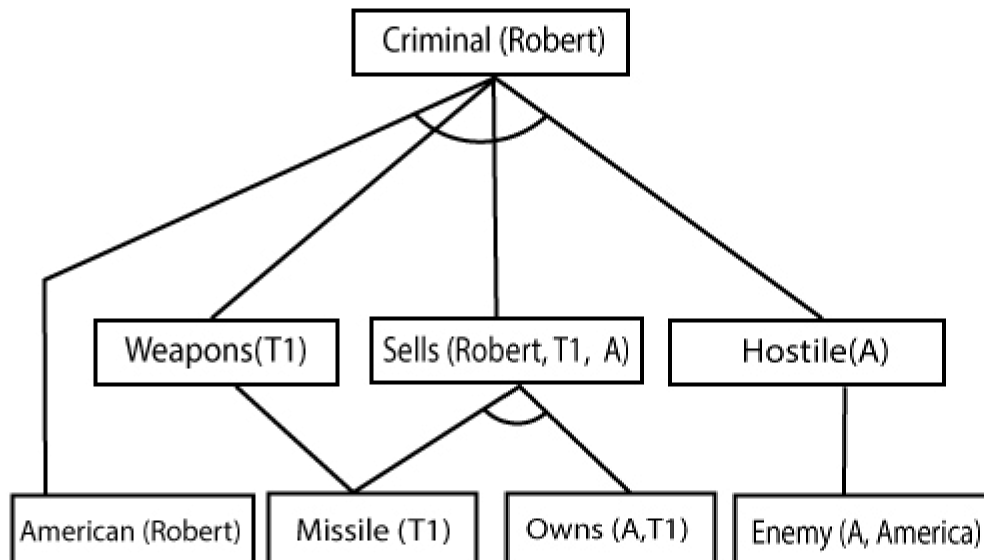
Rule-(4) satisfy with the substitution {p/T1}, so Sells (Robert, T1, A) is added, which infers from the conjunction of Rule (2) and (3).

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:

- It is known as a top-down approach.
- 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) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) ...(1)
  Owns(A, T1) ....(2)
- Missile(T1)
- ?p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A) ......(4)
- Missile(p) → Weapons (p) ....(5)
- Enemy(p, America) →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:

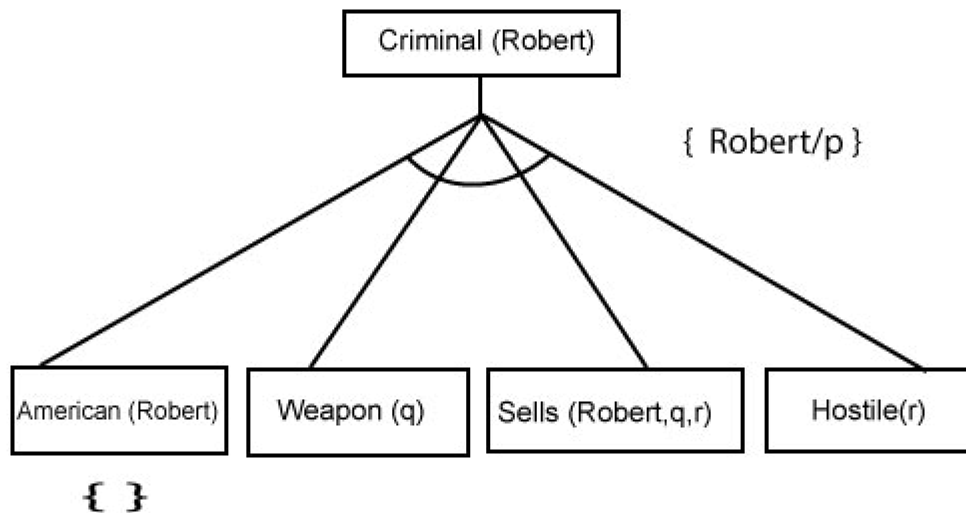
At the first step, we will take the goal fact. And from the goal fact, we will infer other facts, and at last, we will prove those facts true. So our goal fact is "Robert is Criminal," so following is the predicate of it.

Criminal (Robert)

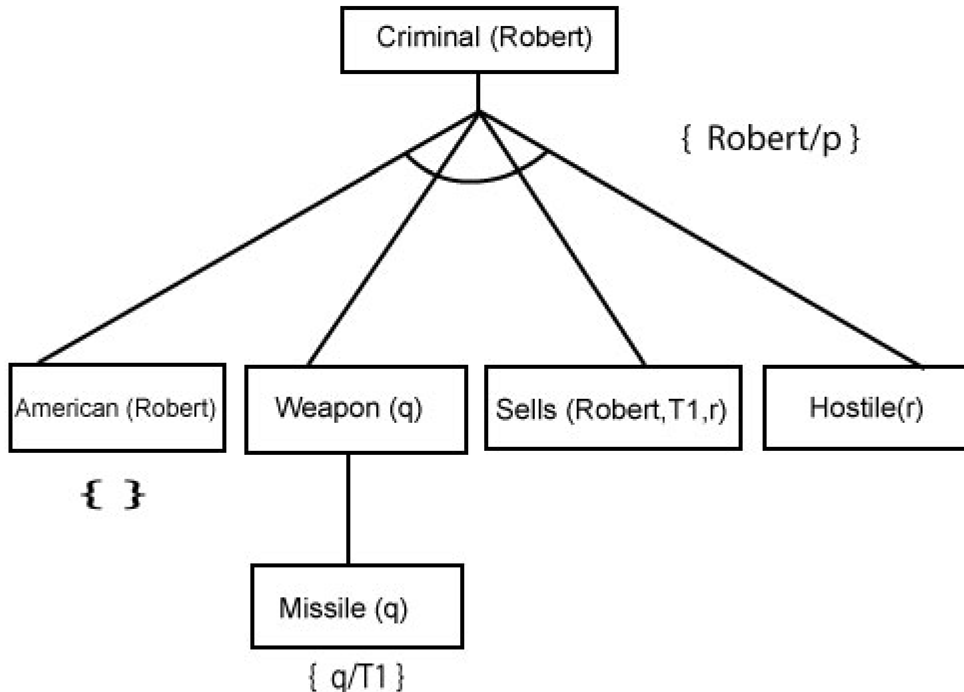
Step-2:

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.

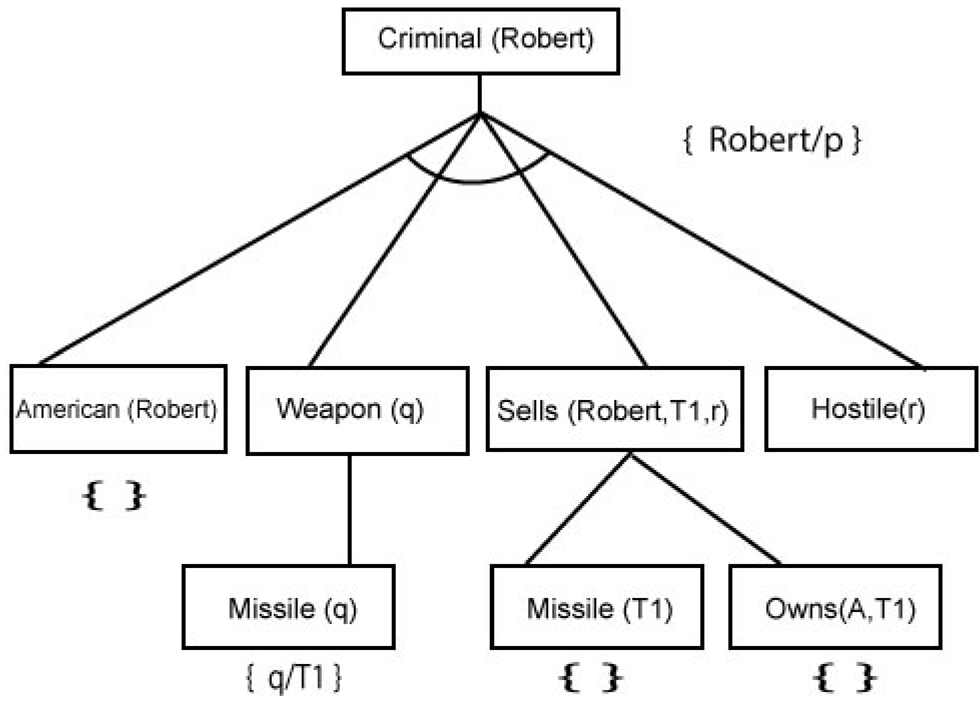


Step-3:t At step-3, we will extract further fact Missile(q) which infer from Weapon(q), as it satisfies Rule-(5). Weapon (q) is also true with the substitution of a constant T1 at q.



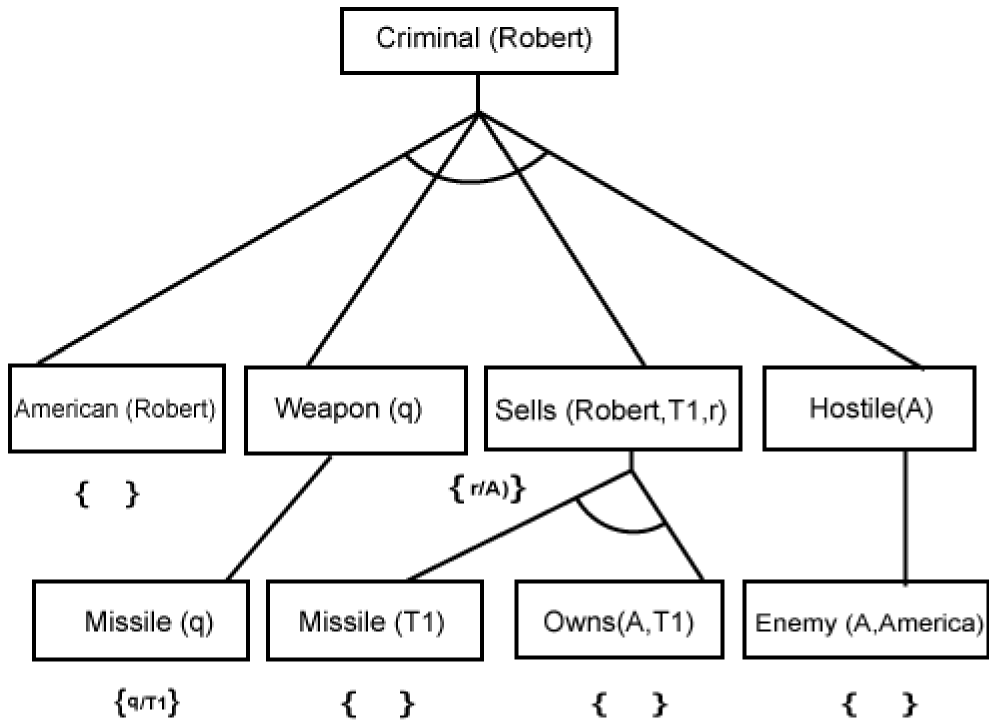
Step-4:

At step-4, we can infer facts Missile(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.



Difference between backward chaining and forward chaining

S. No.	Forward Chaining	Backward Chaining
1.	Forward chaining starts from known facts and applies inference rule to extract more data unit 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.

Unification

- Unification is a process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process.
- It takes two literals as input and makes them identical using substitution.
- Let Ψ_1 and Ψ_2 be two atomic sentences and σ be a unifier such that, $\Psi_1\sigma = \Psi_2\sigma$, then it can be expressed as UNIFY(Ψ_1, Ψ_2).
- Example: Find the MGU for Unify{King(x), King(John)}

Let Ψ_1 = King(x), Ψ_2 = King (John),

Substitution θ = {John/x} is a unifier for these atoms and applying this substitution, and both expressions will be identical.

- The UNIFY algorithm is used for unification, which takes two atomic sentences and returns a unifier for those sentences (If any exist).
- Unification is a key component of all first-order inference algorithms.
- It returns fail if the expressions do not match with each other.
- The substitution variables are called Most General Unifier or MGU.

E.g. Let's say there are two different expressions, P(x, y), and P(a, f(z)).

In this example, we need to make both above statements identical to each other. For this, we will perform the substitution.

P(x, y)......... (i)
P(a, f(z))......... (ii)

- Substitute x with a, and y with f(z) in the first expression, and it will be represented as a/x and f(z)/y.
- With both the substitutions, the first expression will be identical to the second expression and the substitution set will be: [a/x, f(z)/y].

Conditions for Unification:

Following are some basic conditions for unification:

- Predicate symbol must be same, atoms or expression with different predicate symbol can never be unified.
- Number of Arguments in both expressions must be identical.
- Unification will fail if there are two similar variables present in the same expression.

Unification Algorithm:

Algorithm: Unify(Ψ_1, Ψ_2)

Step. 1: If Ψ_1 or Ψ_2 is a variable or constant, then:

a) If Ψ_1 or Ψ_2 are identical, then return NIL.

b) Else if $\Psi_1$is a variable,

   a. then if Ψ_1 occurs in Ψ_2, then return FAILURE

   b. Else return { (Ψ_2/ Ψ_1)}.

c) Else if Ψ_2 is a variable,

   a. If Ψ_2 occurs in Ψ_1 then return FAILURE,

   b. Else return {(Ψ_1/ Ψ_2)}.

d) Else return FAILURE.

Step.2: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE.

Step. 3: IF Ψ_1 and Ψ_2 have a different number of arguments, then return FAILURE.

Step. 4: Set Substitution set(SUBST) to NIL.

Step. 5: For i=1 to the number of elements in Ψ_1.

a) Call Unify function with the ith element of Ψ_1 and ith element of Ψ_2, and put the result into S.

b) If S = failure then returns Failure

c) If S ≠ NIL then do,

   a. Apply S to the remainder of both L1 and L2.

   b. SUBST= APPEND(S, SUBST).

Step.6: Return SUBST.

Implementation of the Algorithm

Step.1: Initialize the substitution set to be empty.

Step.2: Recursively unify atomic sentences:

a. Check for Identical expression match.

b. If one expression is a variable v_i, and the other is a term t_i which does not contain variable v_i, then:

a. Substitute t_i / v_i in the existing substitutions

b. Add t_i /v_i to the substitution setlist.

c. If both the expressions are functions, then function name must be similar, and the number of arguments must be the same in both the expression.

For each pair of the following atomic sentences find the most general unifier (If exist).

1. Find the MGU of {p(f(a), g(Y)) and p(X, X)}

Sol: S_0 => Here, Ψ_1 = p(f(a), g(Y)), and Ψ_2 = p(X, X)
SUBST θ= {f(a) / X}
S1 => Ψ_1 = p(f(a), g(Y)), and Ψ_2 = p(f(a), f(a))
SUBST θ= {f(a) / g(y)}, Unification failed.

Unification is not possible for these expressions.

2. Find the MGU of {p(b, X, f(g(Z))) and p(Z, f(Y), f(Y))}

Here, Ψ_1 = p(b, X, f(g(Z))) , and Ψ_2 = p(Z, f(Y), f(Y))
S_0 => { p(b, X, f(g(Z))); p(Z, f(Y), f(Y))}
SUBST θ={b/Z}

S_1 => { p(b, X, f(g(b))); p(b, f(Y), f(Y))}
SUBST θ={f(Y) /X}

S_2 => { p(b, f(Y), f(g(b))); p(b, f(Y), f(Y))}
SUBST θ= {g(b) /Y}

S_2 => { p(b, f(g(b)), f(g(b)); p(b, f(g(b)), f(g(b))} Unified Successfully.
And Unifier = { b/Z, f(Y) /X , g(b) /Y}.

3. Find the MGU of {p (X, X), and p (Z, f(Z))}

Here, Ψ_1 = {p (X, X), and Ψ_2 = p (Z, f(Z))
S_0 => {p (X, X), p (Z, f(Z))}
SUBST θ= {X/Z}
S1 => {p (Z, Z), p (Z, f(Z))}
SUBST θ= {f(Z) / Z}, Unification Failed.

Hence, unification is not possible for these expressions.

4. Find the MGU of UNIFY(prime (11), prime(y))

Here, Ψ_1 = {prime(11) , and Ψ_2 = prime(y)}
S_0 => {prime(11) , prime(y)}
SUBST θ= {11/y}

S_1 => {prime (11) , prime(11)} , Successfully unified.
Unifier: {11/y}.

5. Find the MGU of Q(a, g(x, a), f(y)), Q(a, g(f(b), a), x)}

Here, Ψ_1 = Q(a, g(x, a), f(y)), and Ψ_2 = Q(a, g(f(b), a), x)
S_0 => {Q(a, g(x, a), f(y)); Q(a, g(f(b), a), x)}
SUBST θ= {f(b)/x}
S_1 => {Q(a, g(f(b), a), f(y)); Q(a, g(f(b), a), f(b))}

SUBST θ= {b/y}
S_1 => {Q(a, g(f(b), a), f(b)); Q(a, g(f(b), a), f(b))}, Successfully Unified.

Unifier: [a/a, f(b)/x, b/y].

6. UNIFY(knows(Richard, x), knows(Richard, John))

Here, Ψ_1 = knows(Richard, x), and Ψ_2 = knows(Richard, John)
S_0 => { knows(Richard, x); knows(Richard, John)}
SUBST θ= {John/x}
S_1 => { knows(Richard, John); knows(Richard, John)}, Successfully Unified.
Unifier: {John/x}.

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.

$$\frac{l_1 \vee \ldots\ldots \vee l_k, \qquad m_1 \vee \ldots\ldots \vee m_n}{\text{SUBST}(\theta, l_1 \vee \ldots\ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k \vee m_1 \vee \ldots\ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots\ldots \vee m_n)}$$

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:

[Animal (g(x) V Loves (f(x), x)] and [¬ Loves(a, b) V ¬Kills(a, b)]

Where two complimentary literals are: Loves (f(x), x) and ¬ Loves (a, b)

These literals can be unified with unifier θ= [a/f(x), and b/x] , and it will generate a resolvent clause:

[Animal (g(x) V ¬ 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. $\forall x$: food(x) → likes(John, x)
b. food(Apple) Λ food(vegetables)
c. $\forall x \forall y$: eats(x, y) Λ ¬ killed(x) → food(y)
d. eats (Anil, Peanuts) Λ alive(Anil).
e. $\forall x$: eats(Anil, x) → eats(Harry, x)
f. $\forall x$: ¬ killed(x) → alive(x) } **added predicates.**
g. $\forall x$: alive(x) →¬ killed(x) }
h. likes(John, 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.

- Eliminate all implication (→) and rewrite
  a. $\forall x$ ¬ food(x) V likes(John, x)
  b. food(Apple) Λ food(vegetables)
  c. $\forall x \forall y$ ¬ [eats(x, y) Λ ¬ killed(x)] V food(y)
  d. eats (Anil, Peanuts) Λ alive(Anil)

e. ∀x ¬ eats(Anil, x) V eats(Harry, x)

f. ∀x¬ [¬ killed(x)] V alive(x)

g. ∀x ¬ alive(x) V ¬ killed(x)

h. likes(John, Peanuts).

- Move negation (¬)inwards and rewrite

  . ∀x ¬ food(x) V likes(John, x)

  a. food(Apple) Λ food(vegetables)

  b. ∀x ∀y ¬ eats(x, y) V killed(x) V food(y)

  c. eats (Anil, Peanuts) Λ alive(Anil)

  d. ∀x ¬ eats(Anil, x) V eats(Harry, x)

  e. ∀x ¬killed(x)] V alive(x)

  f. ∀x ¬ alive(x) V ¬ killed(x)

  g. likes(John, Peanuts).

- Rename variables or standardize variables

. ∀x ¬ food(x) V likes(John, x)

  a. food(Apple) Λ food(vegetables)

  b. ∀y ∀z ¬ eats(y, z) V killed(y) V food(z)

  c. eats (Anil, Peanuts) Λ alive(Anil)

  d. ∀w¬ eats(Anil, w) V eats(Harry, w)

  e. ∀g ¬killed(g)] V alive(g)

  f. ∀k ¬ alive(k) V ¬ killed(k)

  g. likes(John, Peanuts).

- Eliminate existential instantiation quantifier by elimination. In this step, we will eliminate existential quantifier ∃, 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.

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.

. ¬ food(x) V likes(John, x)

a. food(Apple)
b. food(vegetables)
c. ¬ eats(y, z) V killed(y) V food(z)
d. eats (Anil, Peanuts)
e. alive(Anil)
f. ¬ eats(Anil, w) V eats(Harry, w)
g. killed(g) V alive(g)
h. ¬ alive(k) V ¬ killed(k)
i. likes(John, Peanuts).

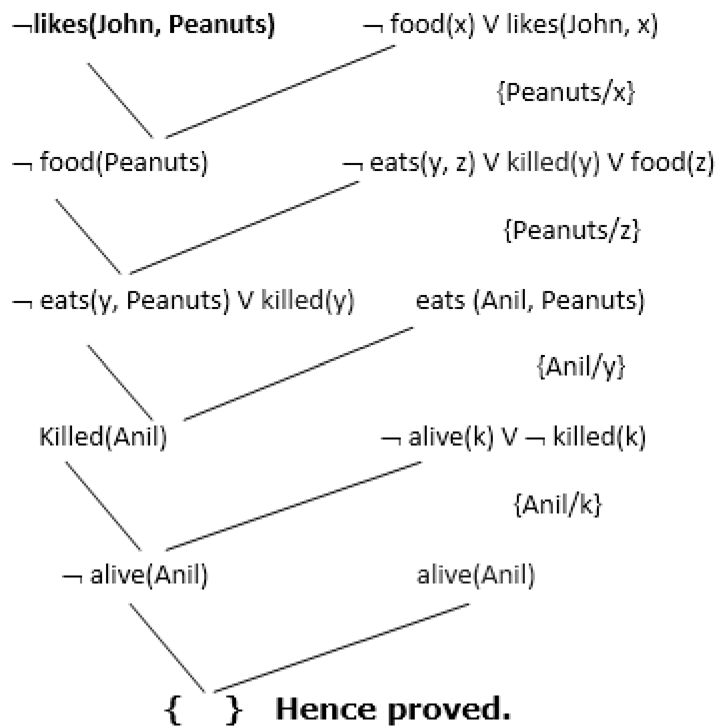
- Distribute conjunction ∧ over disjunction ¬.
  This step will not make any change in this problem.

Step-3: Negate the statement to be proved

In this statement, we will apply negation to the conclusion statements, which will be written as ¬likes(John, 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, ¬likes(John, Peanuts) , and likes(John, x) get resolved(canceled) by substitution of {Peanuts/x}, and we are left with ¬ food(Peanuts)
- In the second step of the resolution graph, ¬ food(Peanuts) , and food(z) get resolved (canceled) by substitution of { Peanuts/z}, and we are left with ¬ eats(y, Peanuts) V killed(y) .
- In the third step of the resolution graph, ¬ eats(y, Peanuts) and eats (Anil, Peanuts) get resolved by substitution {Anil/y}, and we are left with Killed(Anil) .
- In the fourth step of the resolution graph, Killed(Anil) and ¬ killed(k) get resolve by substitution {Anil/k}, and we are left with ¬ alive(Anil) .
- In the last step of the resolution graph ¬ alive(Anil) and alive(Anil) get resolved.

MCQ

1. Knowledge and reasoning also play a crucial role in dealing with __________________ environment.

a) Completely Observable
b) Partially Observable
c) Neither Completely nor Partially Observable
d) Only Completely and Partially Observable

2. Wumpus World is a classic problem, best example of _______
a) Single player Game
b) Two player Game
c) Reasoning with Knowledge
d) Knowledge based Game

3. Which is not a property of representation of knowledge?

a) **Representational Verification**
b) Representational Adequacy
c) Inferential Adequacy
d) Inferential Efficiency

4. Which is not Familiar Connectives in First Order Logic?

a)and
b) iff
c) or
d) not

5. Inference algorithm is complete only if _____________

a)It can derive any sentence
b) It can derive any sentence that is an entailed version
c) It is truth preserving
d) It can derive any sentence that is an entailed version & It is truth preserving

6. Which of the following is not the style of inference?

a)Forward Chaining
b) Backward Chaining
c) Resolution Refutation
d) Modus Ponen

7. Forward chaining systems are _____________ where as backward chaining systems are ___________

   a)Goal-driven, goal-driven
   b) Goal-driven, data-driven
   c) Data-driven, goal-driven
   d) Data-driven, data-driven

8. What are the main components of the expert systems?

   a)Inference Engine
   b) Knowledge Base
   c) Inference Engine & Knowledge Base
   d) None of the mentioned

9. Which is a refutation complete inference procedure for propositional logic?
   a)Clauses
   b) Variables
   c) Propositional resolution
   d) Proposition

10. What kinds of clauses are available in Conjunctive Normal Form?

   a)**Disjunction of literals**
   b) Disjunction of variables
   c) Conjunction of literals
   d) Conjunction of variables

CONCLUSION:

Upon completion of this, Students should be able to

- ❖ Understand the Logical Agents in AI.
- ❖ Understand Wumpus World in AI.
- ❖ Understand First Order logic in AI.
- ❖ Forward Chaining – Backward Chaining in AI.

REFERENCES

1. David Poole, Alan Mackworth, Randy Goebel, “Computational Intelligence: a Logical Approach”, Oxford University Press, 2004.

2. G. Luger, “Artificial Intelligence: Structures and Strategies for Complex Problem Solving”, Fourth Edition, Pearson Education, 2002.

ASSIGNMENT

1. Define Unification and Resolution.
2. Explain the Knowledge Based Agents in AI.
3. Explain the Wumpus World in AI.
4. Explain First Order logic and its inferences.
5. Compare Forward Chaining and Backward Chaining.

UNIT-5

Planning with state space search – Partial-order planning – Planning graphs – Planning and acting in the real world.

AIM & OBJECTIVES

- To understand Planning in AI.
- To understand Partial-order planning in AI.
- To understand Planning and acting in the real world.

PRE- REQUISITE: Basic knowledge of Computer Architecture.

The task of coming up with a sequence of actions that will achieve a goal is called planning. We have seen two examples of planning agents so far: the search-based problem-solving agent and the logical planning agent.

Planning With State-Space Search

Now we turn our attention to planning algorithms. The most straightforward approach is to use state-space search. Because the descriptions of actions in a planning problem specify both preconditions and effects, it is possible to search in either direction: either forward from the initial state or backward from the goal, as shown in Figure.

We can also use the explicit action and goal representations to derive effective heuristics automatically.

Forward state-space search planning with forward state-space search is similar to the problem-solving approach.

It is sometimes called progression planning, because it moves in the forward direction.

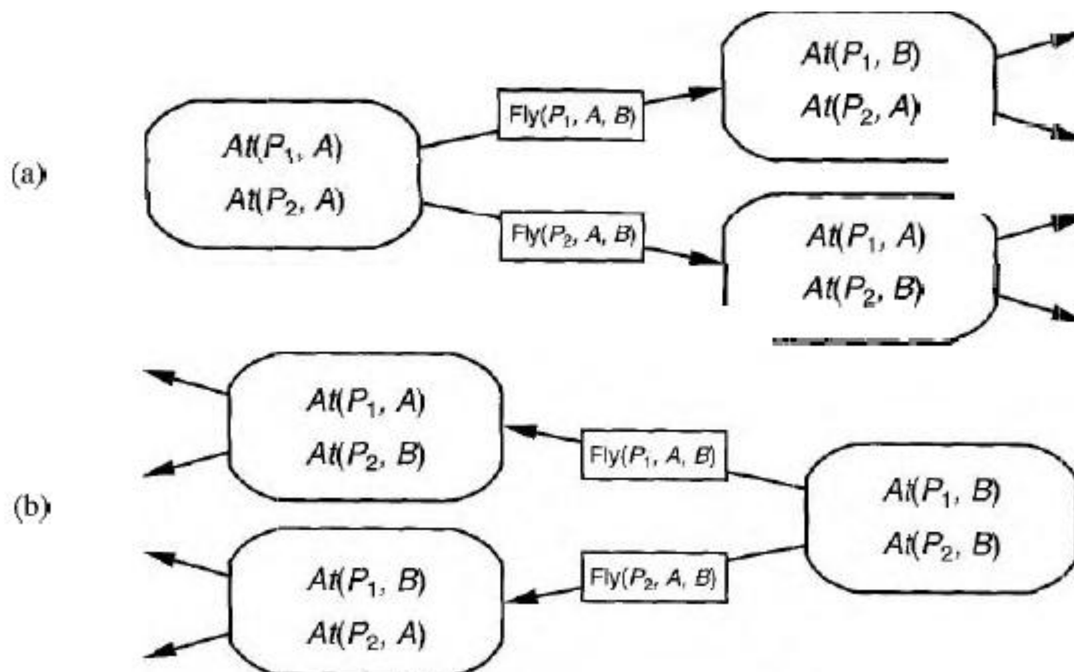
$Init(On(A, Table) \wedge On(B, Table) \wedge On(C, Table)$
$\wedge Block(A) \wedge Block(B) \wedge Block(C)$
$\wedge Clear(A) \wedge Clear(B) \wedge Clear(C))$
$Goal(On(A, B) \wedge On(B, C))$
$Action(Move(b, x, y),$
PRECOND: $On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge$
$(b \neq x) \wedge (b \neq y) \wedge (x \neq y),$
EFFECT: $On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y))$
$Action(MoveToTable(b, x),$
PRECOND: $On(b, x) \wedge Clear(b) \wedge Block(b) \wedge (b \neq x),$
EFFECT: $On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))$

A planning problem in the blocks world: building a three-block tower. One solution is the sequence [Move(B,T able, C) ,Move(A, Table, B)].

Two approaches to searching for a plan.

(a) Forward (progression) state-space search, starting in the initial state and using the problem's actions to search forward for the goal state.

(b) Backward (regression) state-space search: a belief-state search starting at the goal state(s) and using the inverse of the actions to search backward for the initial state.



We start in the problem's initial state, considering sequences of actions until we find a sequence that reaches a goal state. The formulation of planning problems as state-space search problems is as follows:

The initial state of the search is the initial state from the planning problem. In general, each state will be a set of positive ground literals; literals not appearing are false.

The actions that are applicable to a state are all those whose preconditions are satisfied. The successor state resulting from an action is generated by adding the positive effect literals and deleting the negative effect literals. (In the first-order case, we must apply the unifier from the preconditions to the effect literals.) Note that a single successor function works for all planning problems-a consequence of using an explicit action representation.

The goal test checks whether the state satisfies the goal of the planning problem.
The step cost of each action is typically 1. Although it would be easy to allow different costs for different actions, this is seldom done by STRIPS planners.

Recall that, in the absence of function symbols, the state space of a planning problem is finite. Therefore, any graph search algorithm that is complete-for example, A*-will be a complete planning algorithm.

From the earliest days of planning research (around 1961) until recently (around 1998) it was assumed that forward state-space search was too inefficient to be practical. First, forward search does not address the irrelevant action problem-all applicable actions are considered from each state. Second, the approach quickly bogs down without a good heuristic. Consider an air cargo problem with 10 airports, where each airport has *5* planes and 20 pieces of cargo.

The goal is to move all the cargo at airport A to airport B. There is a simple solution to the problem: load the 20 pieces of cargo into one of the planes at A, fly the plane to B, and unload the cargo. But finding the solution can be difficult because the average branching factor is huge: each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded), or loaded into any plane at its airport (if it is unloaded). On average, let's say

there are about 1000 possible actions, so the search tree up to the depth of the obvious solution has about 1000n~o~de s. It is clear that a very accurate heuristic will be needed to make this kind of search efficient.

Backward state-space search

We noted there that backward search can be difficult to implement when the goal states are described by a set of constraints rather than being listed explicitly. In particular, it is not always obvious how to generate a description of the possible predecessors of the set of goal states. We will see that the STRIPS representation makes this quite easy because sets of states can be described by the literals that must be true in those states.

The main advantage of backward search is that it allows us to consider only relevant actions. An action is relevant to a conjunctive goal if it achieves one of the conjuncts of the goal. For example, the goal in our 10-airport air cargo problem is to have 20 pieces of cargo at airport B, or more precisely,

At(C1,B) A At(C2,B)A . . . A At(Czo,B).

Now consider the conjunct *At (C1B,).*Working backwards, we can seek actions that have this as an effect. There is only one: Unload *(C1p, , B),*w here plane *p* is unspecified.

Searching backwards is sometimes called regression planning. The principal question in regression planning is this: what are the states from which applying a given action leads to the goal? Computing the description of these states is called regressing the goal through the action.

Heuristics for state-space search

It turns out that neither forward nor backward search is efficient without a good heuristic function. A heuristic function estimates the distance from a state to the goal; in STRIPS planning, the cost of each action is 1, so the distance is the number of actions. The basic idea is to look at the effects of the actions and at the goals that must be achieved and to guess how many actions are needed to achieve all the goals. Finding the exact number is NP hard, but it is possible to find reasonable estimates most of the time without too much computation. We might also be able to derive an admissible heuristic-one that does not overestimate. This could be used with A* search to find optimal solutions.

There are two approaches that can be tried. The first is to derive a relaxed problem from the given problem specification. The optimal solution cost for the relaxed problem-which we hope is very easy to solve-gives an admissible heuristic for the original problem. The second approach is to pretend that a pure divide-and-conquer algorithm will work. This is called the subgoal independence assumption: the cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving each subgoal independently. The subgoal independence assumption can be optimistic or pessimistic. It is optimistic when there are negative interactions between the subplans for each sub goal for example, when an action in one subplan deletes a goal achieved by another subplan.

It is pessimistic, and therefore inadmissible, when subplans contain redundant actions-for instance, two actions that could be replaced by a single action in the merged plan.

Let us consider how to derive relaxed planning problems. Since explicit representations of preconditions and effects are available, the process will work by modifying those representations.

(Compare this approach with search problems, where the successor function is a black box.) The simplest idea is to relax the problem by removing all preconditions from the actions.

Then every action will always be applicable, and any literal can be achieved in one step (if there is an applicable action-if not, the goal is impossible). This almost implies that the number of steps required to solve a conjunction of goals is the number of unsatisfied goals-almost but not quite, because
(1) there may be two actions, each of which deletes the goal literal achieved by the other, and
(2) some action may achieve multiple goals.

If we combine our relaxed problem with the subgoal independence assumption, both of these issues are assumed away and the resulting heuristic is exactly the number of unsatisfied goals.

In many cases, a more accurate heuristic is obtained by considering at least the positive interactions arising from actions that achieve multiple goals.

First, we relax the problem further by removing negative effects. Then, we count the minimum number of actions required such that the union of those actions' positive effects satisfies the goal.

For example, consider
Goal (A A B A C)
Action(X, EFFECT:A A P)
Action(Y, EFFECT:B A C A Q)
Action(Z, EFFECT:B A P A Q) .

The minimal set cover of the goal {A, B, C) is given by the actions {X,Y), so the set cover heuristic returns a cost of 2. This improves on the subgoal independence assumption, which gives a heuristic value of 3. There is one minor irritation: the set cover problem is NP hard. A simple greedy set-covering algorithm is guaranteed to return a value that is within a factor of log n of the true minimum value, where n is the number of literals in the goal, and usually works much better than this in practice. Unfortunately, the greedy algorithm loses the guarantee of admissibility for the heuristic.

It is also possible to generate relaxed problems by removing negative effects without removing preconditions. That is, if an action has the effect A A 1B in the original problem, it will have the effect A in the relaxed problem. This means that we need not worry about negative interactions between subplans, because no action can delete the literals achieved by another action. The solution cost of the resulting relaxed problem gives what is called the empty-delete-list heuristic. The heuristic is quite accurate, but computing it involves actually running a (simple) planning algorithm.

Partial order planning

Forward and backward state-space search are particular forms of totally ordered plan search. They explore only strictly linear sequences of actions directly connected to the start or goal. This means that they cannot take advantage of problem decomposition. Rather than work on each subproblem separately, they must always make decisions about how to sequence actions from all the subproblems. We would prefer an approach that works on several subgoals independently, solves them with several subplans, and then combines the subplans.

Such an approach also has the advantage of flexibility in the order in which it constructs the plan. That is, the planner can work on "obvious" or "important7' decisions first, rather than being forced to work on steps in chronological order.

For example, a planning agent that is in Berkeley and wishes to be in Monte Carlo might first try to find a flight from San Francisco to Paris; given information about the departure and arrival times, it can then work on ways to get to and from the airports.

The general strategy of delaying a choice during search is called a least commitment strategy. There is no formal definition of least commitment, and clearly some degree of commitment is necessary, lest the search would make no progress. Despite the informality, least commitment is a useful concept for analyzing when decisions should be made in any search problem.

Our first concrete example will be much simpler than planning a vacation. Consider the simple problem of putting on a pair of shoes. We can describe this as a formal planning problem as follows:

$$
\begin{aligned}
&\text{Goal}(RightShoeOn \wedge LeftShoeOn)\\
&Init()\\
&Action(RightShoe, \text{PRECOND:}RightSockOn, \text{EFFECT:}RightShoeOn)\\
&Action(RightSock, \text{EFFECT:}RightSockOn)\\
&Action(LeftShoe, \text{PRECOND:}LeftSockOn, \text{EFFECT:}LeftShoeOn)\\
&\text{Action}(LeftSock, \text{EFFECT:}LeftSockOn)\,.
\end{aligned}
$$

A planner should be able to come up with the two-action sequence Rightsock followed by Rightshoe to achieve the first conjunct of the goal and the sequence Leftsock followed by LeftShoe for the second conjunct. Then the two sequences can be combined to yield the final plan. In doing this, the planner will be manipulating the two subsequences independently, without committing to whether an action in one sequence is before or after an action in the other.

Any planning algorithm that can place two actions into a plan without specifying which comes first is called a partial-order planner. Figure shows the partial-order plan that is the solution to the shoes and socks problem. Note that the solution is represented as a graph of actions, not a sequence. Note also the "dummy" actions called Start and Finish, which mark the beginning and end of the plan.

Calling them actions symplifies things, because now every step of a plan is an action. The partial-order solution corresponds to six possible total-order plans; each of these is called a linearization of the partial-order plan.

Partial-order planning can be implemented as a search in the space of partial-order plans. (From now on, we will just call them "plans.") That is, we start with an empty plan. Then we consider ways of refining the plan until we come up with a complete plan that solves the problem. The actions in this search are not actions in the world, but actions on plans: adding a step to the plan, imposing an ordering that puts one action before another, and so on.

We will define the POP algorithm for partial-order planning. It is traditional to write out the POP algorithm as a stand-alone program, but we will instead formulate partial-order planning as an instance of a search problem. This allows us to focus on the plan refinement steps that can be applied, rather than worrying about how the algorithm explores the space. In fact, a wide variety of uninformed or heuristic search methods can be applied once the search problem is formulated.

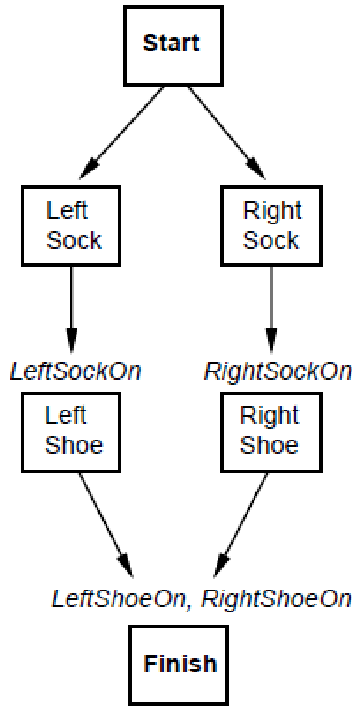
Remember that the states of our search problem will be (mostly unfinished) plans. To avoid confusion with the states of the world, we will talk about plans rather than states.

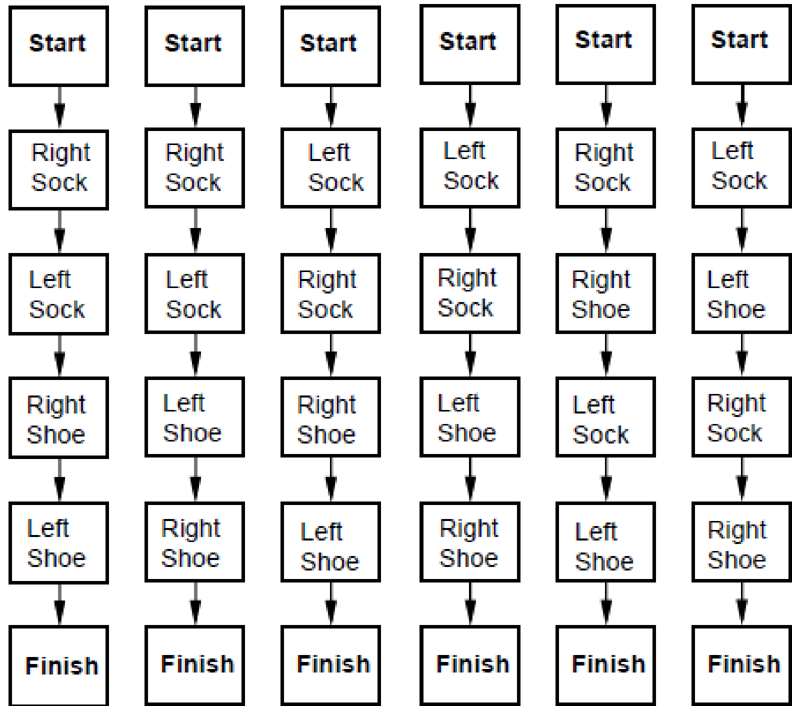
Each plan has the following four components, where the first two define the steps of the plan and the last two serve a bookkeeping function to determine how plans can be extended:

• A set of actions that make up the steps of the plan. These are taken from the set of actions in the planning problem.

The "empty" plan contains just the Start and Finish actions. Start has no preconditions and has as its effect all the literals in the initial state of the planning problem.

Finish has no effects and has as its preconditions the goal literals of the planning problem.





A partial-order plan for putting on shoes and socks, and the six corresponding linearization into total-order plans.

A set of ordering constraints.
Each ordering constraint is of the form A ≺ B, which is read as "A before B" and means that action A must be executed sometime before action B, but not necessarily immediately before. The ordering constraints must describe a proper partial order.

Any cycle—such as A ≺ B and B ≺ A—represents a contradiction, so an ordering constraint cannot be added to the plan if it creates a cycle.

• A set of causal links. A causal link between two actions A and B in the plan is written as A p ACHIEVES −→ B and is read as "A achieves p for B." For example, the causal link

RightSock → RightShoe

asserts that RightSockOn is an effect of the RightSock action and a precondition of RightShoe. It also asserts that RightSockOn must remain true from the time of action RightSock to the time of action RightShoe.

In other words, the plan may not be CONFLICTS extended by adding a new action C that conflicts with the causal link. An action C conflicts with A p→ B if C has the effect ¬p and if C could (according to the ordering constraints) come after A and before B. Some authors call causal links protection intervals, because the link A p → B protects p from being negated over the interval from A to B.

A set of open preconditions. A precondition is open if it is not achieved by some action OPEN PRECONDITIONS in the plan. Planners will work to reduce the set of open preconditions to the empty set, without introducing a contradiction.

For example, the final plan has the following components (not shown are the ordering constraints that put every other action after Start and before Finish):

Actions:{RightSock, RightShoe, LeftSock, LeftShoe, Start, Finish}

Orderings :{RightSock ≺ RightShoe, LeftSock ≺ LeftShoe}

Links :{RightSock RightSockOn −→ RightShoe,
LeftSock LeftSockOn −→ LeftShoe,
RightShoe RightShoeOn −→ Finish,
LeftShoe LeftShoeOn −→ Finish}

Open Preconditions :{} .

We define a consistent plan as a plan in which there are no cycles in the ordering constraints and no conflicts with the causal links. A consistent plan with no open preconditions is a solution. A moment's thought should convince the reader of the following fact: every linearization of a partial-order solution is a total-order solution whose execution from the initial state will reach a goal state. This means that we can extend the notion of "executing a plan" from total-order to partial-order plans. A partial-order plan is executed by repeatedly choosing any of the possible next actions.

The flexible ordering also makes it easier to combine smaller plans into larger ones, because each of the small plans can reorder its actions to avoid conflict with the other plans. Now we are ready to formulate the search problem that POP solves. We will begin with a formulation suitable for propositional planning problems, leaving the first-order complications for later.

As usual, the definition includes the initial state, actions, and goal test.

• The initial plan contains Start and Finish, the ordering constraint Start $\prec$ Finish, and no causal links and has all the preconditions in Finish as open preconditions.

• The successor function arbitrarily picks one open precondition p on an action B and generates a successor plan for every possible consistent way of choosing an action A that achieves p.

Consistency is enforced as follows:

1. The causal link A p −→ B and the ordering constraint A $\prec$ B are added to the plan. Action A may be an existing action in the plan or a new one. If it is new, add it to the plan and also add Start $\prec$ A and A $\prec$ Finish.

2. We resolve conflicts between the new causal link and all existing actions and between the action A (if it is new) and all existing causal links. A conflict between A p −→ B and C is resolved by making C occur at some time outside the protection interval, either by adding B $\prec$ C or C $\prec$ A. We add successor states for either or both if they result in consistent plans.

• The goal test checks whether a plan is a solution to the original planning problem. Because only consistent plans are generated, the goal test just needs to check that there are no open preconditions.

Remember that the actions considered by the search algorithms under this formulation are plan refinement steps rather than the real actions from the domain itself. The path cost is therefore irrelevant, strictly speaking, because the only thing that matters is the total cost of the real actions in the plan to which the path leads.

Nonetheless, it is possible to specify a path cost function that reflects the real plan costs: we charge 1 for each real action added to the plan and 0 for all other refinement steps. In this way, g(n), where n is a plan, will be equal to the number of real actions in the plan. A heuristic estimate h(n) can also be used. At first glance, one might think that the successor function should include successors for every open p, not just for one of them.

This would be redundant and inefficient, however, for the same reason that constraint satisfaction algorithms don't include successors for every possible variable: the order in which we consider open preconditions (like the order in which we consider CSP variables) is commutative. Thus, we can choose an arbitrary ordering and still have a complete algorithm. Choosing the right ordering can lead to a faster search, but all orderings end up with the same set of candidate solutions.

Heuristics for partial-order planning Compared with total-order planning, partial-order planning has a clear advantage in being able to decompose problems into subproblems. It also has a disadvantage in that it does not represent states directly, so it is harder to estimate how far a partial-order plan is from achieving a goal. At present, there is less understanding of how to compute accurate heuristics for partial-order planning than for total-order planning.

The most obvious heuristic is to count the number of distinct open preconditions. This can be improved by subtracting the number of open preconditions that match literals in the Start state. As in the total-order case, this overestimates the cost when there are actions that achieve multiple goals and underestimates the cost when there are negative interactions between plan steps. The next section presents an approach that allows us to get much more accurate heuristics from a relaxed problem. The heuristic function is used to choose which plan to refine. Given this choice, the algorithm generates successors based on the selection of a single open precondition to work on. As in the case of variable selection on constraint satisfaction algorithms, this selection has a large impact on efficiency.

The most-constrained-variable heuristic from CSPs can be adapted for planning algorithms and seems to work well. The idea is to select the open condition that can be satisfied in the fewest number of ways.

There are two special cases of this heuristic.

First, if an open condition cannot be achieved by any action, the heuristic will select it; this is a good idea because early detection of impossibility can save a great deal of work.

Second, if an open condition can be achieved in only one way, then it should be selected because the decision is unavoidable and could provide additional constraints on other choices still to be made.

Although full computation of the number of ways to satisfy each open condition is expensive and not always worthwhile, experiments show that handling the two special cases provides very substantial speedups.

Planning Graphs

All of the heuristics we have suggested for total-order and partial-order planning can suffer from inaccuracies. This section shows how a special data structure called a planning graph can be used to give better heuristic estimates. These heuristics can be applied to any of the search techniques we have seen so far. Alternatively, we can extract a solution directly from the planning graph, using a specialized algorithm such as the one called GRAPHPLAN.

A planning graph consists of a sequence of levels that correspond to time steps in the plan, where level 0 is the initial state. Each level contains a set of literals and a set of actions. Roughly speaking, the literals are all those that could be true at that time step, depending on the actions executed at preceding time steps. Also roughly speaking, the actions are all those actions that could have their preconditions satisfied at that time step, depending on which of the literals actually hold.

This number of steps in the planning graph provides a good estimate of how difficult it is to achieve a given literal from the initial state. More importantly, the planning graph is defined in such a way that it can be constructed very efficiently.

Planning graphs work only for propositional planning problems ones with no variables. As we mentioned in both STRIPS and ADL representations can be propositionalized. For problems with large numbers of objects, this could result in a very substantial blowup in the number of action schemata.

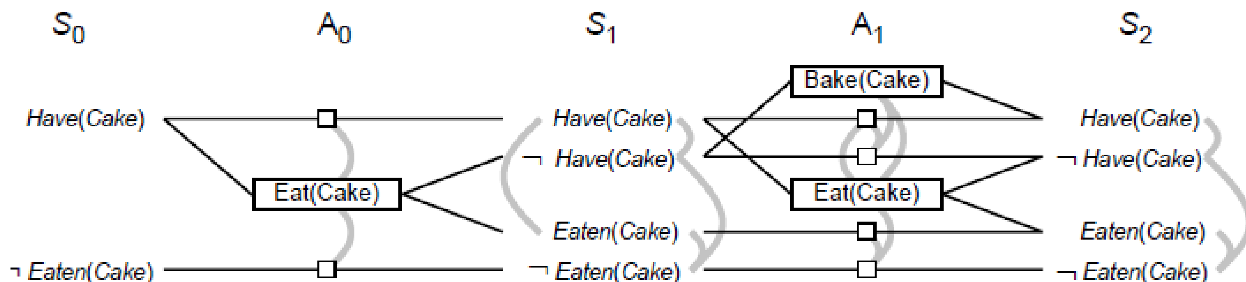
Despite this, planning graphs have proved to be effective tools for solving hard planning problems.

We will illustrate planning graphs with a simple example. (More complex examples lead to graphs that won't fit on the page.) We start with state level S_0, which represents the problem's initial state. We follow that with action level A_0, in which we place all the actions whose preconditions are satisfied in the previous level. Each action is connected to its preconditions in S_0 and its effects in S_1, in this case introducing new literals into S1 that were not in S_0.

The planning graph needs a way to represent inaction as well as action. That is, it needs the equivalent of the frame axioms in situation calculus that allow a literal to remain true from one situation to the next if no action alters it. In a planning graph this is done with a set of persistence actions.

$$
\begin{array}{l}
\mathit{Init}(\mathit{Have}(\mathit{Cake})) \\
\mathit{Goal}(\mathit{Have}(\mathit{Cake}) \land \mathit{Eaten}(\mathit{Cake})) \\
\mathit{Action}(\mathit{Eat}(\mathit{Cake}) \\
\quad \textsc{Precond}{:}\ \mathit{Have}(\mathit{Cake}) \\
\quad \textsc{Effect}{:}\ \neg\, \mathit{Have}(\mathit{Cake}) \land \mathit{Eaten}(\mathit{Cake})) \\
\mathit{Action}(\mathit{Bake}(\mathit{Cake}) \\
\quad \textsc{Precond}{:}\ \neg\, \mathit{Have}(\mathit{Cake}) \\
\quad \textsc{Effect}{:}\ \mathit{Have}(\mathit{Cake})
\end{array}
$$

The “have cake and eat cake too” problem.



The planning graph for the “have cake and eat cake too” problem up to level S2. Rectangles indicate actions (small squares indicate persistence actions) and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines.

For every positive and negative literal C, we add to the problem a persistence action with precondition C and effect C. Figure shows one “real” action, Eat (Cake) in A_0, along with two persistence actions drawn as small square boxes.

Level A_0 contains all the actions that could occur in state S0, but just as importantly it records conflicts between actions that would prevent them from occurring together. The gray lines in Figure indicate mutual exclusion (or mutex) links.

For example, Eat (Cake) MUTEX is mutually exclusive with the persistence of either Have (Cake) or ¬Eaten(Cake). We shall see shortly how mutex links are computed. Level S1 contains all the literals that could result from picking any subset of the actions in A_0. It also contains mutex links (gray lines) indicating literals that could not appear together, regardless of the choice of actions.

For example, Have(Cake) and Eaten(Cake) are mutex: depending on the choice of actions in A0, one or the other, but not both, could be the result.

In other words, S1 represents multiple states, just as regression state-space search does, and the mutex links are constraints that define the set of possible states. We continue in this way, alternating between state level Si and action level Ai until we reach a level where two consecutive levels are identical has leveled off.

Every subsequent level will be identical, so further expansion is unnecessary. What we end up with is a structure where every Ai level contains all the actions that are applicable in Si, along with constraints saying which pairs of actions cannot both be executed.

Every Si level contains all the literals that could result from any possible choice of actions in Ai−1, along with constraints saying which pairs of literals are not possible. It is important to note that the process of constructing the planning graph does not require choosing among actions, which would entail combinatorial search.

Instead, it just records the impossibility of certain choices using mutex links. The complexity of constructing the planning graph is a low-order polynomial in the number of actions and literals, whereas the state space is exponential in the number of literals.

We now define mutex links for both actions and literals. A mutex relation holds between two actions at a given level if any of the following three conditions holds:

• Inconsistent effects: one action negates an effect of the other. For example Eat(Cake) and the persistence of Have(Cake) have inconsistent effects because they disagree on the effect Have(Cake).

• Interference: one of the effects of one action is the negation of a precondition of the other. For example Eat(Cake) interferes with the persistence of Have(Cake) by negating its precondition.

• Competing needs: one of the preconditions of one action is mutually exclusive with a precondition of the other. For example, Bake(Cake) and Eat(Cake) are mutex because they compete on the value of the Have(Cake) precondition.

A mutex relation holds between two literals at the same level if one is the negation of the other or if each possible pair of actions that could achieve the two literals is mutually exclusive. This condition is called inconsistent support.

For example, Have(Cake) and Eaten(Cake) are mutex in S1 because the only way of achieving Have(Cake), the persistence action, is mutex with the only way of achieving Eaten(Cake), namely Eat(Cake). In S2 the two literals are not mutex because there are new ways of achieving them, such as Bake(Cake) and the persistence of Eaten(Cake), that are not mutex.

Planning graphs for heuristic estimation

A planning graph, once constructed, is a rich source of information about the problem. For example, a literal that does not appear in the final level of the graph cannot be achieved by any plan. This observation can be used in backward search as follows: any state containing an unachievable literal has a cost $h(n) = \infty$. Similarly, in partial-order planning, any plan with an unachievable open condition has $h(n) = \infty$.

This idea can be made more general. We can estimate the cost of achieving any goal literal as the level at which it first appears in the planning graph. We will call this the level cost of the goal. In Figure, Have (Cake) has level cost 0 and Eaten (Cake) has level cost 1. It is easy to show that these estimates are admissible for the individual goals. The estimate might not be very good, however, because planning graphs allow several actions at each level whereas the heuristic counts just the level and not the number of actions.

For this reason, it is common to use a serial planning graph for computing heuristics.

A serial graph insists that only one action can actually occur at any given time step; this is done by adding mutex links between every pair of actions except persistence actions. Level costs extracted from serial graphs are often quite reasonable estimates of actual costs.

To estimate the cost of a conjunction of goals, there are three simple approaches. The max-level heuristic simply takes the maximum level cost of any of the goals; this is admissible, but not necessarily very accurate.

The level sum heuristic, following the subgoal independence assumption, returns the sum of the level costs of the goals; this is inadmissible but works very well in practice for problems that are largely decomposable. It is much more accurate than the number-of-unsatisfied-goals heuristic.

For our problem, the heuristic estimate for the conjunctive goal Have(Cake)∧Eaten(Cake) will be 0+1 = 1, whereas the correct answer is 2. Moreover, if we eliminated the Bake(Cake) action, the estimate would still be 1, but the conjunctive goal would be impossible.

Finally, the set-level heuristic finds the level at which all the literals in the conjunctive goal appear in the planning graph without any pair of them being mutually exclusive. This heuristic gives the correct values of 2 for our original problem and infinity for the problem without Bake(Cake). It dominates the max-level heuristic and works extremely well on tasks in which there is a good deal of interaction among subplans.

As a tool for generating accurate heuristics, we can view the planning graph as a relaxed problem that is efficiently soluble. To understand the nature of the relaxed problem, we need to understand exactly what it means for a literal g to appear at level Si in the planning graph. Ideally, we would like it to be a guarantee that there exists a plan with i action levels that achieves g, and also that if g does not appear that there is no such plan.

Unfortunately, making that guarantee is as difficult as solving the original planning problem. So the planning graph makes the second half of the guarantee (if g does not appear, there is no plan), but if g

does appear, then all the planning graph promises is that there is a plan that possibly achieves g and has no "obvious" flaws.

An obvious flaw is defined as a flaw that can be detected by considering two actions or two literals at a time—in other words, by looking at the mutex relations. There could be more subtle flaws involving three, four, or more actions, but experience has shown that it is not worth the computational effort to keep track of these possible flaws. This is similar to the lesson learned from constraint satisfaction problems that it is often worthwhile to compute 2-consistency before searching for a solution, but less often worthwhile to compute 3-consistency or higher.

Planning and Acting in the Real World:

The classical planning representation talks about what to do, and in what order, but the representation cannot talk about time: how long an action takes and when it occurs. For example, the planners could produce a schedule for an airline that says which planes are assigned to which flights, but we really need to know departure and arrival times as well. This is the subject matter of scheduling. The real world also imposes many resource constraints; for example, an airline has a limited number of staff—and staff, who are on one flight cannot be on another at the same time. This section covers methods for representing and solving planning problems that include temporal and resource constraints.

Time, Schedules, and Resources - extends the classical language for planning to talk about actions with durations and resource constraints

Hierarchical Planning - describes methods for constructing plans that are organized hierarchically. This allows human experts to communicate to the planner what they know about how to solve the problem.

Hierarchy also lends itself to efficient plan construction because the planner can solve a problem at an abstract level before delving into details

Planning and Acting in Non-deterministic Domains - presents agent architectures that can handle uncertain environments and

interleave deliberation with execution, and gives some examples of real-world systems.

Multi-Agent Planning - shows how to plan when the environment contains other agents

The approach we take in this section is "plan first, schedule later": that is, we divide the overall problem into a planning phase in which actions are selected, with some ordering constraints, to meet the goals of the problem, and a later scheduling phase, in which temporal information is added to the plan to ensure that it meets resource and deadline constraints.

$$
\begin{aligned}
&Jobs(\{AddEngine1 \prec AddWheels1 \prec Inspect1\},\\
&\quad \{AddEngine2 \prec AddWheels2 \prec Inspect2\})\\
&Resources(EngineHoists(1), WheelStations(1), Inspectors(2), LugNuts(500))\\
&Action(AddEngine1, \text{DURATION}{:}30,\\
&\quad \text{USE}{:}EngineHoists(1))\\
&Action(AddEngine2, \text{DURATION}{:}60,\\
&\quad \text{USE}{:}EngineHoists(1))\\
&Action(AddWheels1, \text{DURATION}{:}30,\\
&\quad \text{CONSUME}{:}LugNuts(20), \text{USE}{:}\ WheelStations(1))\\
&Action(AddWheels2, \text{DURATION}{:}15,\\
&\quad \text{CONSUME}{:}LugNuts(20), \text{USE}{:}\ WheelStations(1))\\
&Action(Inspect_i, \text{DURATION}{:}10,\\
&\quad \text{USE}{:}Inspectors(1))
\end{aligned}
$$

A job-shop scheduling problem for assembling two cars, with resource constraints.The notation A<B means that action A must precede action B.

This approach is common in real-world manufacturing and logistical settings, where the planning phase is often performed by human experts. The automated methods can also be used for the planning phase, provided that they produce plans with just the minimal ordering constraints required for correctness.

Time, Schedules, and Resources

The classical planning representation talks about what to do and in what order, but the representation cannot talk about

Temporal ordering constraints - when an action should occur (before and/or after a specified time and/or specific action(s))

Resource constraints - describes the resources needed for an action to be executed

The available resources - described below

Representing Temporal Constraints, Resource Constraints, and the Available Resources each action is represented by:

- a duration
- a set of temporal ordering constraints (action(s) that must be completed before this action can be executed)
- a set of resource constraints

Each resource is represented by 3 things:

- the type of resource (e.g. bolts, wrenches, or pilots)
- the number of that resource available at start
- whether that resource is:
  Consumable - e.g. the bolts are no longer available for use
  Reusable - e.g. a pilot is occupied during a flight but is available again when the flight is over
- sharable

Resources can be produced by actions; a solution must satisfy all the temporal ordering constraints of actions and resource constraints

Representing temporal and resource constraints

A typical job-shop scheduling problem, consists of a set of jobs, each of which JOB consists a collection of actions with ordering constraints among them. Each action has duration and a set of resource constraints required by the action. Each constraint specifies a type of resource (e.g., bolts, wrenches, or pilots), the number of that resource required, and whether that resource is consumable (e.g., the bolts are no longer available for use) or reusable (e.g., a pilot is occupied during a flight but is available again when the flight is over). Resources can also be produced by actions with negative consumption, including manufacturing, growing, and resupply actions.

A solution to a job-shop scheduling problem must specify the start times for each action and must satisfy all the temporal ordering constraints and resource constraints. As with search and planning problems, solutions can be evaluated according to a cost function;

this can be quite complicated, with nonlinear resource costs, time-dependent delay costs, and so on.

For simplicity, we assume that the cost function is just the total duration of the plan, which is called the makespan. A simple example: a problem involving the assembly of two cars. The problem consists of two jobs, each of the form [AddEngine, AddWheels, Inspect].

Then the Resources statement declares that there are four types of resources and gives the number of each type available at the start: 1 engine hoist, 1 wheel station, 2 inspectors, and 500 lug nuts. The action schemas give the duration and resource needs of each action. The lug nuts are consumed as wheels are added to the car, whereas the other resources are “borrowed” at the start of an action and released at the action’s end.

The representation of resources as numerical quantities, such as Inspectors (2), rather than as named entities, such as Inspector (I1) and Inspector (I2), is an example of a very general technique called aggregation. The central idea of aggregation is to group individual objects into quantities when the objects are all indistinguishable with respect to the purpose at hand. In our assembly problem, it does not matter which inspector inspects the car, so there is no need to make the distinction.

Aggregation is essential for reducing complexity. Consider what happens when a proposed schedule has 10 concurrent Inspect actions but only 9 inspectors are available. With inspectors represented as quantities, a failure is detected immediately and the algorithm backtracks to try another schedule. With inspectors represented as individuals, the algorithm backtracks to try all 10! ways of assigning inspectors to actions.

Hierarchical Planning

The problem-solving and planning methods of the preceding chapters all operate with a fixed set of atomic actions. Actions can be strung together into sequences or branching networks; state-of-the-art algorithms can generate solutions containing thousands of actions. For plans executed by the human brain, atomic actions are muscle activations.

In very round numbers, we have about 10^3 muscles to activate (639, by some counts, but many of them have multiple subunits); we can modulate their activation perhaps 10 times per second; and we are alive and awake for about 10^9 seconds in all. Thus, a human life contains about 10^{13} actions, give or take one or two orders of magnitude.

Even if we restrict ourselves to planning over much shorter time horizons—for example, a two-week vacation in Hawaii—a detailed motor plan would contain around 10^{10} actions. This is a lot more than 1000. To bridge this gap, AI systems will probably have to do what humans appear to do: plan at higher levels of abstraction. A reasonable plan for the Hawaii vacation might be "Go to San Francisco airport; take Hawaiian Airlines flight 11 to Honolulu; do vacation stuff for two weeks; take Hawaiian Airlines flight 12 back to San Francisco; go home." Given such a plan, the action "Go to San Francisco airport" can be viewed as a planning task in itself, with a solution such as "Drive to the long-term parking lot; park; take the shuttle to the terminal."

Each of these actions, in turn, can be decomposed further, until we reach the level of actions that can be executed without deliberation to generate the required motor control sequences.

In this example, we see that planning can occur both before and during the execution of the plan; for example, one would probably defer the problem of planning a route from a parking spot in long-term parking to the shuttle bus stop until a particular parking spot has been found during execution. Thus, that particular action will remain at an abstract level prior to the execution phase. For example, complex software is created from a hierarchy of subroutines or object classes; armies operate as a hierarchy of units; governments and corporations have hierarchies of departments, subsidiaries, and branch offices.

The key benefit of hierarchical structure is that, at each level of the hierarchy, a computational task, military mission, or administrative function is reduced to a small number of activities at the next lower level, so the computational cost of finding the correct way to arrange those activities for the current problem is small. Nonhierarchical methods, on the other hand, reduce a task to a large number of individual actions; for large-scale problems, this is completely impractical.

Planning and Acting with Nondeterminism

- Conformant planning
  (w/o observations)
- Contingency planning
  (for partially observable/nondeterministic environments)
- Online planning/replanning (for unknown environments)

Indeterminacy in the World

Bounded indeterminacy: actions can have unpredictable effects, but the possible effects can be listed in the action description axioms

Unbounded indeterminacy: set of possible preconditions or effects either is unknown or is too large to be completely enumerated closely related to qualification problem

Solutions

Conformant or sensorless planning

Devise a plan that works regardless of state or outcome such plans may not exist

Conditional planning

Plan to obtain information (observation actions) Subplan for each contingency, e.g., [Check(Tire1), if Intact(Tire1) then Inflate(Tire1) else CallAAA Expensive because it plans for many unlikely cases

Monitoring/Replanning

Assume normal states, outcomes

Check progress during execution, replan if necessary

Unanticipated outcomes may lead to failure (e.g., no AAA card) (Really need a combination; plan for likely/serious eventualities, deal with others when they arise, as they must eventually)

Conformant planning

Search in space of belief states (sets of possible actual states)

Conditional planning

If the world is nondeterministic or partially observable then percepts usually provide information, i.e., split up the belief state

Conditional plans check (any consequence of KB +) percept
[. .. , if C then Plan A else Plan B, .. .]
Execution: check C against current KB, execute "then" or "else"

Need to handle nondeterminism by building into the plan conditional steps that check the state of the environment at run time, and then decide what to do.

Augment STRIPS to allow for nondeterminism:

Augment STRIPS to allow for nondeterminism:

Add Disjunctive effects (e.g., to model when action sometimes fails):

$Action(Left,$ PRECOND:$AtR,$ EFFECT: $AtL \vee AtR)$

Add Conditional effects (i.e., depends on state in which it's executed):

Form: **when** $<condition>$: $<effect>$

$Action(Suck,$ PRECOND:,

EFFECT: (**when** AtL: $CleanL$) $\wedge$ (**when** AtR: $CleanR$))

Create Conditional steps:

if $<test>$ **then** *plan-A* **else** *plan-B*

Need some plan for every possible percept and action outcome

(Cf. game playing: some response for every opponent move)
(Cf. backward chaining: some rule such that every premise satisfied

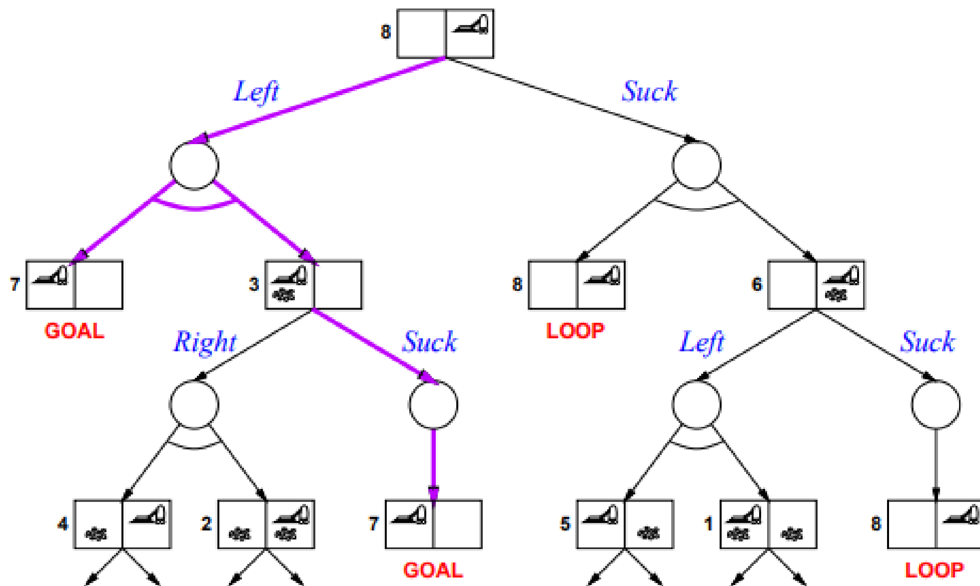
Use: AND–OR tree search (very similar to backward chaining algorithm)
Similar to game tree in minimax search
Differences: Max and Min nodes become OR and AND nodes

- Robot takes action in "state" nodes.
- Nature decides outcome at "chance" nodes.
- Plan needs to take some action at every state it reaches
- (i.e., Or nodes)
- Plan must handle every outcome for the action it takes
- (i.e., And nodes)

Solution is a subtree with (1) goal node at every leaf, (2) one action specified at each state node, and (3) includes every outcome branch at chance nodes.

Example: "Game Tree", Fully Observable World

Double Murphy: sucking or arriving may dirty a clean square



Plan: [*Left*, **if** $AtL \wedge CleanL \wedge CleanR$ **then** [] **else** *Suck*]

Assume: You have a chair, a table, and some cans of paint; all colors are unknown.

Goal: chair and table have same color.

How would each of the following handle this problem?

Classical planning:
Can't handle it, because initial state isn't fully specified.

Sensorless/Conformant planning:
Open can of paint and apply it to both chair and table.

Conditional planning:
Sense the color of the table and chair. If same, then we're done. If not, sense labels on the paint cans; if there is a can that is the same color as one piece of furniture, then apply the paint to the other piece. Otherwise, paint both pieces with any color.

Monitoring/replanning:

Similar to conditional planner, but perhaps with fewer branches at first, which are filled in as needed at runtime. Also, would check for unexpected outcomes (e.g., missed a spot in painting, so repaint)

Incomplete info: use conditional plans; conformant planning (can use belief states)

Incorrect info: use execution monitoring and replanning

MCQ

1. Following is/are the components of the partial order planning.

   a)Bindings
   b) Goal
   c) Causal Links
   d) All of the mentioned

2. A plan that describe how to take actions in levels of increasing refinement and specificity is ____________

   a)Problem solving
   b) Planning
   c) Non-hierarchical plan
   d) Hierarchical plan

3. A constructive approach in which no commitment is made unless it is necessary to do so, is ____________
   a)Least commitment approach
   b) Most commitment approach
   c) Nonlinear planning
   d) Opportunistic planning

4. Uncertainty arises in the Wumpus world because the agent's sensors give only ____________
   a)Full & Global information
   b) Partial & Global Information
   c) Partial & local Information
   d) Full & local information

5. Which of the following search belongs to totally ordered plan search?

a)**Forward state-space search**
b) Hill-climbing search
c) Depth-first search
d) Breadth-first search

6. Which cannot be taken as advantage for totally ordered plan search?

a)Composition
b) State search
c) Problem decomposition
d) None of the mentioned

7. What is the advantage of totally ordered plan in constructing the plan?

a)Reliability
b) Flexibility
c) Easy to use
d) All of the mentioned

8. Which strategy is used for delaying a choice during search?

a)First commitment
b) Least commitment
c) Both First & Least commitment
d) None of the mentioned

9. Which algorithm places two actions into a plan without specifying which should come first?

a)Full-order planner
b) Total-order planner
c) Semi-order planner
d) Partial-order planner

10.What is the other name of each and every total-order plans?

a)Polarization
b) Linearization
c) Solarization
d) None of the mentioned

CONCLUSION:

Upon completion of this, Students should be able to

- Understand Planning in AI.
- Understand Partial-order planning in AI.
- Understand Planning and acting in the real world.

REFERENCES

1. David Poole, Alan Mackworth, Randy Goebel, "Computational Intelligence: a Logical Approach", Oxford University Press, 2004

2. G. Luger, "Artificial Intelligence: Structures and Strategies for Complex Problem Solving", Fourth Edition, Pearson Education, 2002.

ASSIGNMENT

1. Explain the planning in AI.
2. Explain the Partial-order planning in AI.
3. Explain Planning and acting in the real world.