

ARTIFICIAL INTELLIGENCE

UNIT I: INTRODUCTION (9)

AI problems, foundation of AI, history of AI, intelligent agents: Agents and Environments, the concept of rationality, the nature of environments, structure of agents, problem solving agents, problem formulation.

UNIT 2: SEARCHING (9)

Searching for solutions, uninformed search strategies – Breadth first search, depth first Search. Search with partial information (Heuristic search) Hill climbing, A^* , AO^* Algorithms, Problem reduction, Game Playing-Adversarial search, Games, mini-max algorithm, optimal decisions in multiplayer games, Problem in Game playing, Alpha-Beta pruning, Evaluation functions.

UNIT 3: REPRESENTATION OF KNOWLEDGE (9)

Knowledge representation issues, predicate logic- logic programming, semantic nets- frames and inheritance, constraint propagation, representing knowledge using rules, rules based deduction systems. Reasoning under uncertainty, review of probability, Bayes' probabilistic interferences and dempstershafer theory.

UNIT 4: LOGIC CONCEPTS (9)

First order logic. Inference in first order logic, propositional vs. first order inference, unification & lifts forward chaining, Backward chaining, Resolution, Learning from observation Inductive learning, Decision trees, Explanation based learning, Statistical Learning methods, Reinforcement Learning.

UNIT 5: EXPERT SYSTEMS (9)

Architecture of expert systems, Roles of expert systems – Knowledge Acquisition Meta knowledge Heuristics. Typical expert systems – MYCIN, DART, XCON: Expert systems shells.

COURSE OUTCOMES:

On successful completion of the course, students will be able to		POs
CO1	Study the concepts of Artificial Intelligence.	PO1, PO2
CO2	Learn the methods of solving problems using Artificial Intelligence	PO1, PO2, P03, PO4, PO5
CO3	Introduce the concepts of Expert Systems.	PO1, PO2, P03, PO4, PO5
CO4	Understand the applications of AI, namely game playing, theorem proving, and machine learning.	PO1, PO2, P03, PO4, PO5
CO5	Learn different knowledge representation techniques	PO1, PO2, P03, PO4

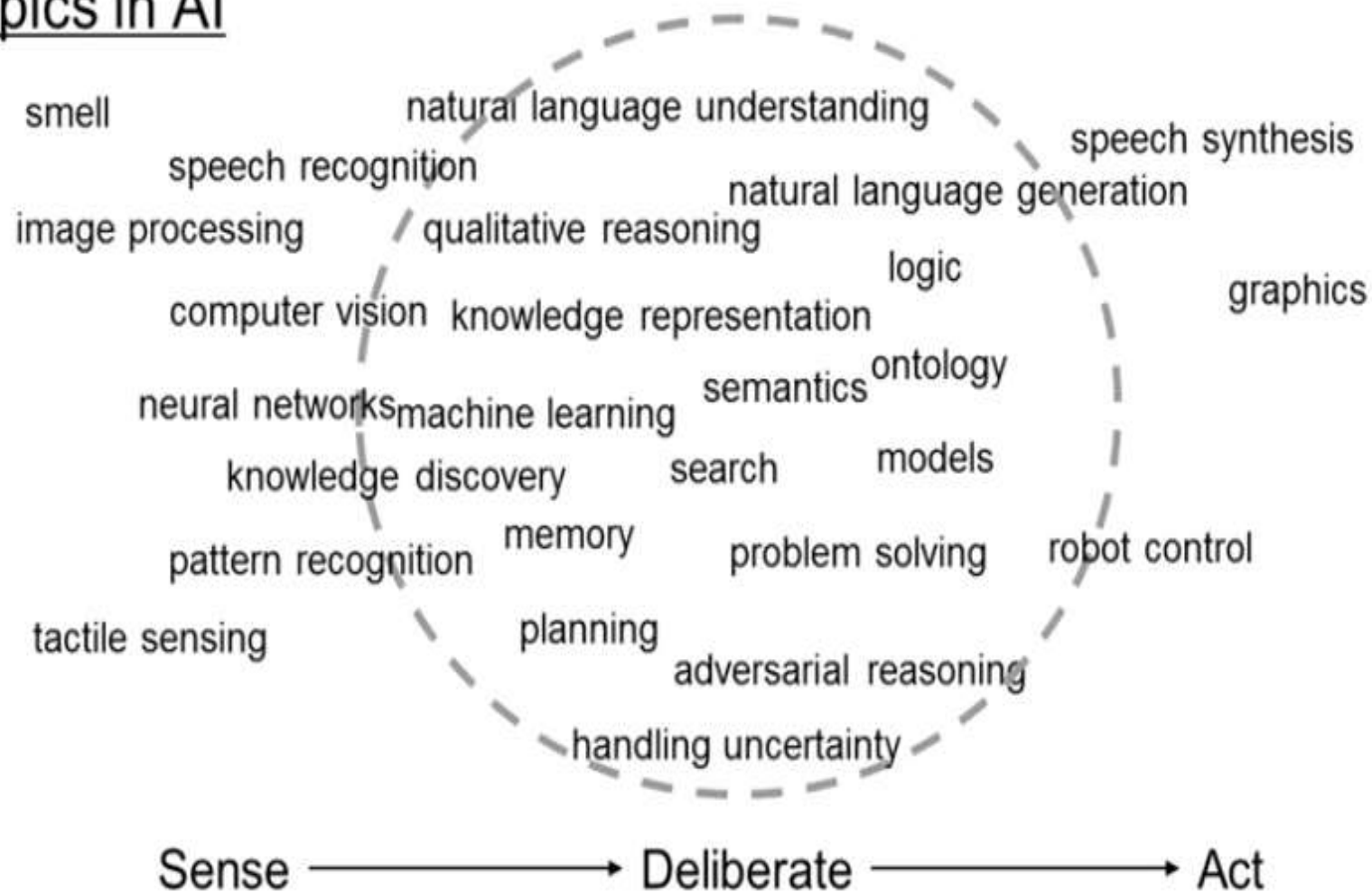
TEXT BOOKS:

- S. Russel and P. Norvig, “Artificial Intelligence – A Modern Approach”, Second Edition, Pearson Education, 2007.
- Kevin Night and Elaine Rich, Nair B., “Artificial Intelligence (SIE)”, Mc Graw Hill, 2008.

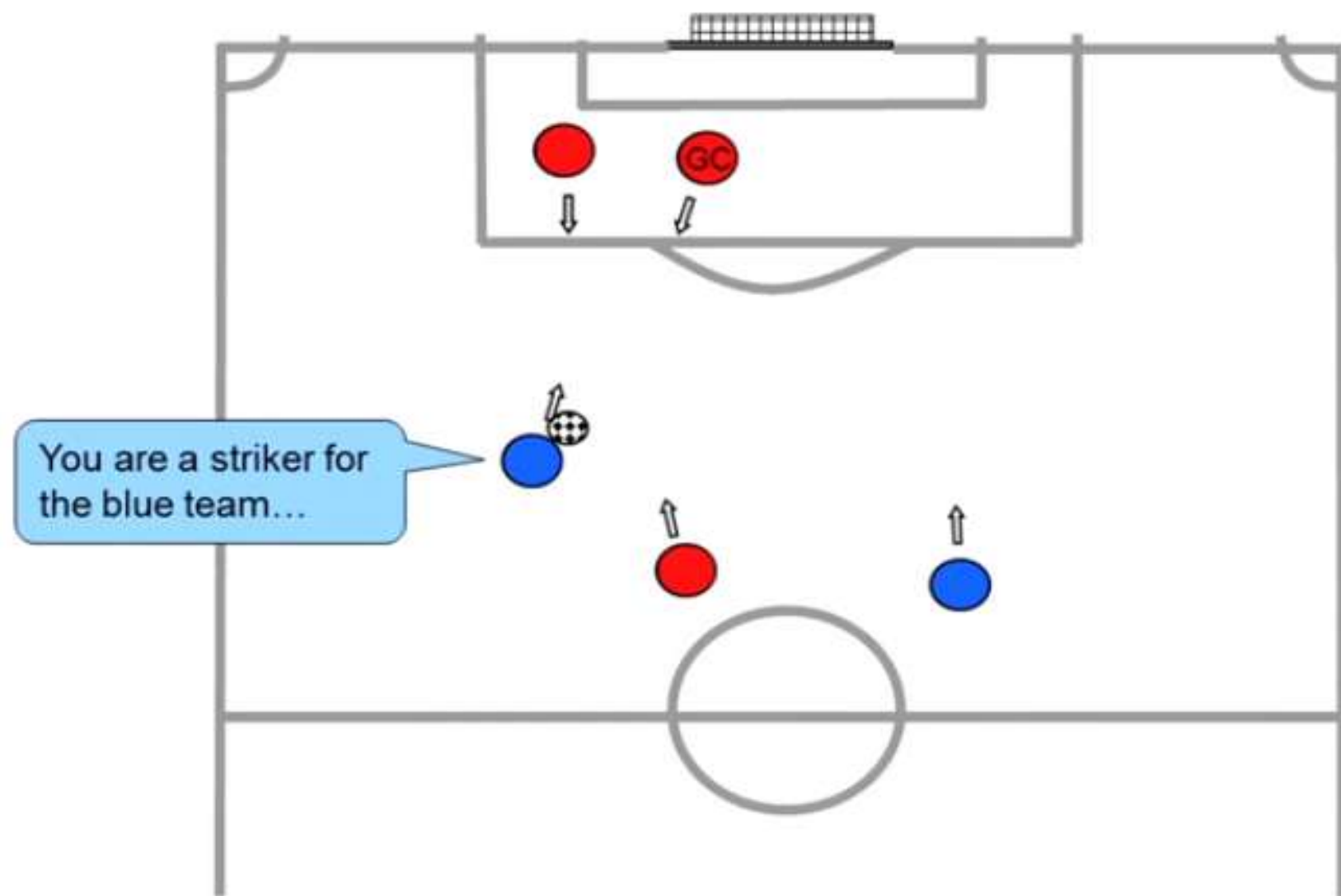
REFERENCE BOOKS:

1. David Poole, Alan Mackworth, Randy Goebel, “Computational Intelligence: a logical approach”, Oxford University Press.
2. G. Luger, “Artificial Intelligence: Structures and Strategies for complex problemsolving”, Fourth Edition, Pearson Education.
3. J. Nilsson, “Artificial Intelligence: A new Synthesis”, Elsevier Publishers.
4. Artificial Intelligence, SarojKaushik, CENGAGE Learning.

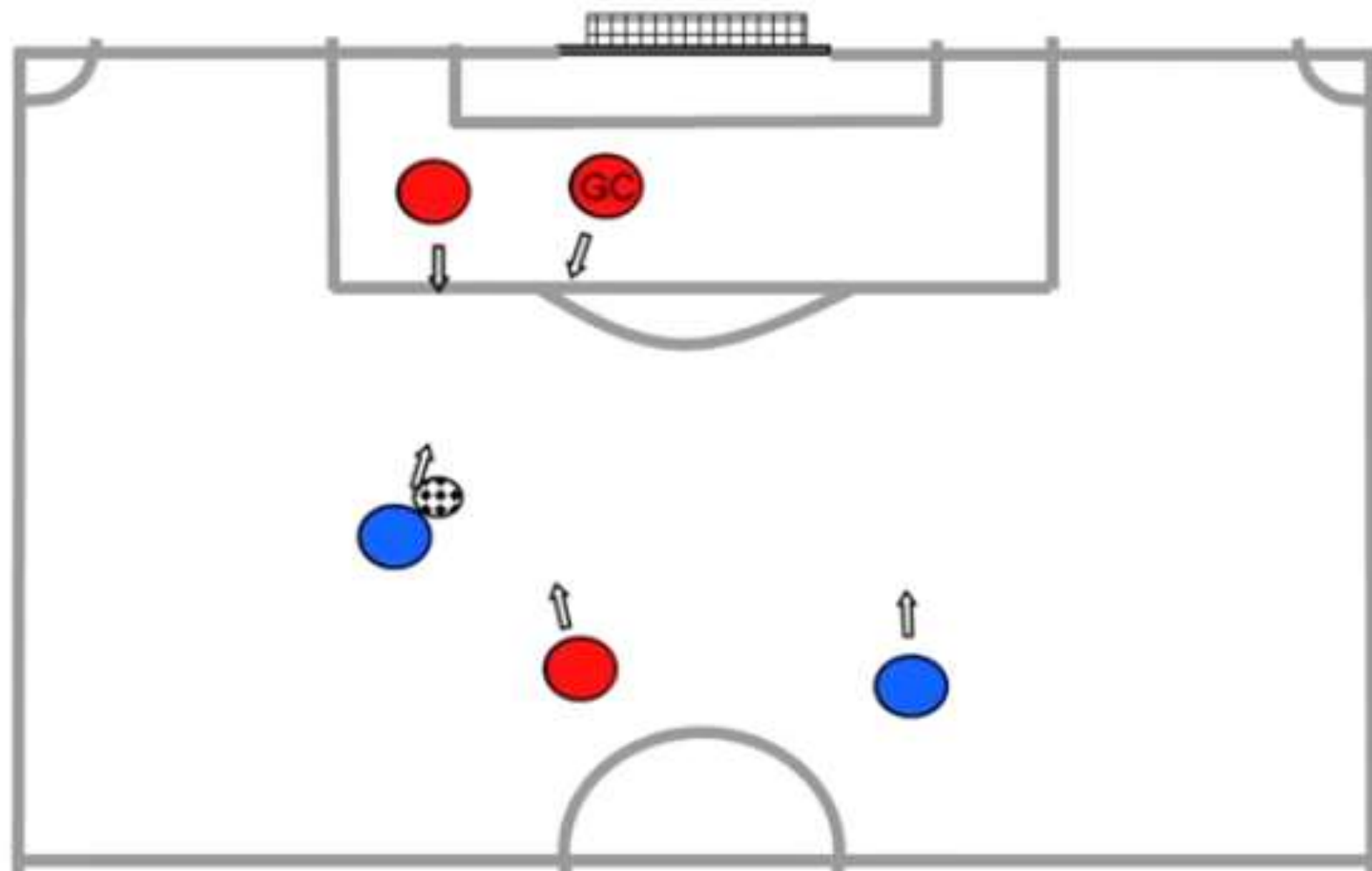
Topics in AI



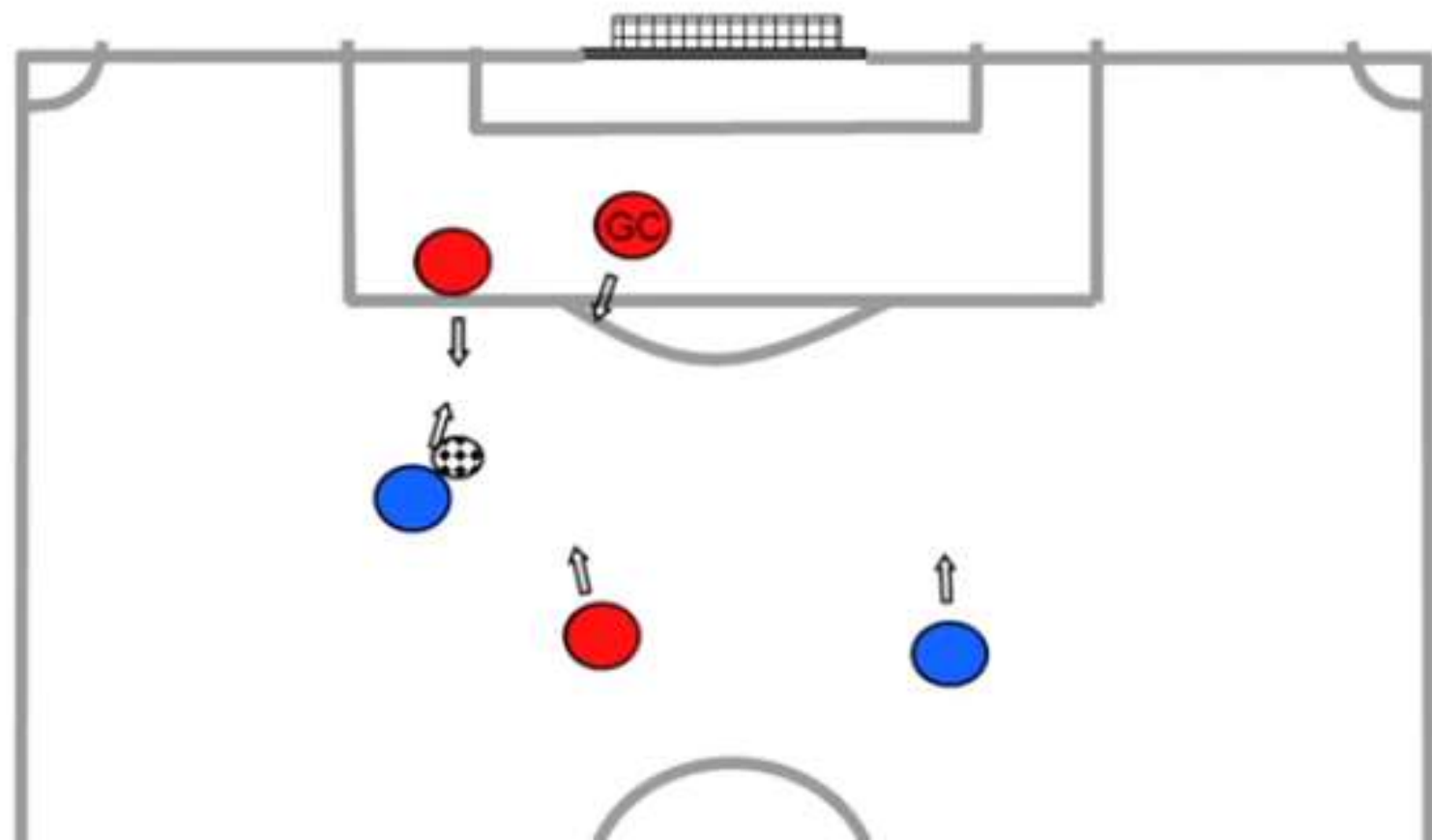
On a football field...



.. moving forward with the ball...



... the defenders are closing in ... what should you do?



Problem solving

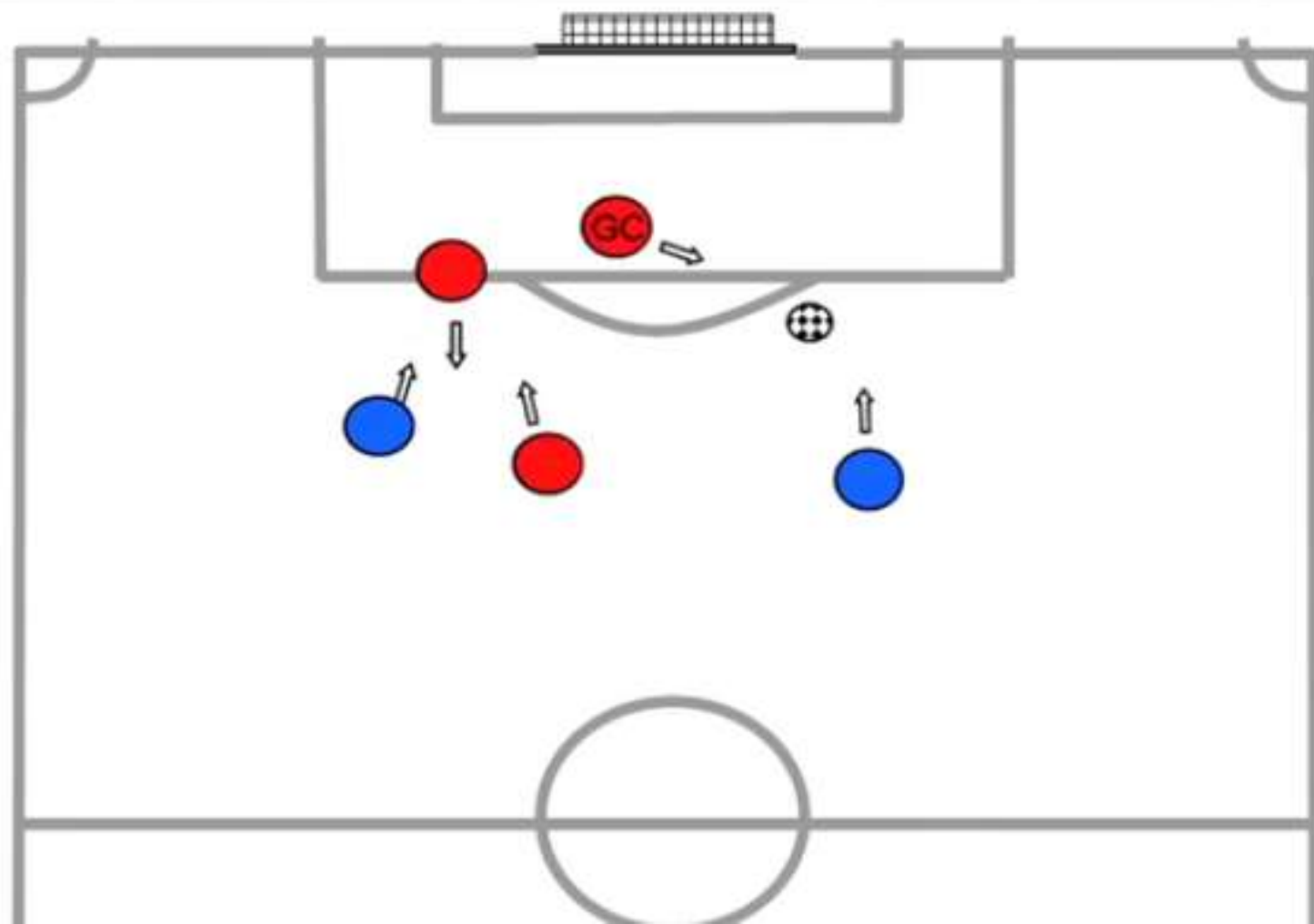
An autonomous agent
in some world

has a **goal** to achieve

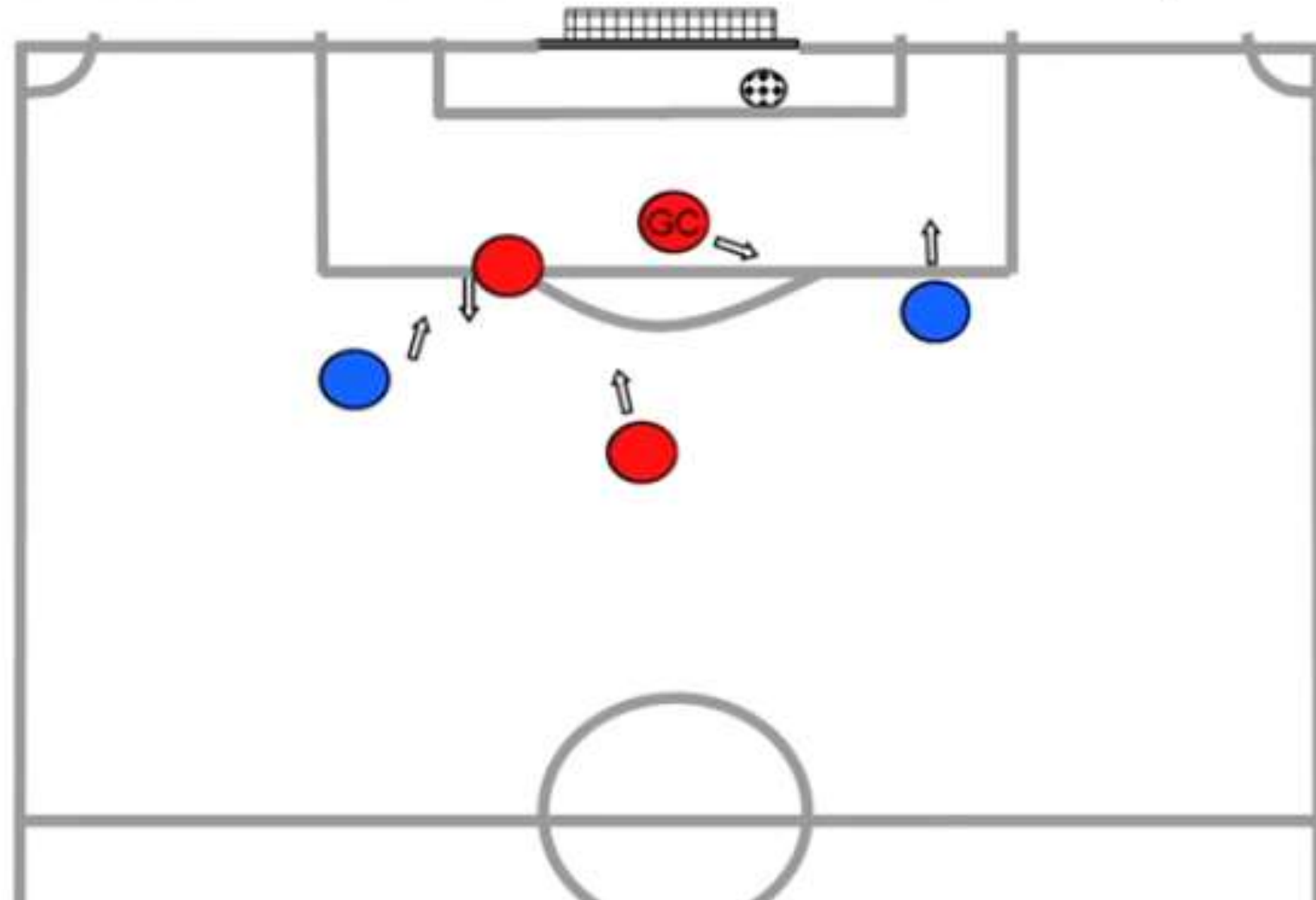
and

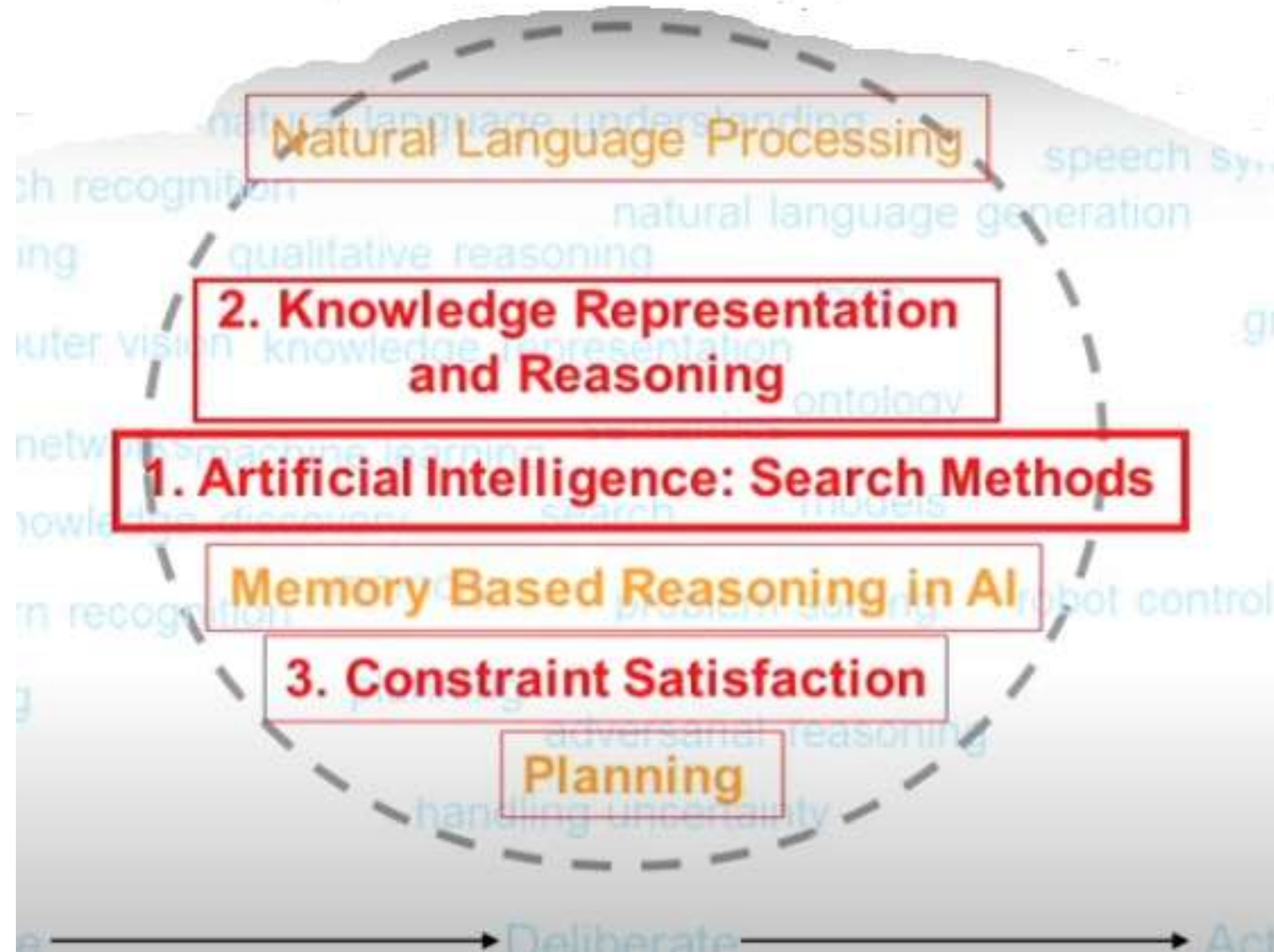
a set of actions to choose from
to strive for the goal

Pass the ball to a teammate who is free...



... and has a better chance to score





- **INTELLIGENCE**

- The capacity to learn and solve problems.
- In particular,
 - the ability to solve novel problems (i.e solve new problems)
 - the ability to act rationally (i.e act based on reason)
- **What is involved in intelligence?**
 - **Ability to interact with the real world**
 - to perceive, understand, and act
 - e.g., speech recognition and understanding and synthesis
 - e.g., image understanding
 - e.g., ability to take actions

- **Reasoning and Planning**

- modeling the external world, given input
- solving new problems, planning, and making decisions
- ability to deal with unexpected problems, uncertainties

- **Learning and Adaptation**

- we are continuously learning and adapting
- The internal models are always being updated.
e.g., a baby learning to categorize and recognize animals

What is artificial intelligence?

- 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*."
- **Artificial Intelligence** is the 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."
- Major AI textbooks define artificial intelligence as "the study and design of intelligent agents," where an **intelligent agent** is a system that **perceives** its **environment** and **takes actions** which maximize its chances of success.
- **John McCarthy**, who coined the term in 1956, defines it as "the science and engineering of making intelligent machines, especially intelligent computer programs."

ARTIFICIAL INTELLIGENCE

The term AI is defined by each author in own ways which falls into 4 categories

1. System that think like humans.
2. System that act like humans.
3. Systems that think rationally.
4. Systems that act rationally.

Thinking Humanly “The exciting new effort to make computers think . . . <i>machines with minds</i>, in the full and literal sense.” (Haugeland, 1985) “[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . .” (Bellman, 1978)	Thinking Rationally “The study of mental faculties through the use of computational models.” (Charniak and McDermott, 1985) “The study of the computations that make it possible to perceive, reason, and act.” (Winston, 1992)
Acting Humanly “The art of creating machines that perform functions that require intelligence when performed by people.” (Kurzweil, 1990) “The study of how to make computers do things at which, at the moment, people are better.” (Rich and Knight, 1991)	Acting Rationally “Computational Intelligence is the study of the design of intelligent agents.” (Poole <i>et al.</i>, 1998) “AI . . . is concerned with intelligent behavior in artifacts.” (Nilsson, 1998)
Figure 1.1 Some definitions of artificial intelligence, organized into four categories.	

SOME DEFINITIONS OF AI

- **Building systems that think like humans**

- The exciting new effort to make computers think ... machines with minds, in the full and literal sense -- Haugeland, 1985
- The automation of activities that we associate with human thinking, ... such as decision-making, problem solving, learning, ...|| -- Bellman, 1978

- **Building systems that act like humans**

- The art of creating machines that perform functions that require intelligence when performed by people -- Kurzweil, 1990
- The study of how to make computers do things at which, at the moment, people are better -- Rich and Knight, 1991

- **Building systems that think rationally**

- The study of mental faculties through the use of computational models -- Charniak and McDermott, 1985
- The study of the computations that make it possible to perceive, reason, and act-Winston, 1992

- **Building systems that act rationally**

- A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes -- Schalkoff, 1990
- The branch of computer science that is concerned with the automation of intelligent behavior -- Luger and Stubblefield, 1993

- we see eight definitions of AI, laid out along two dimensions. The definitions on top are concerned with *thought processes* and *reasoning*, whereas the ones on the bottom address *behavior*.
- The definitions on the left measure success in terms of fidelity to *human* performance, whereas the ones on the right measure against an *ideal* performance measure, called **rationality**.
- A system is rational if it does the “right thing,” given what it knows.

Acting Humanly: The Turing Test Approach

- Turing Test proposed by Alan Turing in 1950
- The computer is asked questions by a human interrogator.
- The computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the responses come from a person or not. Programming a computer to pass, the computer need to possess the following capabilities:

Natural language processing to enable it to communicate successfully in English.

Knowledge representation to store what it knows or hears

Automated reasoning to use the stored information to answer questions and to draw new conclusions.

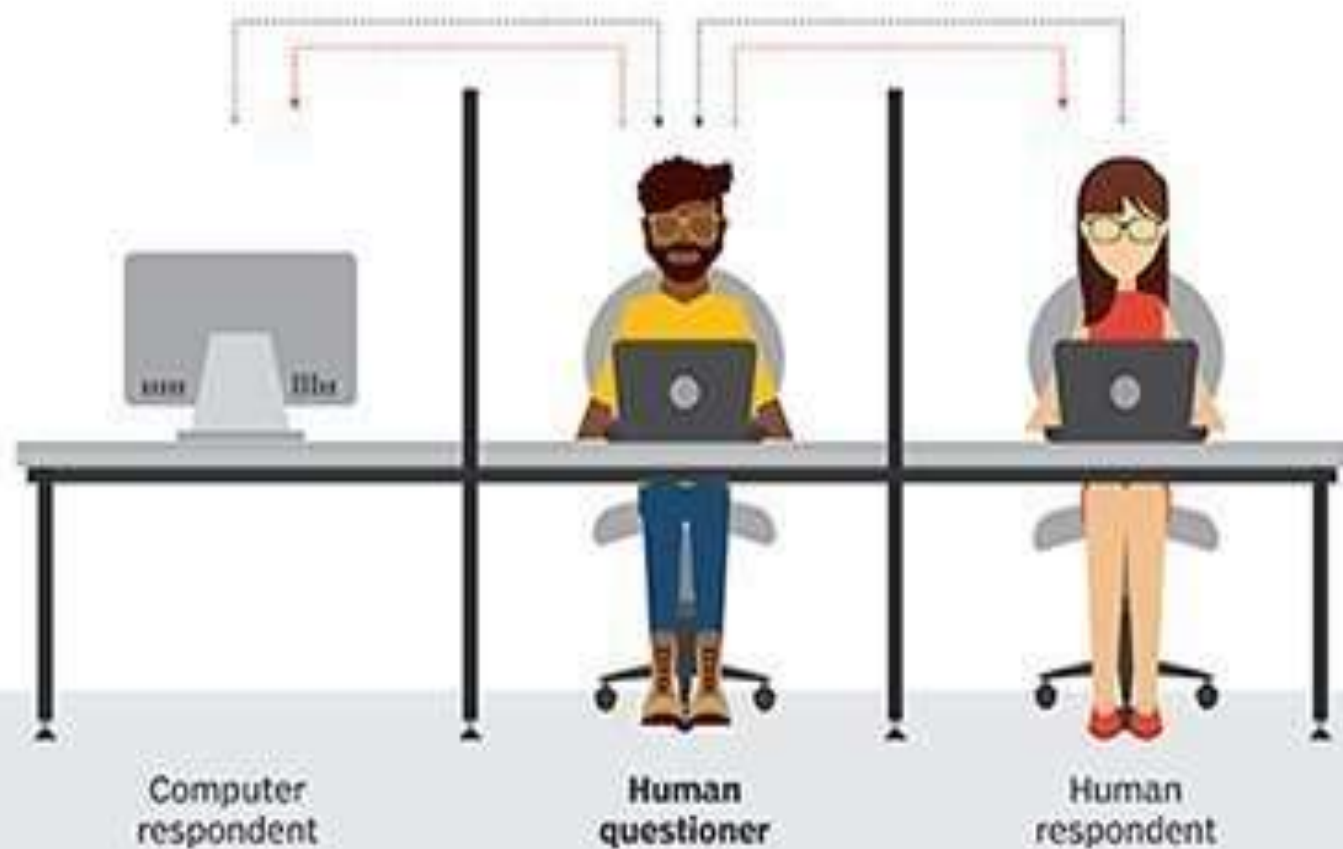
Machine learning to adapt to new circumstances and to detect and extrapolate patterns.

- **Total Turing Test** includes a video signal so that the interrogator can test the subject's perceptual abilities, as well as the opportunity for the interrogator to pass physical objects as input. To pass the total Turing Test, the computer will need
- **Computer vision** to perceive objects, and
- **Robotics** to manipulate objects and move about.

Turing test

During the Turing test, the human questioner asks a series of questions to both respondents.
After the specified time, the questioner tries to decide which terminal is operated by the human respondent and which terminal is operated by the computer.

■ QUESTION TO RESPONDENTS ■ ANSWERS TO QUESTIONER



Thinking humanly: The cognitive modeling approach

- To say that a given program thinks like a human, we must have some way of determining how humans think
- We need to get inside actual working of the human mind, this can be done:
 - Through introspection – trying to capture our own thoughts as they go by;
 - Through psychological experiments
- Once we have a sufficiently precise theory of the mind, it becomes possible to express the theory as a computer program. If the program's input–output behavior matches corresponding human behavior, that is evidence that some of the program's mechanisms could also be operating in humans.
- Allen Newell and Herbert Simon, who developed GPS, the —General Problem Solver‖ tried to trace the reasoning steps to traces of human subjects solving the same problems. The interdisciplinary field of cognitive science brings together computer models from AI and experimental techniques from psychology to try to construct precise and testable theories of the workings of the human mind

Thinking rationally : The “laws of thought approach”

- The Greek philosopher Aristotle was one of the first to attempt to codify —right thinking that is irrefutable (ie. Impossible to deny) reasoning processes. His syllogism provided patterns for argument structures that always yielded correct conclusions when given correct premises
- for example, Socrates is a man; all men are mortal; therefore Socrates is mortal.
- These laws of thought were supposed to govern the operation of the mind; their study initiated a field called logic.
- The **logician** tradition within artificial intelligence is used to build programs to create intelligent systems.

Acting rationally : The rational agent approach

- An **agent** is just something that acts (*agent* comes from the Latin *agere*, to do). Of course, all computer programs do something, but
- Computer agents are not mere programs, but they are expected to have the following attributes also:
 - (a) operating under autonomous control,
 - (b) perceiving their environment,
 - (c) persisting over a prolonged time period,
 - (d) adapting to change and create and pursue goals.
- A **rational agent** is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome.

Foundations of AI

- The disciplines that contributed ideas, viewpoints, and techniques to AI while addressing some questions.
- **Philosophy**
 - Can formal rules be used to draw valid conclusions?
 - How does the mental mind arise from a physical brain?
 - Where does knowledge come from?
 - How does knowledge lead to action?
- **Mathematics**
 - What are the formal rules to draw valid conclusions?
 - What can be computed?
 - How do we reason with uncertain information?
- **Economics**
 - How should we make decisions so as to maximize payoff?
 - How should we do this when others may not go along?
 - How should we do this when the payoff may be far in the future?

- **Neuroscience**
 - How do brains process information?
- **Psychology**
 - How do humans and animals think and act?
- **Computer engineering**
 - How can we build an efficient computer?
- **Control theory and cybernetics**
 - How can artifacts operate under their own control?

Levels of AI (types of artificial intelligence)

1. ANI: Artificial narrow intelligence
2. AGI: Artificial general intelligence
3. ASI: Artificial super intelligence

1. Narrow AI (Weak AI)

- This is the most common type of AI currently in use.
 - It's designed to perform a specific task extremely well, often better than humans, but it lacks general intelligence.
 - Focuses on one task and cannot perform beyond its limitations (common in our daily lives)
 - makes use of NLP to perform tasks
- Examples include:
 - Email Spam filters
 - Recommendation systems
 - Facial recognition software
 - Robots that perform repetitive tasks on assembly lines
 - Virtual assistants like Siri or Alexa
 - IBMS Watson, Self driving cars,

2. General AI (Strong AI):

- This level of AI would possess human-level cognitive abilities, meaning it could learn, understand, and apply knowledge across various tasks, just like a human.
- General AI could potentially learn new things on its own and apply its knowledge to new situations.
- This level of AI is currently theoretical and not yet achieved.

➤ strong AI or Deep AI

➤ Can understand and learn any intellectual task that a human being can (researchers are striving to reach strong AI)

➤ machines can mimic human intelligence

➤ ability to apply their intelligence to solve problems

➤ properties of Deep AI are: recognition, recall, imagination

Ex: Advanced robotics

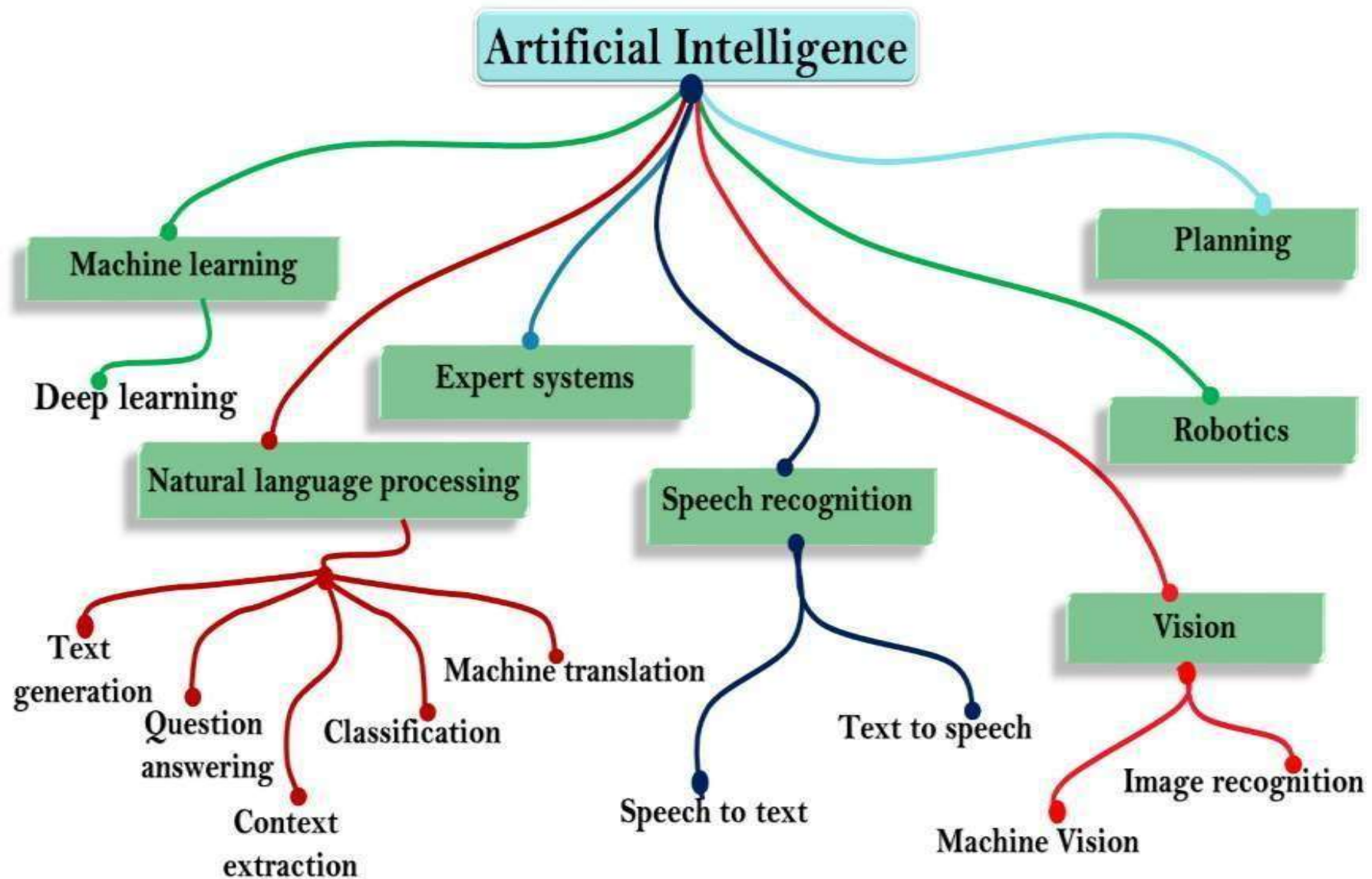
ASI Artificial super intelligence

- Surpasses human intelligence and can perform any task better than a human (still a concept)
- level where machines super passes human capabilities and become self aware, able to build emotions of its own, better than human in sports, math, science etc., in movies only

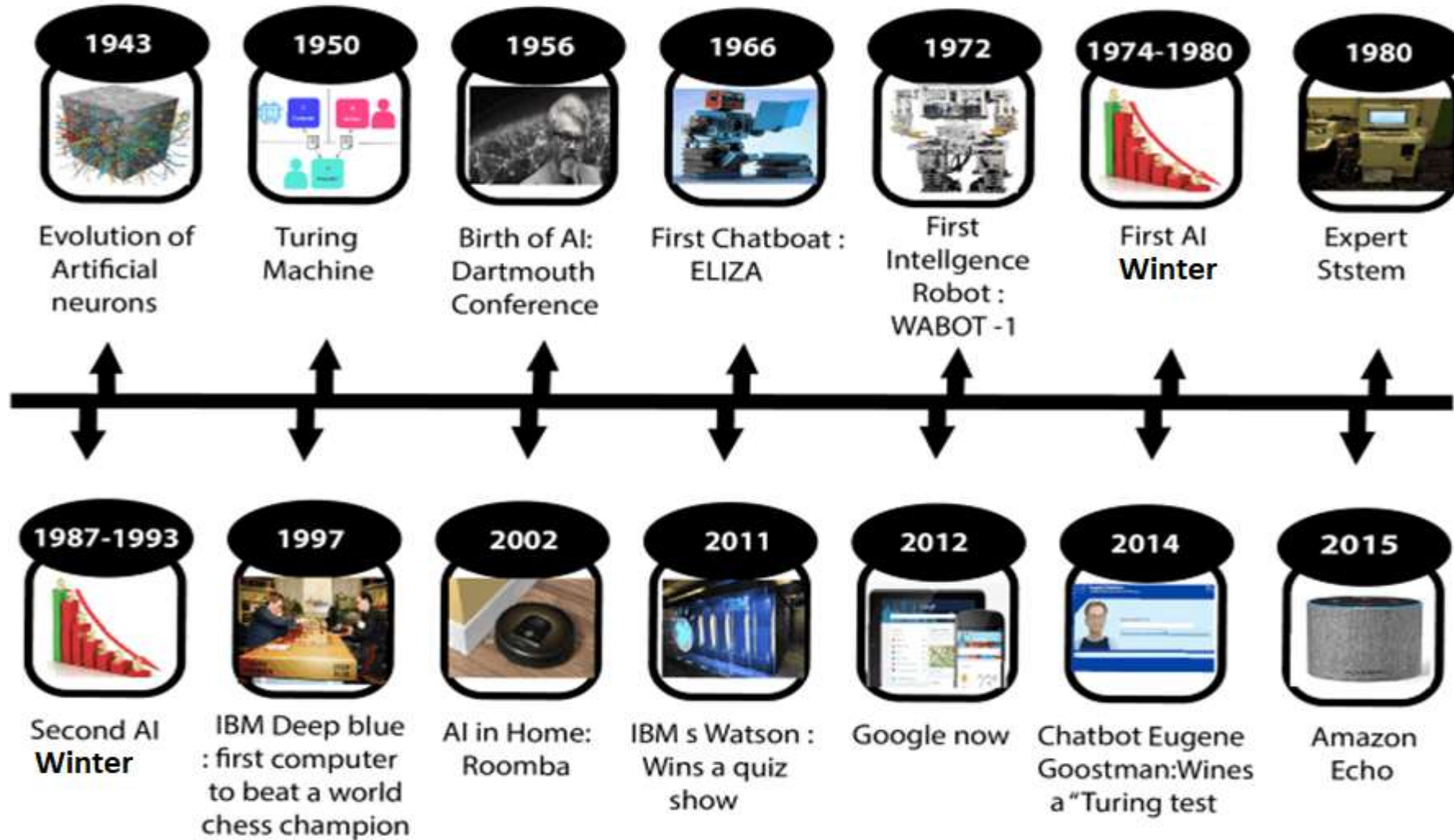
Levels of AI



Subsets of AI



History of AI



History of AI

- **The gestation of artificial intelligence (1943–1955)**
 - **Year 1943:** The first work which is now recognized as AI was done by Warren McCulloch and Walter Pitts 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:** Alan Turing publishes "**Computing Machinery and Intelligence**" in which he proposed a test called a **Turing test**.
- **The birth of artificial intelligence (1956)**
 - **Year 1955:** the "first artificial intelligence program was created , 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.
- **The golden years-Early enthusiasm (1956-1974)**
 - **Year 1966:** The researchers emphasized developing algorithms which can solve mathematical problems. the first chatbot was created 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.

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

Shakey the mobile robot



Shakey the Robot in its display case at the Computer History Museum.

Manufacturer	SRI International
Year of creation	1966 - 1972
Replaced by	Flakey the robot

WABOT-1 (1970~1973)

WABOT-2 (1980~1984)



John McCarthy (September 4, 1927 – October 24, 2011)



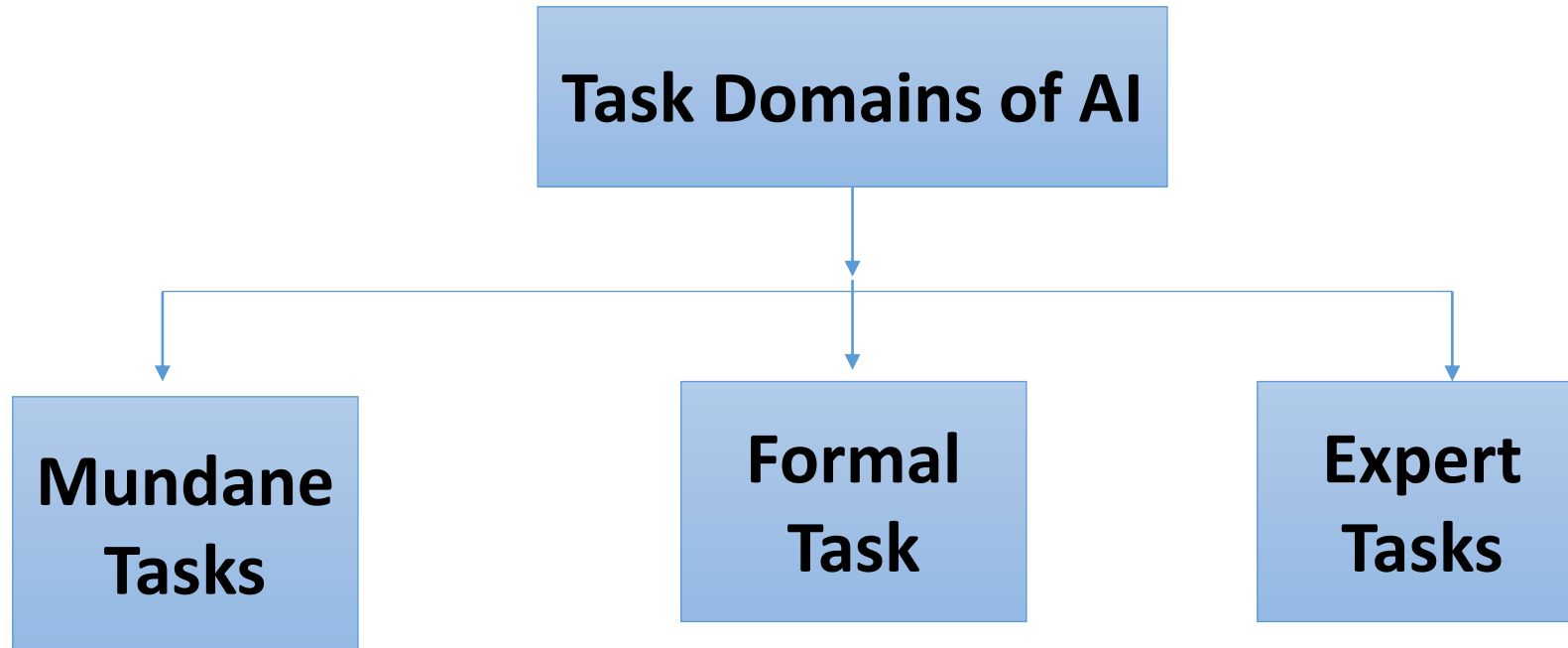
McCarthy at a conference in 2006

Alan Turing

23 June 1912 – 7 June 1954



AI Problems or Task Domains of AI



AI Problems or Task Domains of AI

Mundane tasks	Formal tasks	Expert tasks
Perception – Computer Vision – Speech, Voice	Games – Go – Chess (Deep Blue) – Checkers	Engineering – Design – Fault Finding – Manufacturing – Monitoring
Natural Language Processing – Understanding – Language Generation – Language Translation	Mathematics – Geometry – Logic – Integration and Differentiation	Scientific Analysis
Common Sense Reasoning	Theorem Proving	Financial Analysis
Planning		Medical Diagnosis
Robot Control		

The State of the Art(the application of Artificial Intelligence)

- **Robotic vehicles:** A driverless robotic car named STANLEY raced through the rough terrain of the Mojave dessert at 22 mph, finishing the 132-mile distance first to win the 2005 DARPA Grand Challenge. STANLEY is a Volkswagen Touareg outfitted with cameras, radar, and laser rangefinders to sense the environment and onboard software to command the steering, braking, and acceleration.

- **Speech recognition:** A traveler calling United Airlines to book a flight can have the entire conversation guided by an automated speech recognition and dialog management system.
- **Autonomous planning and scheduling:** A hundred million miles from Earth, NASA's Remote Agent program became the first on-board autonomous planning program to control the scheduling of operations for a spacecraft (Jonsson *et al.*, 2000). REMOTE AGENT generated plans from high-level goals specified from the ground and monitored the execution of those plans—detecting, diagnosing, and recovering from problems as they occurred.

- **Game playing:** IBM's DEEP BLUE became the first computer program to defeat the world champion(Garry Kasparov) in a chess match. Human champions studied Kasparov's loss and were able to draw a few matches in subsequent years, but the most recent human-computer matches have been won convincingly by the computer.
- **Spam fighting:** Each day, learning algorithms classify over a billion messages as spam, saving the recipient from having to waste time deleting what, for many users, could comprise 80% or 90% of all messages, if not classified away by algorithms. Because the spammers are continually updating their tactics, it is difficult for a static programmed approach to keep up, and learning algorithms work best

- **Logistics planning:** During the Persian Gulf crisis of 1991, U.S. forces deployed a Dynamic Analysis and Replanning Tool, DART (Cross and Walker, 1994), to do automated logistics planning and scheduling for transportation. This involved up to 50,000 vehicles, cargo, and people at a time, and had to account for starting points, destinations, routes, and conflict resolution among all parameters. The AI planning techniques generated in hours a plan that would have taken weeks with older methods. The Defense Advanced Research Project Agency (DARPA) stated that this single application more than paid back DARPA's 30-year investment in AI.

- **Robotics:** The iRobot Corporation has sold over two million Roomba robotic vacuum cleaners for home use. The company also deploys the more rugged PackBot to Iraq and Afghanistan, where it is used to handle hazardous materials, clear explosives, and identify the location of snipers.
- **Machine Translation:** A computer program automatically translates from Arabic to English. The program uses a statistical model built from examples of Arabic-to-English translations and from examples of English text totaling two trillion words (Brants *et al.*, 2007). None of the computer scientists on the team speak Arabic, but they do understand statistics and machine learning algorithms.

iRobot Roomba 614 Robotic Vacuum Cleaner :



PACKBOT



Advantages of Artificial Intelligence

1. High Accuracy with less errors

Artificial Intelligence can significantly reduce errors and **increase accuracy and precision**. The decisions taken by AI in every step is decided by information previously gathered and a certain set of algorithms. When programmed properly, these errors can be reduced to null.

2. 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.

3. High reliability: AI machines are highly reliable and can perform the same action multiple times with high accuracy.

- **4. Useful for risky areas:** Another big advantage of AI is that humans can overcome many risks by letting AI robots do them for us. Whether it be defusing a bomb, going to space, exploring the deepest parts of oceans, machines with metal bodies are resistant in nature and can survive unfriendly atmospheres. Moreover, they can provide accurate work with greater responsibility and not wear out easily.
- **5. 24x7 Availability:** There are many studies that show humans are productive only about 3 to 4 hours in a day. Humans also need breaks and time offs to balance their work life and personal life. But AI can work endlessly without breaks. They think much faster than humans and perform multiple tasks at a time with accurate results. They can even handle tedious repetitive jobs easily with the help of AI algorithms.

6. Digital Assistance

Some of the most technologically advanced companies engage with users using digital assistants, which eliminates the need for human personnel. Many websites utilize digital assistants to deliver user-requested content. We can discuss our search with them in conversation. Some chatbots are built in a way that makes it difficult to tell whether we are conversing with a human or a chatbot.

We all know that businesses have a customer service crew that must address the doubts and concerns of the patrons. Businesses can create a chatbot or voice bot that can answer all of their clients' questions using AI.

AI technology is currently used by various E-commerce websites to show the products as per customer requirement.

7. New Inventions

- In practically every field, AI is the driving force behind numerous innovations that will aid humans in resolving the majority of challenging issues.
- For instance, recent advances in AI-based technologies have allowed doctors to detect cancer at an earlier stage.

8. Unbiased Decisions

- Human beings are driven by emotions, whether we like it or not. AI on the other hand, is devoid of emotions and highly practical and rational in its approach. A huge advantage of Artificial Intelligence is that it doesn't have any biased views, which ensures more accurate decision-making.

9. Perform Repetitive Jobs

- We will be doing a lot of repetitive tasks as part of our daily work, such as checking documents for flaws and mailing thank-you notes, among other things. We may use artificial intelligence to efficiently automate these menial chores and even eliminate "boring" tasks for people, allowing them to focus on being more creative.
- Example: In banks, it's common to see multiple document checks to obtain a loan, which is a time-consuming task for the bank's owner. The owner can expedite the document verification process for the advantage of both the clients and the owner by using AI Cognitive Automation.

10. Daily Applications

- Today, our everyday lives are entirely dependent on mobile devices and the internet. We utilize a variety of apps, including Google Maps, Alexa, Siri, Cortana on Windows, OK Google, taking selfies, making calls, responding to emails, etc. With the use of various AI-based techniques, we can also anticipate today's weather and the days ahead.
- Example: About 20 years ago, you must have asked someone who had already been there for instructions when you were planning a trip. All you need to do now is ask Google where Bangalore is. The best route between you and Bangalore will be displayed, along with Bangalore's location, on a Google map.

Disadvantages of Artificial Intelligence

1. High Costs

The ability to create a machine that can simulate human intelligence is no small feat. It requires plenty of time and resources and can cost a huge deal of money. AI also needs to operate on the latest hardware and software to stay updated and meet the latest requirements, thus making it quite costly.

2. No creativity

A big disadvantage of AI is that it cannot learn to think outside the box. AI is capable of learning over time with pre-fed data and past experiences, but cannot be creative in its approach. A classic example is the bot Quill who can write [Forbes earning reports](#). These reports only contain data and facts already provided to the bot. Although it is impressive that a bot can write an article on its own, it lacks the human touch present in other Forbes articles.

3. Unemployment

One application of artificial intelligence is a robot, which is displacing occupations and increasing unemployment (in a few cases). Therefore, some claim that there is always a chance of unemployment as a result of chatbots and robots replacing humans.

For instance, robots are frequently utilized to replace human resources in manufacturing businesses in some more technologically advanced nations like Japan. This is not always the case, though, as it creates additional opportunities for humans to work while also replacing humans in order to increase efficiency.

4. Make Humans Lazy

- [AI applications](#) automate the majority of tedious and repetitive tasks. Since we do not have to memorize things or solve puzzles to get the job done, we tend to use our brains less and less. This addiction to AI can cause problems to future generations.
- With the increment of technology, people are getting more dependent on devices and hence they are losing their mental capabilities.

5. No Ethics

- Ethics and morality are important human features that can be difficult to incorporate into an AI. The rapid progress of AI has raised a number of concerns that one day, AI will grow uncontrollably, and eventually wipe out humanity. This moment is referred to as the AI singularity.

6. Emotionless

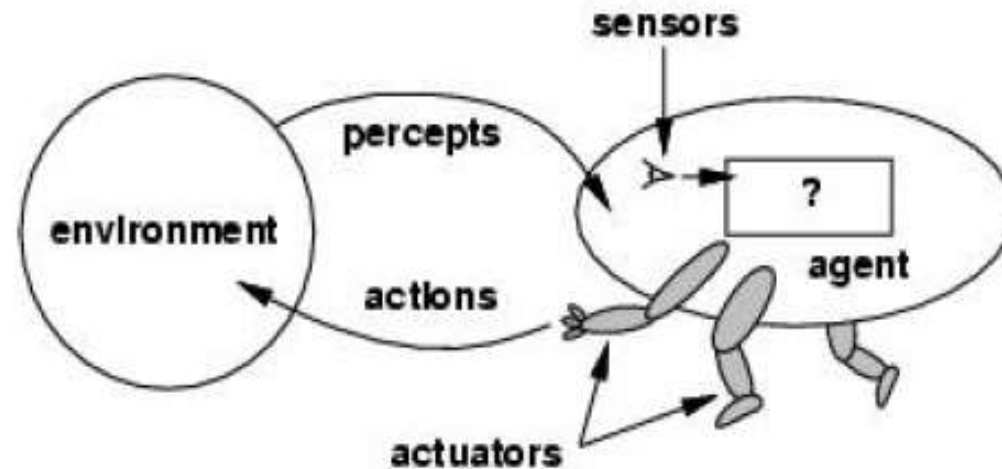
- Since early childhood, we have been taught that neither computers nor other machines have feelings. Humans function as a team, and team management is essential for achieving goals. However, there is no denying that robots are superior to humans when functioning effectively, but it is also true that human connections, which form the basis of teams, cannot be replaced by computers.

7. No Improvement

- Artificial intelligence is a technology based on pre-loaded facts and experience. AI is proficient at repeatedly carrying out the same task, but if we want any adjustments or improvements, we must manually alter the codes. AI cannot be accessed and utilized akin to human intelligence, but it can store infinite data.
- Machines can only complete tasks they have been developed or programmed for; if they are asked to complete anything else, they frequently fail or provide useless results, which can have significant negative effects. Thus, we are unable to make anything conventional.

AGENTS

- An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**. This simple idea is illustrated in Figure.



- A **human agent** has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts for actuators.
- A **robotic agent** might have cameras and infrared range finders for sensors and various motors for actuators.
- A **software agent** receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

Percept: **Percept is used** to refer to the agent's perceptual inputs at any given instant.

Percept Sequence: An agent's **percept sequence** is the complete history of everything the agent has ever perceived.

In general an agent's choice of action at any given instant can depend on the entire percept sequence observed to date.

Agent function

Mathematically speaking, an agent's behavior is described by the **agent function** that maps any given percept sequence to an action.

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

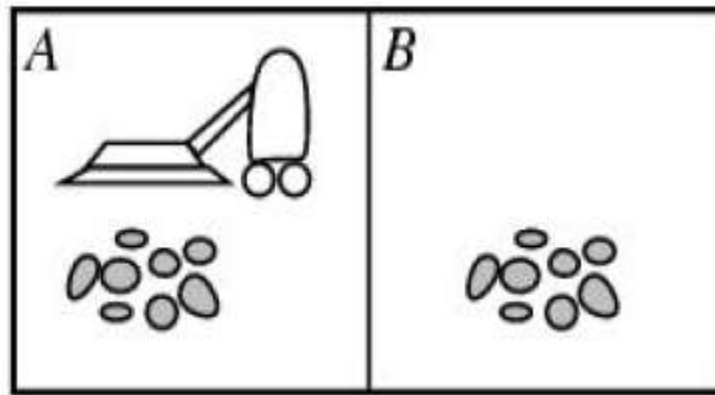
- for **most agents**, Agent function would be a **very large table**

The **agent program** runs on the physical **architecture** to produce agent function(f).

agent = architecture + program

Agent program

- The agent function for an artificial agent will be implemented by an **agent program**.
- It is important to keep these two ideas distinct.
 - ❖ The agent function is an abstract mathematical description;
 - ❖ The agent program is a concrete implementation, running on the agent architecture.
- **Example**-the vacuum-cleaner world.
- This particular world has just two locations: squares A and B.
- The vacuum agent perceives **which square** it is in and whether there is **dirt in the square**.
- It can choose to **move left, move right, suck up the dirt, or do nothing**.
- One very simple agent function is the following:
 - if the current square is dirty, then suck, otherwise,
 - it move to the other square.



Percepts: location and status, [A , Dirty]

Actions: *Left, Right, Suck, NoOp*

Partial tabulation of a simple agent function for the vacuum cleaner world

Percept Sequence	Action
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
.....	

Example of Agent program

```
function Reflex-VACUUM-AGENT ([locations, status]) returns an action  
if status = Dirty then return Suck  
else if location = A then return Right  
elseif location = B then return Left
```


Good Behavior: The concept of Rationality

- A **rational agent** is one that does the right thing-based on what it can perceive and the actions it can perform.
- every entry in the table for the agent function is filled out correctly.
- The right action is the one that will cause the agent to be most successful.

- **A performance measure** represents the **criterion for success** of an agent's behavior.
- When an agent is placed in an environment, it generates a sequence of actions according to the percepts it receives.
- This sequence of actions causes the environment to go through a sequence of states.
- If the sequence is desirable, then the agent has performed well.
- E.g., performance measure of a vacuum-cleaner agent could be amount of dirt cleaned up, amount of time taken, amount of electricity consumed, amount of noise generated, etc.

Rationality

- The rationality at any given time depends on four things:
 - The performance measure that defines the criterion of success.
 - The agent's prior knowledge of the environment.
 - The actions that the agent can perform.
 - The agent's percept sequence to date.

Definition of a rational agent:

For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and some built-in knowledge the agent has about the environment.

- Is a vacuum-agent **rational**? (Depends!)
 - **Performance measure:** awards one point for each clean square at each time step, over a lifetime of 1000 time steps.
 - **Geography** of the environment is **known a priori** (two squares, **A** is on **left** and **B** is on **right**, etc.) .
 - Clean squares stay clean and sucking cleans the current square.
 - The **Left** and **Right** actions move the agent left and right
 - The only available actions are **Left**, **Right**, and **Suck** .
 - The agent correctly perceives its location and whether it contains dirt.
- We claim that under these circumstances the agent is indeed rational

- The same agent would be **irrational** under **different circumstances**.
- For ex., once all the dirt is **cleaned up**, the agent will **oscillate needlessly back and forth**;
- if the **performance measure** includes a **penalty of one point** for each **movement** left or right, the agent will fare poorly.
- A **better** agent for this case would **do nothing** once it is sure that all the **squares are clean**.
- If clean squares can become **dirty again**, the agent should occasionally **check and re-clean** them if needed.
- If the **geography** of the environment is **unknown**, the **agent will need to explore it** rather than stick to squares A and B.

Omniscience, learning, and autonomy

- An **omniscient agent** knows the actual outcome of its actions and can act accordingly; but omniscience is impossible in reality.
- Rationality maximizes expected performance while perfection maximizes actual performance
- Definition of rationality does not require omniscience, then, because the rational choice depends only on the percept *sequence to date*.
- Doing actions in order to modify future percepts-sometimes called **information gathering**-is an important part of rationality.
- a rational agent requires not only to gather information, but also to **learn** as much as possible from what it perceives.

- **An agent is autonomous if its behavior is determined by its own experience** (with ability to learn and adapt)
- A rational agent should be **autonomous**-it should learn what it can to compensate for partial or incorrect prior knowledge.
- It would be reasonable to provide an AI agent with some initial knowledge as well as an ability to learn. After sufficient experience of its environment, the behavior of a rational agent can become effectively independent of its prior knowledge.
- Rational $\Rightarrow$ exploration, learning, autonomy

Task environments

- **Task environments**, which are essentially the "**problems**" to which rational agents are the "**solutions**."

Specifying the task environment

- The rationality of the simple agent, needs specification of
 - the performance measure
 - the environment
 - the agent's actuators
 - Sensors.

PEAS

- All these are grouped together under the heading of the **task environment**.
- We call this the **PEAS** (Performance, Environment, Actuators, Sensors) description.
- In designing an agent, the **first step must always be to specify the task environment as fully as possible**.

PEAS

How about a vacuum cleaner?



iRobot Roomba series

- **P**erformance: cleanness, efficiency: distance traveled to clean, battery life, security.
- **E**nvironment: room, table, wood floor, carpet, different obstacles.
- **A**ctuators: wheels, different brushes, vacuum extractor.
- **S**ensors: camera, dirt detection sensor, cliff sensor, bump sensors, infrared wall sensors.

PEAS Example: Autonomous taxi

- **Performance measure**
 - Safe, fast, legal, comfortable trip, maximize profits
- **Environment**
 - Roads, other traffic, pedestrians, customers
- **Actuators**
 - Steering wheel, accelerator, brake, signal, horn
- **Sensors**
 - Cameras, LIDAR, speedometer, GPS, odometer, engine sensors, keyboard



Another PEAS example: Spam filter

- **Performance measure**
 - Minimizing false positives, false negatives
- **Environment**
 - A user's email account, email server
- **Actuators**
 - Mark as spam, delete, etc.
- **Sensors**
 - Incoming messages, other information about user's account



PEAS for a medical diagnosis system

- **Performance measure:** Healthy patient, minimize costs, lawsuits
- **Environment:** Patient, hospital, staff
- **Actuators:** Screen display (questions, tests, diagnoses, treatments, referrals)
- **Sensors:** Keyboard (entry of symptoms, findings, patient's answers)



PEAS for a satellite image analysis system

- **Performance measure:** correct image categorization
- **Environment:** downlink from orbiting satellite
- **Actuators:** display categorization of scene
- **Sensors:** color pixel arrays

PEAS for a part-picking robot

- **Performance measure:** Percentage of parts in correct bins
- **Environment:** Conveyor belt with parts, bins
- **Actuators:** Jointed arm and hand
- **Sensors:** Camera, joint angle sensors



Rectangular Snip

PEAS for a refinery controller

- **Performance measure:** maximize purity, yield, safety
- **Environment:** refinery, operators
- **Actuators:** valves, pumps, heaters, displays
- **Sensors:** temperature, pressure, chemical sensors

- **PEAS description of the “online shopping agent”**
- We need to describe the PEAS for the “shopping for Data Warehousing books on the internet” activity.

- **Performance measures:**

Price of the book, Author of the book, Quality of the book, Book reviews on google.

Obtain interested/desired books, Cost minimization.

- **Environment:**

Internet websites, Web pages of a particular website,
Vendors/Sellers

Shippers

- **Actuators:**

Filling in the forms, Display to the user, Follow URL

- **Sensors:**

Keyboard entry, Browser used to find web pages, HTML

PEAS for Interactive English tutor

- **Performance measure:** Maximize student's score on test
- **Environment:** Set of students
- **Actuators:** Screen display (exercises, suggestions, corrections)
- **Sensors:** Keyboard

© 2004 Morgan Kaufmann Publishers, Inc.

Agent Type	Performance Measure	Environments	Actuators	Sensors
Taxi driver	Safe: fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, Signal, horn, display	Cameras, sonar, Speedometer, GPS, Odometer, engine sensors, keyboards, accelerometer

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, minimize costs, lawsuits	Patient, hospital, staff	Display questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display categorization of scene	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Maximize purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Maximize student's score on test	Set of students, testing agency	Display exercises, suggestions, corrections	Keyboard entry

Properties of task environments (types of task environment) (Environment types)

- 1. Fully observable vs. partially observable**
- 2. Deterministic vs. stochastic**
- 3. Episodic vs. sequential**
- 4. Static vs. dynamic**
- 5. Discrete vs. continuous**
- 6. Single agent vs. multi agent**

Fully observable vs. partially observable.

- If an agent's sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is fully observable.
- A task environment is effectively fully observable if the sensors detect all aspects that are *relevant* to the choice of action;
- An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data.
- Fully observable environments are convenient, because the agent need not maintain any internal state to keep track of the world
 - – Playing chess? Fully.
 - – Taxi driving? Partially.
- If the agent has no sensors at all then the environment is unobservable

For example , a **vacuum agent** with only a local dirt sensor cannot tell whether there is dirt in other squares, and an **automated taxi** cannot see what other drivers are thinking.

Deterministic vs. stochastic.

- If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is deterministic;
- Otherwise, it is stochastic.
- If the environment is deterministic except for the actions of other agents, then the environment is strategic
- In principle, an agent need not worry about uncertainty in a fully observable, deterministic environment
- **Taxi driving is stochastic**, because one can never predict the behavior of traffic exactly, and **vacuum world is deterministic**

Episodic vs. sequential

- In an **episodic task environment**, the agent's experience is divided into atomic episodes.
- Each episode consists of the agent perceiving and then performing a single action. The next episode does not depend on the actions taken in previous episodes.
- For example, an agent that has to **spot defective parts** on an assembly line bases each decision on the current part, regardless of previous decisions; moreover, the current decision doesn't affect whether the next part is defective.
- In **sequential environments**, on the other hand, the current decision Could affect all future decisions.
- **Chess and taxi driving are sequential** : in both cases, short-term actions can have long term consequences.
- Episodic environments are much simpler than sequential environments because the agent does not need to think ahead.

Discrete vs. continuous.

- The discrete/continuous distinction can be applied to the *state* of the environment, to the way *time* is handled, and to the *percepts* and *actions* of the agent.
- For example, a discrete-state environment such as a **chess game** has a finite number of distinct states.
- Chess also has a discrete set of percepts and actions.
- **Taxi driving is a continuous-** state and continuous-time problem:
- The speed and location of the taxi and of the other vehicles sweep through a range of continuous values and do so smoothly over time.
- Taxi-driving actions are also continuous (steering angles, etc.)



Single agent vs. multiagent.

- if a single agent is present in the environment then it is Single agent environment, if more than one agent present then it is multiagent.
- An agent solving a crossword puzzle by itself is clearly in a single-agent environment,
- Where as an agent playing chess is in a two-agent environment.
- Multiagent is further classified in to two ways
 - **Competitive multiagent environment**
 - **Cooperative multiagent environment**
- Chess is a Competitive multiagent environment
- Taxi driving environment is a Cooperative multiagent environment

Static Vs dynamic

If the environment can change while an agent is thinking, then we say the environment is dynamic for that agent; otherwise it is static.

- Static environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action.
- Dynamic environments, on the other hand, are continuously asking the agent what it wants to do
- If the environment itself does not change with the passage of time but the agent's performance score does, then we say the environment is **semidynamic**.
- **Taxi driving : dynamic**: the other cars and taxi itself keep moving while the driving algorithm dithers about what to do next
- Chess when played with a clock is semi dynamic
- **Crossword puzzles are static**



Chess with
a clock



Scrabble



Autonomous
driving

Task Environment	Observable	Deterministic /Stochastic	Episodic/ Sequential	Static/Dy namic	Discrete/Co ntinuous	Agents
Crossword puzzle	Fully	Deterministic	Sequential	Static	Discrete	Single
Chess with a clock	Fully	Strategic	Sequential	Semi	Discrete	Multi
Poker	Partially	Strategic	Sequential	Static	Discrete	Multi
Backgammon	Fully	Stochastic	Sequential	Static	Discrete	Multi
Taxi driving	Partially	Stochastic	Sequential	Dynamic	Continuous	multi
Medical dignosis	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Refinery controller	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Interactive English tutor	Partially	Stochastic	Sequential	Dynamic	Discrete	Multi
Image analysis	Fully	Deterministic	Episodic	Semi	Continuous	Single
Part picking robot	Partially	Stochastic	Episodic	Dynamic	Continuous	Single

THE STRUCTURE OF AGENTS(Agent Programs)

- The *job of AI* is to *design* an **agent program** that implements the agent function that maps percepts to actions.
- this program will run on some sort of computing device with physical sensors and actuators—we call this the **architecture**:
- *agent = architecture + program*
- the program we choose has to be appropriate for the architecture.
- Ex. If the program is going to recommend actions like *Walk*, the architecture should better have legs.
- The *architecture might be* just an *ordinary PC*, or it might be *a robotic car* with several *onboard computers, cameras, and other sensors*.
- In general, the architecture makes the percepts from the sensors available to the program, runs the program, and feeds the program's action choices to the actuators as they are generated.

- Agent program that we design, take the **current percept as input** from the sensors and **return an action** to the actuators
- agent program takes the current percept as input where as the agent function takes the entire percept history.
- A trivial agent program that keeps track of the percept sequence and then uses it to index into a table of actions to decide what to do.(Table driven agent) is as follows
- **persistent** :continuing to do something

```
function TABLE-DRIVEN-AGENT(percept ) returns an action
persistent: percepts, a sequence, initially empty
    table, a table of actions, indexed by percept sequences, initially fully
    specified
append percept to the end of percepts
action ← LOOKUP(percepts, table)
return action
```

- To build a rational agent in this way, we as designers must construct a table that contains the appropriate action for every possible percept sequence.
- Table driven approach to agent construction **leads to failure**
- Let P be the set of possible percepts and let T be the lifetime of the agent then The lookup table will contain

$$\sum_{t=1}^T |\mathcal{P}|^t \quad \text{entries}$$

- Consider the automated taxi: the visual input from a single camera comes in at the rate of roughly 27 megabytes per second (30 frames per second, 640×480 pixels with 24 bits of color information)
- This gives a lookup table for automated taxi with over 10 250,000,000,000 entries for one hour of driving
- lookup table for chess would have at least 10^{150} entries.

- Drawbacks:
 - Huge table (P^T , P : set of possible percepts, T : lifetime)
 - Space to store the table
 - Take a long time to build the table
 - No autonomy
 - Even with learning, need a long time to learn the table entries

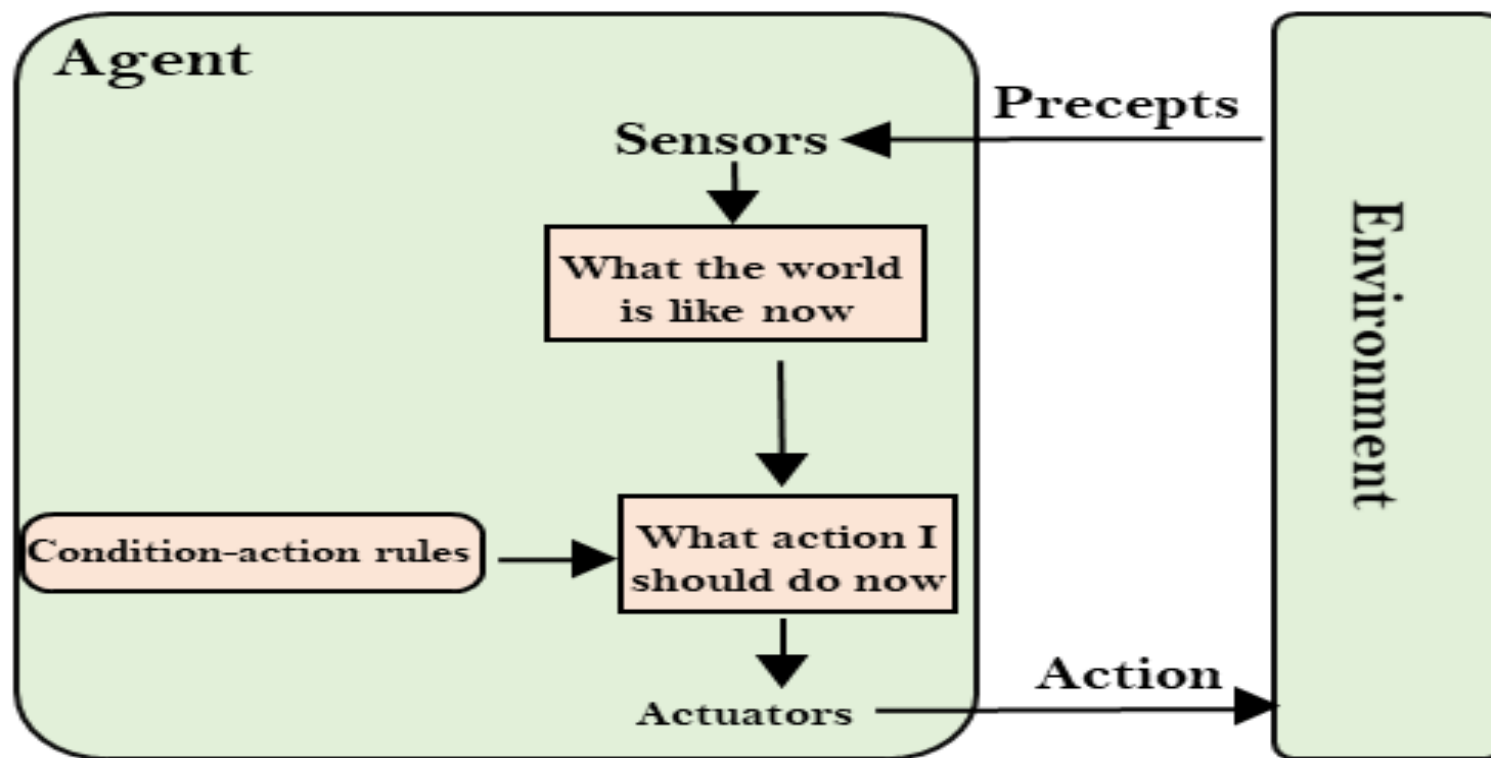
Types of agent programs

- Simple reflex agents;
- Model-based reflex agents;
- Goal-based agents;
- Utility-based agents.
- Learning agents

Simple Reflex Agent

- The simplest kind of agent is the **simple reflex agent**.
- These agents select actions on the basis of the **current percept**, ignoring the rest of the percept history .
- These agents only succeed in the fully observable environment.
- For example, the vacuum agent whose agent function is tabulated is given below,
- a simple reflex agent, because its decision is based only on the current location and on whether that contains dirt.
- Select action on the basis of **only the current** percept. E.g. the vacuum-agent
- Simple reflex agent is implemented through **condition-action rules** which means it maps the current state to action
- Example :If dirty then suck
- Example: if car-in-front-is-breaking then initiate-breaking

Percept Sequence	Action
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
.....	

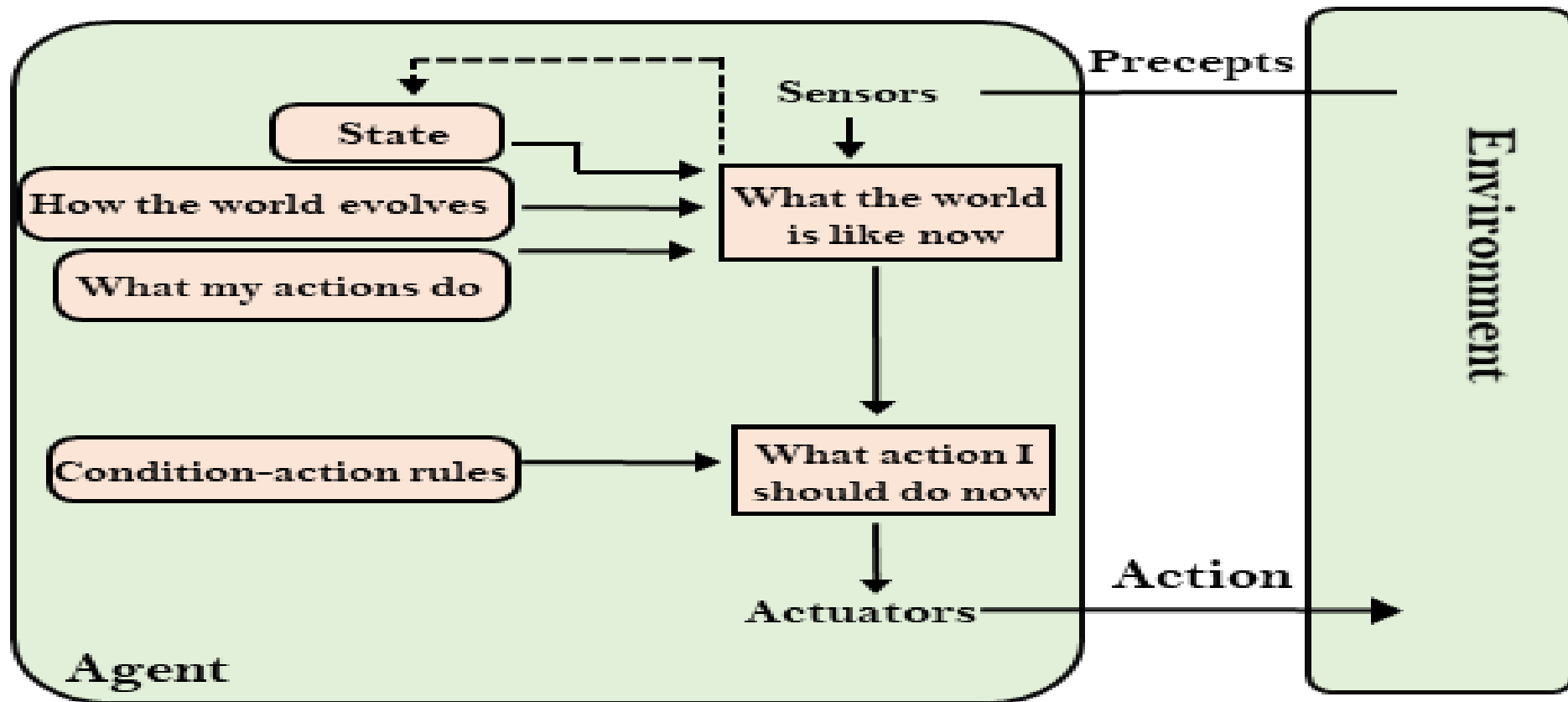


function SIMPLE-REFLEX-AGENT(*percept*) **returns** an action
persistent: *rules*, a set of condition-action rules

state $\leftarrow$ INTERPRET-INPUT(*percept*)
rule $\leftarrow$ RULE-MATCH(*state*, *rules*)
action $\leftarrow$ *rule*.ACTION
return *action*

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:
 - How the world evolves because of other agents.
 - How the agent's action affects the world.



function MODEL-BASED-REFLEX-AGENT(*percept*) **returns** an action

persistent: *state*, the agent's current conception of the world state

model, a description of how the next state depends on current state and action

rules, a set of condition–action rules

action, the most recent action, initially none

state $\leftarrow$ UPDATE-STATE(*state*, *action*, *percept*, *model*)

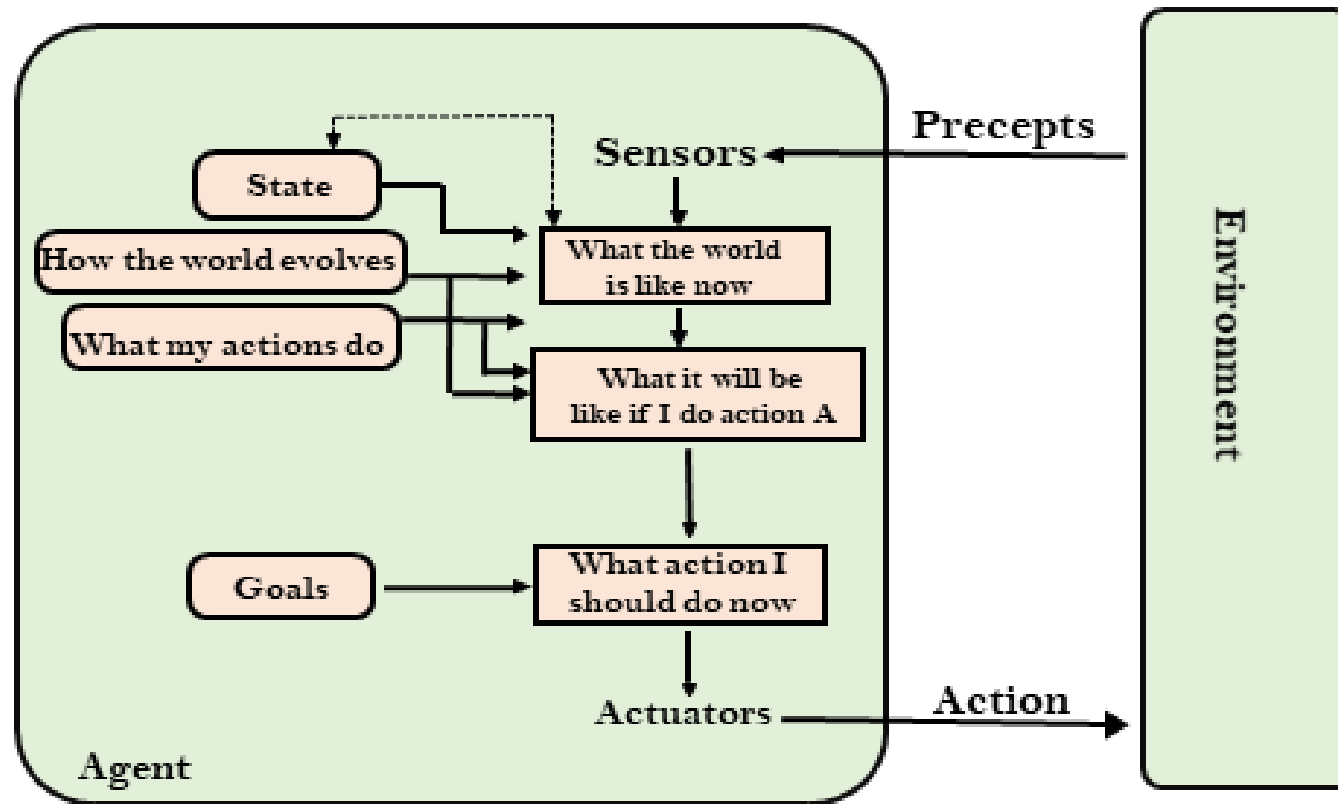
rule $\leftarrow$ RULE-MATCH(*state*, *rules*)

action $\leftarrow$ *rule*.ACTION

return *action*

Goal-based agents

- The knowledge of the current state of 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**



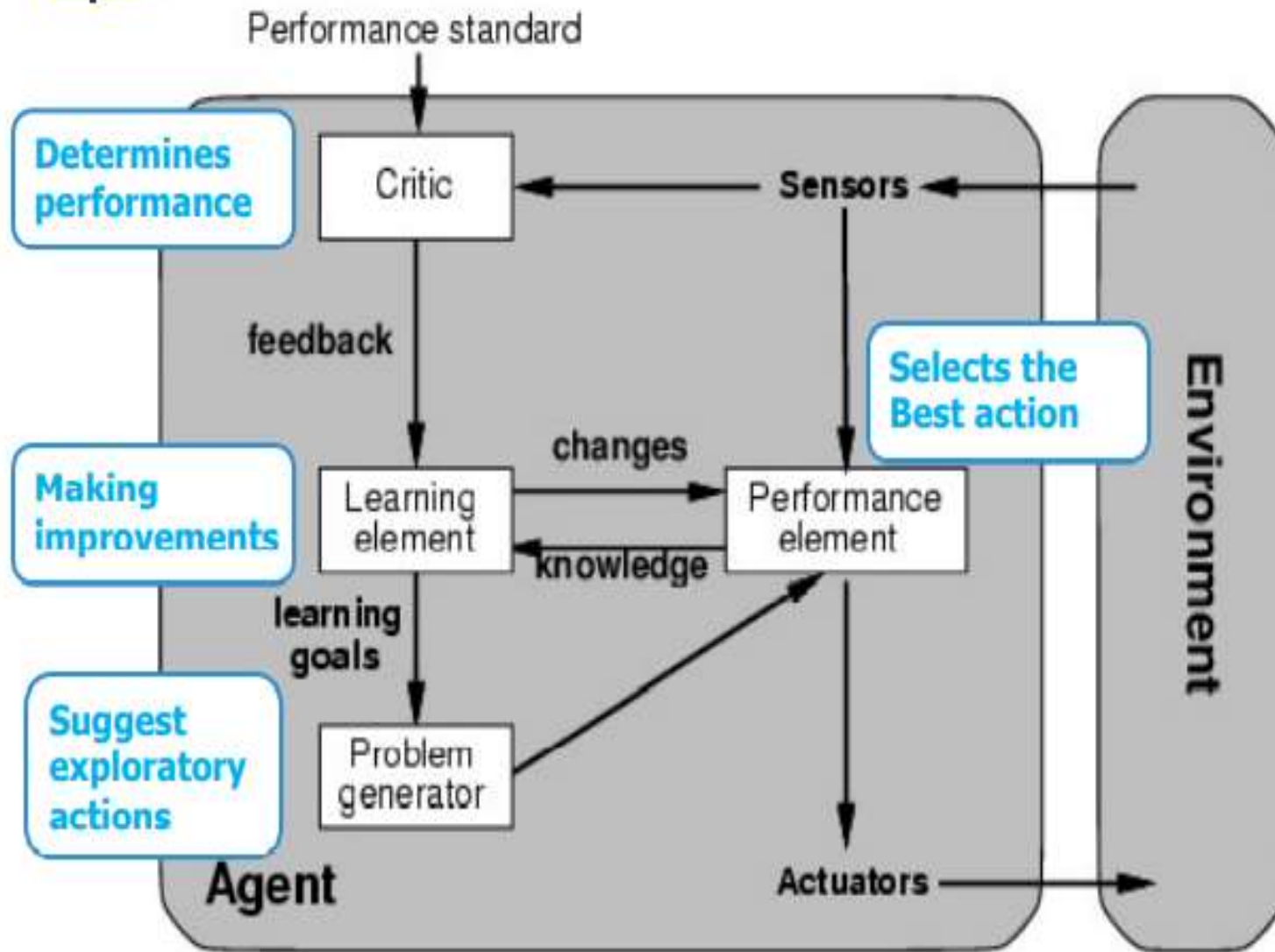
- Reflex agent breaks when it sees brake lights. Goal based agent reasons
 - Brake light -> car in front is stopping -> I should stop -> I should use brake

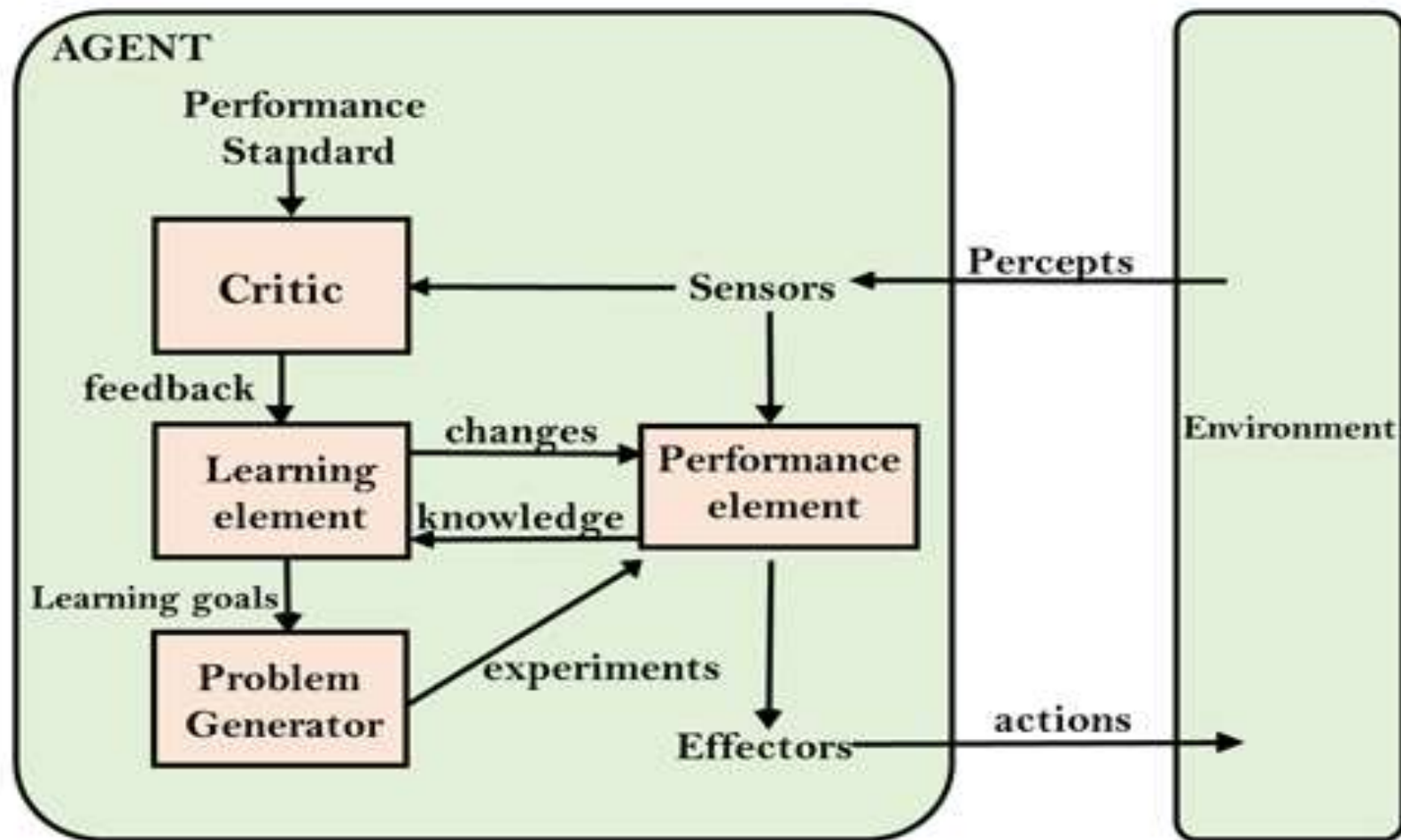
Utility-based agents

- 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.

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:
 - **Learning element:** It is responsible for making improvements by learning from environment
 - **Critic:** Learning element takes feedback from critic which describes that how well the agent is doing with respect to a fixed performance standard.
 - **Performance element:** It is responsible for selecting external action
 - **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.

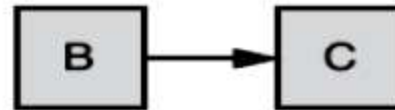




How the components of agent programs work

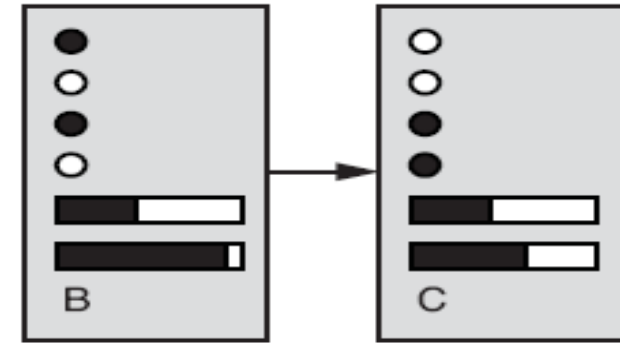
- agent programs consisting of various components whose function is to answer questions such as:
 - “What is the world like now?”
 - “What action should I do now?”
 - “What do my actions do?”
- There are three ways components can represent environment states and the transitions between them.
 - **atomic**
 - **factored**
 - **structured**

- In an **atomic representation** each state of the world is indivisible, a block box that has no internal structure.
- Consider the problem of finding a driving route from one end of a country to the other via some sequence of cities.
- For the purposes of solving this problem, state is only the name of the city – a single atom of knowledge
- The algorithms underlying **search** and **game playing** , work with atomic representation



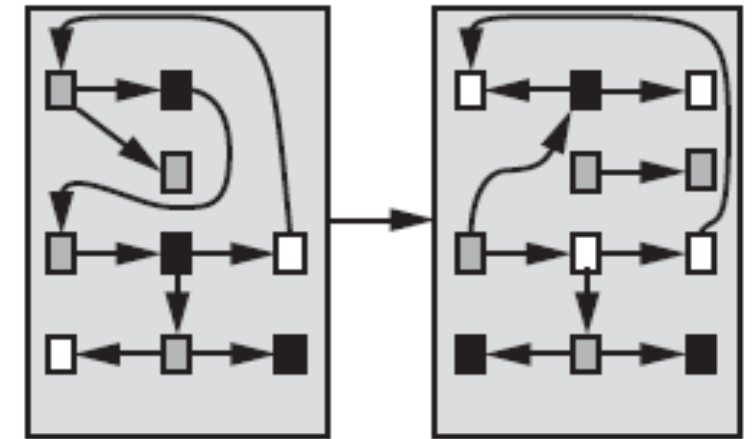
(a) Atomic

- A **factored representation** splits up each state into a (vector) fixed set of **variables** or **attributes**, each of which can have a **value**.
- values can be Boolean, real valued, or one of a fixed set of symbols.
- When we need to be concerned with more than just atomic location in one city (in previous problem), we might need to consider how much gas/fuel is in the tank, our current GPS coordinates, what station is on the radio, and so on
- two different factored states can share some attributes and not others.
- this makes it much easier to work out how to turn one state into another.
- With factored representations, we can also represent **uncertainty**
- Many important areas of AI are based on factored representations, including **constraint satisfaction** algorithms , **propositional logic** , **planning** and the **machine learning** algorithms.



(b) Factored

- In a **structured representation**, we need to understand the world as having *things* in it that are *related* to each other, not just variables with values.
- In a **structured representation**, objects and their various and varying relationships can be described explicitly.
- Structured representations underlie **relational databases** and **first-order logic**, **first-order probability**, **knowledge-based learning** and much of **natural language understanding**
- The expressiveness increases from atomic representation to structured representation.
- More expressive representation can capture more about the environment than less expressive.
- reasoning and learning become more complex as the expressive power of the representation increases



(c) Structured

Problem-solving agents (PSA)

- It's a goal-based agent
- It decides what to do by finding sequences of actions that lead to desirable states.
- Intelligent agents are supposed to adopt a **goal** and aim to satisfy so as to **maximize** the performance measure.
- **Goal formulation** is the first step in problem solving
- **Goal formulation** based on
 - the current situation and
 - the agent's performance measure
- The agent has to find out which sequence of actions will get to a goal state.
- Agent needs to decide what sorts of actions and states are to be considered.

- **Problem formulation** is the process of deciding what actions and states to consider, given a goal
- Traveling agent - Unless the agent is very familiar and has the **previous knowledge** of the geography of destination, it cannot make proper decision to reach destination.
- Therefore, we need to **search** for the possible sequences?
- The process of examining different possible sequences of actions that reaches the goal is called **Search**.
- **A search algorithm** takes a problem as input and returns a solution in the form of an action sequence.
- Once a solution is found, then the recommended actions can be carried out , this phase is known as **execution**

- The complete process of an agent to solve the problems can be shown as “**formulate, search, execute**” design for the agent.
- A simple problem solving agent
 - **Formulate** a goal and a problem
 - **Search** for a sequence of actions to solve the problem
 - **Execute** the actions one at a time
 - When complete, formulate another goal and continue
 - Note that while executing the sequence, the agent ignores its percepts assuming that the solution it has will always work

function SIMPLE-PROBLEM-SOLVING-AGENT(*percept*) **returns** an action

persistent: *seq*, an action sequence, initially empty

state, some description of the current world state

goal, a goal, initially null

problem, a problem formulation

state $\leftarrow$ UPDATE-STATE(*state*, *percept*)

if *seq* is empty **then**

goal $\leftarrow$ FORMULATE-GOAL(*state*)

problem $\leftarrow$ FORMULATE-PROBLEM(*state*, *goal*)

seq $\leftarrow$ SEARCH(*problem*)

if *seq* = *failure* **then return** a null action

action $\leftarrow$ FIRST(*seq*)

seq $\leftarrow$ REST(*seq*)

return *action*

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

Problem formulation(Problem Definition)

- A problem can be defined formally by five components:

1. Initial state ,the agent starts in.

2. Actions

A description of the possible actions available to the agent

- Given a particular state **S**, ACTIONS (s) returns a set of actions that can be executed in S.
- We say that each of these actions is **applicable** in **S**.

3. Transition Model:

- A description of what each action does.
- Transition model, specified by a function RESULT(s, a) that returns the state that results from doing action a in state s.
- Also use the term **successor** to refer to any state reachable from a given state by a single action.

- **STATE SPACE:** Together, the initial state, actions, and transition model define the state space of the problem that is the set of all states reachable from the initial state by any sequence of actions.
- The state space forms a directed network or **graph** in which the nodes are states and the links between nodes are actions.
- A **path** in the state space is a sequence of states connected by a sequence of actions.

4. **Goal Test**, determines whether a given state is a goal state

- **Explicit** Among the set of possible goal state(s), check whether the given state is one of them
- **implicit**, Goal is specified by an abstract property , for ex. in chess, the goal is to reach a state called “checkmate,” where the opponent’s king is under attack and can’t escape.

5. Path cost : A function that assigns a numeric cost to each path

- PSA should choose a cost function that reflects its own performance measure
- **The step cost** of taking action a to go from state s to state s' is denoted as $c(s, a, s')$, assumed to be ≥ 0
- Five elements define a problem and can be represented as a single data structure that is given as **input** to a problem solving algorithm
- A **solution** to a problem is a path (sequence of actions) from the initial state to a goal state
- Solution quality is measured by the **path cost function**
- An **optimal solution** has the lowest path cost among all solutions

EXAMPLE PROBLEMS

- A **TOY PROBLEM** is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is usable by different researchers to compare the performance of algorithms
- A **REAL-WORLD PROBLEM** is one whose solutions people actually care about. It cannot have a single agreed-upon description, but we can give the general description.

vacuum cleaning world

This can be formulated as a problem as follows:

- **States:** The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt.

Thus, there are $2 \times 2^2 = 8$ possible world states.

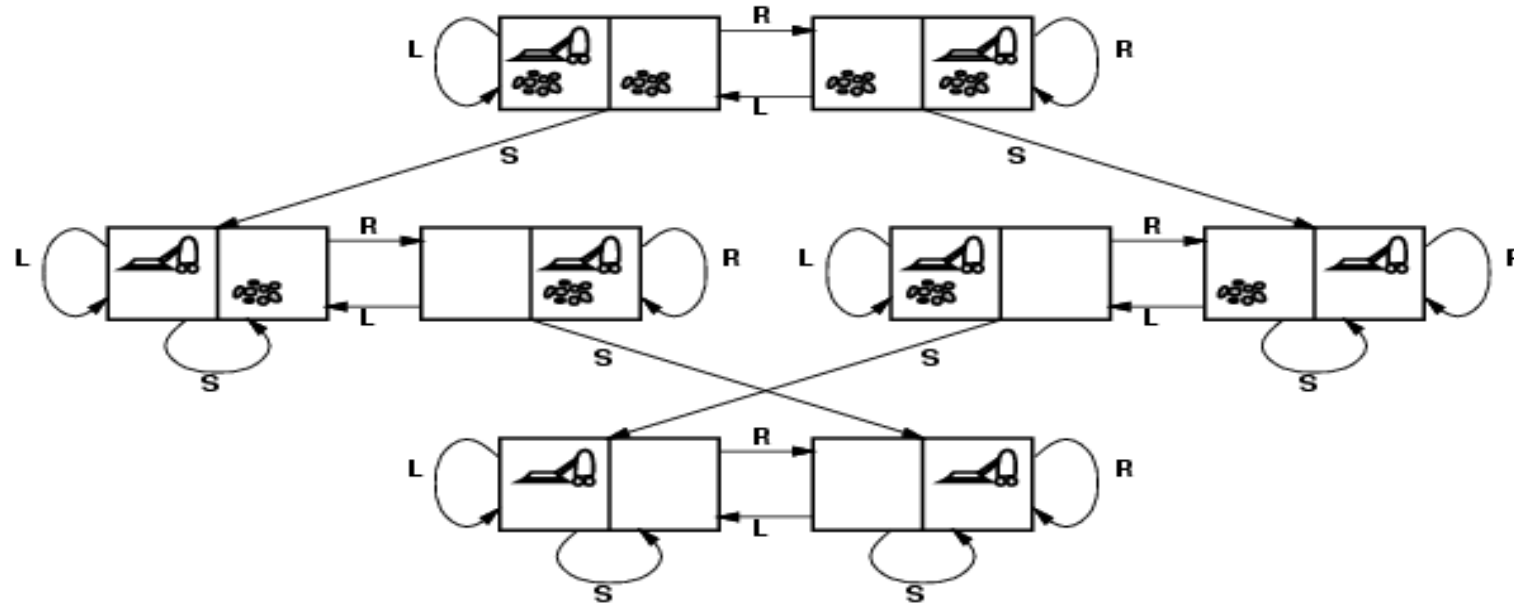
A larger environment with n locations has $n \cdot 2^n$ states.

- **Initial state:** Any state can be designated as the initial state.
- **Actions:** In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*.

Larger environments might also include *Up* and *Down*.

- **Transition model:** The actions have their expected effects, except that moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Sucking* in a clean square have no effect.
- **Goal test:** This checks whether all the squares are clean.
- **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

Vacuum world state space graph - Summary



- Arcs denote actions { L, R, S }
- states? dirt and agent location
- actions? *Left, Right, Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

8-puzzle

- consists of a 3×3 board with eight numbered tiles and a blank space.
- A tile adjacent to the blank space can slide into the space.

7	2	4
5		6
8	3	1

Start State

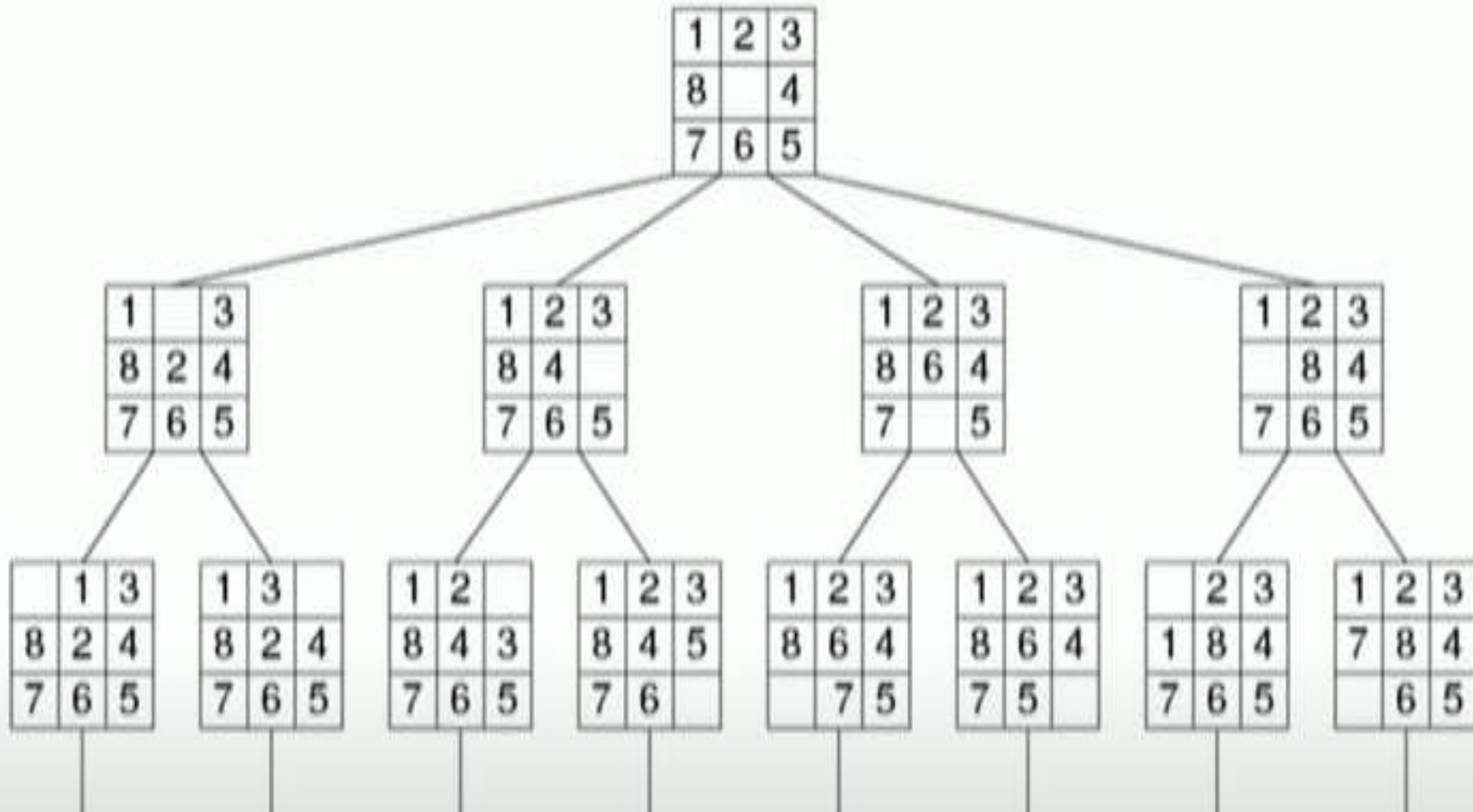
	1	2
3	4	5
6	7	8

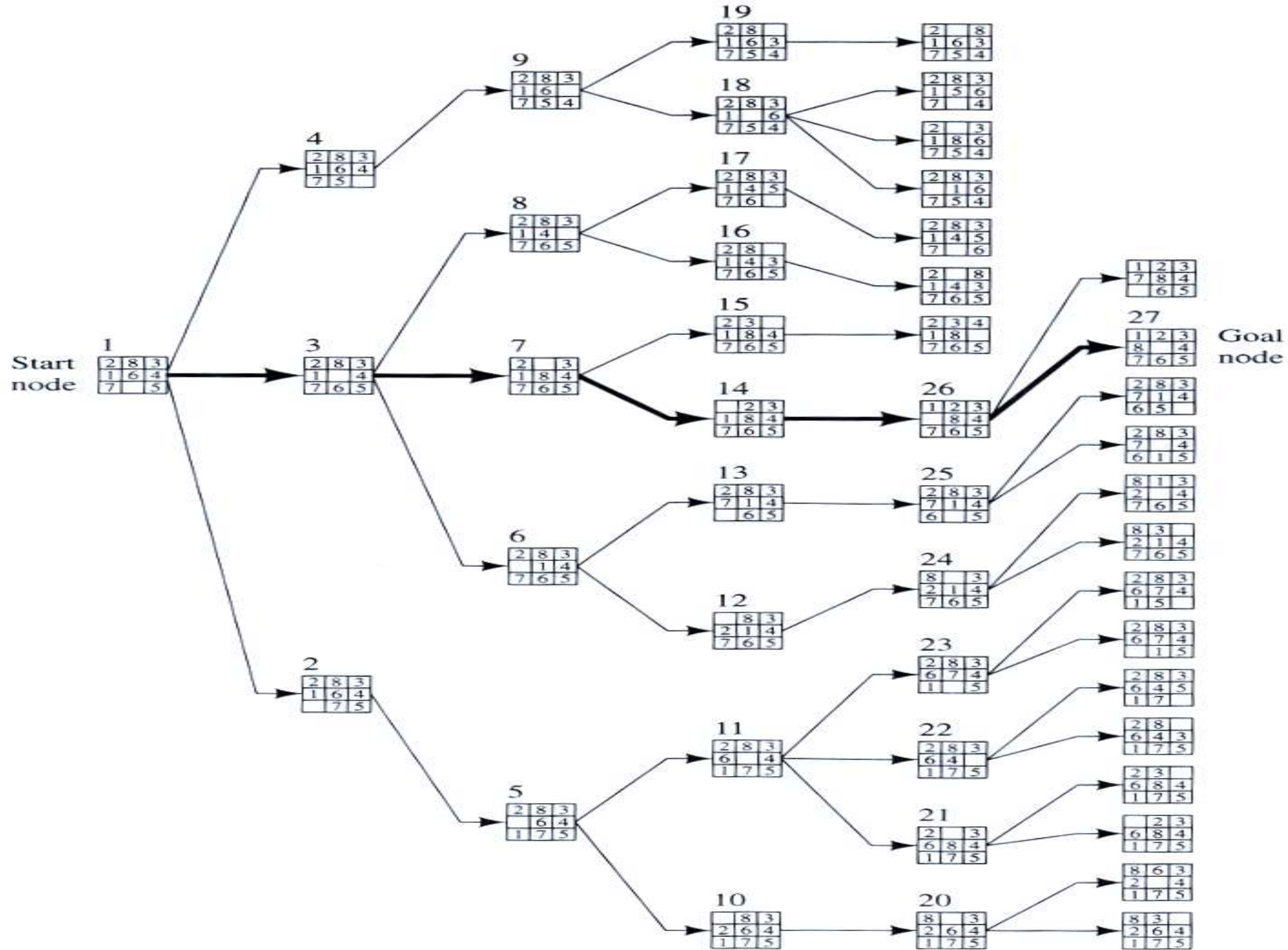
Goal State

- **States:** A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
 - **Initial state:** Any state can be designated as the initial state.
 - **Actions:** The simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or *Down*.

- **Transition model:** Given a state and action, this returns the resulting state; for example, if we apply *Left* to the start state in Figure, the resulting state has the 5 and the blank switched.
- **Goal test:** This checks whether the state matches the goal configuration shown in Figure
- **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

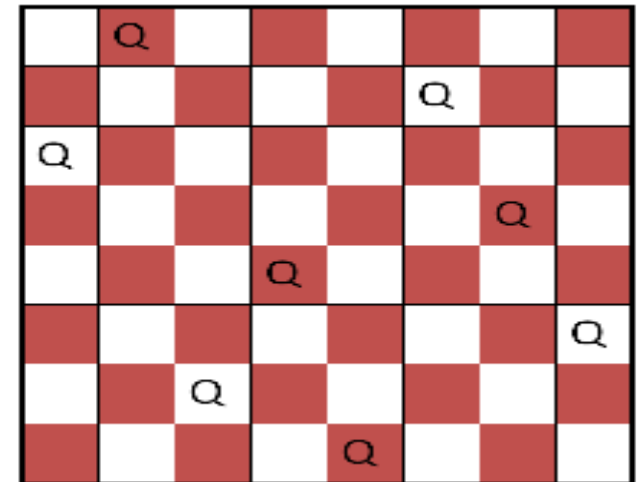
Search Tree Example: Fragment of 8-Puzzle Problem Space





8-queens problem

- The goal of the **8-queens problem** is to place eight queens on a chessboard such that no queen attacks any other.
- A queen attacks any piece in the same row, col or diagonal
- Efficient algorithms exist for whole n-queens family
- Still, an interesting test problem for search algorithms



- There are two main kinds of formulation

Incremental formulation

- Starting with an empty state
- For 8-queen problem, each action adds a queen to the state

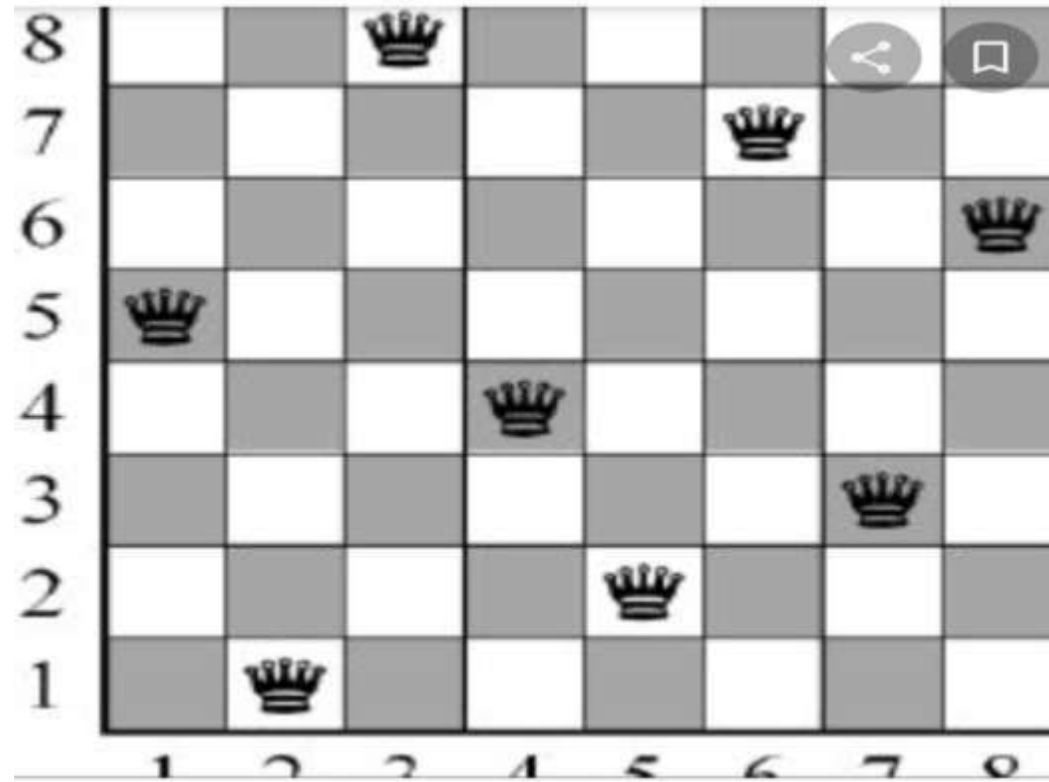
Complete state formulation

- Starts with all 8 queens on the board
- Moves them around
- In both cases, no path cost because only the final state counts!

The first incremental formulation one might try is the following:

- **States:** Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state:** No queens on the board.
- **Actions:** Add a queen to any empty square.
- **Transition model:** Returns the board with a queen added to the specified square.
- **Goal test:** 8 queens are on the board, none attacked.

We have 1.8×10^{14} possible sequences to investigate.



- A better formulation would prohibit placing a queen in any square that is already attacked:
- **States:**
 - Arrangements of n queens ($n = 1$ to 8), one per column in the leftmost n columns, with no queen attacking another are states
- **Actions:**
 - Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.
- 2057 sequences to investigate

- Our final toy problem was devised by Donald Knuth (1964) and illustrates how infinite state spaces can arise.
- starting with the number 4, a sequence of factorial, square root, and floor operations will reach any desired positive integer. For example, we can reach 5 from 4 as follows:

$$\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \rfloor = 5 .$$

- The problem definition is very simple:
- **States:** Positive numbers.
- **Initial state:** 4.
- **Actions:** Apply factorial, square root, or floor operation
- **Transition model:** As given by the mathematical definitions of the operations.
- **Goal test:** State is the desired positive integer.

- To our knowledge there is no bound on how large a number might be constructed in the process of reaching a given target.
- for example, the number 620,448,401,733,239,439,360,000 is generated in the expression for 5
- so the state space for this problem is infinite.

Real-world Problems

- Route finding
- Touring problems (TSP)
- VLSI layout
- Robot Navigation
- Automatic assembly sequencing
- Protein design

Route-finding

The airline travel problems solved by a travel-planning Web site:

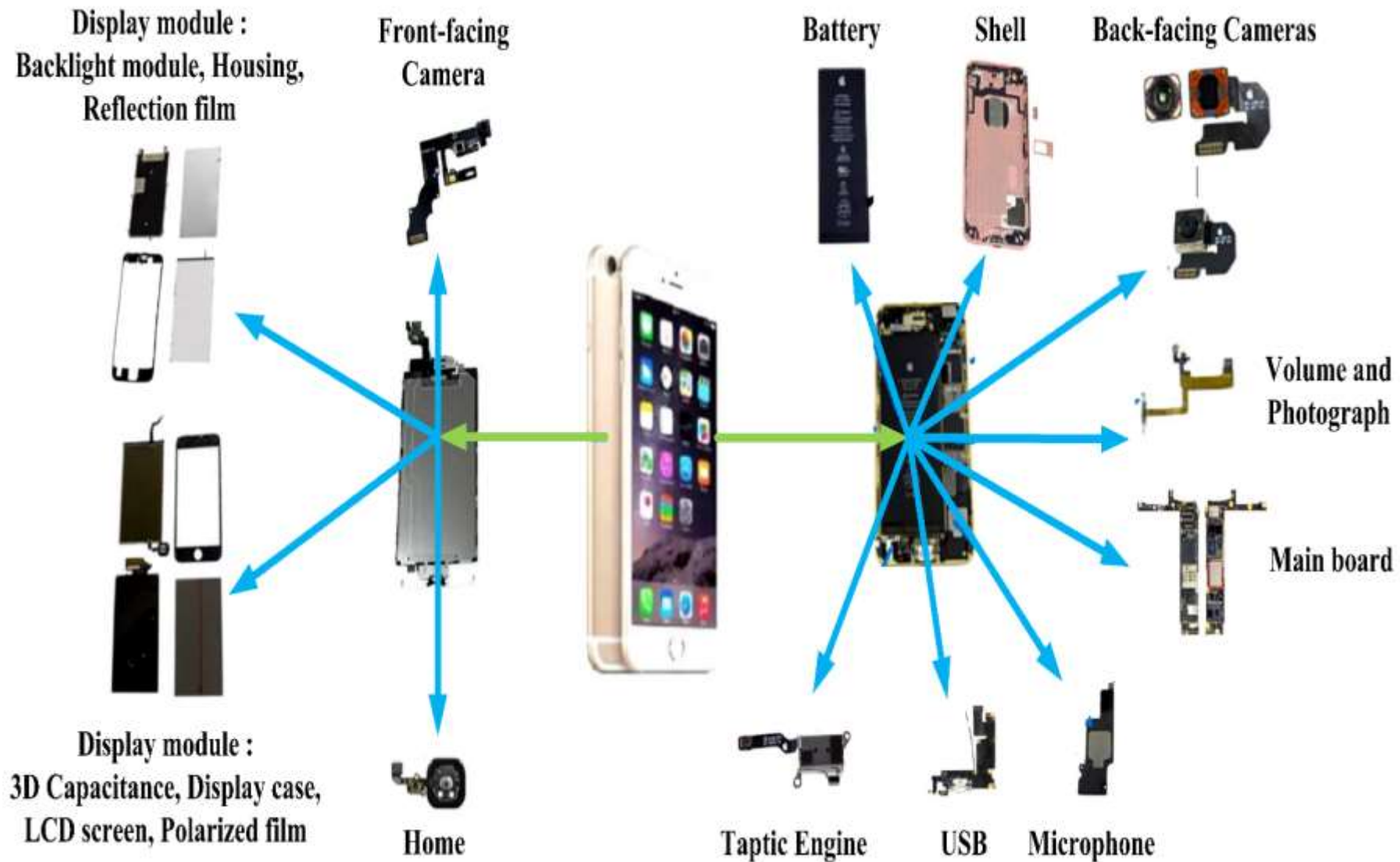
- **States:** Each state includes a location (e.g., an airport) and the current time. Since the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about "historical" aspects.
- **Initial state:** This is specified by the user's query.
- **Actions:** any flight from the current location, in a given seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.

- **Transition model:** The state resulting from taking a flight will have the flight's destination as the current location and the flight's arrival time as the current time.
- **Goal test:** Are we at the final destination specified by the user?
- **Path cost:** This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, etc.

- **Touring problems** are closely related to route-finding problems, but with an important difference. The starting city and destination city is the same and has to visit every city at least once.
- **Traveling salesperson problem** (TSP) is a touring problem in which each city must be visited exactly once. The aim is to find the *shortest* tour.
- A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield.
- usually split into two parts: **cell layout** and **channel routing**.

- In **cell layout**, the primitive components of the circuit are grouped into cells, each of which performs some recognized function.
- The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells.
- **Channel routing** finds a specific route for each wire through the gaps between the cells. These search problems are extremely complex.
- **Robot navigation** is a generalization of the route-finding problem
- Rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states.
- For a circular robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs or wheels that must also be controlled, the search space becomes many dimensional.

- **Automatic assembly sequencing** of complex objects by a robot is to find an order in which to assemble the parts of some object.
- If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done.
- Checking a step in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation.
- Thus, the generation of legal actions is the expensive part of assembly sequencing. Any practical algorithm must avoid exploring all but a tiny fraction of the state space.
- Another important assembly problem is **protein design**, in which the goal is to find a sequence of amino acids that will fold into a three-dimensional protein with the right properties to cure some disease.



Unit-2

Artificial Intelligence

Dr. Munipraveena Rela, M.Tech., Ph.D.,

- **UNIT 2: SEARCHING (9)**
- Searching for solutions, uniformed search strategies – Breadth first search, depth first Search. Search with partial information (Heuristic search) Hill climbing, A* ,AO* Algorithms, Problem reduction, Game Playing-Adversial search, Games, mini-max algorithm, optimal decisions in multiplayer games, Problem in Game playing, Alpha-Beta pruning, Evaluation functions.

✓ **Searching for solution :**

- ✓ After formulating the problem we need to find the solution. This can be done by search through state space.
- ✓ The searching techniques uses search tree that is generated by initial state and successor function.
 - ✓ The searching begins at the root of the search tree.
 - ✓ Check whether it is a goal state, if so return SUCCESS.
 - ✓ If not, expand the state by applying successor function.
 - ✓ We continue choosing, testing and expanding until either the goal state is reached or there are no more nodes to be expanded.

General TREE_SEARCH_ALGORITHM

```
function TREE-SEARCH(problem) returns a solution, or failure
{
  initialize the search tree using the initial state of problem
  loop do
  {
    if there are no states for expansion then return failure
    choose a node for expansion according to strategy
    if the selected node is a goal state then return the
        corresponding solution
    else
      expand the chosen node, adding the resulting nodes to the
        search tree
  }
}
```


General GRAPH_SEARCH_ALGORITHM

function GRAPH-SEARCH(problem) **returns** a solution, or failure

initialize the frontier using the initial state of problem

initialize the explored set to be empty

loop do

if the frontier is empty **then return** failure

choose a leaf node and remove it from the frontier

if the node contains a goal state **then return** the corresponding solution

add the node to the explored set

expand the chosen node, adding the resulting nodes to the frontier

only if not in the frontier or explored set

Measuring problem-solving performance

- The evaluation of a search strategy
 - **Completeness:**
 - is the strategy guaranteed to find a solution when there is one?
 - **Optimality:**
 - does the strategy find the highest-quality solution when there are several different solutions?
 - **Time complexity:**
 - how long does it take to find a solution?
 - **Space complexity:**
 - how much memory is needed to perform the search?

- In AI, complexity is expressed in
 - **b**, branching factor, maximum number of successors of any node
 - **d**, the depth of the shallowest goal node.
(depth of the least-cost solution)
 - **m**, the maximum length of any path in the state space
- Time is measured in number of nodes generated during the search
- Space measured in terms of maximum number of nodes stored in memory

- For effectiveness of a search algorithm
 - we can just consider the total cost
 - **The total cost** = path cost (g) of the solution found + search cost
 - search cost = time necessary to find the solution
- Tradeoff:
 - (long time, optimal solution with least g)
 - vs. (shorter time, solution with slightly larger path cost g)

✓ **Measuring the solutions performance**

- ✓ Completeness: guarantees the solution
 - ✓ Optimality: minimum path cost
 - ✓ Time Complexity: time required
 - ✓ Space Complexity: space required
- } based on
no. of
nodes generated

✓ Search Strategies or Search Algorithm

- ✓ It determines the node which is to be expanded
 - ✓ Search a node
 - ✓ Check Goal state → return success
 - ✓ Otherwise generate all its successor

- ✓ **There are two types of search strategies:**
 - ✓ Uninformed Search Strategy
 - ✓ The Search is carried out without any additional information.
 - ✓ Informed Search Strategy
 - ✓ The Search is carried out by using additional information to determine the next step towards finding the solution.

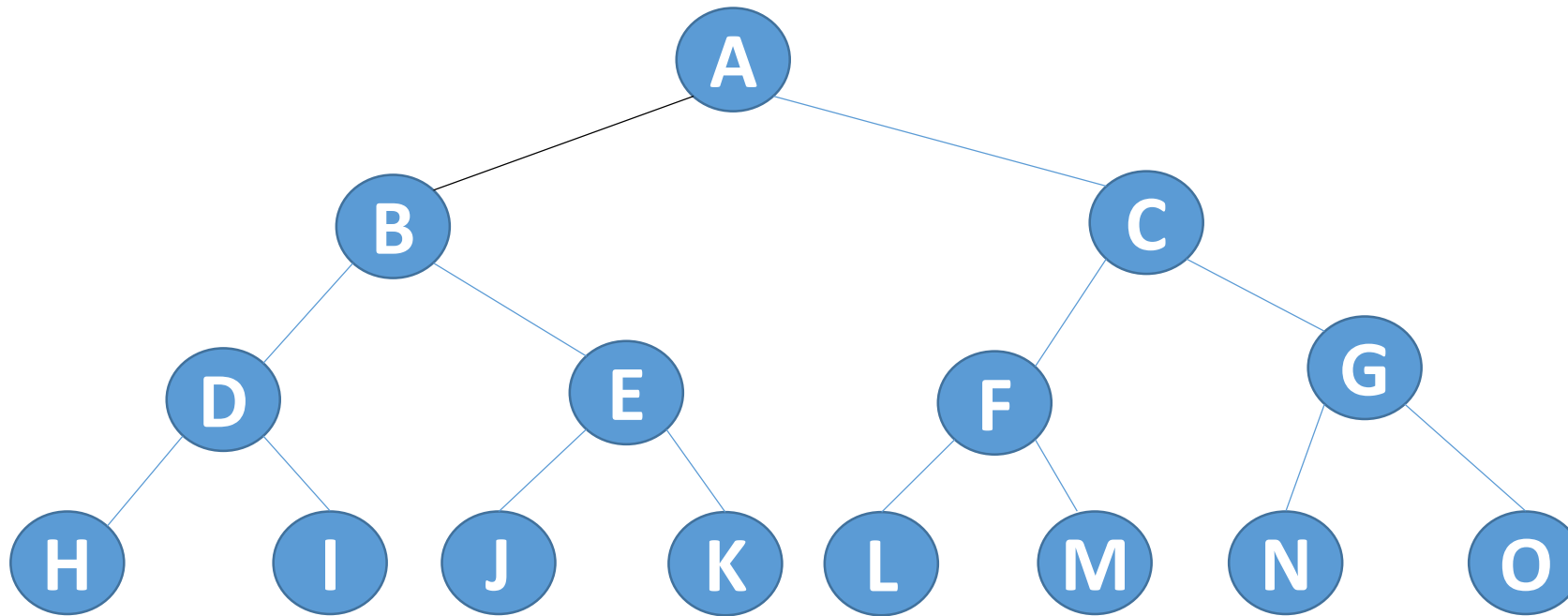
Uninformed search strategies

- **Uninformed**: While searching you have no clue whether one non-goal state is better than any other. This search is also called as blind Search.
- have no additional information about states beyond that provided in the problem definition
- **Various uninformed strategies:**
 1. Breadth-first search
 2. Uniform-cost search
 3. Depth-first search
 4. Depth-limited search
 5. Iterative deepening search
 6. Bidirectional Search

BREADTH FIRST SEARCH (BFS)

- Breadth-first search in which the root node is expanded first, then all successors of the root node are expanded next, then their successors, and so on.
- In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- Breadth-first-search is implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out(FIFO) queue, assuring that the nodes that are visited first will be expanded first.
- In otherwards, calling TREE-SEARCH (problem, FIFO-QUEUE()) results in breadth-first-search.
- The FIFO queue puts all newly generated successors at the end of the queue, which means that Shallow nodes are expanded before deeper nodes.

Consider the following binary tree:



Assume G is Goal State



Marker Node to be expended

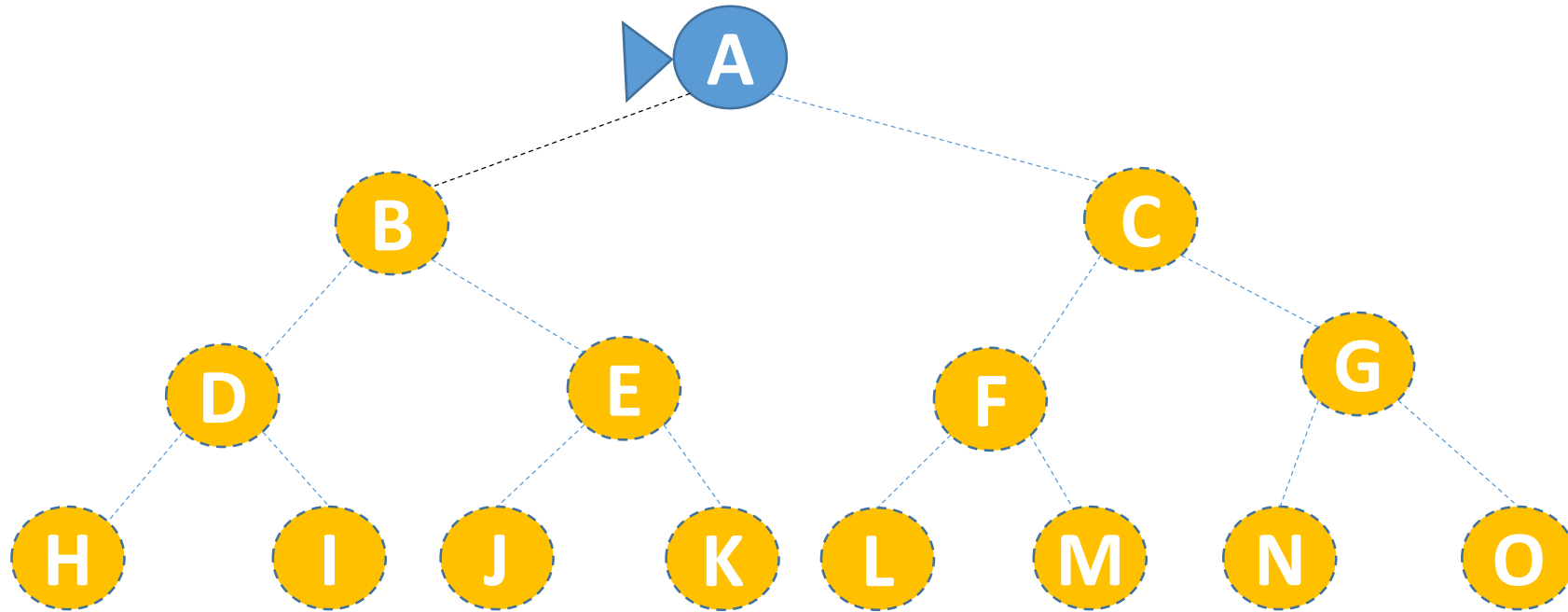


Solid circle - Visited



Dashed circle – not visited

Step 1



Marker Node to be expanded



Solid circle - Visited



Dashed circle – not visited

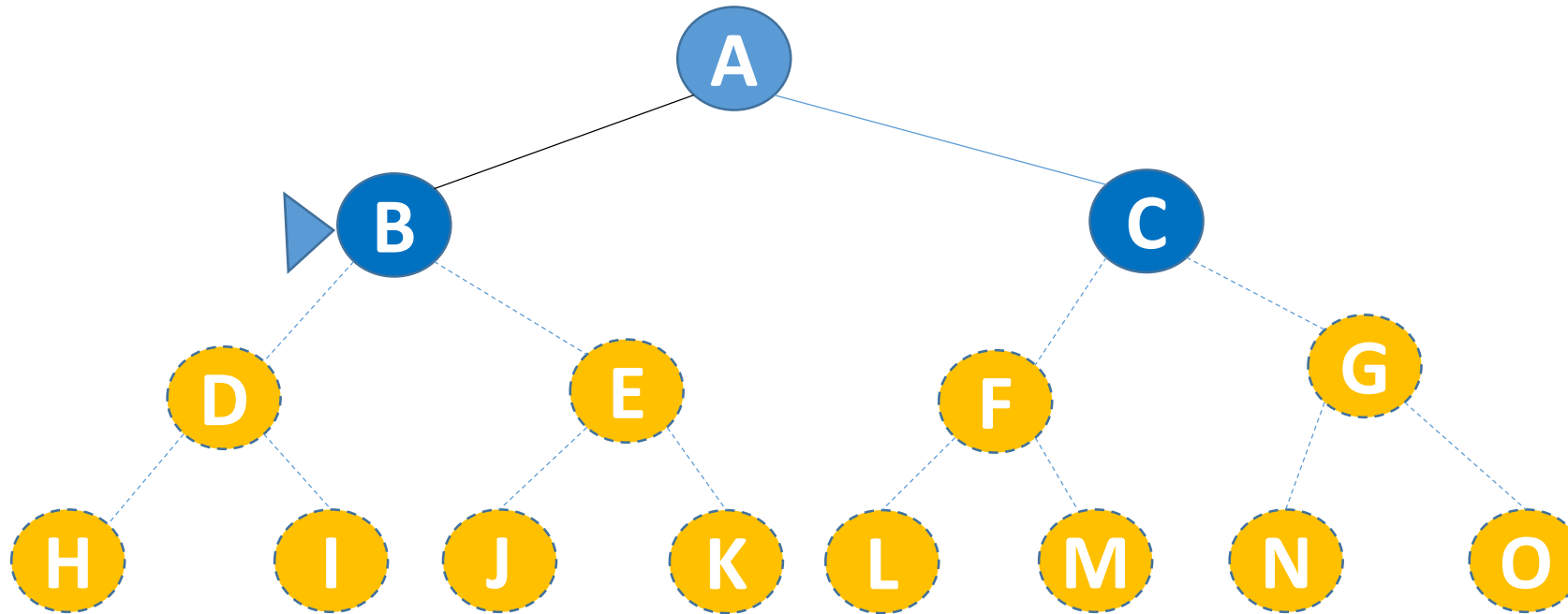
Queue {A}

Check whether A is a goal State

Else

Expand A -> add all its successors at the end of the Queue

Step 2



Queue {B,C}

Check whether B is a goal State

Else

Expand B - > add all its successors at the end of the Queue



Marker Node to be expanded

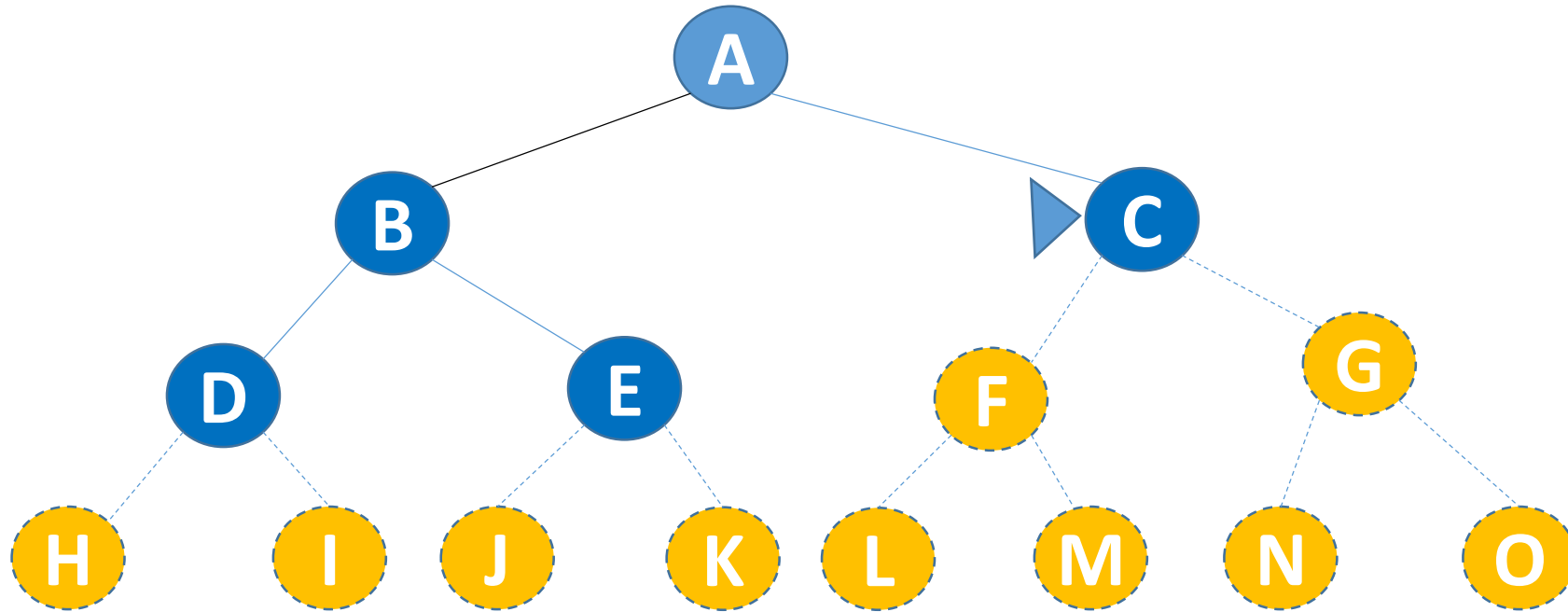


Solid circle - Visited



Dashed circle – not visited

Step 3



Queue {C,D,E}

Check whether C is a goal State

Else

Expand C - > add all its successors at the end of the Queue



Marker Node to be expanded

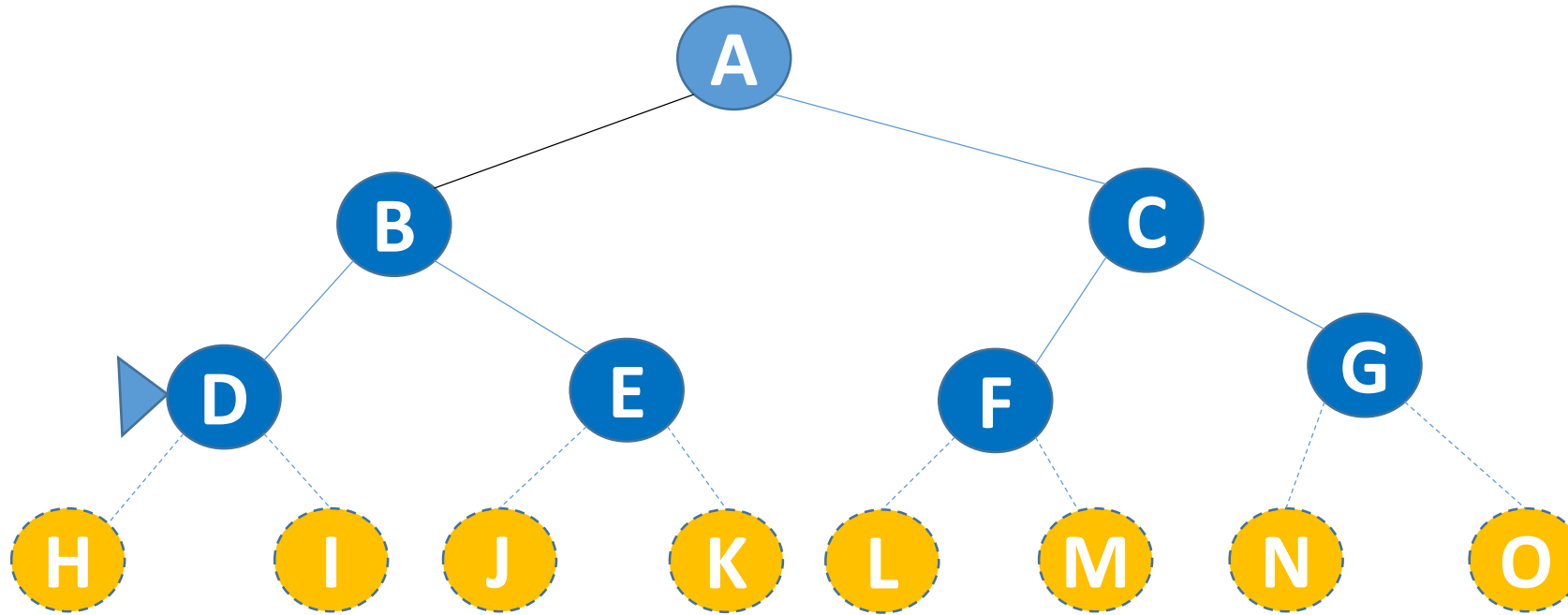


Solid circle - Visited



Dashed circle – not visited

Step 4



Queue {D,E,F,G}

Check whether D is a goal State

Else

Expand D- > add all its successors at the end of the Queue



Marker Node to be expended

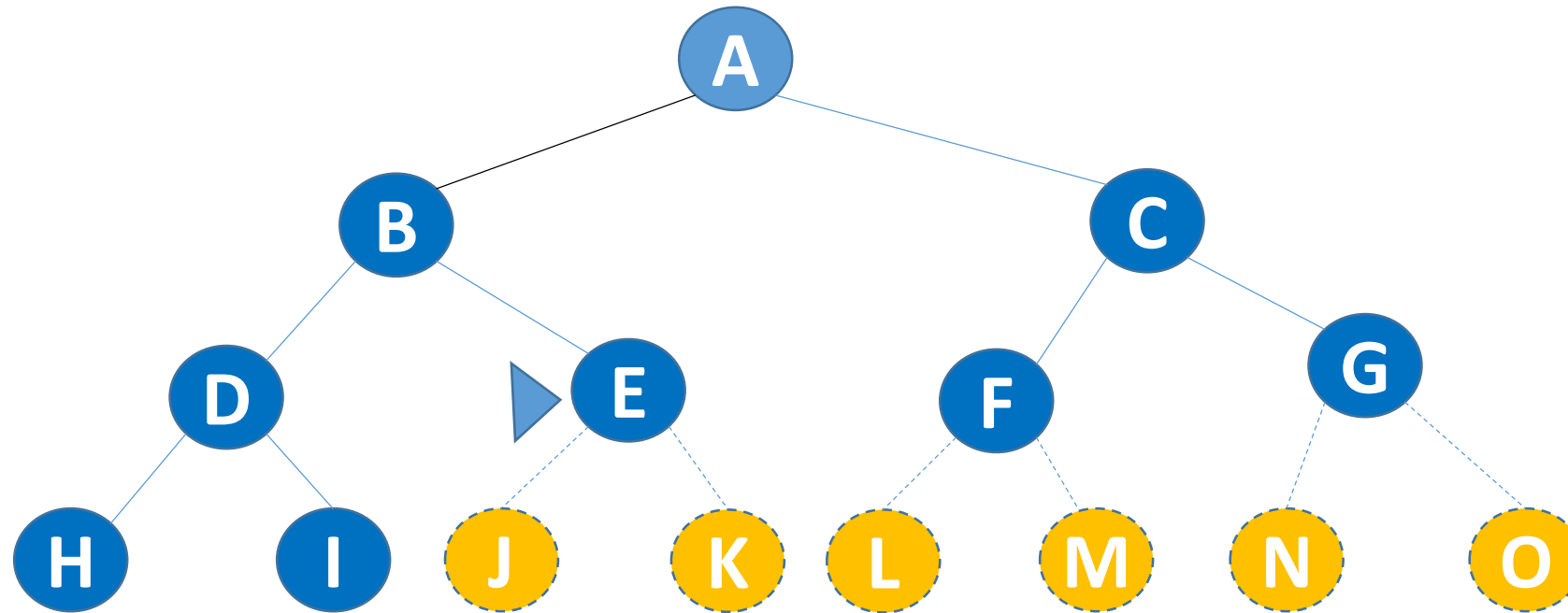


Solid circle - Visited



Dashed circle – not visited

Step 5



Queue {E,F,G,H,I}

Check whether E is a goal State

Else

Expand E -> add all its successors at the end of the Queue



Marker Node to be expended

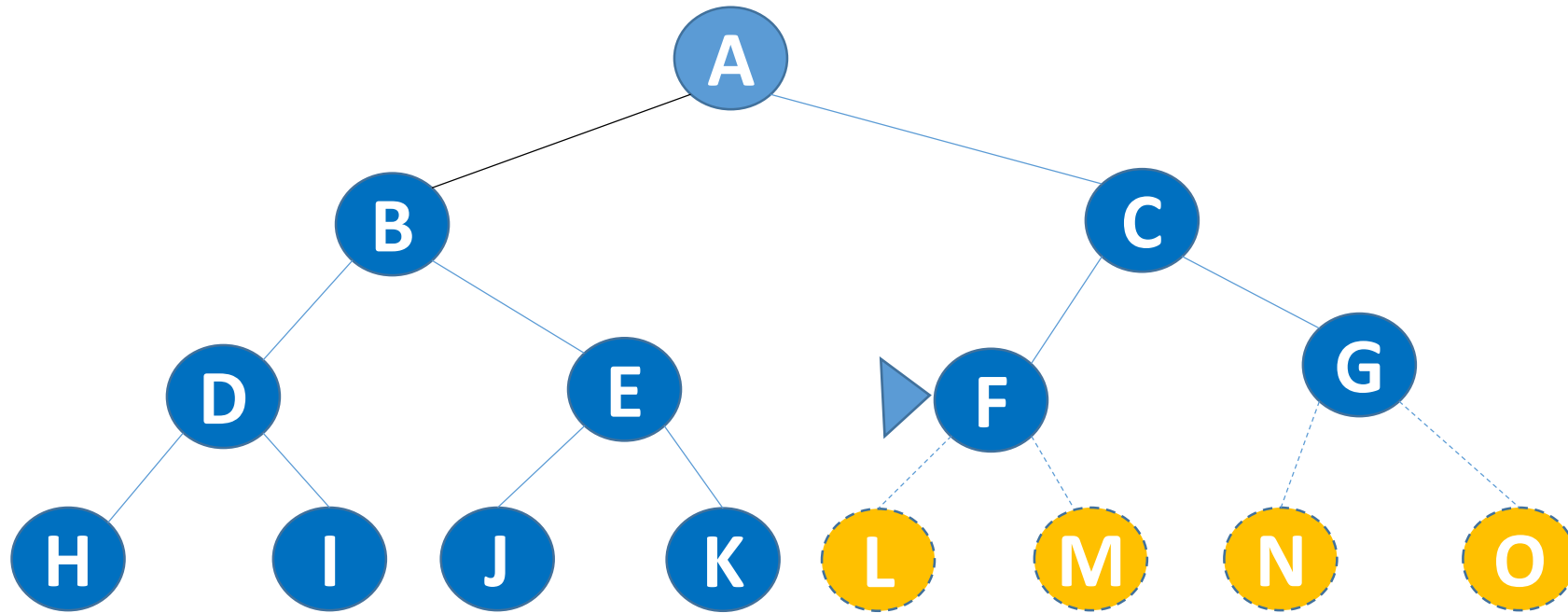


Solid circle - Visited



Dashed circle – not visited

Step 6



Queue {F,G,H,I,J,K}

Check whether F is a goal State

Else

Expand F- > add all its successors at the end of the Queue



Marker Node to be expanded

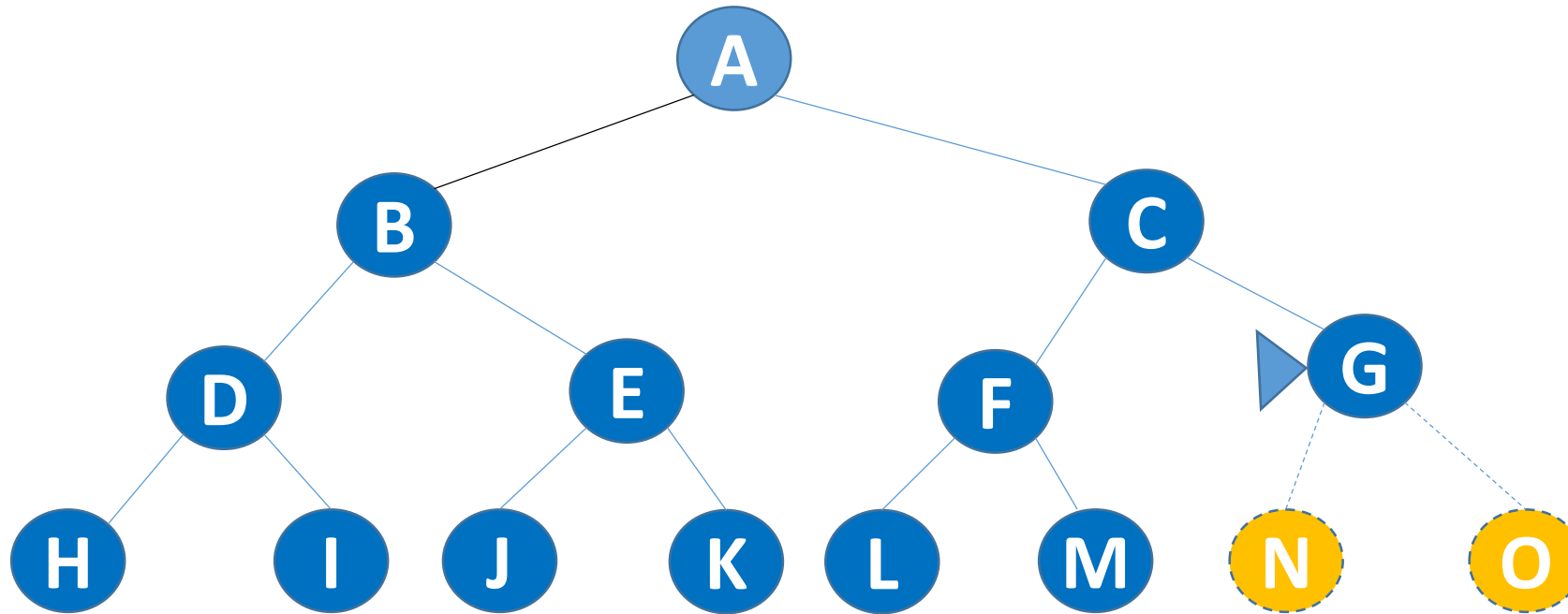


Solid circle - Visited



Dashed circle – not visited

Step 7



Queue {G,H,I,J,K,L,M}

Check whether G is a goal State

Yes G is Goal State -> stop the process.

BFS solution is ABCDEFG



Marker Node to be expanded



Solid circle - Visited



Dashed circle – not visited

- As discussed earlier, Breadth-First Search (BFS) is an algorithm used for traversing graphs or trees.
- Traversing means visiting each node of the graph. Breadth-First Search is a recursive algorithm to search all the vertices of a graph or a tree.
- BFS in python can be implemented by using data structures like a dictionary and lists. Breadth-First Search in tree and graph is almost the same. The only difference is that the graph may contain cycles, so we may traverse to the same node again.
- *Breadth-First Search algorithm is a graph traversing technique, where you select a random initial node (source or root node) and start traversing the graph layer-wise in such a way that all the nodes and their respective children nodes are visited and explored.*

- As breadth-first search is the process of traversing each node of the graph, a standard BFS algorithm traverses each vertex of the graph into two parts: 1) Visited 2) Not Visited. So, the purpose of the algorithm is to visit all the vertex while avoiding cycles.
- BFS starts from a node, then it checks all the nodes at distance one from the beginning node, then it checks all the nodes at distance two, and so on. So as to recollect the nodes to be visited, BFS uses a queue.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure

  node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier  $\leftarrow$  a FIFO queue with node as the only element
  explored  $\leftarrow$  an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node  $\leftarrow$  POP(frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier  $\leftarrow$  INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

Properties of breadth-first search

- Complete? Yes it always reaches goal (if b is finite)
- Time? $1+b+b^2+b^3+\dots +b^d + (b^{d+1}-b)) = O(b^{d+1})$
(this is the number of nodes we generate)
- Space? $O(b^{d+1})$ (keeps every node in memory,
either in fringe or on a path to fringe).
- Optimal? Yes (if we guarantee that deeper solutions are less optimal,
e.g. step-cost=1).
- **Space** is the bigger problem (more than time)

Time complexity for BFS

- Assume every state has b successors.
- The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level.
- Each of these generates b more nodes, yielding b^3 nodes at the third level, and so on.
- Now suppose, that the solution is at depth d .
- In the worst case, we would expand all but the last node at level d , generating $b^{d+1} - b$ nodes at level $d+1$.
- Then the total number of nodes generated is
$$b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1}).$$
- Every node that is generated must remain in memory, because it is either part of the fringe or is an ancestor of a fringe node.
- The space complexity is, therefore, the same as the time complexity

Advantages of Breadth First Search Algorithm

- The main advantage is that it is very simple to implement and understand.
- It is guaranteed to find the shortest path from the starting point to the goal.
- Breadth First Search tends to find paths with fewer steps than other algorithms, such as Depth First Search.
- Breadth First Search can be easily parallelized, which means that it can take advantage of multiple processors to speed up the search.

Disadvantages of Breadth First Search Algorithm

- It can be very memory intensive since it needs to keep track of all the nodes in the search tree.
- It can be slow since it expands all the nodes at each level before moving on to the next level.
- It can sometimes find sub-optimal solutions since it doesn't explore all possible paths through the search tree.

Applications of Breadth First Search Algorithm

- The breadth first search algorithm is a powerful tool that can be used to solve various problems. In this article, we will explore some of the potential applications of this algorithm.
- 1. Graph traversal: Breadth first search can be used to traverse a graph. This can be useful for tasks such as finding the shortest path between two nodes or determining if a graph is connected.
- 2. Network routing: Breadth first search can be used to find the shortest path between two nodes in a network. This can be useful for tasks such as routing traffic or finding the quickest route between two points.
- 3. Pattern matching: Breadth first search can be used to match patterns in data. This can be useful for tasks such as searching for a specific string in a text file or finding all instances of a particular word in a document.
- 4. Data mining: Breadth first search can be used to mine data for insights. This can be useful for tasks such as finding trends in social media data or uncovering hidden relationships in large datasets.

Applications

- **GPS Navigation systems:**

Breadth-First Search is one of the best algorithms used to find neighboring locations by using the GPS system.

- **Find the Shortest Path & Minimum Spanning Tree for an unweighted graph:**

When it comes to an unweighted graph, calculating the shortest path is quite simple since the idea behind the shortest path is to choose a path with the least number of edges. Breadth-First Search can allow this by traversing a minimum number of nodes starting from the source node. Similarly, for a spanning tree, we can use either of the two, Breadth-First Search or Depth-first traversal methods to find a spanning tree.

- **Broadcasting:**

Networking makes use of what we call as packets for communication. These packets follow a traversal method to reach various networking nodes. One of the most commonly used traversal methods is Breadth-First Search. It is being used as an algorithm that is used to communicate broadcasted packets across all the nodes in a network.

- **Peer to Peer Networking:**

Breadth-First Search can be used as a traversal method to find all the neighboring nodes in a Peer to Peer Network. For example, BitTorrent uses Breadth-First Search for peer to peer communication.

Time and memory requirements for breadth-first search

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes
Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.			

Uniform Cost Search

- ✓ **BFS** is optimal only if all step cost or path costs are equal.
- ✓ **BFS** always expands the shallowest node first.
- ✓ **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 is to **find a path to the goal node** which has the **lowest cumulative cost**.
- ✓ It expands nodes according to their path costs from the root node. It can be used to solve any graph/tree where the optimal cost is in demand.
- ✓ A uniform-cost search algorithm is implemented by the **priority queue**. It gives **maximum priority** to the **lowest cumulative cost**.
- ✓ Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.
- ✓ Uniform-cost search **does not care** about the **number of steps** a path has, but **only about their total cost**.

Cost of a node is defined as:

$\text{cost}(\text{node}) = \text{cumulative cost of all nodes from root}$

$\text{cost}(\text{root}) = 0$

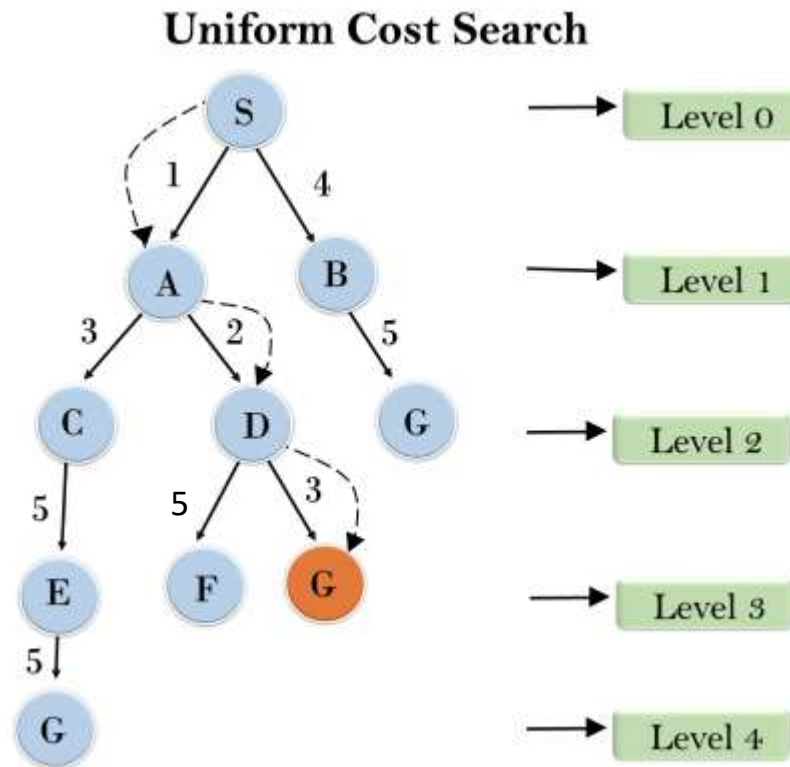
Uniform Cost Search

✓ **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.



function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

node $\leftarrow$ a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0

frontier $\leftarrow$ a priority queue ordered by PATH-COST, with *node* as the only element

explored $\leftarrow$ an empty set

loop do

if EMPTY?(*frontier*) **then return** failure

node $\leftarrow$ POP(*frontier*) /* chooses the lowest-cost node in *frontier* */

if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

 add *node*.STATE to *explored*

for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**

child $\leftarrow$ CHILD-NODE(*problem*, *node*, *action*)

if *child*.STATE is not in *explored* or *frontier* **then**

frontier $\leftarrow$ INSERT(*child*, *frontier*)

else if *child*.STATE is in *frontier* with higher PATH-COST **then**

 replace that *frontier* node with *child*

- Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d . Instead, let C^* be the cost of the optimal solution, and assume that every action costs at least ϵ .
- Then the algorithm's worst-case time and space complexity is $O(b^{1+C^*/\epsilon})$, which can be much greater than b^d . This is because uniform cost search can explore large trees of small steps before exploring paths involving large and perhaps useful steps.

DEPTH FIRST SEARCH (DFS)

- Depth-first-search always expands the deepest node in the current fringe of the search tree.
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- As those nodes are expanded, they are dropped from the fringe, so then the search "backs up" to the next shallowest node that still has unexplored successors.
- This strategy can be implemented by TREE-SEARCH with a last-in- first- out (LIFO) structure, also known as a stack.
- Depth-first-search has very modest memory requirements.
- It needs to store only a single path from the root to a leaf node along with the remaining unexpanded sibling nodes for each node on the path.
- Once the node has been expanded, it can be removed from the memory, as soon as its descendants have been fully explored.
- For a state space with a branching factor b and maximum depth m , depth-first-search requires storage of only $bm + 1$ nodes.

Depth First Search Algorithm

- The DFS algorithm is the search algorithm which begins the searching from the root node and goes down till the leaf of a branch at a time looking for a goal node. If the goal node is not found, the searching retraces its steps back to the point from where the other branch was left unexplored and the same procedure is repeated for that branch.
- DFS follows the following 3 steps:
 - Visit a node “S”.
 - Mark “S” as visited.
 - Recursively visit every unvisited node attached to “S”.
- Since DFS is of **recursive** nature, this can be implemented using **stacks**.

Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at front

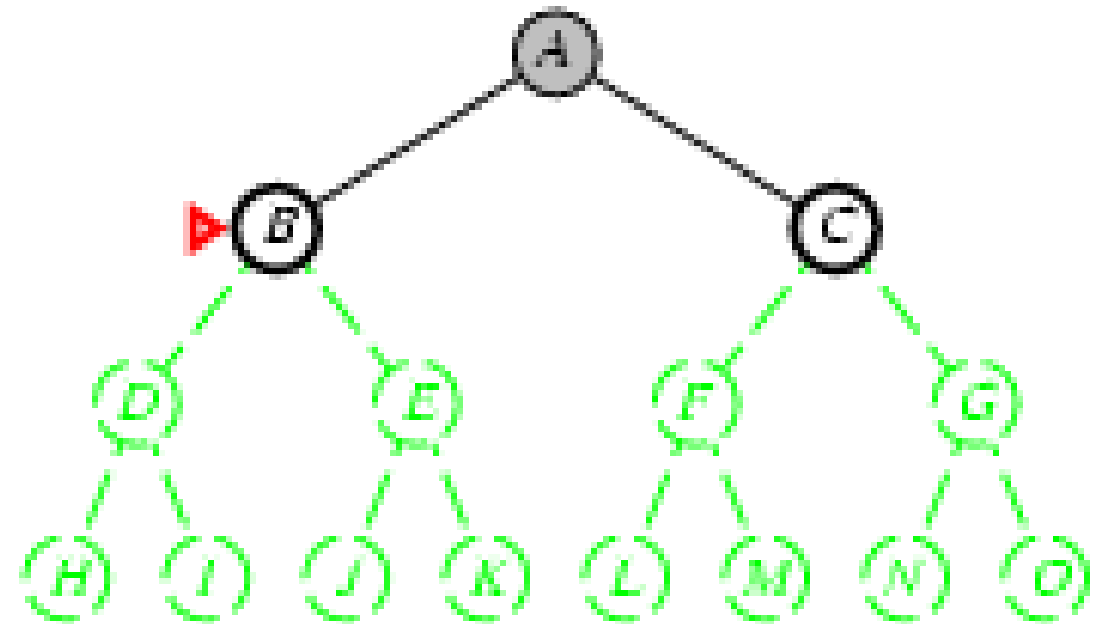
G is the Goal node
A is the start node

Step 1: stack{A},
If A is a goal node, return the solution
Else Expand A → add all its successors at the front of stack



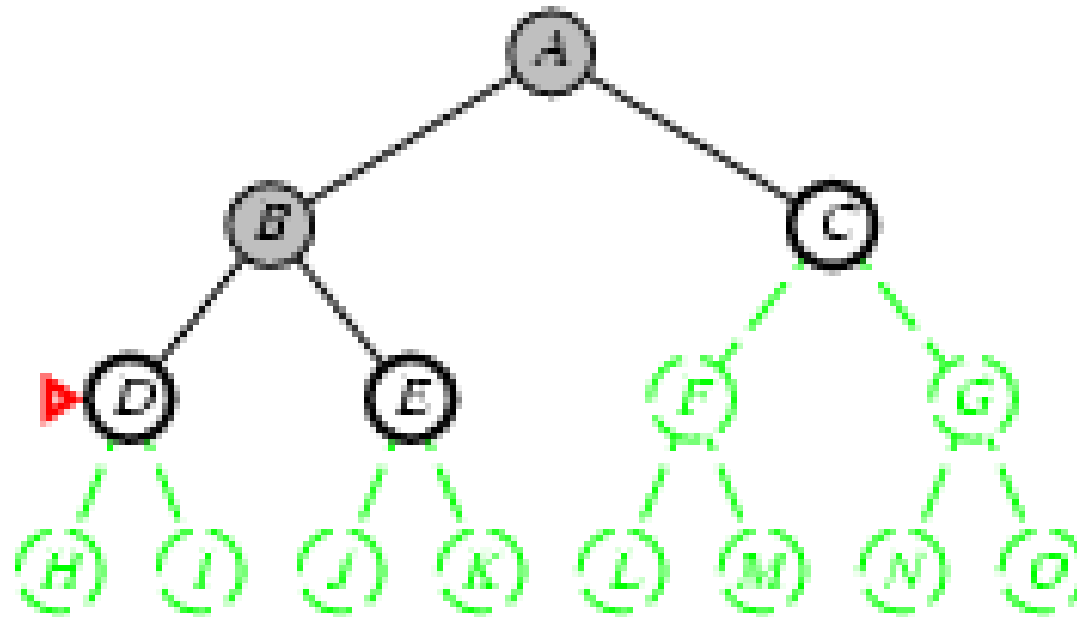
Depth-first search

Step 2: Stack {B,C}
Check whether B is a goal State
Else
Expand B- > add all its successors at the front of the Stack



Depth-first search

Step 3



Stack {D,E,C}

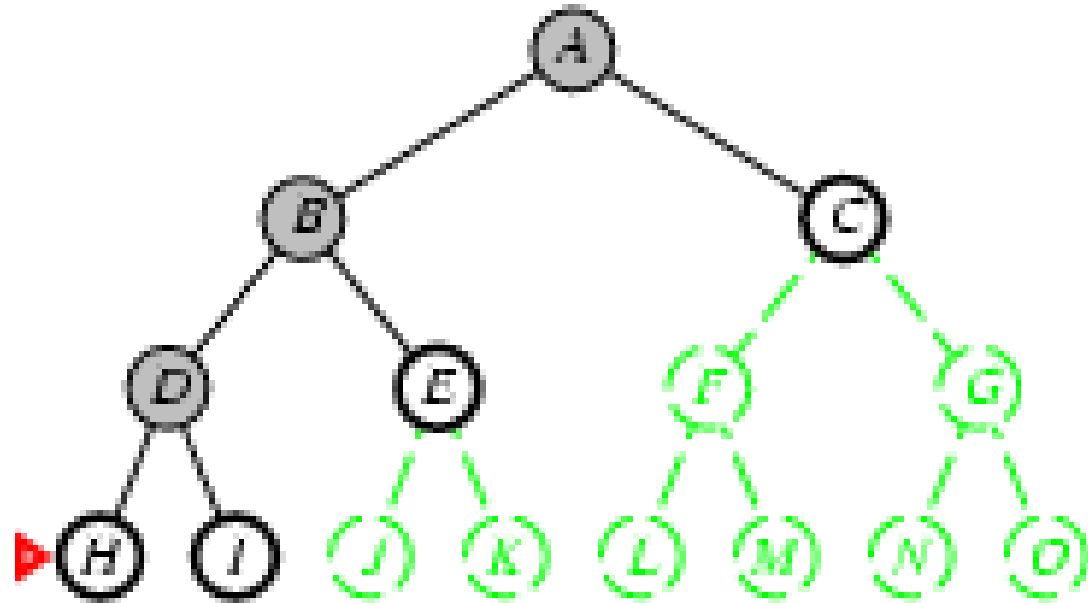
Check whether D is a goal State

Else

Expand D- > add all its successors at the front of the Stack

Depth-first search

Step 4



Stack {H,I,E,C}

Check whether H is a goal State

Else

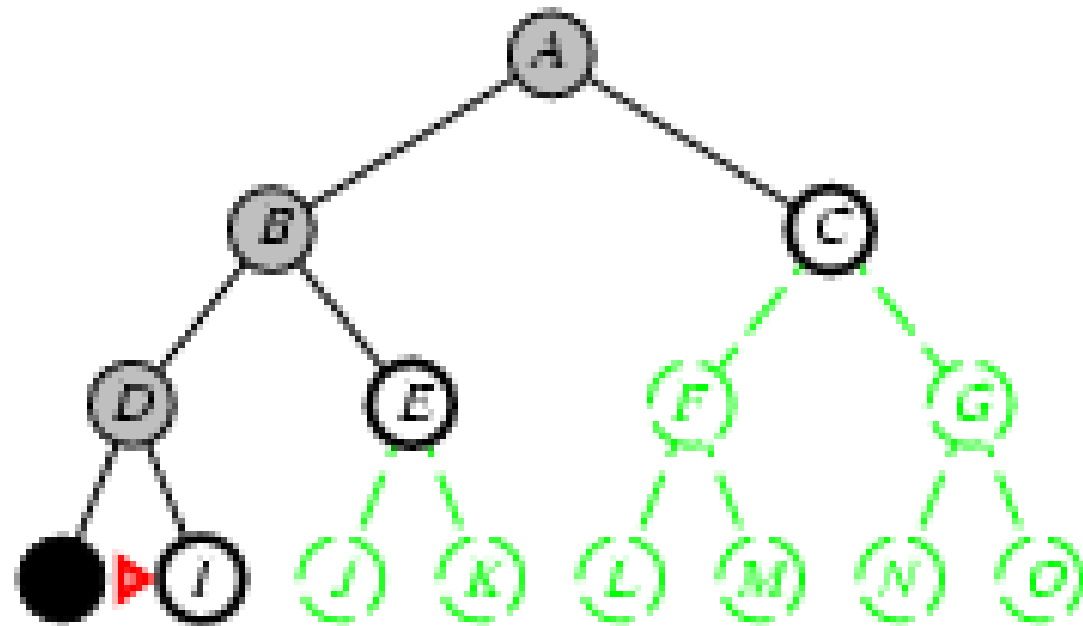
Expand H- > no successor,

Select next node from fringe

Depth-first search

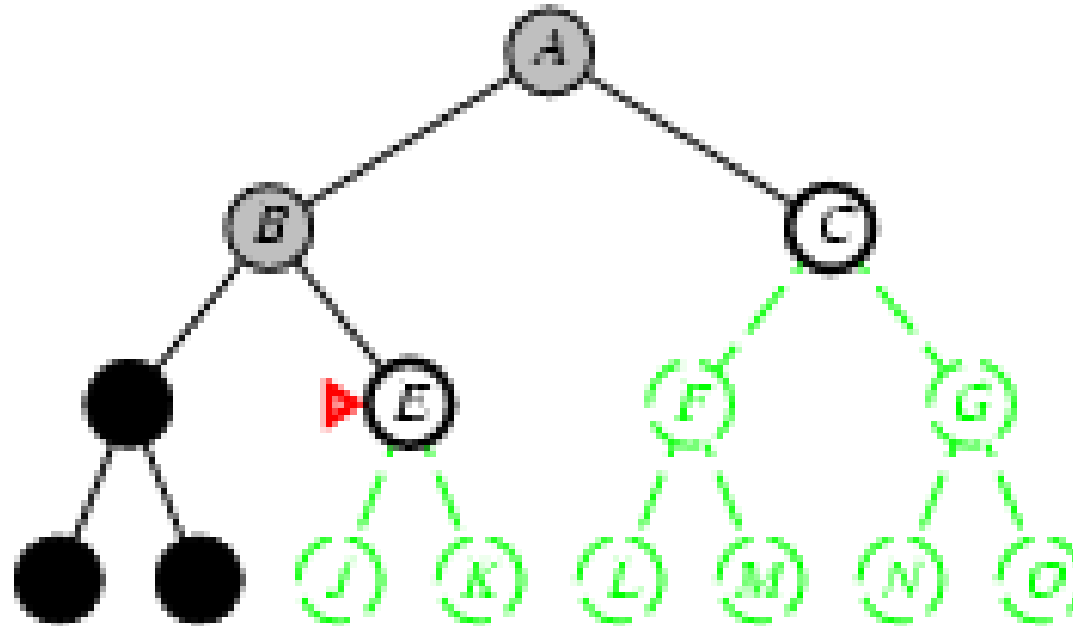
Step-5

Stack {E,C}
Check whether I is a goal State
Else
Expand I -> no successor,
Select next node from fringe



Depth-first search

Step-6



Stack {C}

Check whether E is a goal State

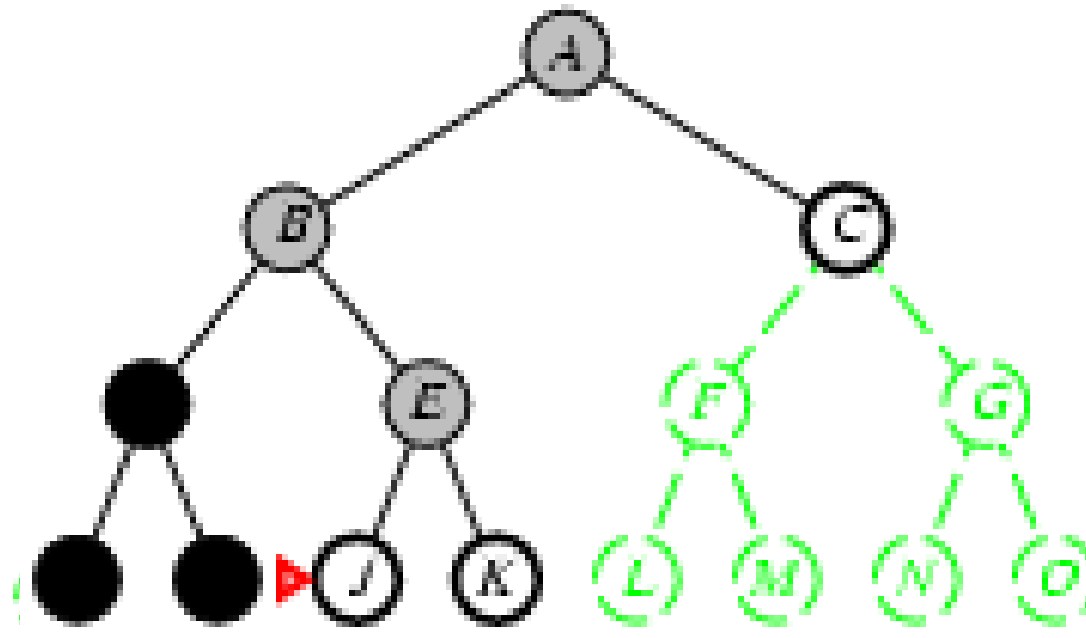
Else

Expand E- > add all its successors at the front of the Stack

Depth-first search

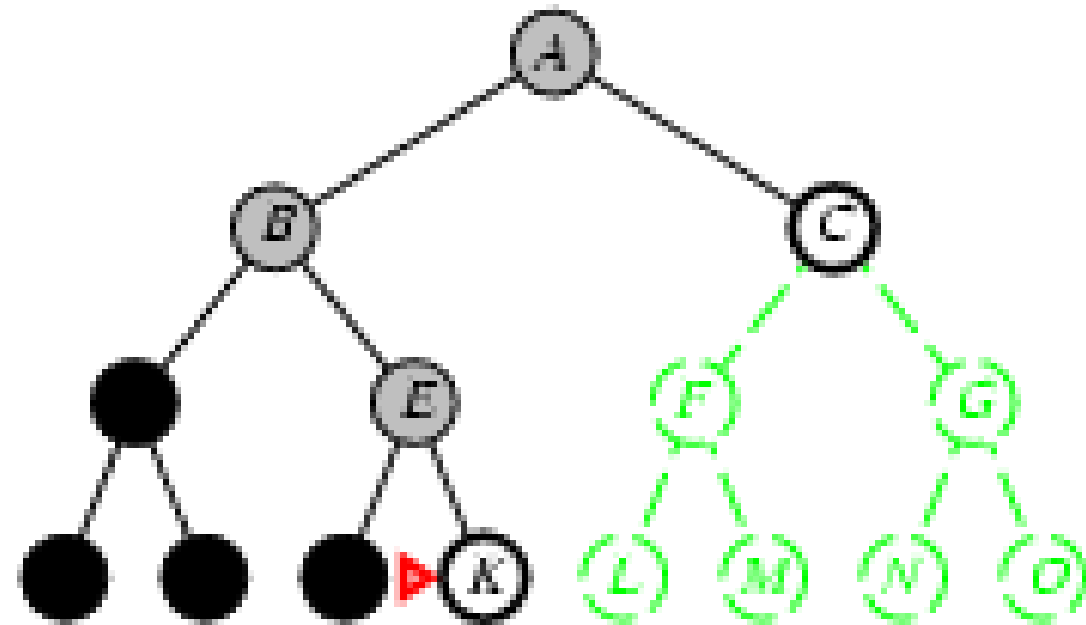
Step-7

Stack {J,K, C}
Check whether J is a goal State
Else
Expand J- > no successor,
Select next node from fringe



Depth-first search

- Step-8



Stack {K, C}

Check whether K is a goal State

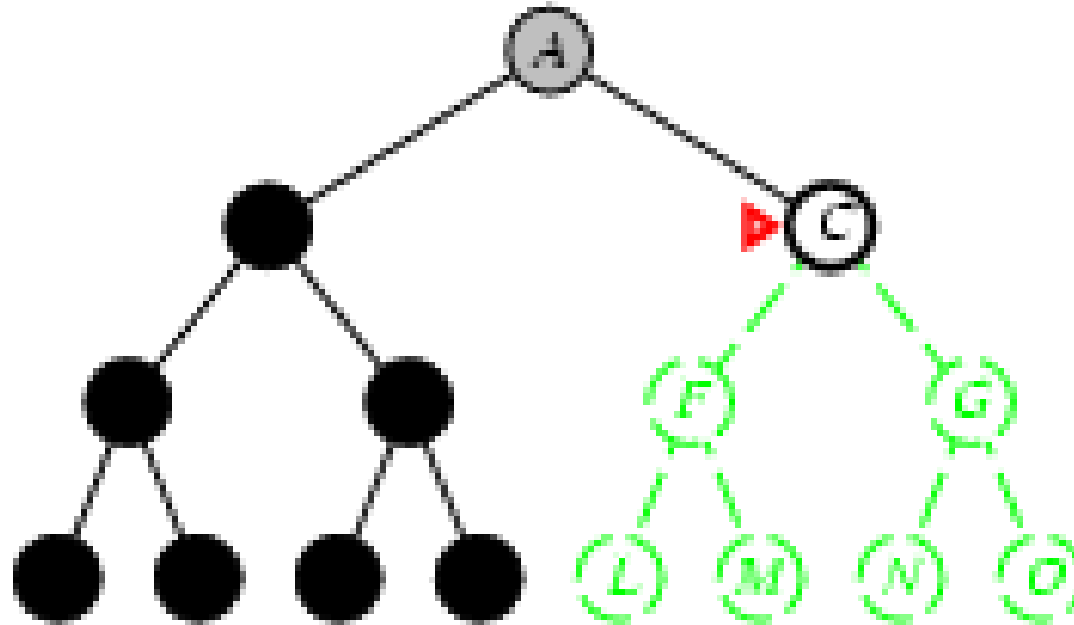
Else

Expand K- > no successor,

Select next node from fringe

Depth-first search

Step-9



Stack { C }

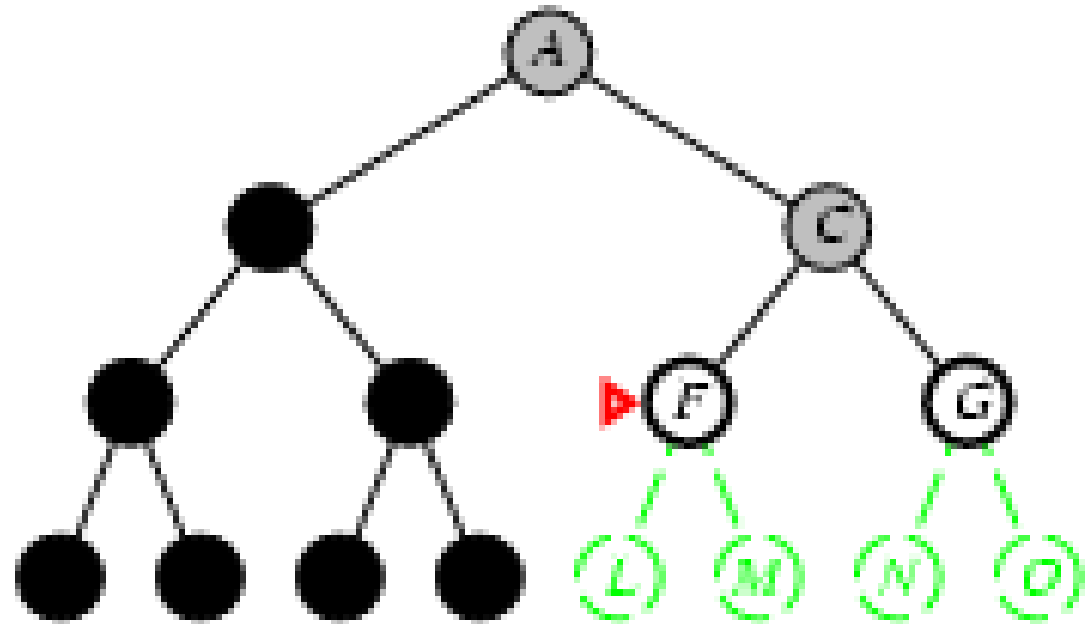
Check whether C is a goal State

Else

Expand C -> add all its successors at the front of the Stack

Depth-first search

Step-10



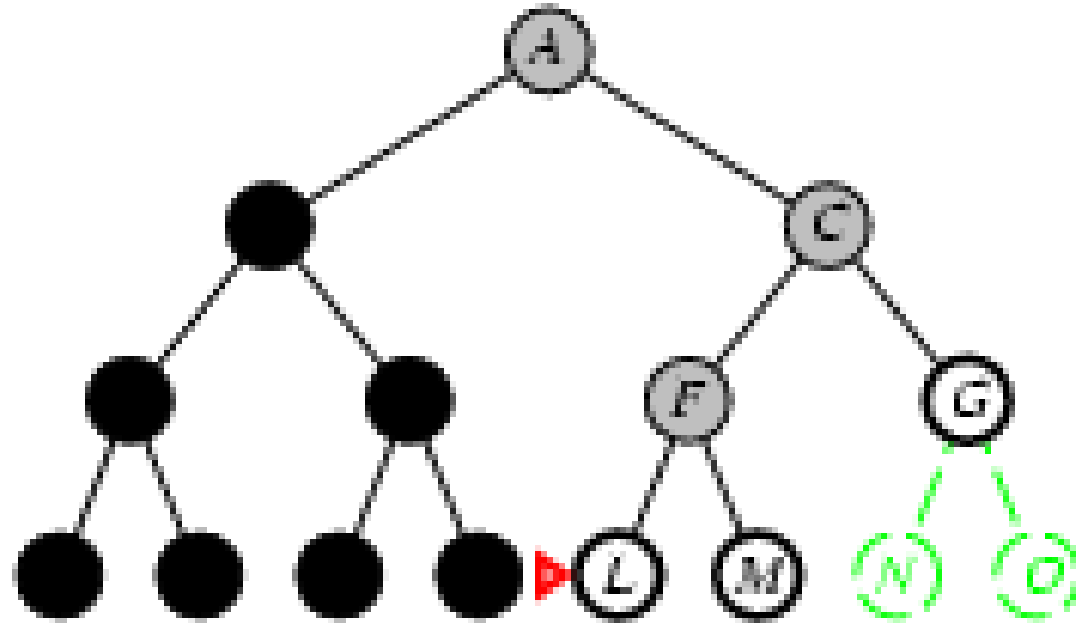
Stack { F,G}

Check whether F is a goal State

Else

Expand F- > add all its successors at the front of the Stack

- Step-11



Stack { L,M,G}

Check whether L is a goal State

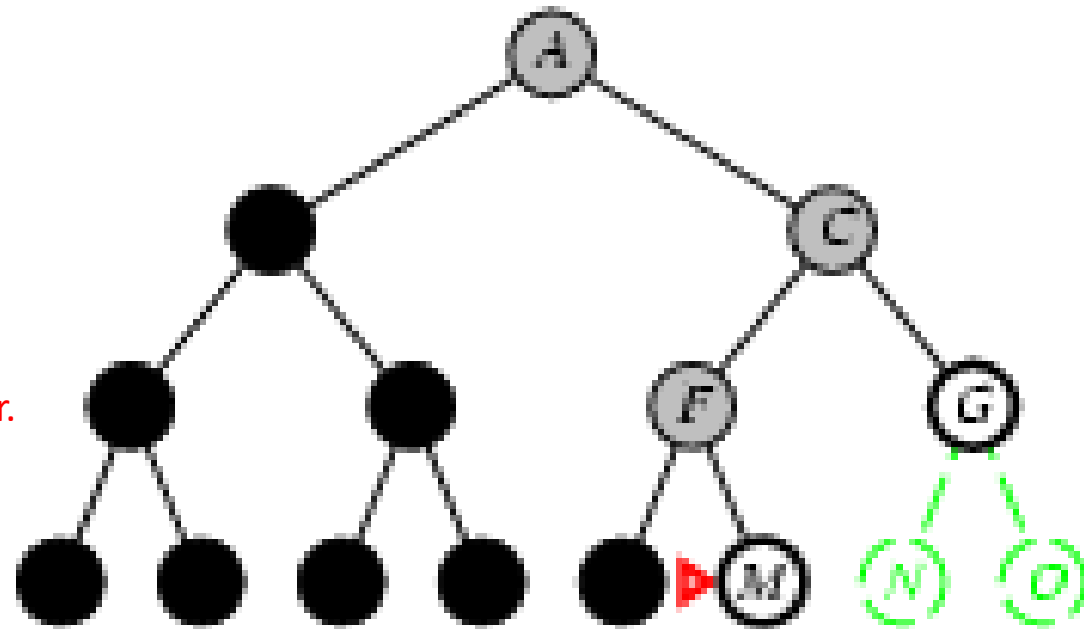
Else

Expand L- > add all its successors at the front of the Stack. No successor. Select next node from stack for expansion

Depth-first search

Step-12

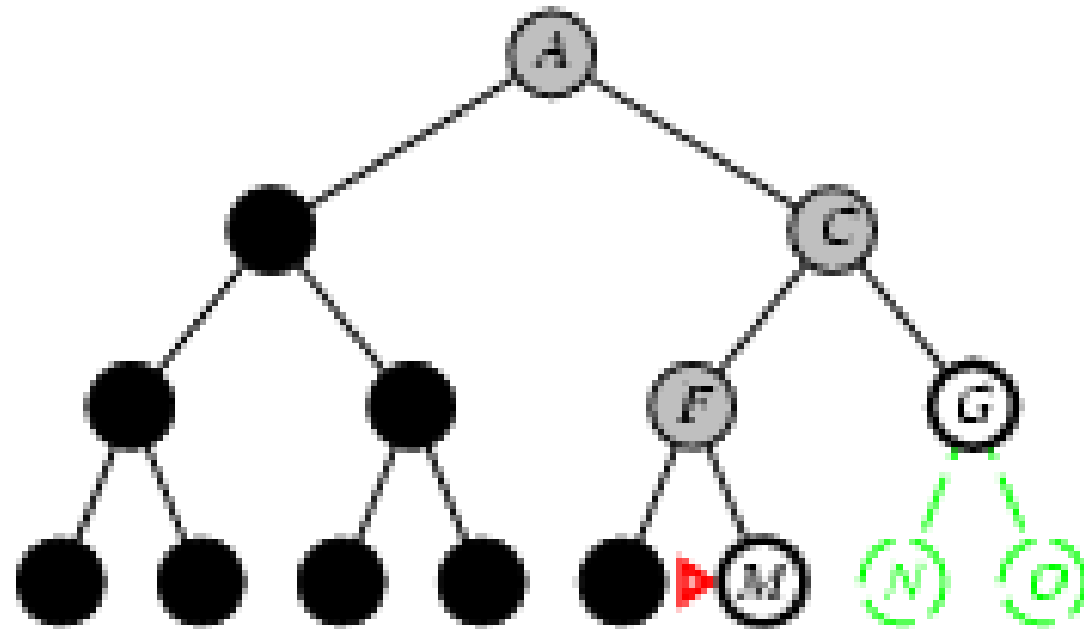
Stack { M,G}
Check whether M is a goal State
Else
Expand M- > add all its successors. No successor.
Select next node from stack for expansion



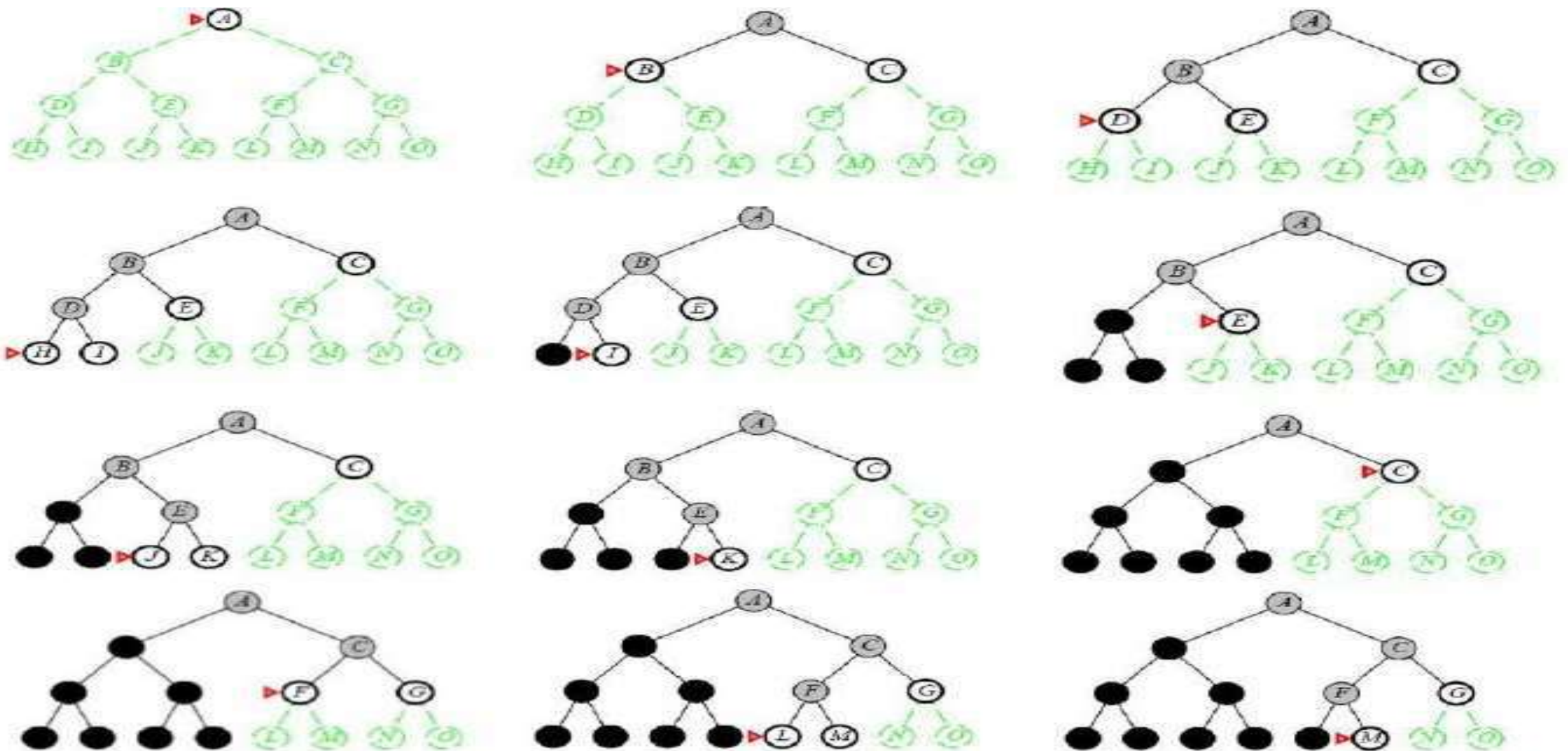
Depth-first search

Step-13

Stack { G }
Check whether G is a goal State
Return the solution



DFS Solution: ABDHIEJKCFLMG



Depth-first-search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from the memory; these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

The DFS algorithm works as follows:

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

Depth-first search

- ✓ **Completeness:**
 - ✓ DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree. fails in infinite-depth spaces
- ✓ **Time Complexity:**
 - ✓ Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by: $O(b^m)$
 - ✓ Where, m = the maximum length of any path in the state space
- ✓ **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 frontier, 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.

✓ 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).
- ✓ A variant of DFS is backtracking search, which generates only one successor at a time. Space complexity is $O(m)$

✓ 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.

Properties of depth-first search

- Complete? No: fails in infinite-depth spaces
Can modify to avoid repeated states along path
- Time? $O(b^m)$ with m =maximum depth
- terrible if m is much larger than d
 - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$, i.e., linear space! (we only need to remember a single path + expanded unexplored nodes)
- Optimal? No (It may find a non-optimal goal first)

Depth Limited Search

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

```

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

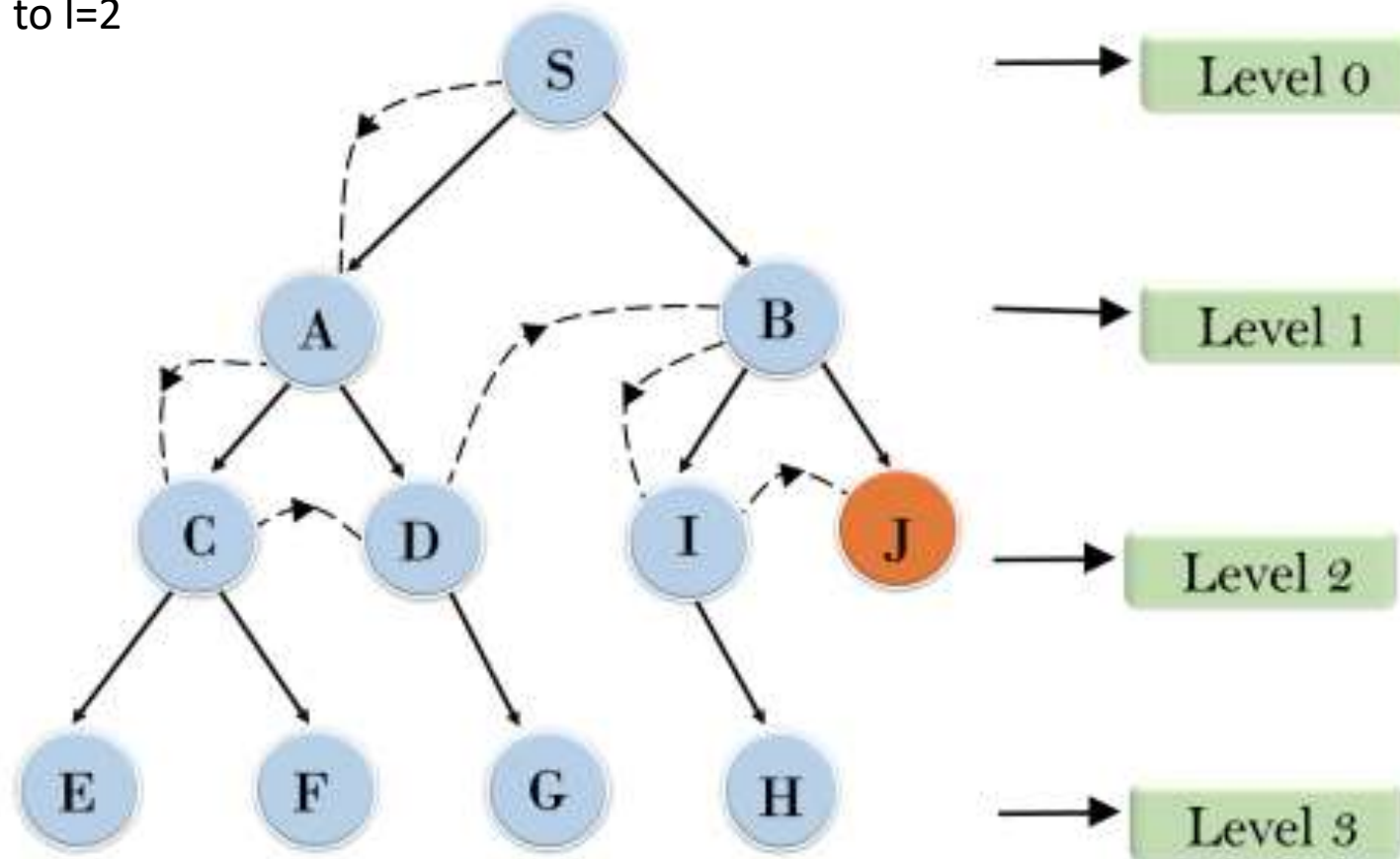
function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
    cutoff_occurred?  $\leftarrow$  false
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action)
      result  $\leftarrow$  RECURSIVE-DLS(child, problem, limit - 1)
      if result = cutoff then cutoff_occurred?  $\leftarrow$  true
      else if result  $\neq$  failure then return result
    if cutoff_occurred? then return cutoff else return failure

```

A recursive implementation of depth-limited tree search.

Depth Limited Search

Depth is limited to $l=2$



Depth Limited Search

- ✓ **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 $\ell > d$.

Iterative deepening depth-first search

- ✓ The iterative deepening search is a general strategy used in combination with depth first tree search
- ✓ 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.

ITERATIVE DEEPENING DEPTH-FIRST SEARCH ALGORITHM

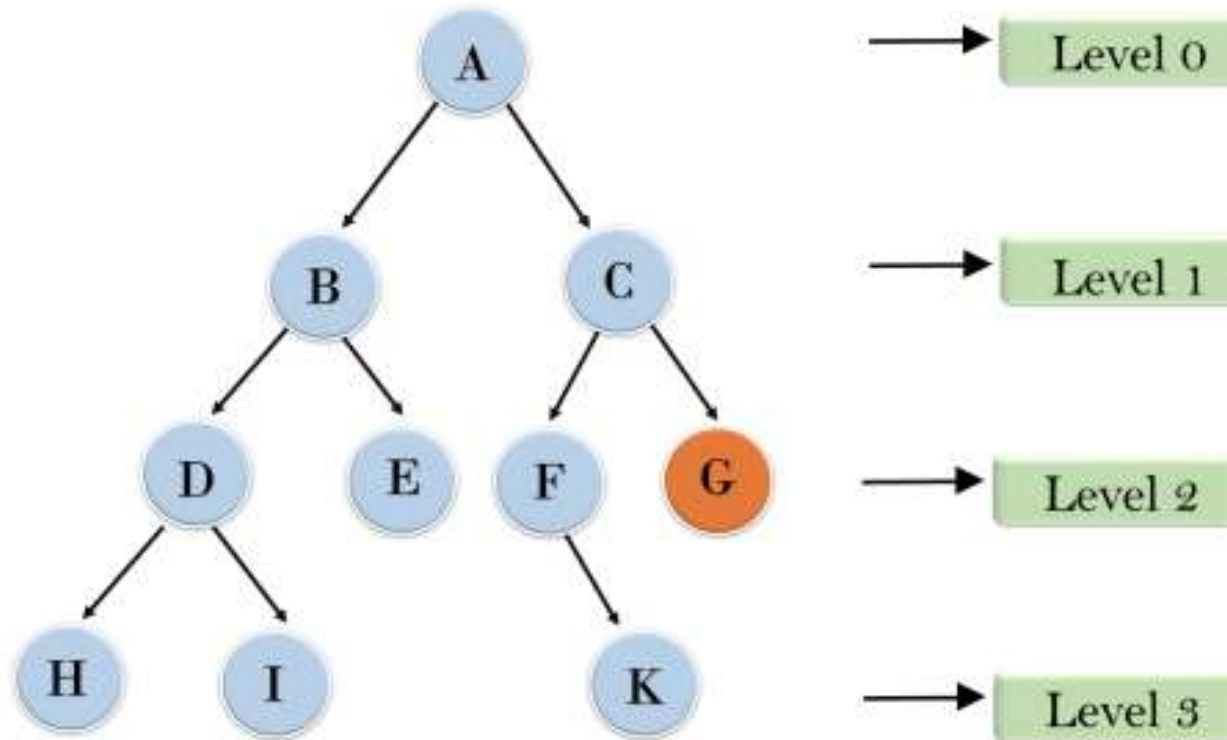
function ITERATIVE-DEEPENING-SEARCH(*problem*) *returns a solution, or failure*

for *depth* = 0 **to** ∞ **do**

result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)

if *result* $\neq$ *cutoff* **then return** *result*

Iterative deepening depth first search



Depth limited to level $l=0$: 1'st Iteration-----> A

Depth limited to level $l=1$: 2'nd Iteration----> A, B, C

Depth limited to level $l=2$: 3'rd Iteration----->A, B, D, E, C, F, G

Depth limited to level $l=3$: 4'th Iteration----->A, B, D, H, I, E, C, F, K, G

Iterative deepening depth first search algorithm will find the goal node in the third iteration.

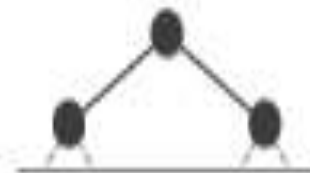
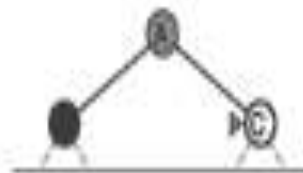
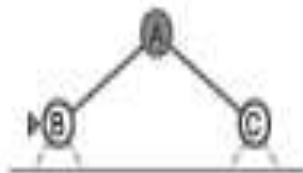
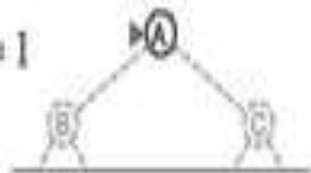
DFS finds the solution in fourth iteration

Limit = 0

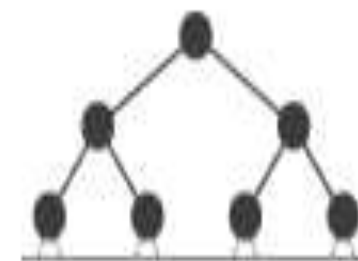
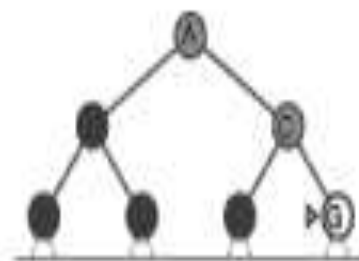
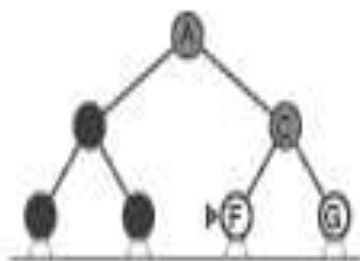
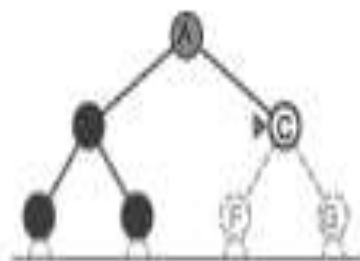
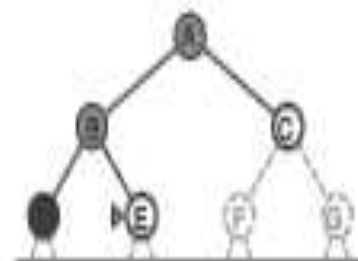
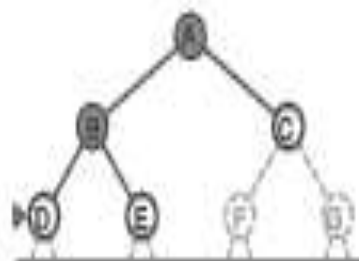
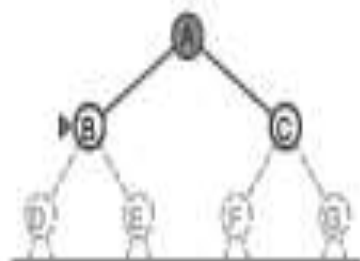
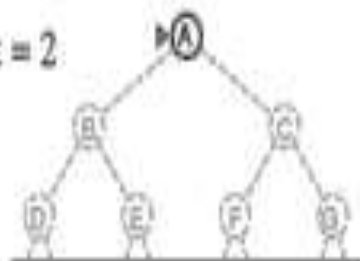


1

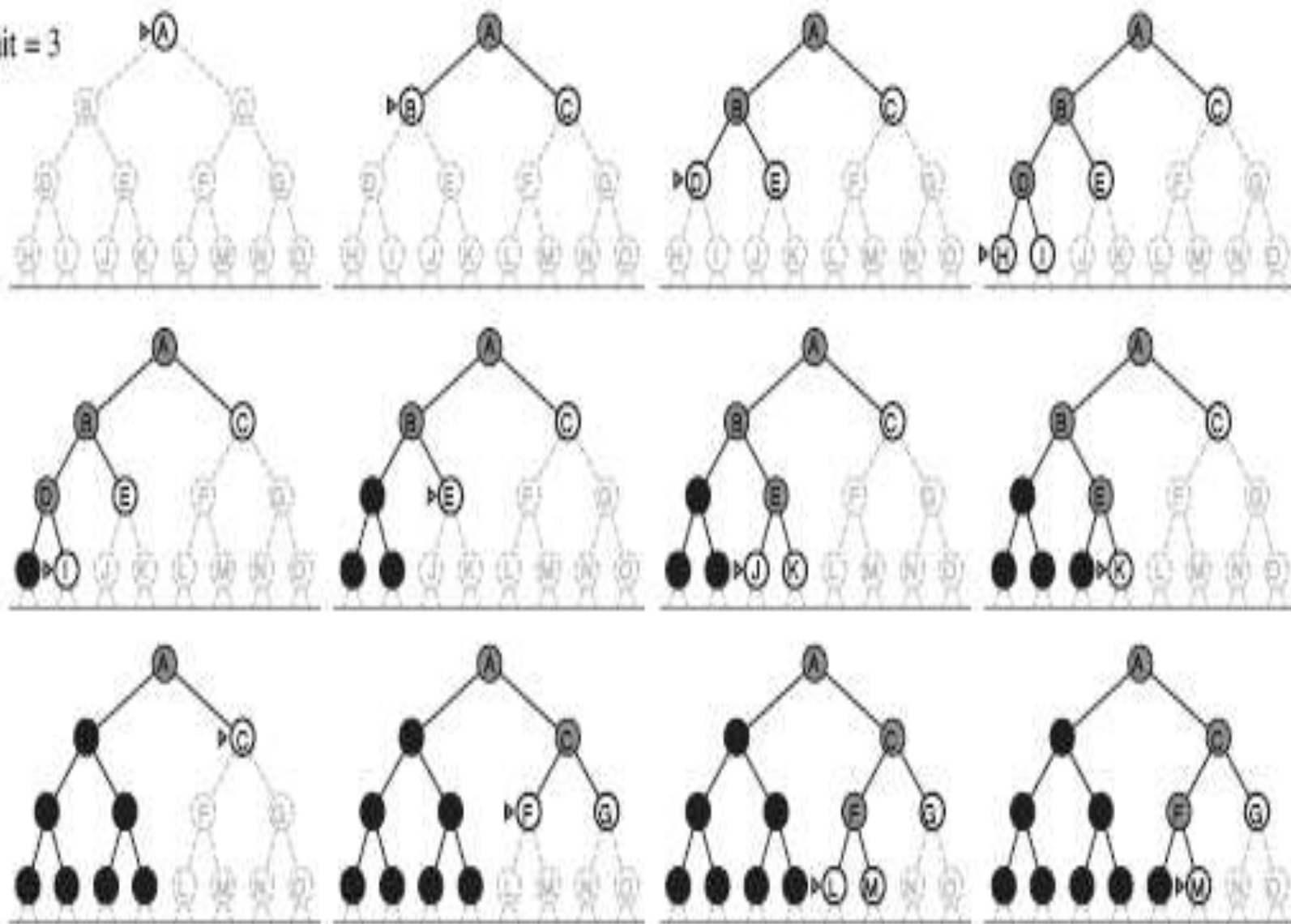
Limit = 1



Limit = 2



Limit = 3



Iterative deepening depth-first search

- ✓ **Completeness:**

- ✓ This algorithm is complete if the branching factor is finite.

- ✓ **Time Complexity:**

- ✓ Time complexity is $O(b^d)$

- ✓ **Space Complexity:**

- ✓ Space complexity is $O(b \times d)$.

- ✓ **Optimal:**

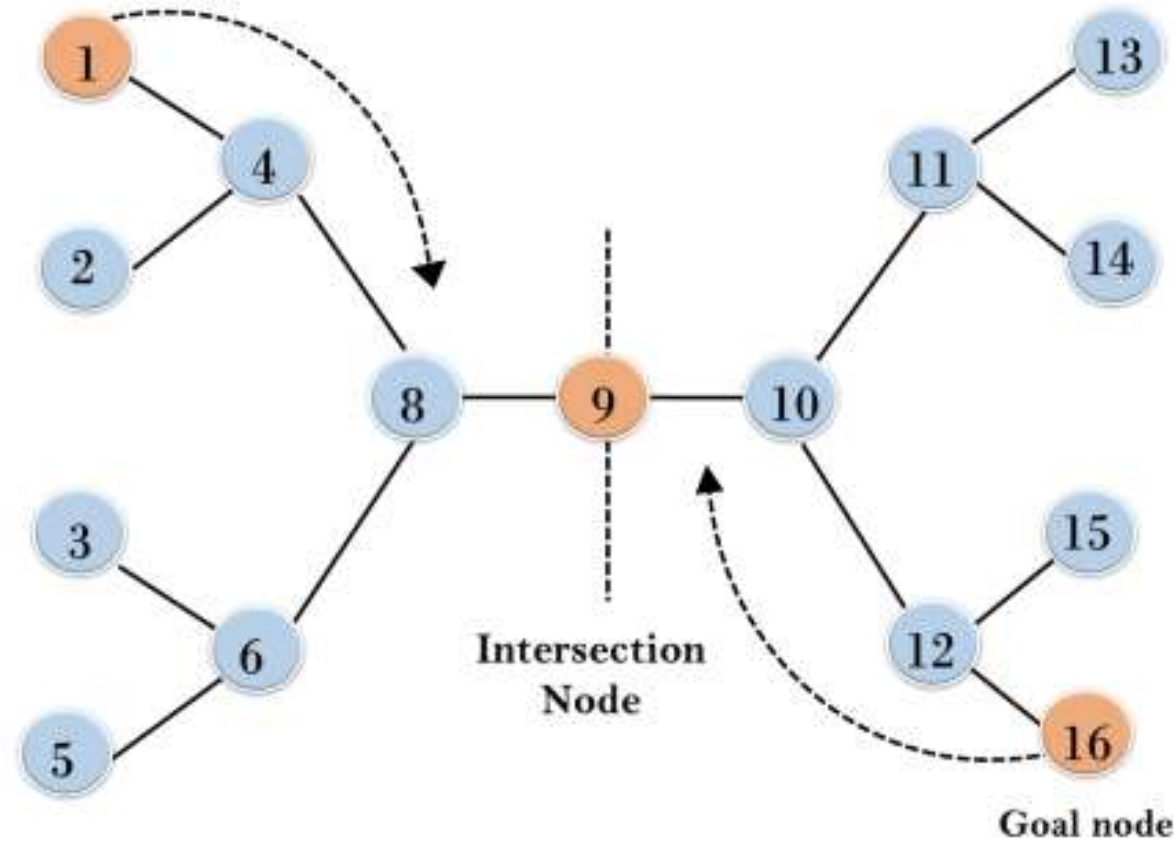
- ✓ IDDFS algorithm is optimal if path cost is a non-decreasing function of the depth of the node.

Bidirectional Search Algorithm

- ✓ Bidirectional search algorithm runs two simultaneous searches, one from 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.

Bidirectional Search

Root node



Bidirectional Search Algorithm

- ✓ **Completeness:**
 - ✓ Bidirectional Search is complete if we use BFS in both searches.
- ✓ **Time Complexity:**
 - ✓ Time complexity is $O(b^{d/2})$
- ✓ **Space Complexity:**
 - ✓ Space complexity is $O(b^{d/2})$
- ✓ **Optimal:**
 - ✓ Bidirectional search is Optimal.

Comparing uninformed search strategies

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

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: ^a complete if b is finite; ^b complete if step costs $\geq \epsilon$ for positive ϵ ; ^c optimal if step costs are all identical; ^d if both directions use breadth-first search.

Open list and Closed List

Open List

The **open list** (also: **frontier**) organizes the leaves of a search tree.

It must support two operations efficiently:

- determine and remove the next node to expand
- insert a new node that is a candidate node for expansion

Methods of an Open List *open*

open.is_empty() test if the open list is empty

open.pop() removes and returns the next node to expand

open.insert(*n*) inserts node *n* into the open list

- Different search algorithm use different strategies for the decision which node to return in *open.pop*.
- The choice of a suitable data structure depends on this strategy (e.g., stack, deque, min-heap).

Closed List

The **closed list** remembers expanded states to avoid duplicated expansions of the same state.

It must support two operations efficiently:

- insert a node whose state is not yet in the closed list
- test if a node with a given state is in the closed list; if yes, return it

Methods of a Closed List *closed*

closed.insert(*n*) insert node *n* into *closed*;
if a node with this state already exists in *closed*,
replace it

closed.lookup(*s*) test if a node with state *s* exists in the closed list;
if yes, return it; otherwise, return **none**

- **All search algorithms use open and closed list**
- **The node which is to expanded next, that is `open.pop()` is determined by the search algorithm**
- **Ex: DFS, BFS**

Algorithm

- **Create 2 empty lists: OPEN and CLOSED**
- **Start from the initial node (say N) and put it in the 'ordered' OPEN list**
- **Repeat the next steps until GOAL node is reached**
 - **If OPEN list is empty, then EXIT the loop returning 'False'**
 - **Select the first/top node (say N) in the OPEN list and move it to the CLOSED list. Also capture the information of the parent node**
 - **If N is a GOAL node, then move the node to the Closed list and exit the loop returning 'True'. The solution can be found by backtracking the path**
 - **If N is not the GOAL node, expand node N to generate the 'immediate' next nodes linked to node N and add all those to the OPEN list if not in closed list**
 - **Reorder the nodes in the OPEN list in according to the searching algorithm**

If you take this as an example

Step 1: open list: A

closed list:

Step 2: after visiting first node, performing open.pop() and closed.insert() successors of A will be added in open list and A will move to closed list

open list: B,C

closed list: A

Step 3: next by selecting node B for expansion , ie, moving B to closed list and adding successors of B to open list

open list: C,D,E

closed list: A,B

Step 4: next node to expand depend on search algorithm

if we use BFS, next node for expansion is C ,ie, moving C to closed list and adding successors of C to open list

open list: D,E,F,G

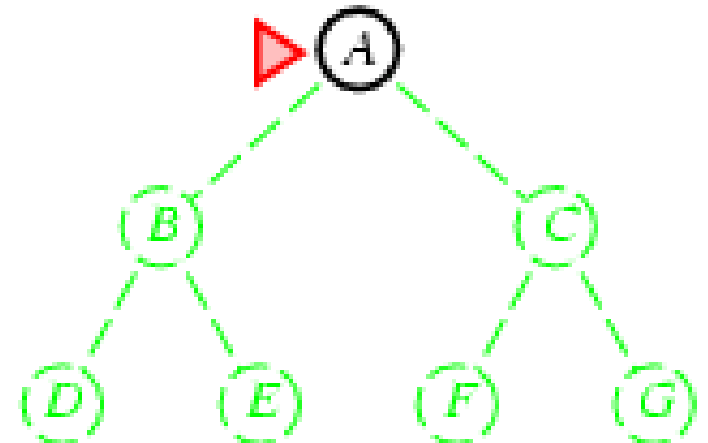
closed list: A,B,C

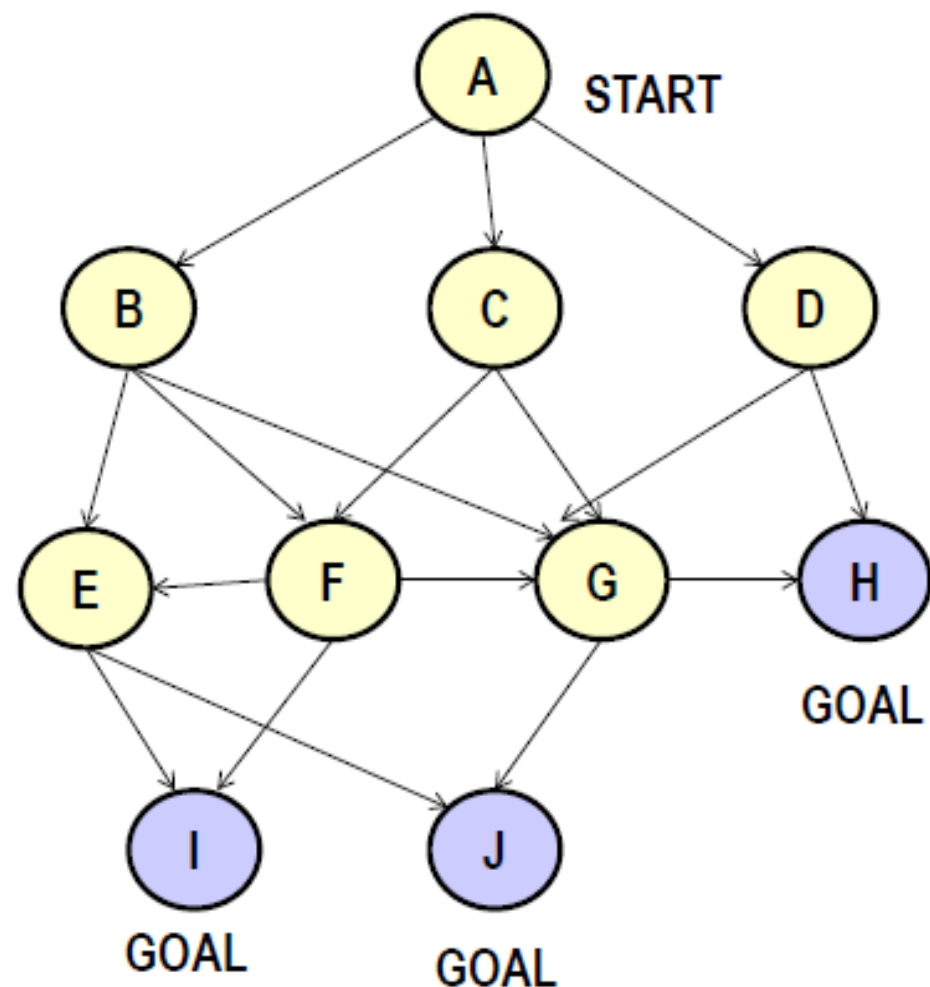
if we use DFS, next node for expansion is D then

open list: C,E,

closed list: A,B,D

This continues till the open list is empty



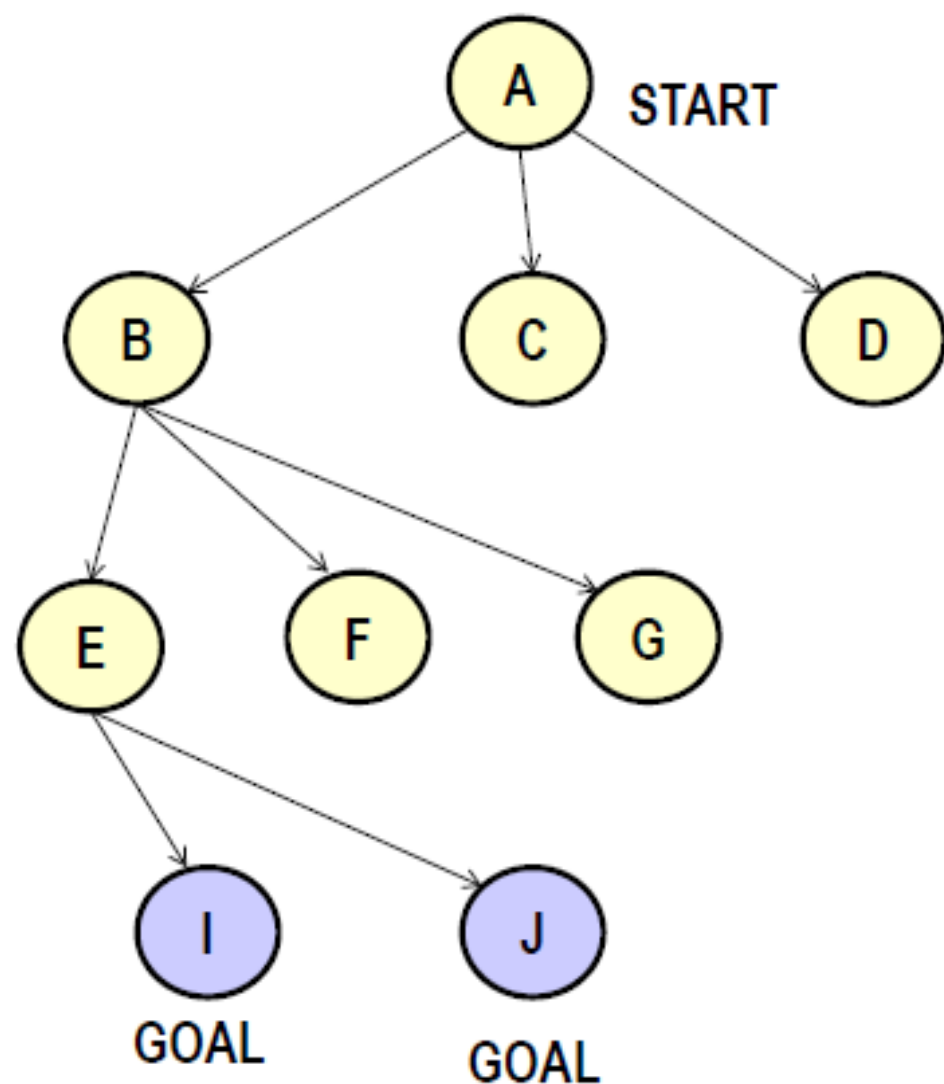


BREADTH-FIRST SEARCH:

1. OPEN = {A}, CLOSED = {}
2. OPEN = {B,C,D}, CLOSED = {A}
3. OPEN = {C,D,E,F,G}, CLOSED = {A,B}
4. OPEN = {D,E,F,G}, CLOSED = {A,B,C}
5. OPEN = {E,F,G,H}, CLOSED = {A,B,C,D}
6. OPEN = {F,G,H,I,J}, CLOSED = {A,B,C,D,E}
7. OPEN = {G,H,I,J}, CLOSED = {A,B,C,D,E,F}
8. OPEN = {H,I,J}, CLOSED = {A,B,C,D,E,F,G}
9. Goal Node H Found. Can Terminate with Path from A to H. This is guaranteed to be the minimum length path.

BFS GUARANTEES SHORTEST LENGTH PATH TO GOAL BUT HAS HIGHER MEMORY REQUIREMENT

- **Breadth first search** explores the space level by level only when there are no more states to be explored at a given level does the algorithm move on to the next level.
- We implement **BFS** using lists open and closed to keep track of progress through the state space.
- In the open list, the elements will be those who have been generated but whose children have not been examined.
- The closed list records the states that have been examined and whose children have been generated.
- The order of removing the states from the open list will be the order of searching.
- The open is maintained as a queue on the first in first out data structure. States are added to the right of the list and removed from the left.



DEPTH-FIRST SEARCH:

1. OPEN = {A}, CLOSED = {}
2. OPEN = {B,C,D}, CLOSED = {A}
3. OPEN = {E,F,G,C,D}, CLOSED = {A,B}
4. OPEN = {I,J,F,G,C,D}, CLOSED = {A,B,E}
5. Goal Node I Found. Can Terminate with Path from A to I or may Continue for more Goal nodes if minimum length or cost is a criteria

DFS MAY NOT TERMINATE IF THERE IS AN INFINITE DEPTH PATH EVEN IF THERE IS A GOAL NODE AT FINITE DEPTH

DFS HAS LOW MEMORY REQUIREMENT

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 from the goal, path cost, how to reach to goal node, etc.
- This knowledge help agents to explore less to the search space and find more efficiently the goal node.
- The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

INFORMED SEARCH AND EXPLORATION

Informed (Heuristic) Search Strategies

- **Informed search strategy** is one that uses problem-specific knowledge beyond the definition of the problem itself.
- It can find solutions more efficiently than uninformed strategy.

Heuristics function:

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

Admissibility of the heuristic function is given as:

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

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.

Various informed search strategies are

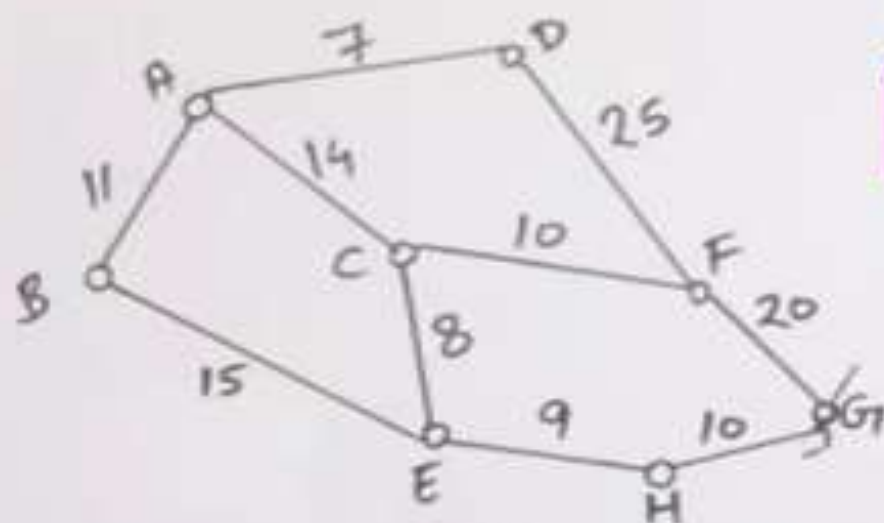
- ✓ Best-first Search Algorithm (Greedy Search)
- ✓ A* Search
- ✓ AO* search

Best-first search

- **Best-first search** is an instance of general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function** $f(n)$.
- The node with lowest evaluation is selected for expansion, because the evaluation measures the distance to the goal.
- This can be implemented using a priority-queue, a data structure that will maintain the fringe in ascending order of f -values.
- 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 cost is estimated by heuristic function,

BEST FIRST SEARCH(BFS)

- ❖ The implementation of Best First Search Algorithm involves maintaining two lists- OPEN and CLOSED.
- ❖ **OPEN** contains those nodes that have been evaluated by the heuristic function but have not been expanded into successors yet.
- ❖ **CLOSED** contains those nodes that have already been visited.



OPEN	CLOSE
A]	[]
[C,B,D]	[A]
B,D	A,C
F,E,B,D	A,C
G,E,B,D	A,C,F
E,B,D	A,C,F,G

Straight line dist

$A \rightarrow G = 40$
 $B \rightarrow G = 32$
 $C \rightarrow G = 25$
 $D \rightarrow G = 35$
 $E \rightarrow G = 19$
 $F \rightarrow G = 17$
 $H \rightarrow G = 10$
 $G \rightarrow G = 0$

PATH={A->C->F->G AND COST=44}

BEST FIRST SEARCH(BFS)

Algorithm

- Priority queue 'PQ' containing initial states

Loop

- If PQ=Empty Return Fail

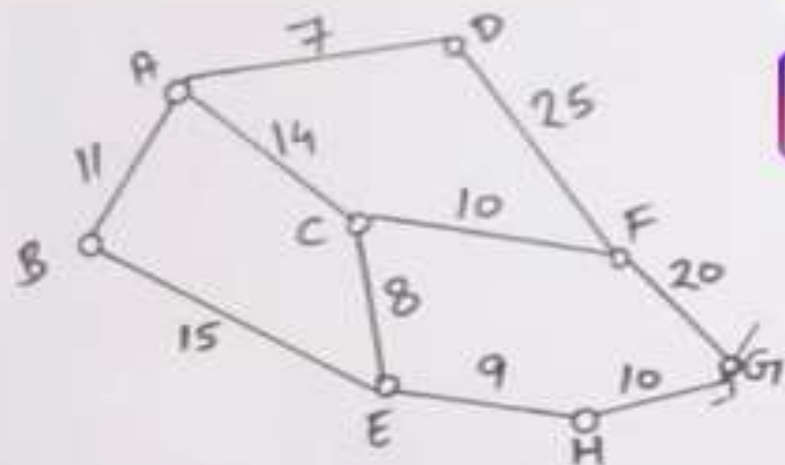
Else

- Insert Node into PQ(Open-List)
- Remove First(PQ) -> NODE(Close-List) <
- If NODE=GOAL
- Return path from initial state to NODE

ELSE

- Generate all successor of NODE and insert newly generated NODE into 'PQ' according to cost value.

END LOOP



OPEN	CLOSE
[A]	[]
[C,B,D]	[A]
B,D	A,C
F,E,B,D	A,C
G,E,B,D	A,C,F
E,B,D	A,C,F,G

Straight line dist

A → G = 40
B → G = 32
C → G = 25
D → G = 35
E → G = 19
F → G = 17
H → G = 10
G → G = 0

PATH={A->C->F->G} AND COST=44

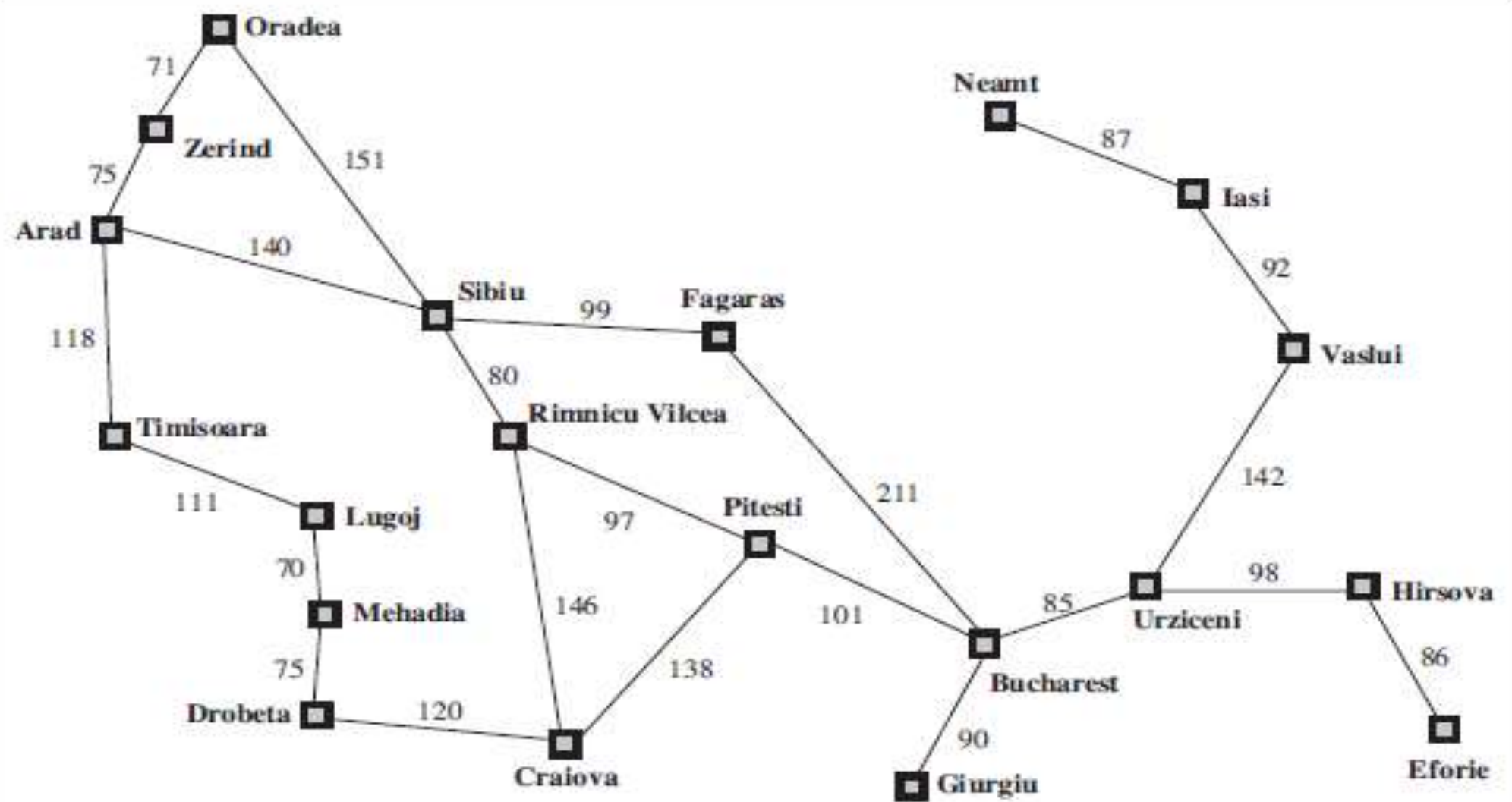


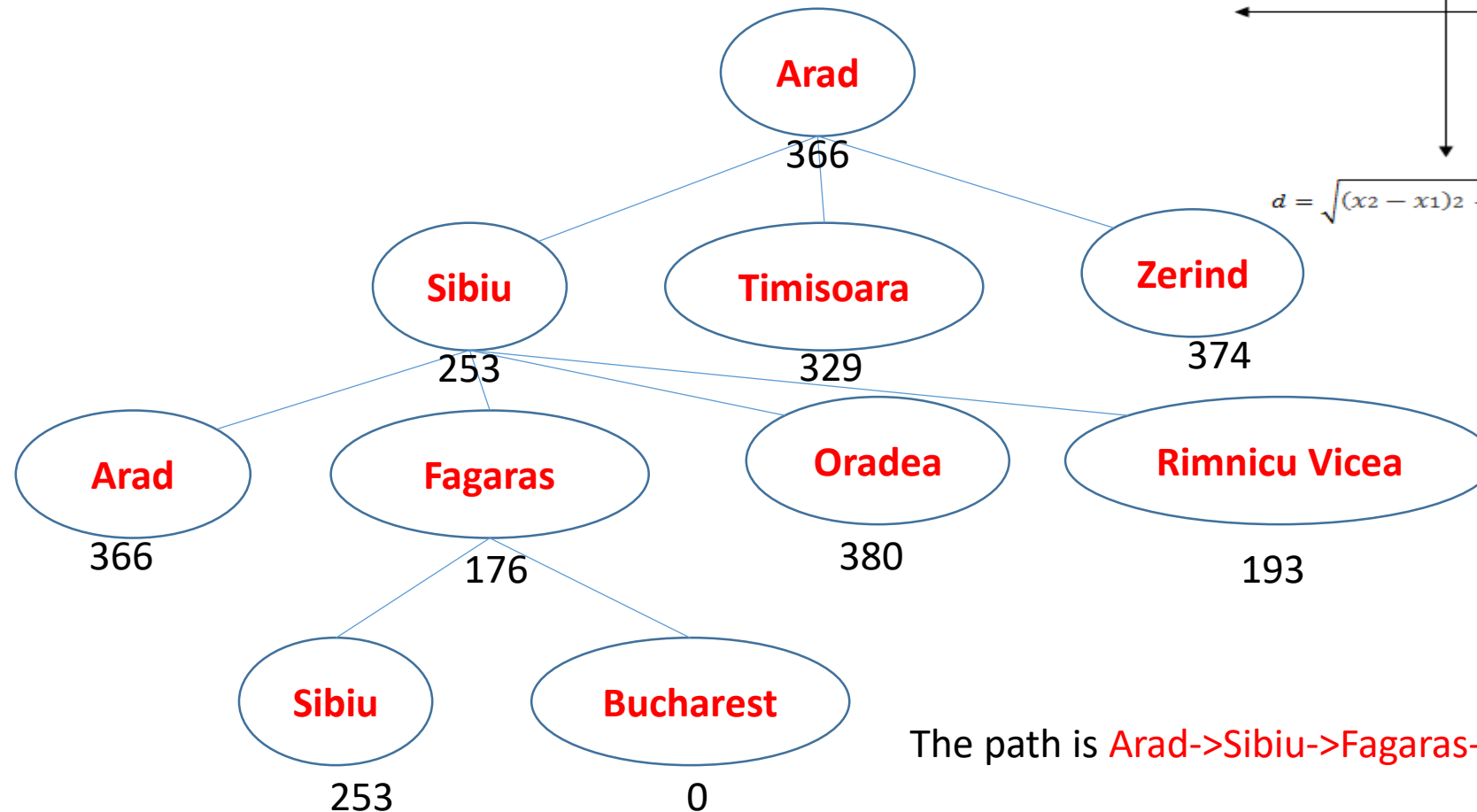
Figure 3.2 A simplified road map of part of Romania.

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

Figure 3.22 Values of h_{SLD} —straight-line distances to Bucharest.

Best-first Search Algorithm (Greedy Search)

- ✓ $h_{\text{SLD}}(n)$ = straight-line distance from n to Bucharest Greedy best-first search expands the node that appears to be closest to goal
- ✓ Greedy best-first search expands the node that appears to be closest to goal



$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The path is **Arad->Sibiu->Fagaras->Bucharest**

Best-first Search Algorithm (Greedy Search)

- ✓ **Completeness:**
 - ✓ Complete
- ✓ **Time Complexity:**
 - ✓ Time complexity is $O(b^m)$
- ✓ **Space Complexity:**
 - ✓ Space complexity is $O(b^m)$
- ✓ **Optimal:**
 - ✓ No Optimal.

Advantages:

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

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.

A* Search

- ✓ 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.

A* Search Algorithm

- ✓ In the A* search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

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

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

$g(n)$ = cost to reach node n from start state.

$f(n)$ = estimated cost of the cheapest solution through n , Evaluation function

A* Search Algorithm

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.

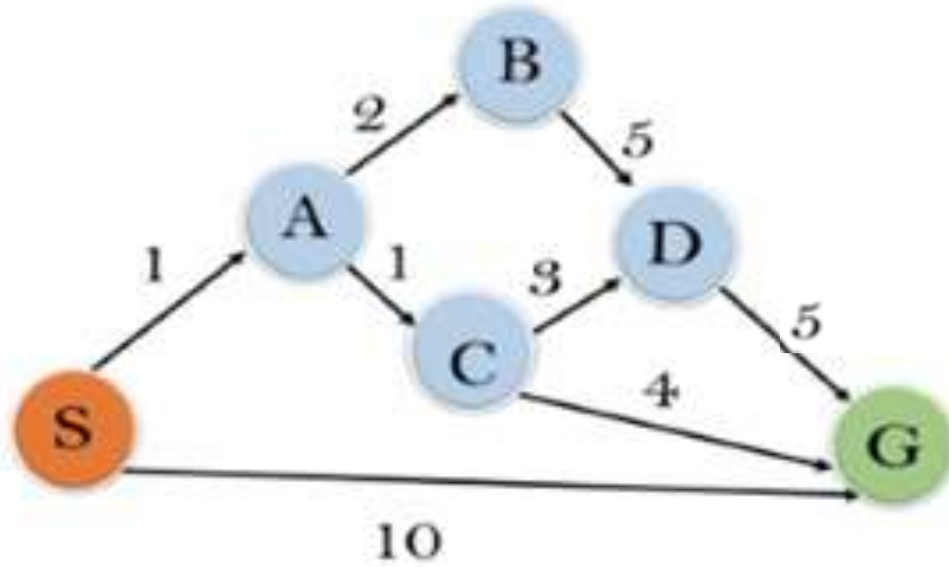
Step 3: Select the node from the OPEN list which has the smallest value of evaluation function ($g+h$), if node n is goal node then return success and stop, otherwise

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.

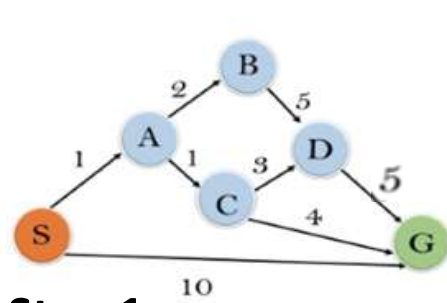
Step 6: Return to Step 2.

A* Algorithm



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

A* Algorithm



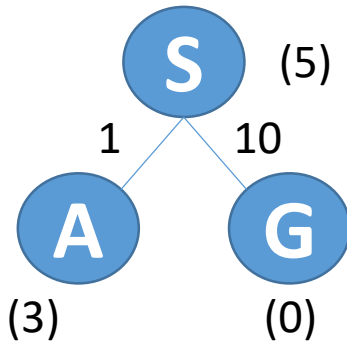
Step 1:

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



$$S \rightarrow S ; f(n) = g(n) + h(n) = 0 + 5 = 5$$

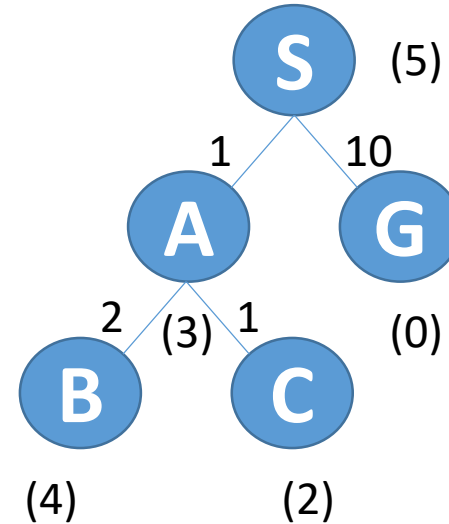
Step 2:



$$S \rightarrow A ; f(n) = g(n) + h(n) = 1 + 3 = 4$$

$$S \rightarrow G ; f(n) = g(n) + h(n) = 10 + 0 = 10$$

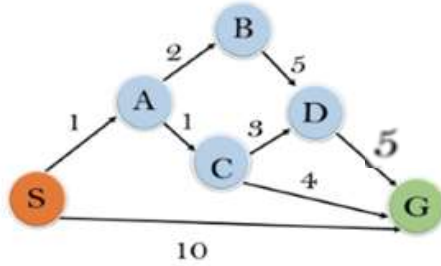
Step 3:



$$S \rightarrow A \rightarrow B ; f(n) = g(n) + h(n) = 3 + 4 = 7$$

$$S \rightarrow A \rightarrow C ; f(n) = g(n) + h(n) = 2 + 2 = 4$$

A* Algorithm



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

$$S \rightarrow S ; f(n) = g(n) + h(n) = 0 + 5 = 5$$

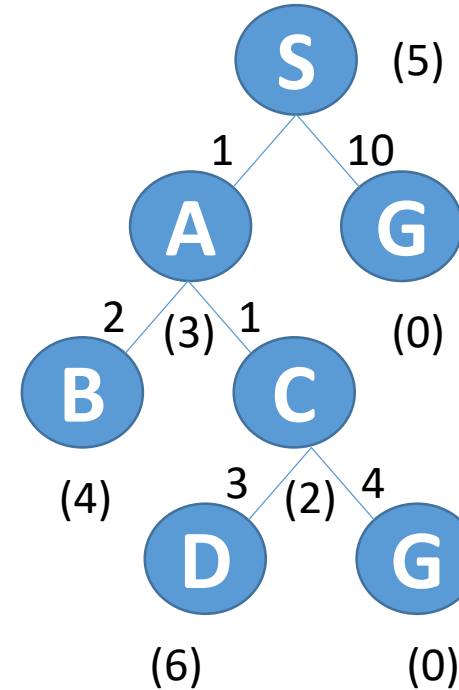
$$S \rightarrow A ; f(n) = g(n) + h(n) = 1 + 3 = 4$$

$$S \rightarrow G ; f(n) = g(n) + h(n) = 10 + 0 = 10$$

$$S \rightarrow A \rightarrow B ; f(n) = g(n) + h(n) = 3 + 4 = 7$$

$$S \rightarrow A \rightarrow C ; f(n) = g(n) + h(n) = 2 + 2 = 4$$

Step 4:



$$S \rightarrow A \rightarrow C \rightarrow D ;$$

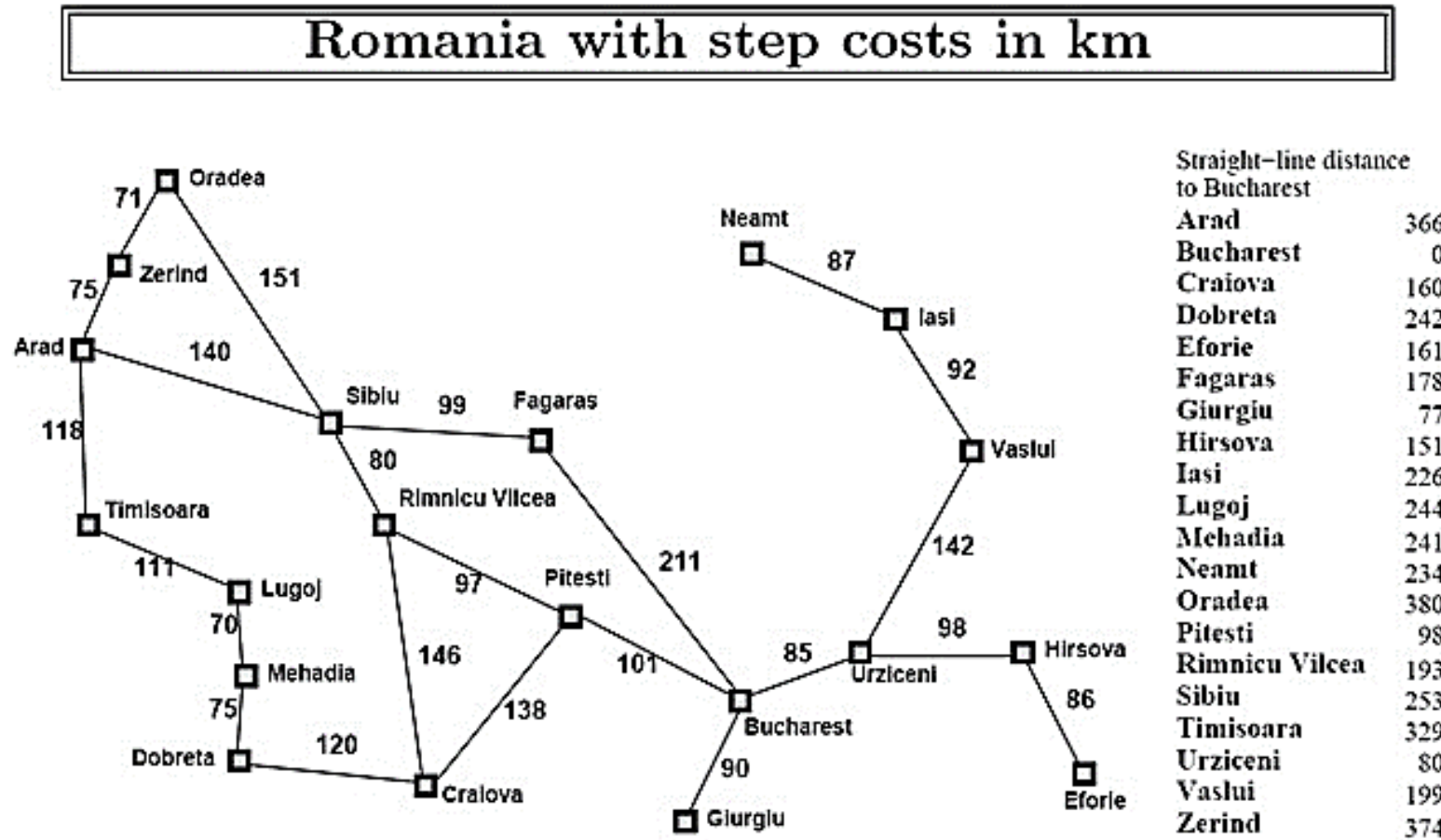
$$f(n) = g(n) + h(n) = 5 + 6 = 11$$

$$S \rightarrow A \rightarrow C \rightarrow G ;$$

$$f(n) = g(n) + h(n) = 6 + 0 = 6$$

A* Search Algorithm

Consider the following route map



- ✓ Find route from city Arad to city Bucharest
 - ✓ Here Goal state is Bucharest, we will need to know the straight line distances to Bucharest.

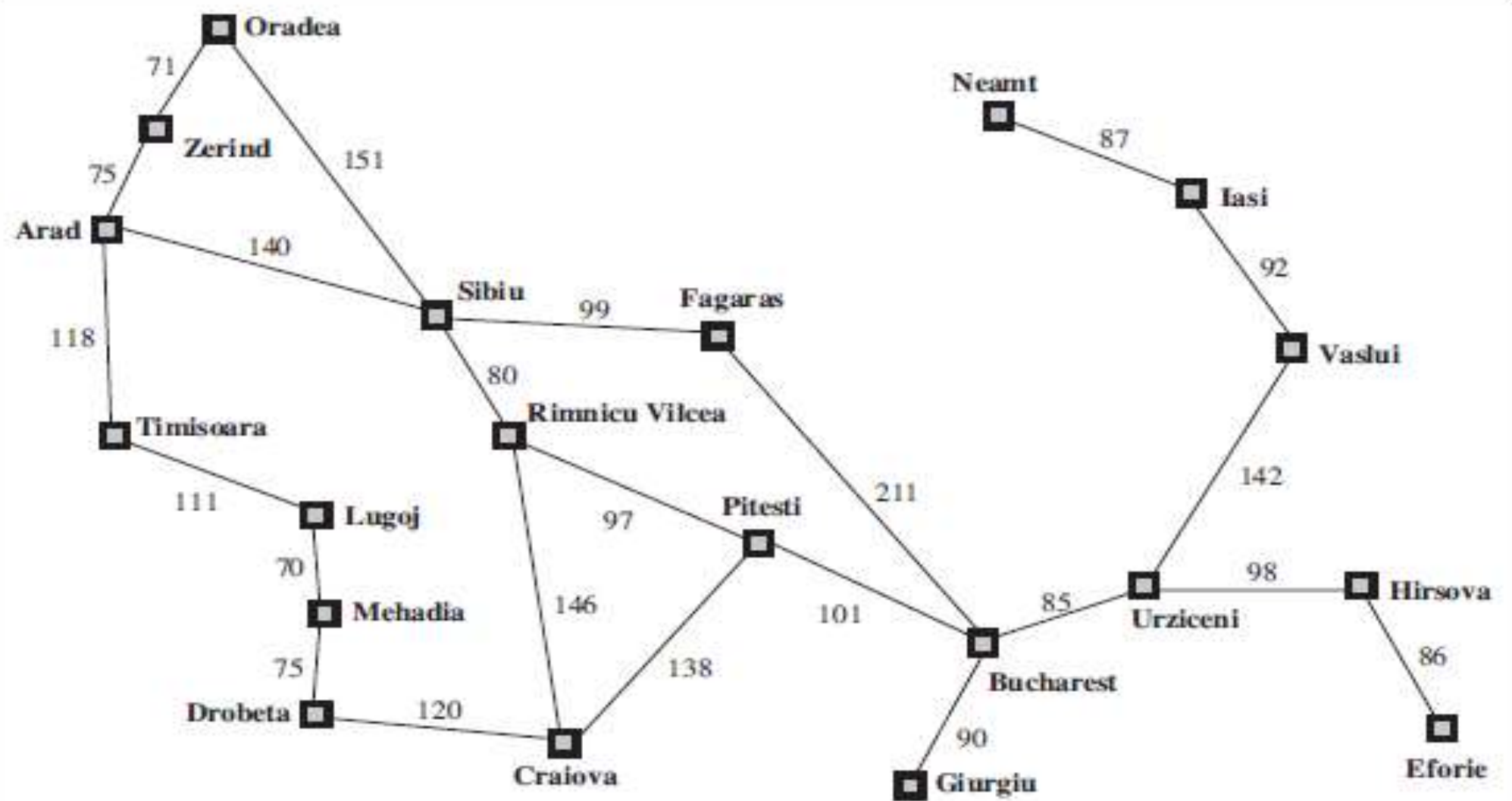


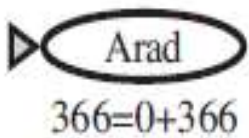
Figure 3.2 A simplified road map of part of Romania.

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

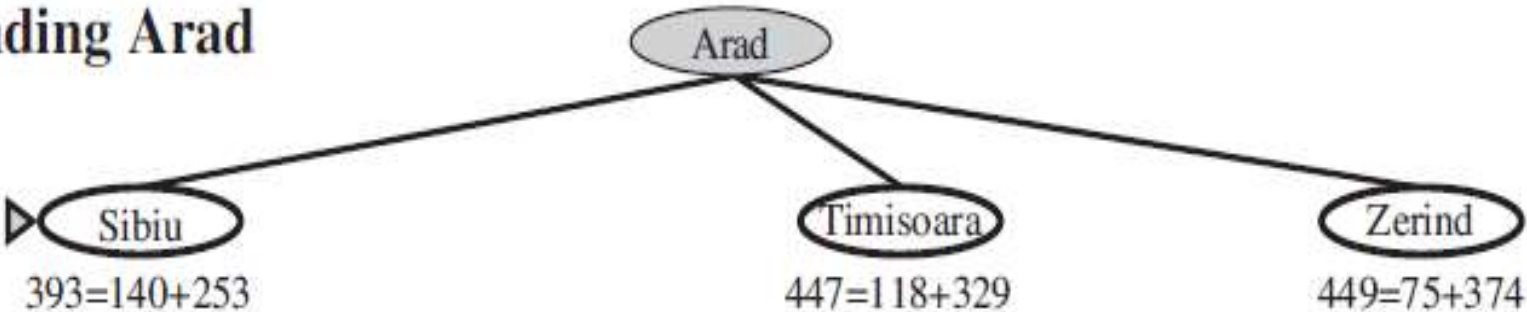
Figure 3.22 Values of h_{SLD} —straight-line distances to Bucharest.

Step 1: Open List: {(A,366)}

(a) The initial state

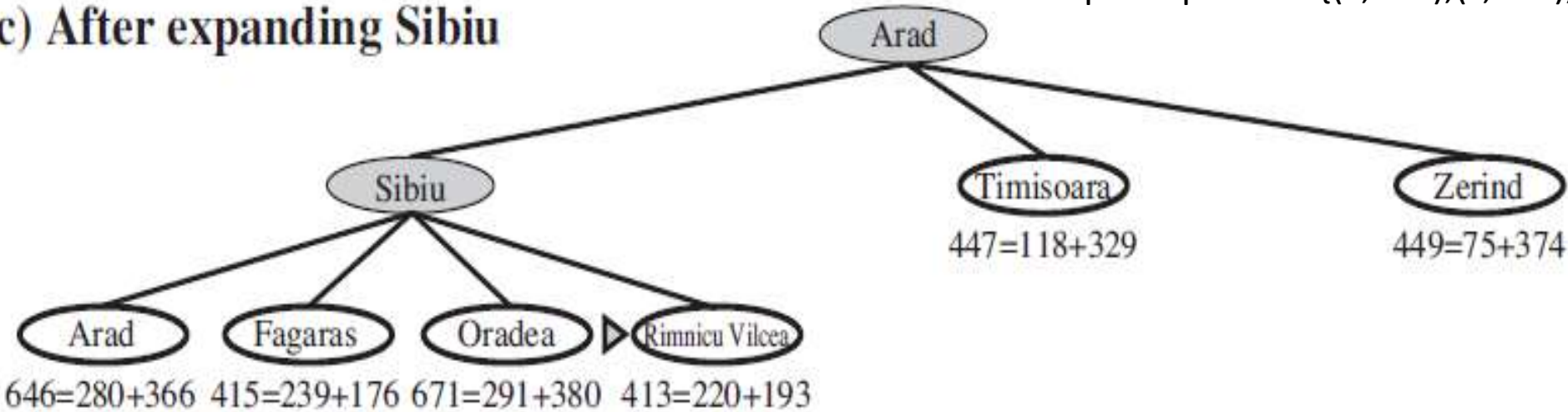


(b) After expanding Arad

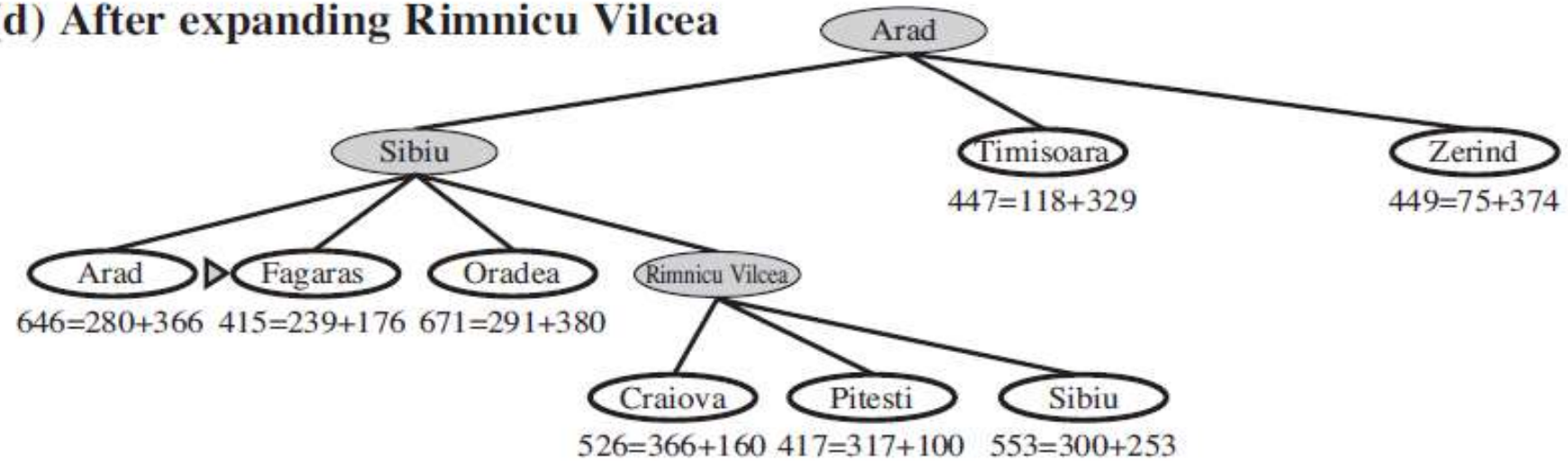


Step 2: Open List: {(S,393),(T,447),(Z,374)}

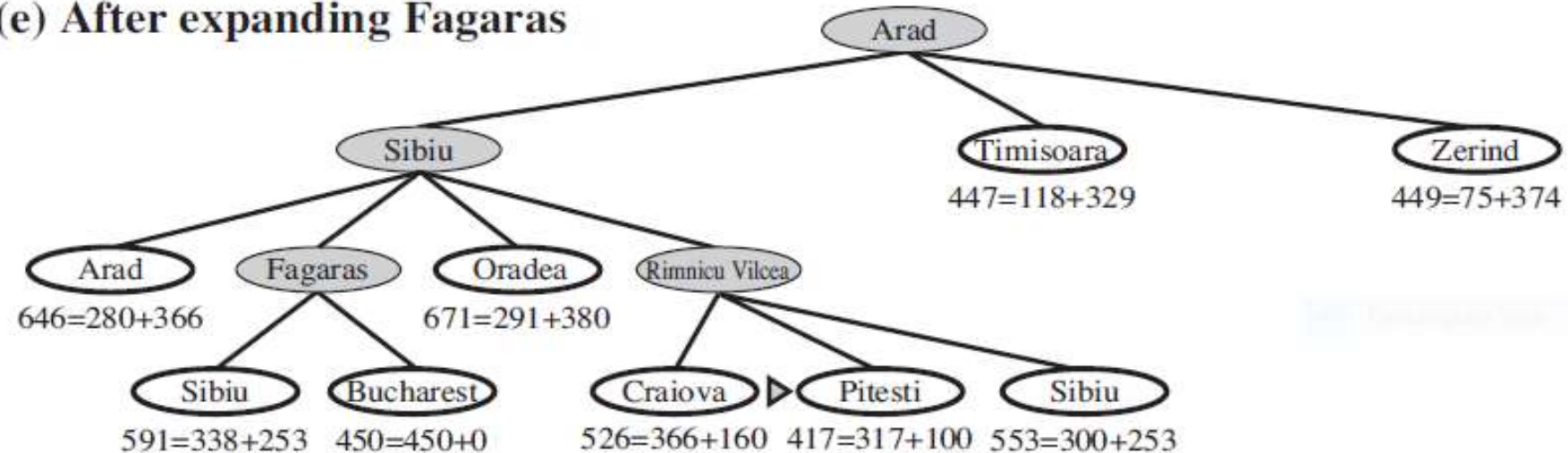
(c) After expanding Sibiu



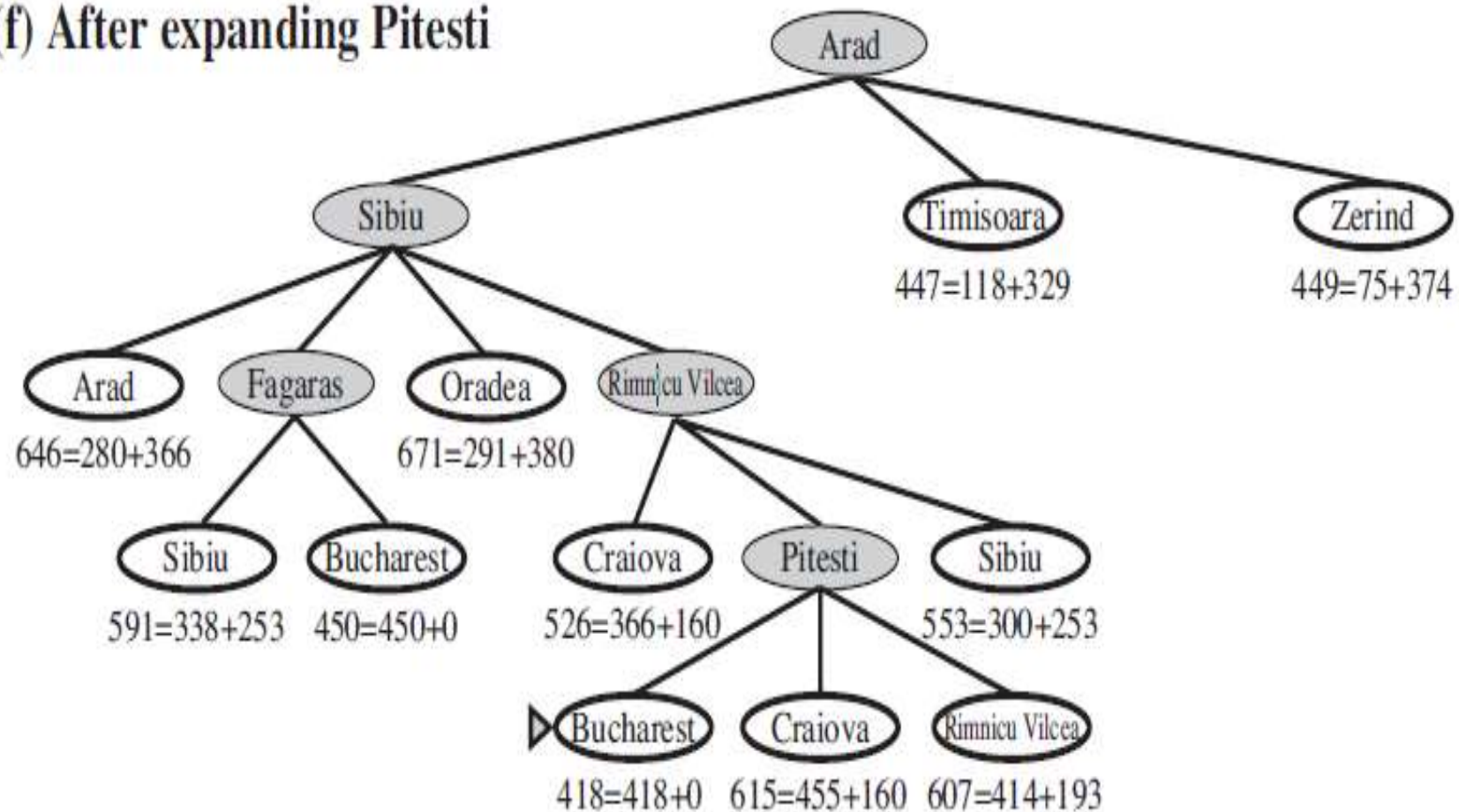
(d) After expanding Rimnicu Vilcea

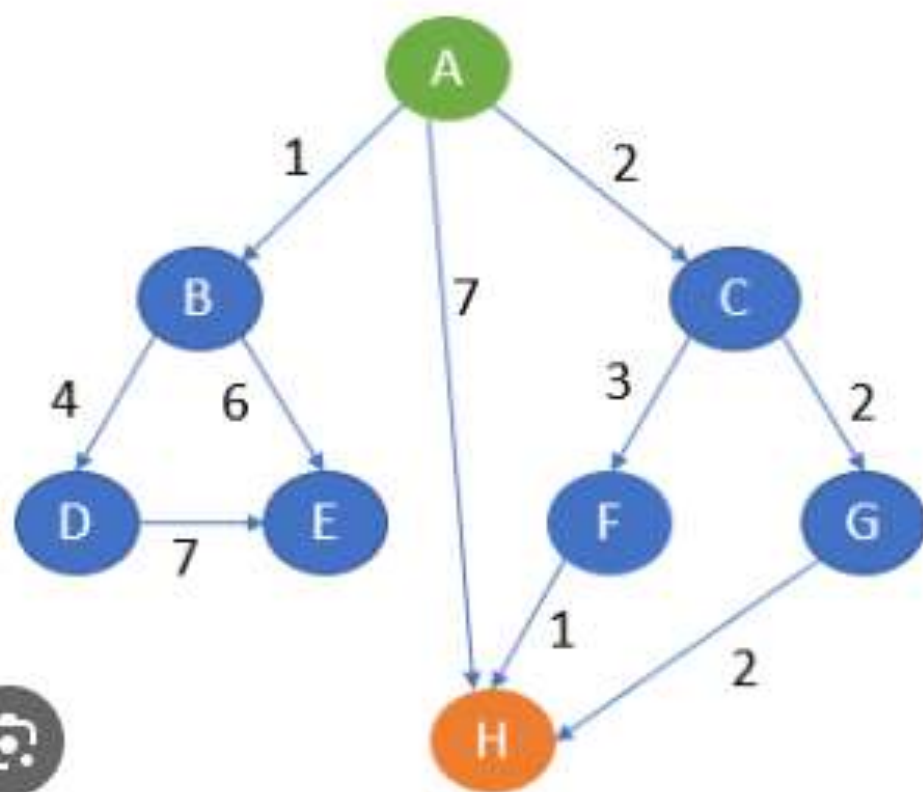


(e) After expanding Fagaras



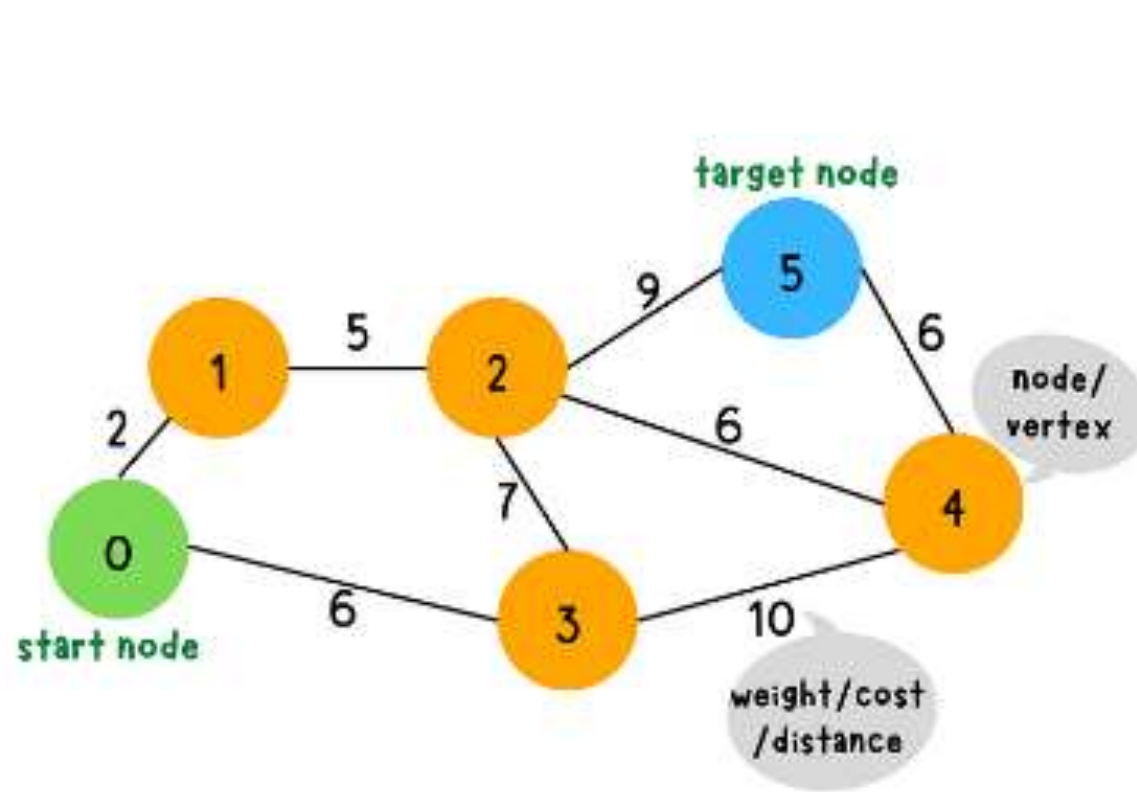
(f) After expanding Pitesti





NODES	HEURISTICS
A	5
B	3
C	4
D	2
E	6
F	3
G	1
H	0

Example



$h(n)$

node	Heuristic distance to target node
0	20
1	16
2	6
3	10
4	4
5	0

Properties :

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, Consistency.**

Admissible: the first condition requires for optimality is that $h(n)$ should be an admissible heuristic for A* tree search. An admissible heuristic is one that never overestimates the cost to reach the goal.

Consistency: Second required condition is consistency for only A* graph-search. heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by any action a , the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n' :

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

- 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 $O(b^d)$

Advantages:

- it always expands the nodes with optimal solution
- it is optimal among all other algorithms for any given heuristic search
- it generates fewer nodes when compares to remaining algorithms

Disadvantage

- it is not practical for large scale problems
- It keeps all generated nodes in memory and usually runs out of space, we can overcome this problem by memory bounded heuristic

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.

Problem Reduction

- A solution to a problem can be obtained by decomposing it into smaller sub-problems. Each of this sub-problem can then be solved to get its sub-solution. These sub-solutions can then be recombined to get a solution as a whole. That is called is Problem Reduction.
- This method generates arc which is called as AND arcs. One AND arc may point to any number of successor nodes, all of which must be solved for an arc to point to a solution

AND-OR graphs

- AND-OR graph (or tree) is useful for representing the solution of problems that can be solved by decomposing them into a set of smaller problems, all of which must then be solved.
- This decomposition or reduction generates arcs that we call AND arcs.
- One AND arc may point to any numbers of successor nodes. All of which must then be solved in order for the arc to point solution.
- In order to find solution in an AND-OR graph we need an algorithm similar to best-first search but with the ability to handle the AND arcs appropriately.
- In AI's [problem reduction search](#), "FUTILITY" is a value representing a threshold to stop a search when the estimated cost of finding a solution exceeds a predefined limit, effectively indicating the search is no longer worthwhile. This threshold, called the futility value, is set to a level where any further expenditure of computational resources is deemed impractical or unnecessary for finding a solution.

Problem Reduction algorithm:

- 1. Initialize the graph to the starting node.
- 2. Loop until the starting node is labelled SOLVED or until its cost goes above FUTILITY:
 - (i) Traverse the graph, starting at the initial node and following the current best path and accumulate the set of nodes that are on that path and have not yet been expanded.
 - (ii) Pick one of these unexpanded nodes and expand it. If there are no successors, assign FUTILITY as the value of this node. Otherwise, add its successors to the graph and for each of them compute $f'(n)$. If $f'(n)$ of any node is 0, mark that node as SOLVED.
 - (iii) Change the $f'(n)$ estimate of the newly expanded node to reflect the new information provided by its successors. Propagate this change backwards through the graph. If any node contains a successor arc whose descendants are all solved, label the node itself as SOLVED

Step cost of each action is 1

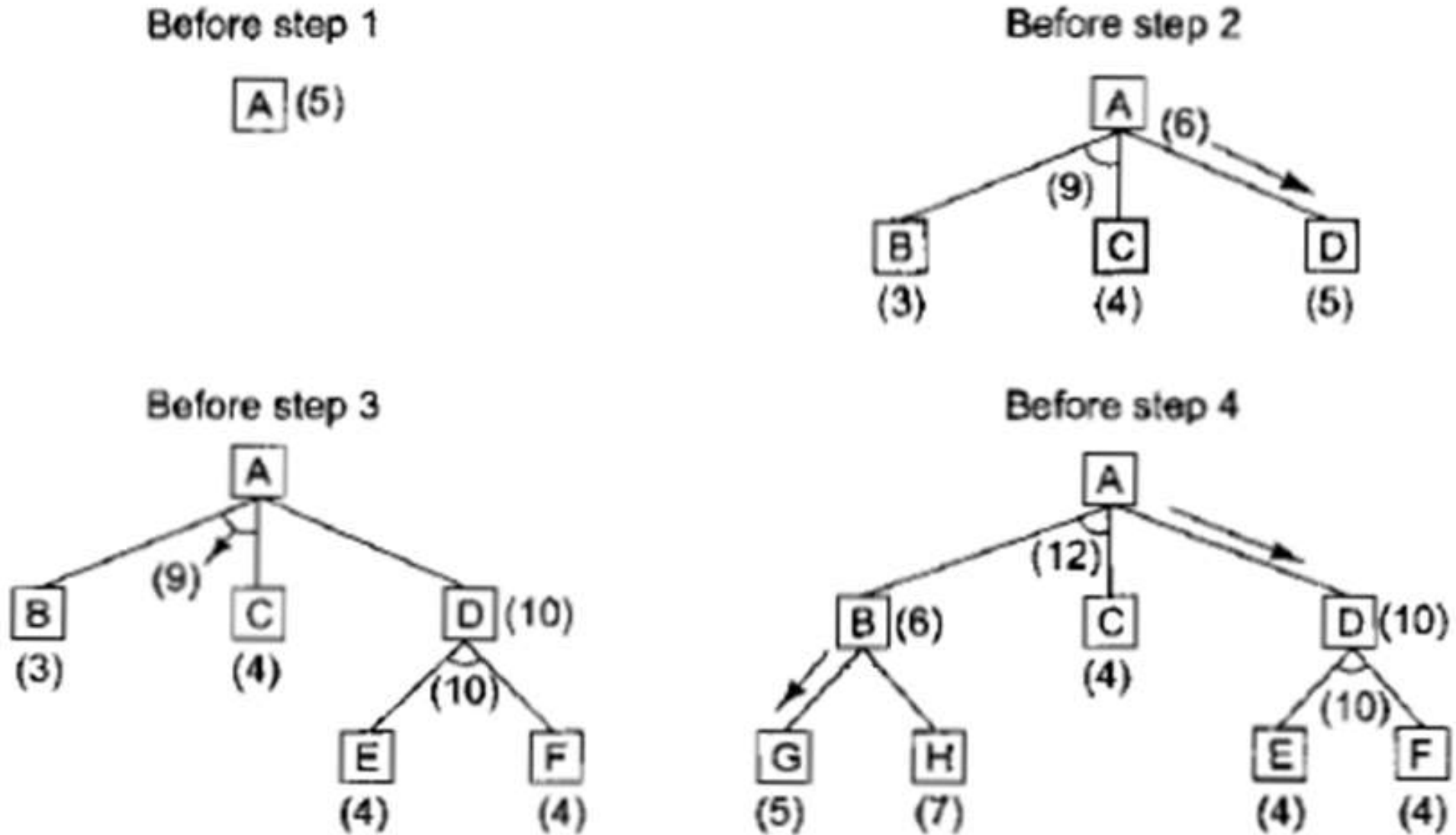


Fig. 3.8 *The Operation of Problem Reduction .*

- Step cost of each path is 1.
- At step 1, A is the only node. It is expanded, gives nodes B, C, and D. The arrow mark to D is indicating as the most promising path from A, since it costs 6 compared to B and C, which costs 9.
- In step 2, node D is chosen for expansion. This process produces the AND arc to E and F, with a combined cost estimate of 10. so we update the f' value of D to 10.
- Going back one more level, we see that this makes the AND arc B-C better than the arc to D. so it is labeled as the current best path.
- At step 3, we will have to expand both B and C eventually, so let's choose to explore B first. This generates two new arcs, the ones to G and to H. Propagating their f' values backward, we update f' of B to 6. This requires updating the cost of the AND arc B-C to 12 ($6+4+2$). After doing that, the arc to D is again the better path from A, so either node E or node F will be chosen for expansion at step 4.
- This process continues until either a solution is found or all paths have led to dead ends, indicating that there is no solution.

AO* Algorithm

- The AO* method **divides** any given difficult **problem into a smaller group** of problems that are then resolved **using the AND-OR** graph concept. AND OR graphs are specialized graphs that are used in problems that can be divided into smaller problems.

AO* Algorithm

- The AO* algorithm is based on AND-OR graphs to break complex problems into smaller ones and then solve them.
- The AND side of the graph represents those tasks that need to be done with each other to reach the goal, while the OR side stands alone for a single task. Let's see an example below:



- The start state and the target state are already known in the knowledge-based search strategy known as the **AO* algorithm**, and the best path is identified by heuristics.
- The informed search technique considerably reduces the algorithm's **time complexity**.
- The AO* algorithm is far more effective in searching AND-OR trees **than** the A* algorithm.

Working of AO* algorithm:

- The evaluation function in AO* looks like this:
 $f(n) = g(n) + h(n)$

here,

$f(n)$ = The actual cost of traversal.

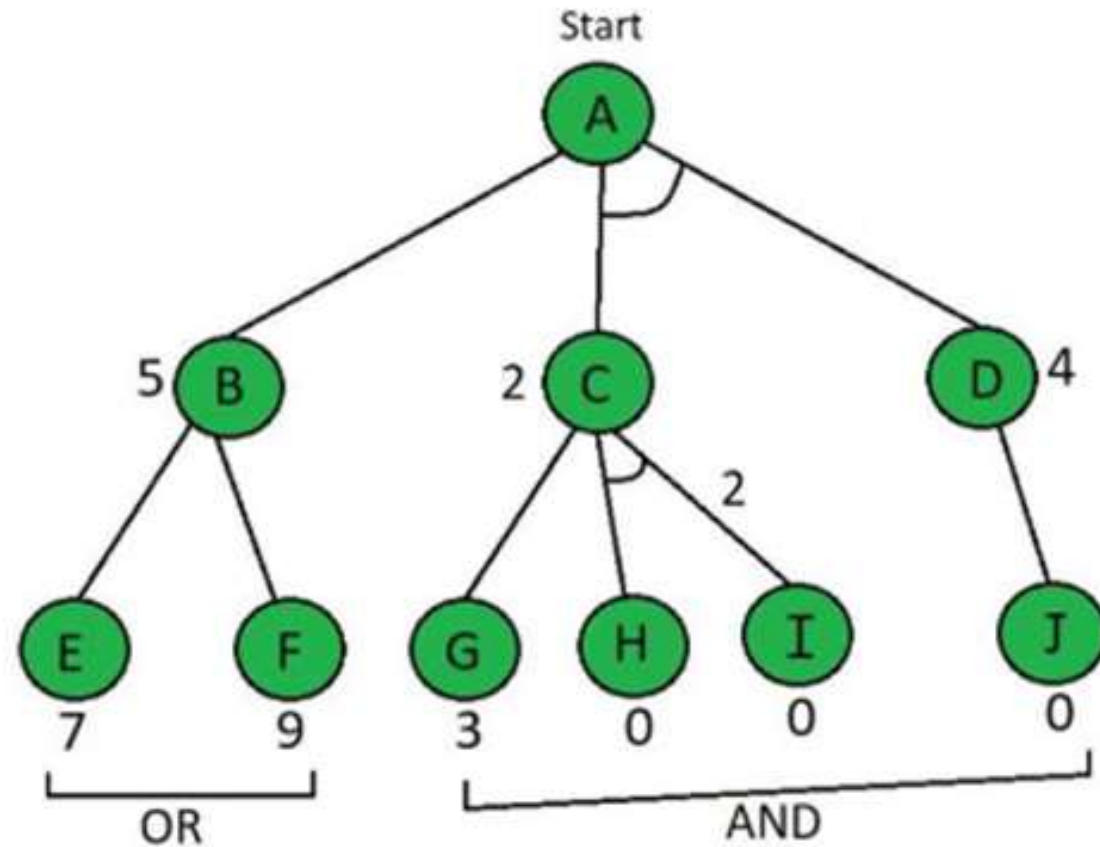
$g(n)$ = the cost from the initial node to the current node.

$h(n)$ = estimated cost from the current node to the goal state.

Difference between the A* Algorithm and AO* algorithm

- A* algorithm and AO* algorithm both works on the **best first search**.
- They are both **informed search** and works on given heuristics values.
- **A*** always **gives** the **optimal solution** but AO* doesn't guarantee to give the optimal solution.
- Once AO* got a solution **doesn't explore** all possible paths but A* explores all paths.
- When compared to the A* algorithm, the AO* algorithm uses **less memory**.
- opposite to the A* algorithm, the AO* algorithm cannot go into an endless **loop**.
- A* represents an OR graph algorithm that is used to find a single solution.
- AO* represents an AND_OR graph algorithms that is used to find more than one solution by ANDing more than one branch.

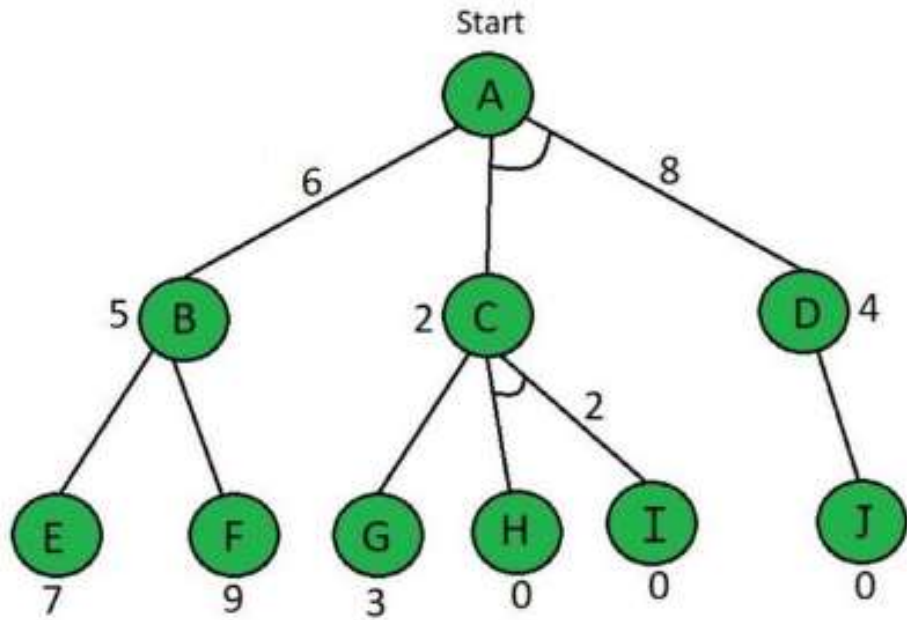
Example:



AO* Algorithm – Question tree

Here in the above example below the Node which is given is the heuristic value i.e $h(n)$.
Edge length is considered as **1**.

- Step1



With help of $f(n) = g(n) + h(n)$ evaluation function,

Start from node A,

$$f(A \rightarrow B) = g(B) + h(B)$$

$$= 1 + 5$$

for path cost

$$= 6$$

.....here $g(n)=1$ is taken by default

$$f(A \rightarrow C+D) = g(c) + h(c) + g(d) + h(d)$$

$$= 1 + 2 + 1 + 4$$

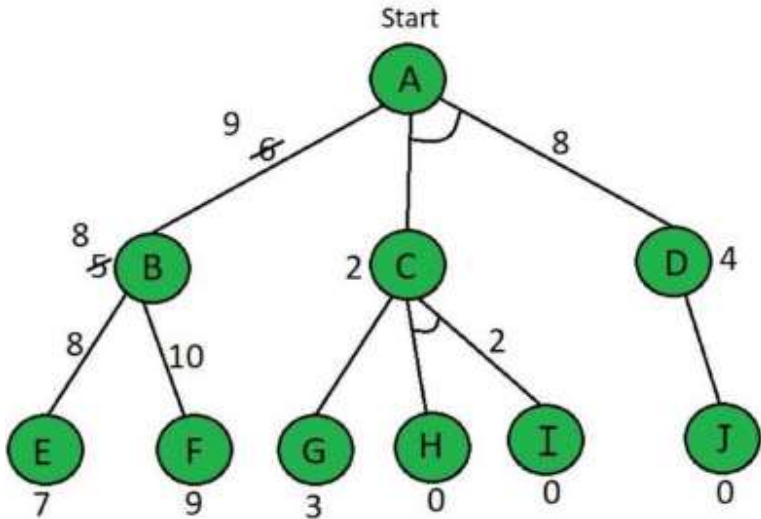
they are in AND

$$= 8$$

.....here we have added **C & D** because

So, by calculation $A \rightarrow B$ path is chosen which is the minimum path, i.e
 $f(A \rightarrow B)$

• Step2



According to the answer of step 1, explore node B
Here the value of E & F are calculated as follows,

$$\begin{aligned} f(B \rightarrow E) &= g(e) + h(e) \\ f(B \rightarrow E) &= 1 + 7 \\ &= 8 \end{aligned}$$

$$\begin{aligned} f(B \rightarrow f) &= g(f) + h(f) \\ f(B \rightarrow f) &= 1 + 9 \\ &= 10 \end{aligned}$$

So, by above calculation $B \rightarrow E$ path is chosen which is minimum path, i.e $f(B \rightarrow E)$

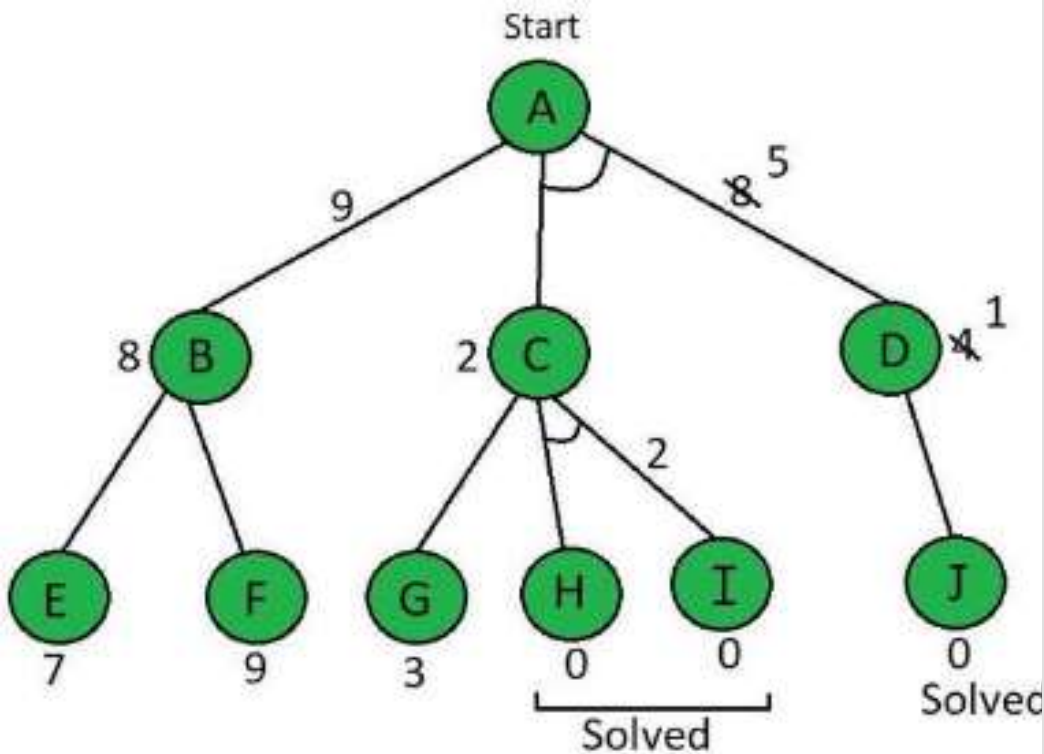
because **B's** heuristic value is different from its actual value The heuristic is

updated and the minimum cost path is selected. The minimum value in our situation is **8**.

Therefore, the heuristic for **A** must be updated due to the change in **B's** heuristic.

So we need to calculate it again.

- Step 3



By comparing $f(A \rightarrow B)$ & $f(A \rightarrow C+D)$

$f(A \rightarrow C+D)$ is shown to be **smaller**. i.e $8 < 9$

Now explore $f(A \rightarrow C+D)$

So, the current node is **C**

$$f(C \rightarrow G) = g(g) + h(g)$$

$$f(C \rightarrow G) = 1 + 3$$

$$= 4$$

$$f(C \rightarrow H+I) = g(h) + h(h) + g(i) + h(i)$$

$$f(C \rightarrow H+I) = 1 + 0 + 1 + 0$$

$$= 2$$

.....here we have added **H & I** because

they are in **AND**

$f(C \rightarrow H+I)$ is selected as the path with the lowest cost and the heuristic is also left unchanged

because it matches the actual cost. Paths H & I are solved because the heuristic for those paths is **0**,

but Path $A \rightarrow D$ needs to be calculated because it has an **AND**.

$$f(D \rightarrow J) = g(j) + h(j)$$

$$f(D \rightarrow J) = 1 + 0$$

$$= 1$$

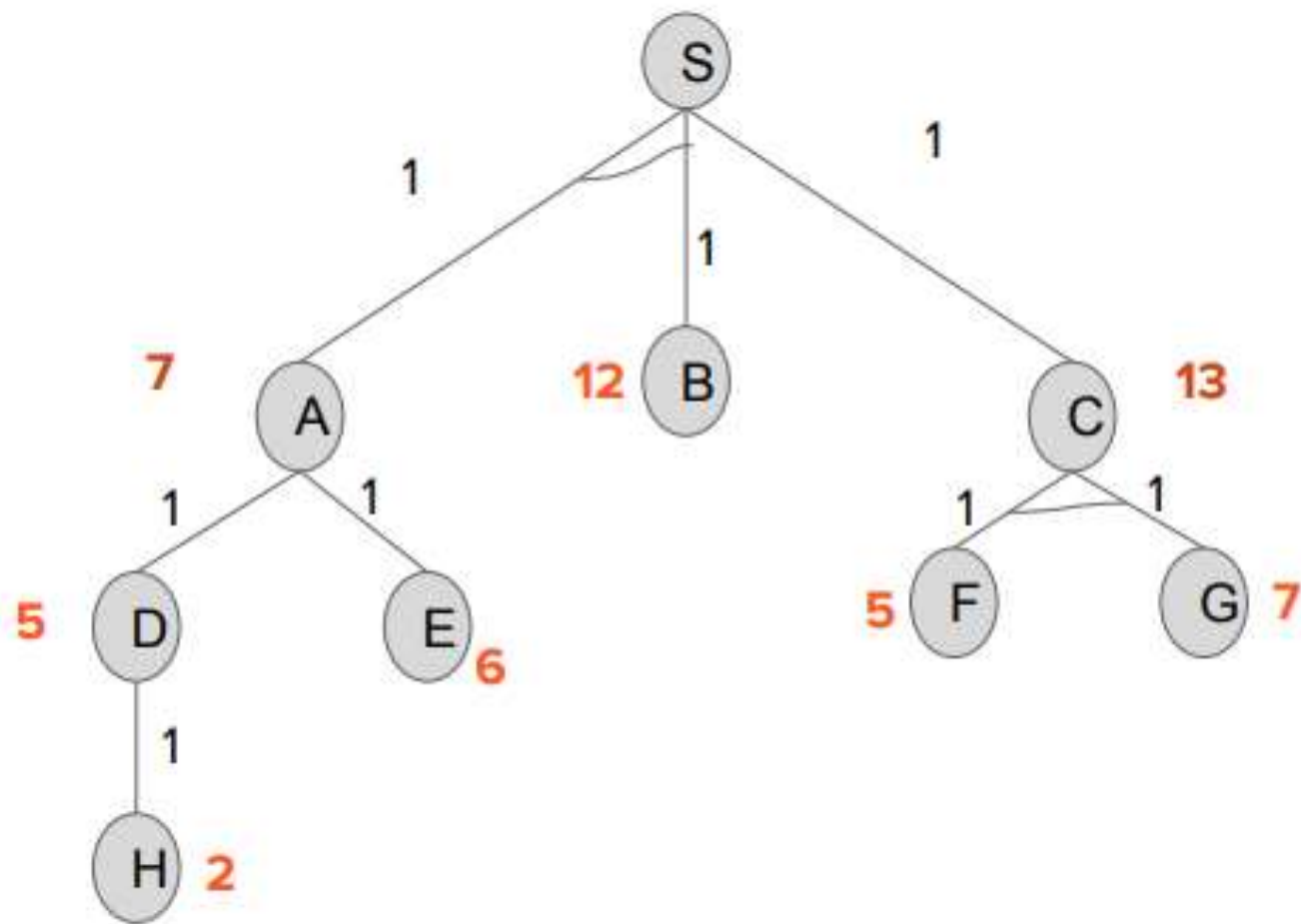
the heuristic of node D needs to be updated to 1.

$$\begin{aligned}f(A \rightarrow C+D) &= g(c) + h(c) + g(d) + h(d) \\&= 1 + 2 + 1 + 1 \\&= 5\end{aligned}$$

as we can see that path **f(A→C+D)** is get solved and this **tree has become a solved tree** now.

In simple words, the main flow of this algorithm is that we have to find **firstly level 1st** heuristic value and **then level 2nd** and after that **update the values** with going upward means towards the root node.

AO* Example 1



AO* Example 1

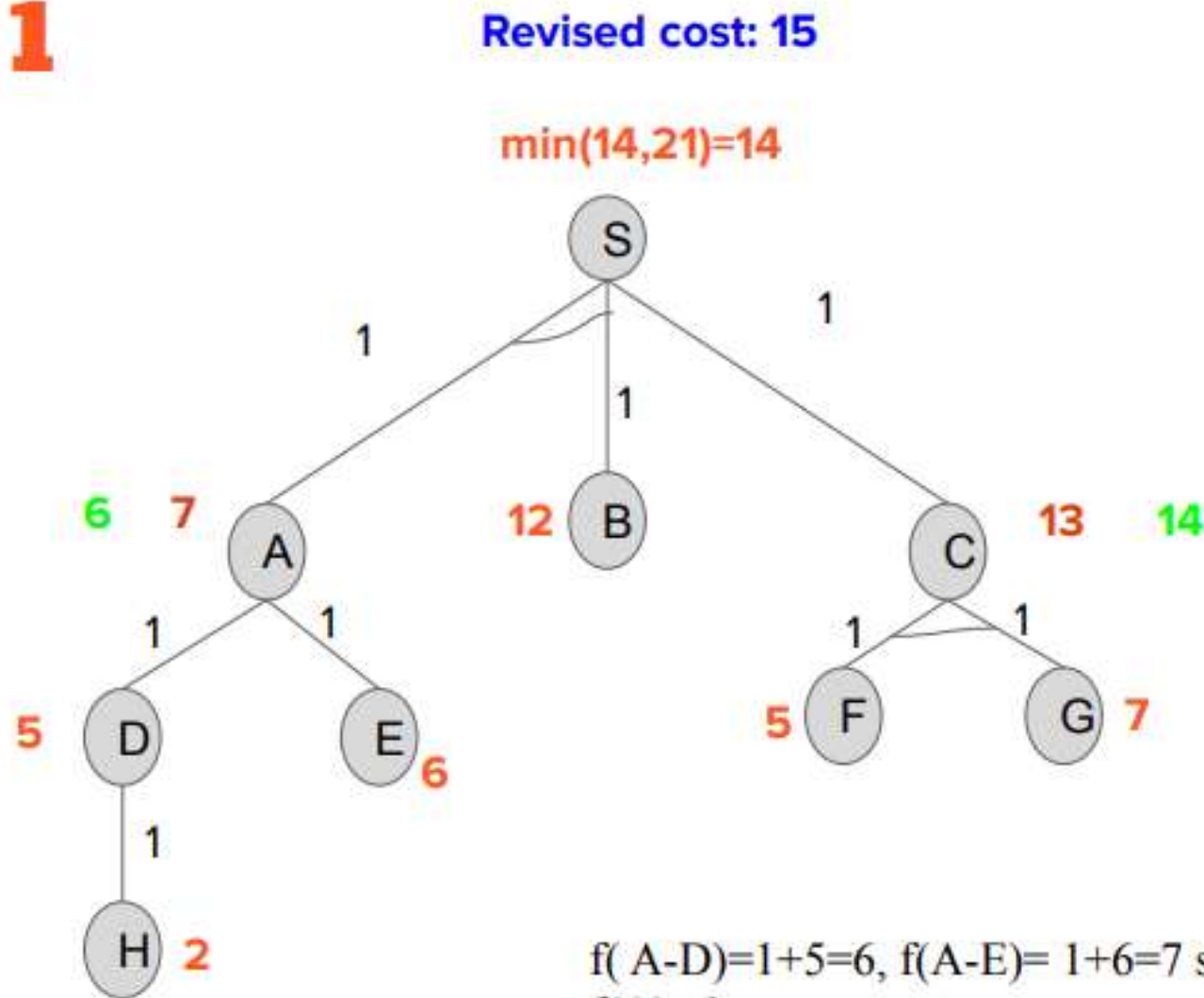
$$f(n) = g(n) + h(n)$$

$$\text{Path-1: } f(S-C) = 1 + 13 = 14$$

$$\text{Path-2: } f(S-A-B) = 1 + 1 + 7 + 12 = 21$$

$$f(C-F-G) = 1 + 1 + 5 + 7 = 14$$

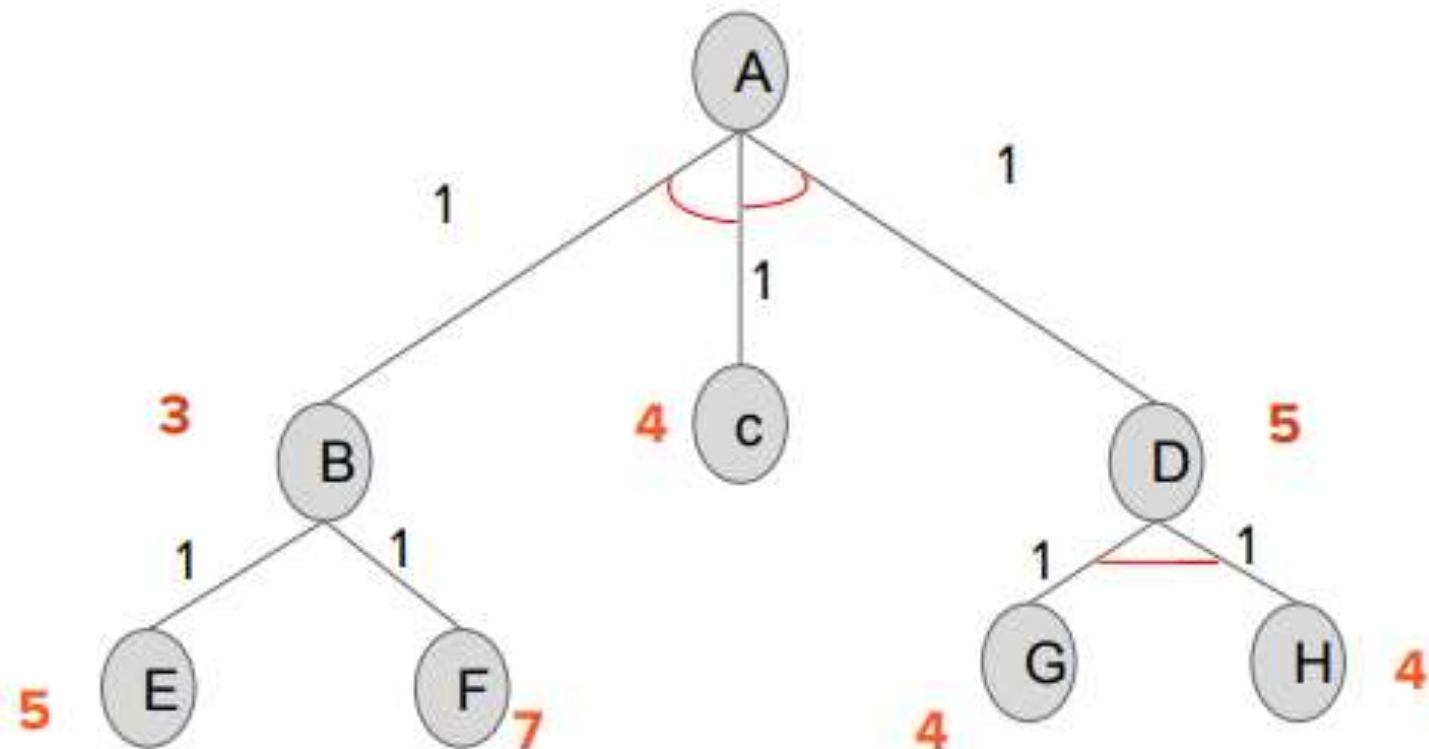
$$f(S-C) = 1 + 14 = 15 \text{ (revised)}$$



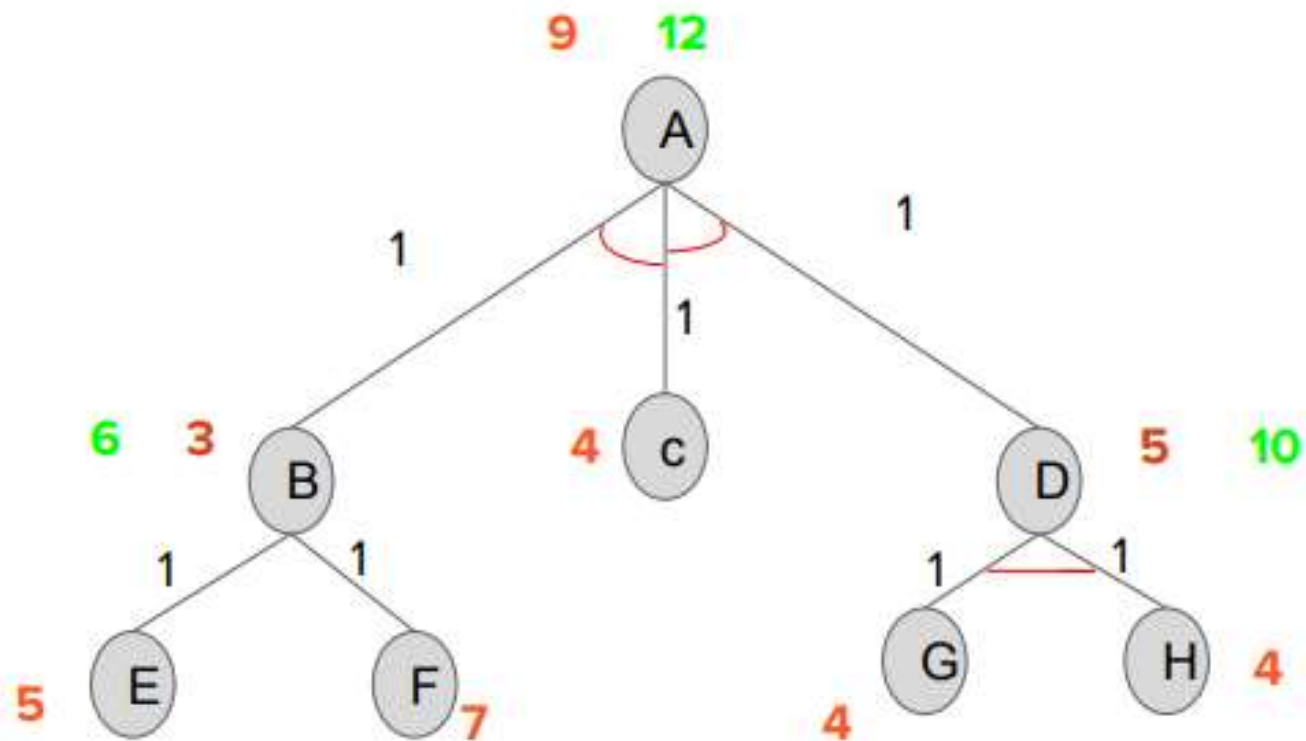
$$f(A-D) = 1 + 5 = 6, f(A-E) = 1 + 6 = 7 \text{ so revised } f(A) = 6$$

$$\text{Revised } (S-A-B) = 1 + 6 + 1 + 12 = 20 \quad \text{X}$$

AO* Example 2



AO* Example 2



Path -1: $f(A-B-C) = 1+1+3+4 = 9$

$f(B-E) = 1+5 = 6$

$f(B-F) = 1+7 = 8$

$f(A-B-C) = 1+1+6+4 = 12$

Path-2: $f(A-C-D) = 1+1+4+5 = 11$

$f(D-G-H) = 1+1+4+4 = 10$

$f(A-C-D) = 1+1+4+10 = 16$

Advantages and Disadvantages of AO* Algorithm

Advantages

- **Adaptability:**
AO* can adapt to changing requirements and computational resources, making it suitable for real-time systems.
- **Incremental Solutions:**
It provides incremental solutions, allowing users to make progress while the search continues.
- **Optimistic Estimates:**
The use of optimistic cost estimates can guide the search efficiently.
- **Heuristic Guidance:**
Like A*, it benefits from heuristic guidance, improving search efficiency.

Disadvantages

- **Quality vs. Time Trade-off:**

AO* sacrifices optimality for adaptability. It may not always find the absolute best solution but provides a good compromise between quality and time.

- **Complexity:**

Implementing AO* can be more complex than simpler algorithms due to its adaptability and nature.

- **Heuristic Quality:**

The effectiveness of AO* heavily depends on the quality of the heuristic function. Poor heuristics can lead to suboptimal solutions.

Real-life Applications of AO* Algorithm

AO* finds applications in various fields:

- **Robotics:**

- AO* is widely used in robotic path planning. Robots can navigate complex environments while continuously improving their routes.

- **Video Games:**

- In video games, AO* is used for character path finding, ensuring that game characters move efficiently and avoid obstacles.

- **Network Routing:**

- In computer networking, AO* helps in finding optimal routes for data packets in dynamic networks.

- **Autonomous Vehicles:**

- Autonomous vehicles use AO* for real-time route planning, taking into account traffic conditions and obstacles.

- **Natural Language Processing:**

- AO* is used in parsing and grammar-checking algorithms to generate syntactically correct sentences incrementally.

- **Resource Management:**

- It is used in resource allocation and scheduling, such as allocating resources in cloud computing.

A*

It represents an OR graph algorithm that is used to find a single solution (either this or that).

It is a computer algorithm which is used in path-finding and graph traversal. It is used in the process of plotting an efficiently directed path between a number of points called nodes.

In this algorithm you traverse the tree in depth and keep moving and adding up the total cost of reaching the cost from the current state to the goal state and add it to the cost of reaching the current state.

AO*

It represents an AND-OR graph algorithm that is used to find more than one solution by ANDing more than one branch.

In this algorithm you follow a similar procedure but there are constraints traversing specific paths.

When you traverse those paths, cost of all the paths which originate from the preceding node are added till that level, where you find the goal state regardless of the fact whether they take you to the goal state or not.

- **AO*** Performance
- **The AO* algorithm is not optimal** because it stops as soon as it finds a solution and does not explore all the paths.
- However, **AO* is complete**, meaning it finds a solution, if there is any, and does not fall into an infinite loop. Moreover, the AND feature in this algorithm reduces the demand for memory.
- The space complexity comes in polynomial order, while the time complexity is $O(b^m)$, where b stands for branching and m is the maximum depth or number of levels in the search tree.

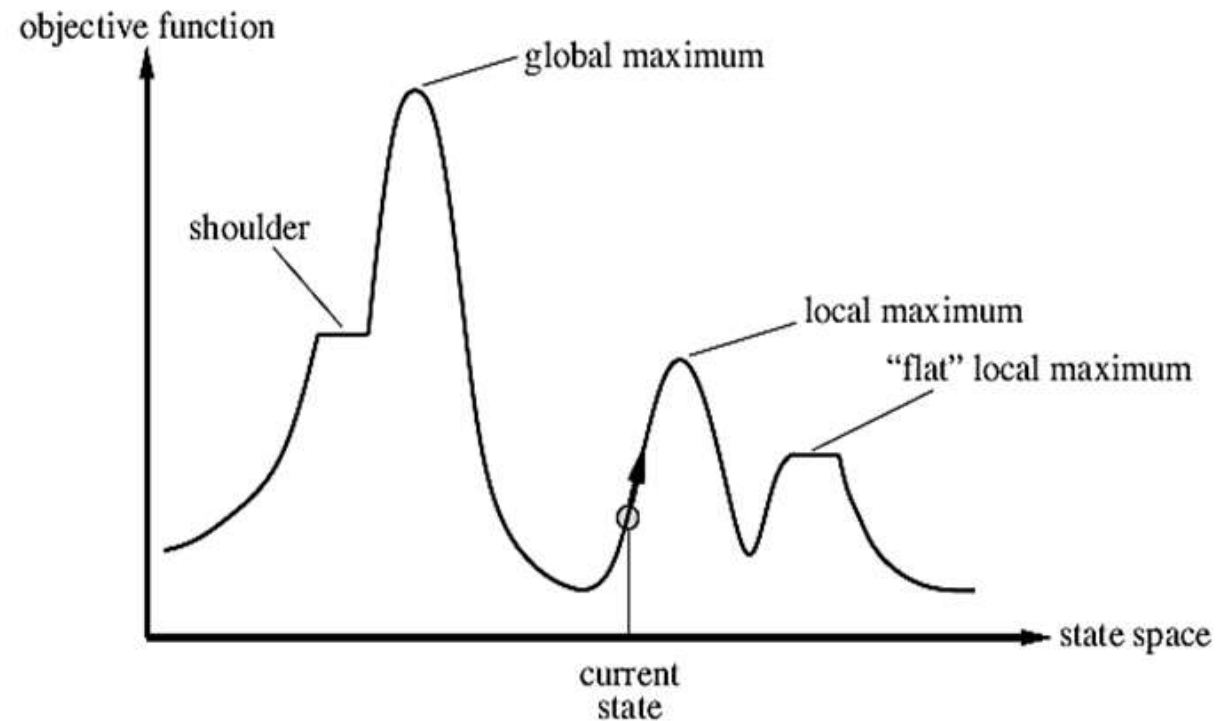
Local search algorithms (Hill-climbing search algorithm)

- ✓ The search algorithms that we have seen so far, explores search space systematically. i.e. when a goal state is found, the path to the goal state constitutes the solution.
- ✓ In many problems, however the path to the goal state is irrelevant.
 - ✓ Example : 8 queen problem
- ✓ If the path to goal state doesn't matter then, we can use local search algorithms.
- ✓ Local Search Algorithms:
 - ✓ Local search algorithms operates using a single current state, and generally move only to neighbours of that state.
 - ✓ The path followed by search is not maintained
 - ✓ Advantages:
 - ✓ It uses less memory
 - ✓ Useful for solving pure optimization problems
 - ✓ It can find reasonable solution in a large or infinite state space for which systematic algorithms are not suitable.

Local search algorithms and optimization problems

- ✓ To understand local search, we consider the **state-space landscape**.
- ✓ A landscape has both “location” (defined by the state) and “elevation” (defined by the value of the heuristic cost function or objective function).
- ✓ If elevation corresponds to cost, then the aim is to find the lowest valley—a **global minimum**
- ✓ if elevation corresponds to an objective function, then the aim is to find the highest peak—a **global maximum**.
- ✓ Local search algorithms explore this landscape.
- ✓ A **complete** local search algorithm always finds a goal if one exists;
- ✓ an **optimal** algorithm always finds a global minimum/maximum.

Local search algorithms and optimization problems



A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow.

Hill-climbing search algorithm

- ✓ It is a local search algorithm which continuously moves in the direction of increasing value to find best solution to the problem (uphill).
- ✓ It terminates when it reaches a peak value where no neighbour has a higher value.
- ✓ The algorithm does not maintain a search tree, so the data structure for the current node need only to record the state and the value of the objective function.
- ✓ Hill climbing does not look ahead beyond the immediate neighbors of the current state.
- ✓ Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about

- ✓ **Ex. 8-queens problem**, Local search algorithms typically use a **complete-state formulation**, where each state has 8 queens on the board, one per column.
- ✓ The successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has $8 \times 7 = 56$ successors).
- ✓ The heuristic cost function h is the number of pairs of queens that are attacking each other, either directly or indirectly.
- ✓ The global minimum of this function is zero, which occurs only at perfect solutions.

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

current $\leftarrow$ MAKE-NODE(*problem*.INITIAL-STATE)

loop do

neighbor $\leftarrow$ a highest-valued successor of *current*

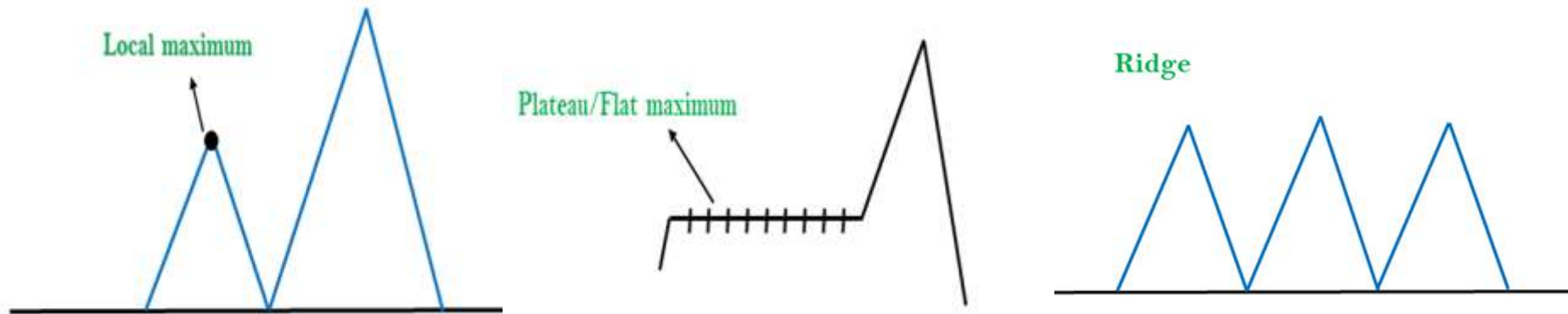
if *neighbor*.VALUE $\leq$ *current*.VALUE **then return** *current*.STATE

current $\leftarrow$ *neighbor*

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h .

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 neighbouring states, but there is another state also present which is higher than the local maximum.
2. **Plateau:** A plateau is the flat area of the search space landscape. it can be a flat local maximum, from which no uphill exit exists, or shoulder from which progress is possible. A hill-climbing search might be lost in the plateau area.
3. **Ridges:** A ridge is a special form of the sequence of local maximum. Which is difficult for the algorithm to navigate



Problems in Different Regions in Hill climbing

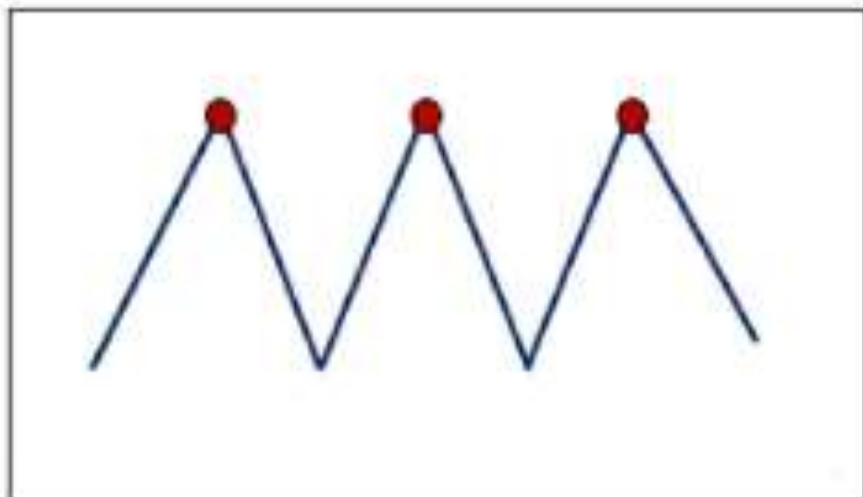
- **1. Local Maximum**

- The algorithm terminates when the current node is local maximum as it is better than its neighbours. However, there exists a global maximum where objective function value is higher
- **Solution:** Back Propagation can mitigate the problem of Local maximum as it starts exploring alternate paths when it encounters Local Maximum

Problems in Different Regions in Hill climbing

- **2. Ridge**

- Ridge occurs when there are multiple peaks and all have the same value or in other words, there are multiple local maxima which are same as global maxima



- **Solution:** Ridge obstacle can be solved by moving in several directions at the same time

Problems in Different Regions in Hill climbing

- **3. Plateau**

- Plateau is the region where all the neighbouring nodes have the same value of objective function so the algorithm finds it hard to select an appropriate direction.



- **Solution:** Plateau obstacle can be solved by taking making a big jump from the current state which will land you in non-plateau region

Types of Hill Climbing Algorithm:

1. Simple Hill Climbing

- ✓ It only evaluates the neighbour node state one at a time and selects the first one which optimizes current cost and set it as a current state. It only checks its one successor state, and if it finds better than the current state, then move else be in the same state.

2. Steepest-Ascent hill climbing

- ✓ This algorithm examines all the neighbouring nodes of the current state and selects one neighbour node which is closest to the goal state. This algorithm consumes more time as it searches for multiple neighbours

3. Stochastic hill climbing

- ✓ Stochastic hill climbing chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move.

4. First-choice hill climbing

- It implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.
- This is a good strategy when a state has many (e.g., thousands) of successors.

5. Random-restart hill climbing

- It adopts the well-known adage, “If at first you don’t succeed, try, try again.”
- It conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found.
- If each hill-climbing search has a probability p of success, then the expected number of restarts required is $1/p$.

Advantage of Hill Climbing Algorithm

- Hill Climbing is very useful in **routing-related problems** like **Travelling Salesmen Problem**, Job Scheduling, Chip Designing, and Portfolio Management
- It is good in solving **the optimization problem** while using only limited computation power
- It is more **efficient than other search** algorithms

Advantages and Disadvantages of Hill Climbing Algorithms

Advantages	Disadvantages
Simplicity: The algorithm is straightforward to understand and implement.	Susceptibility to Local Optima: The algorithm can become stuck at locally optimal solutions that aren't the best overall.
Memory Efficiency: It's memory-efficient, maintaining only the current state's data.	Limited Exploration: Its tendency to focus on the immediate vicinity limits its exploration, potentially overlooking globally optimal solutions.
Rapid Convergence: It often converges swiftly to a solution, which is beneficial in scenarios where time is critical.	Dependence on Initial State: The quality and effectiveness of the solution found heavily depend on the starting point.

Applications of Hill Climbing Algorithm

- **Marketing:** The Hill Climbing algorithm is a game-changer for marketing managers crafting top-notch strategies. It's instrumental in solving the classic Traveling-Salesman problems, optimizing sales routes, and reducing travel time. This leads to more efficient sales operations and better resource utilization.
- **Robotics:** The algorithm plays a critical role in robotics, enhancing the performance and coordination of various robotic components. This leads to more sophisticated and efficient robotic systems performing complex tasks.
- **Job Scheduling:** Within computing systems, Hill Climbing is key in job scheduling, optimizing the allocation of system resources for various tasks. Efficiently managing the distribution of jobs across different nodes ensures optimal use of computational resources, enhancing overall system efficiency.

Game playing

- 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.
- 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 known as **multi-agent environment**, in which each agent is an opponent of other agent and playing against each other.
- Searches in which two or more competitive agents trying to explore the same search space for the solution, are called **adversarial searches**

- When agent play two-player games (chess, tic-tac-toe)
 - Agent wants its best move
 - The opponent wants his best move
 - Both the agents have same information.
 - opponent wants to win
- **Game trees** are used to represent two-player games.
- Alternate moves in the game are represented by alternate levels in the tree.
- Nodes in the tree represent positions.
- Edges between nodes represent moves.
- Leaf nodes represent won, lost or drawn positions.
- For most games, the game tree can be huge

Games as Search

- Two players: MAX and MIN
- MAX moves first and they take turns until the game is over
 - Winner gets reward, loser gets penalty.
 - “Zero sum” means the sum of the reward and the penalty is a constant.
- **Formal definition of game as a search problem:**
 - **Initial state:** Set-up specified by the rules, e.g., initial board configuration of chess.
 - **Player(s):** Defines which player has the move in a state.
 - **Actions(s):** Returns the set of legal moves in a state.
 - **Result(s,a):** Transition model defines the result of a move.
 - **Terminal-Test(s):** Is the game finished? True if finished, false otherwise.
 - **Utility function(s,p):** Gives numerical value of terminal state s for player p . E.g., win (+1), lose (-1), and draw (0) in tic-tac-toe.
- MAX uses search tree to determine next move.

The initial state, ACTIONS function, and RESULT function define the **game tree** for the game

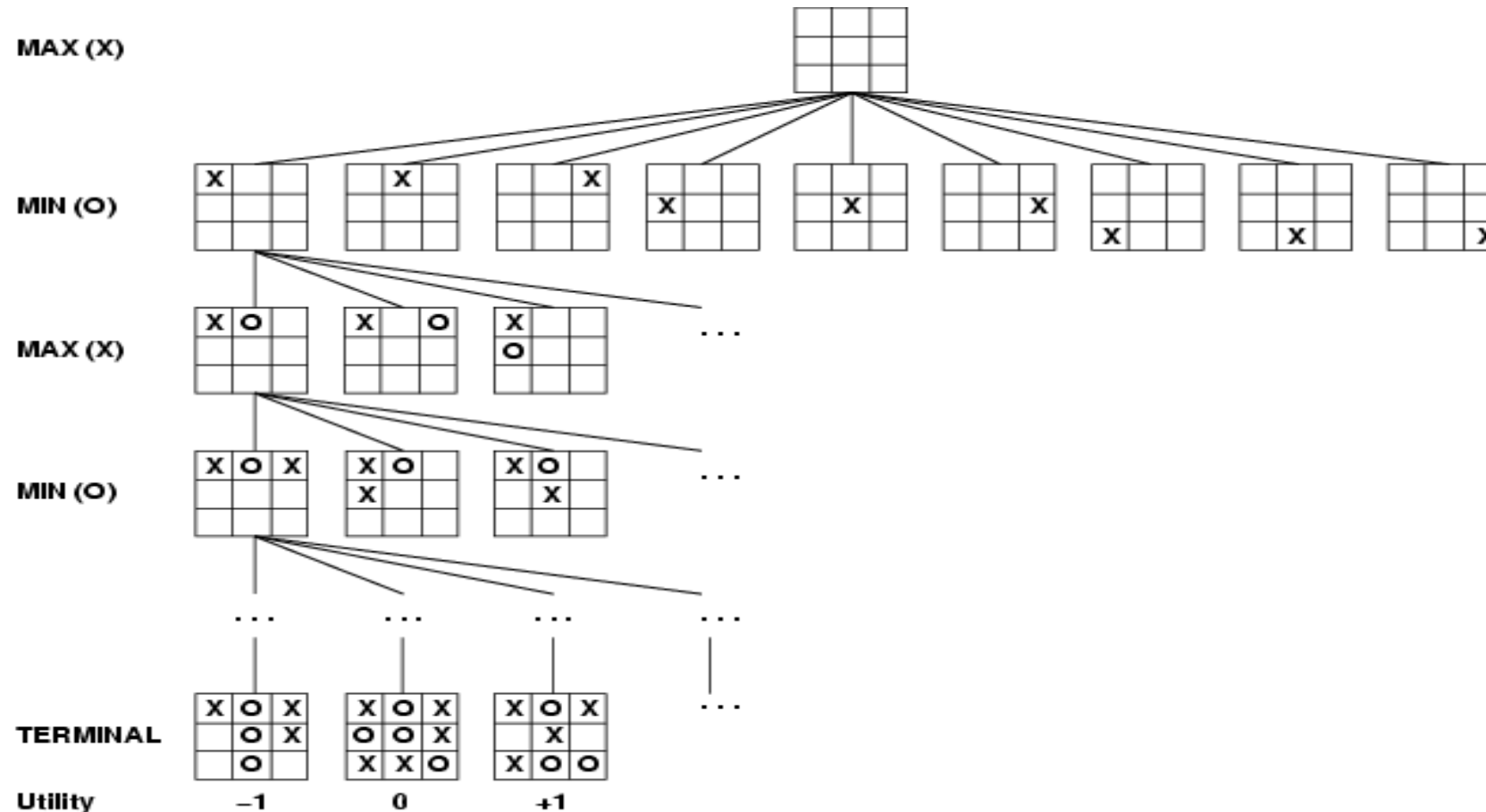


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

- 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:

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

The minimax algorithm

- The minimax algorithm computes the minimax decision from the current state.
- It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations.
- The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds.
- The minimax algorithm performs a complete depth-first exploration of the game tree.

MAX

MIN

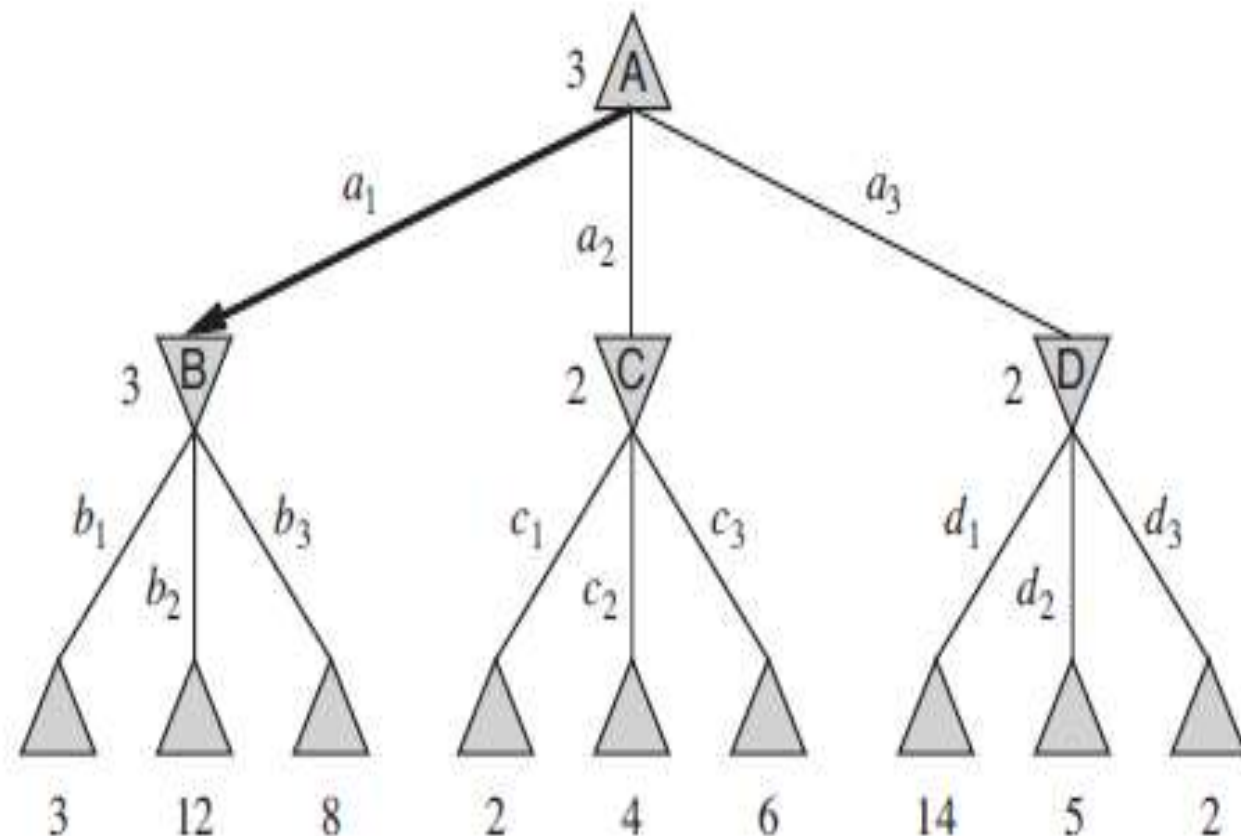
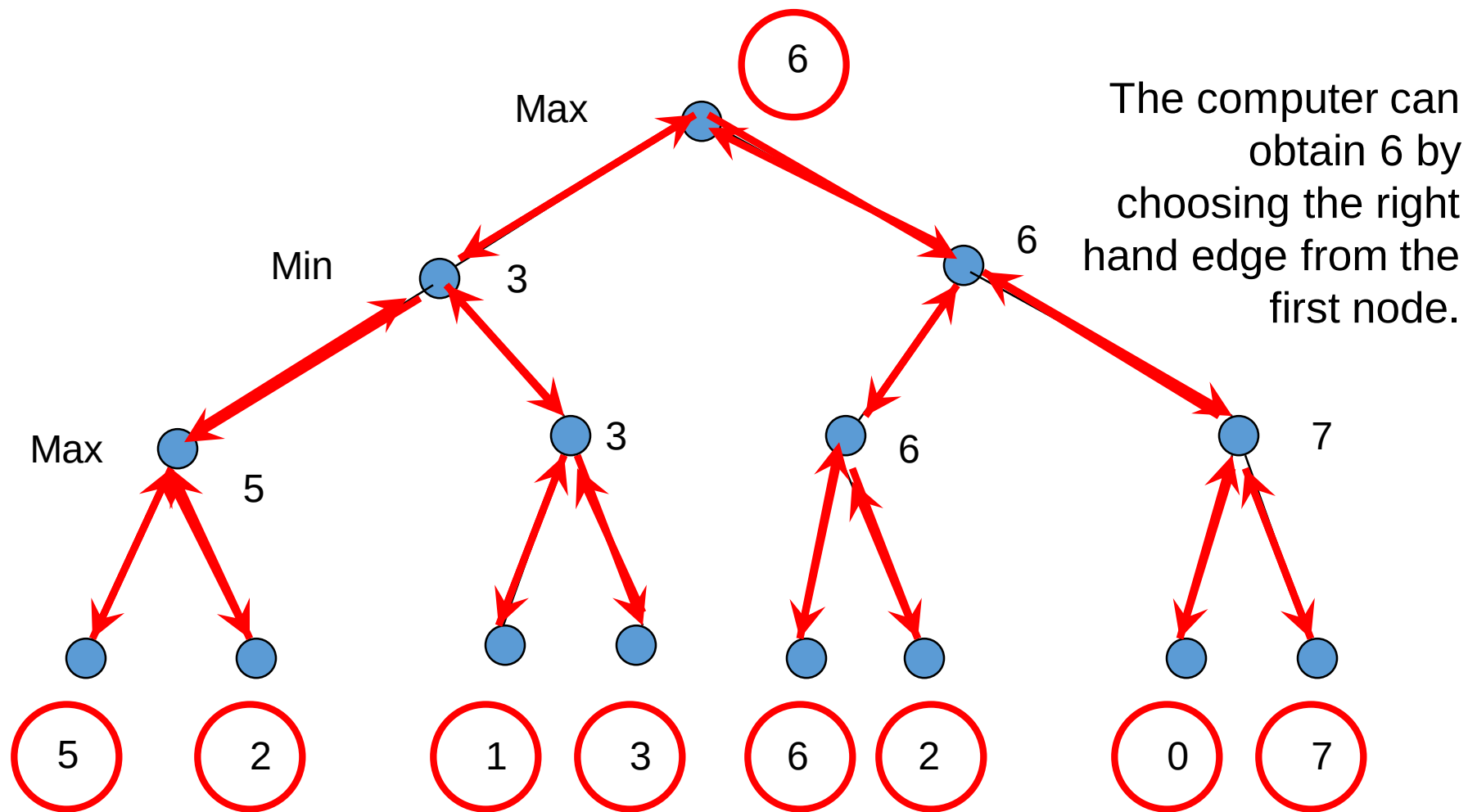


Figure 5.2 A two-ply game tree. The $\triangle$ nodes are “MAX nodes,” in which it is MAX’s turn to move, and the ∇ nodes are “MIN nodes.” The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX’s best move at the root is a_1 , because it leads to the state with the highest minimax value, and MIN’s best reply is b_1 , because it leads to the state with the lowest minimax value.



Pseudocode for Minimax Algorithm

function MINIMAX-DECISION(*state*) **returns** *an action*

inputs: *state*, current state in game

return $\arg \max_{a \in \text{ACTIONS}(\textit{state})} \text{MIN-VALUE}(\text{Result}(\textit{state}, a))$

function MAX-VALUE(*state*) **returns** *a utility value*

if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

$v \leftarrow -\infty$

for *a* **in** ACTIONS(*state*) **do**

$v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{Result}(\textit{state}, a)))$

return *v*

function MIN-VALUE(*state*) **returns** *a utility value*

if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

$v \leftarrow +\infty$

for *a* **in** ACTIONS(*state*) **do**

$v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{Result}(\textit{state}, a)))$

return *v*

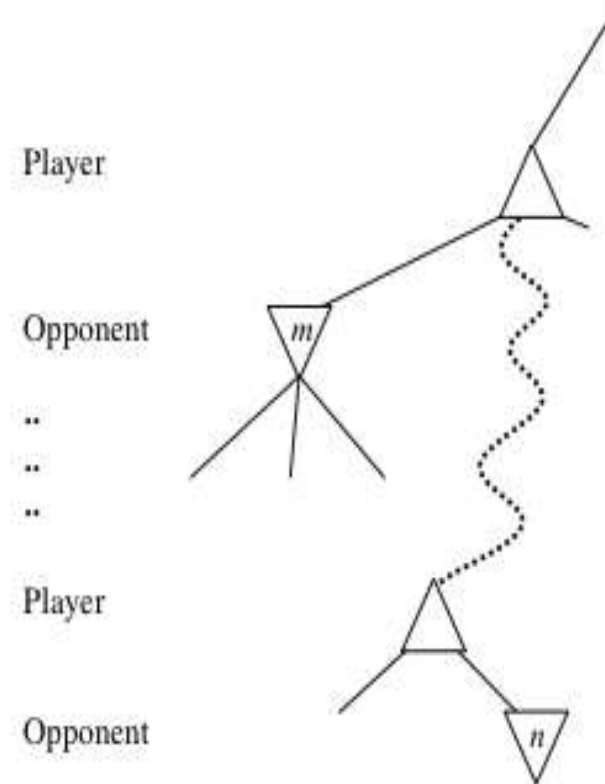
- Complete?
 - Yes (if tree is finite).
- Optimal?
 - Yes (against an optimal opponent).
- Time complexity?
 - $O(b^m)$
- Space complexity?
 - $O(bm)$ (depth-first search, generate all actions at once)
 - $O(m)$ (backtracking search, generate actions one at a time)

Alpha-Beta Pruning

- The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree.
- Alpha-beta pruning is a modified version of the minimax algorithm.
- In this technique, 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**.
- 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.

General alpha-beta pruning

- Consider a node n in the tree ---
- If player has a better choice at:
 - Parent node of n
 - Or any choice point further up
- Then n will never be reached in play
- Hence, when that much is known about n , it can be pruned.



Alpha-beta Algorithm

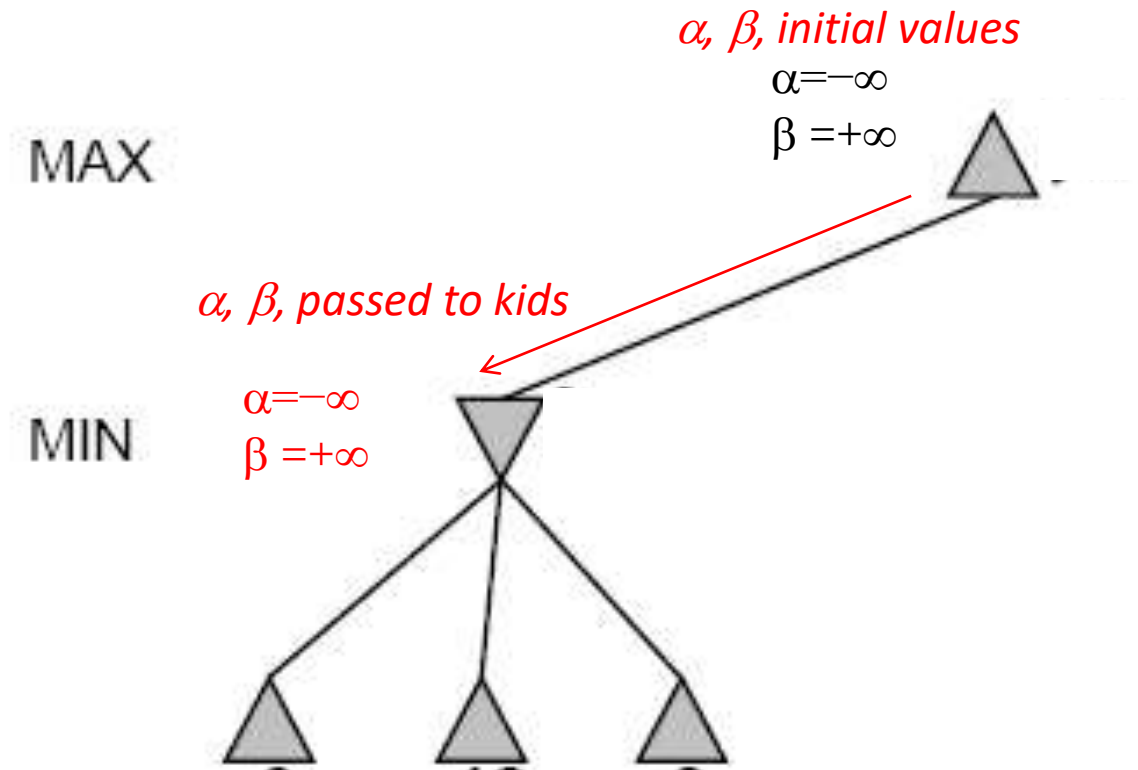
- Depth first search
 - only considers nodes along a single path from root at any time
- α = highest-value choice found at any choice point of path for MAX
(initially, $\alpha = -\text{infinity}$)
- β = lowest-value choice found at any choice point of path for MIN
(initially, $\beta = +\text{infinity}$)
- Pass current values of α and β down to child nodes during search.
 - Update values of α and β during search:
 - MAX updates α at MAX nodes
 - MIN updates β at MIN nodes
 - Prune remaining branches at a node when $\alpha \geq \beta$

When to Prune

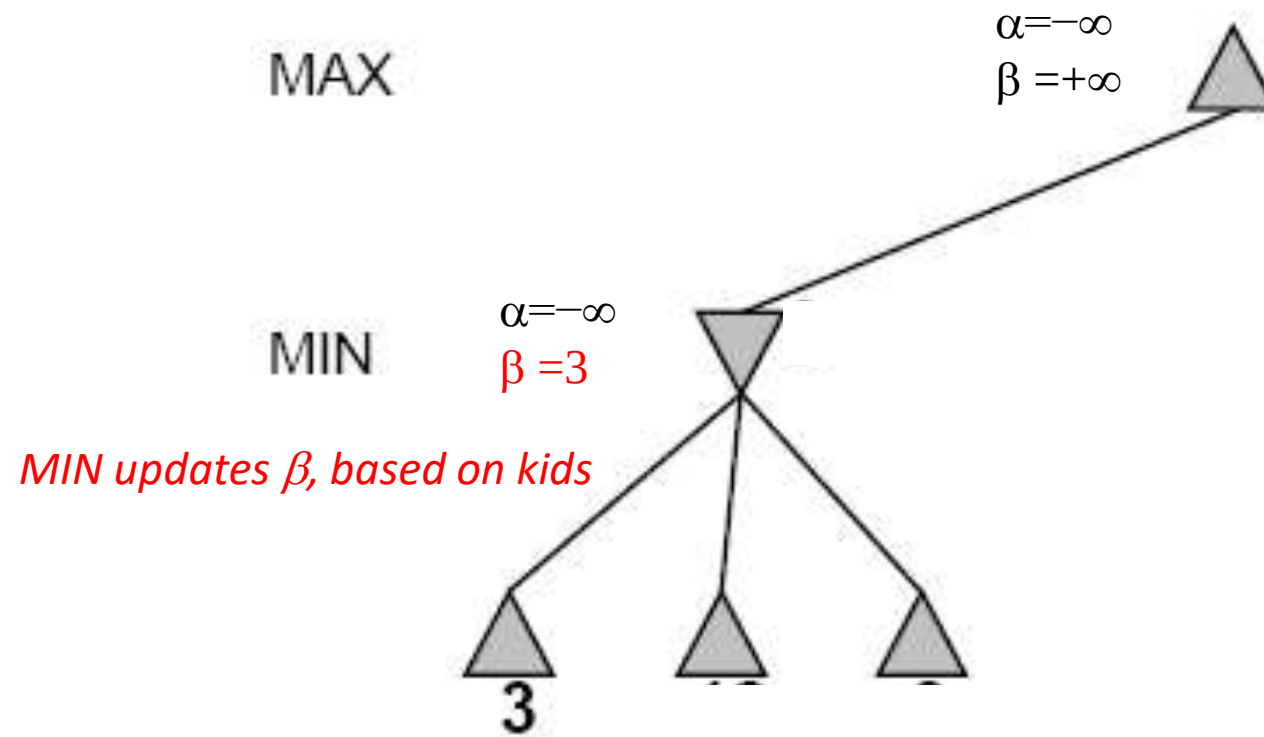
- Prune whenever $\alpha \geq \beta$.
 - Prune below a Max node whose alpha value becomes greater than or equal to the beta value of its ancestors.
 - **Max nodes update alpha** based on children's returned values.
 - Prune below a Min node whose beta value becomes less than or equal to the alpha value of its ancestors.
 - **Min nodes update beta** based on children's returned values.

Alpha-Beta Example

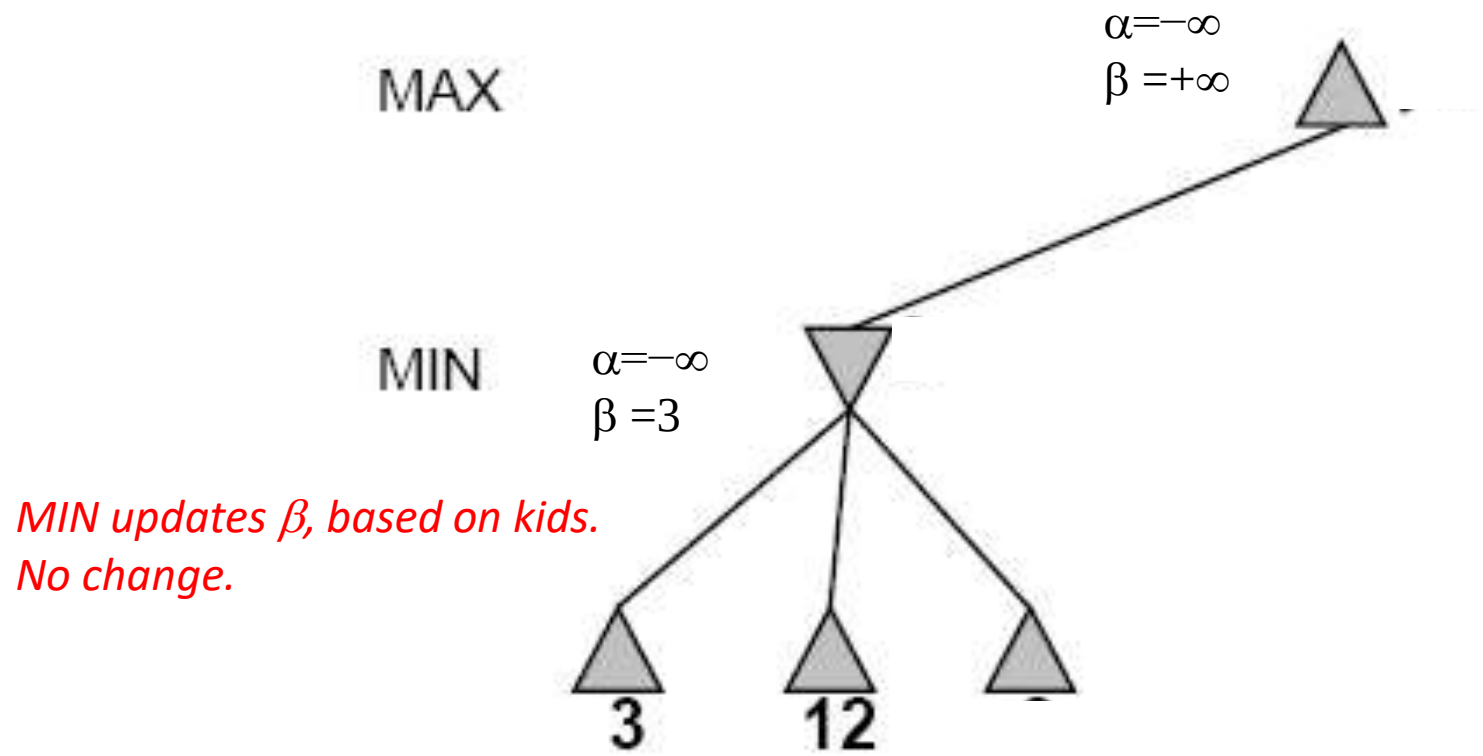
Do DF-search until first leaf



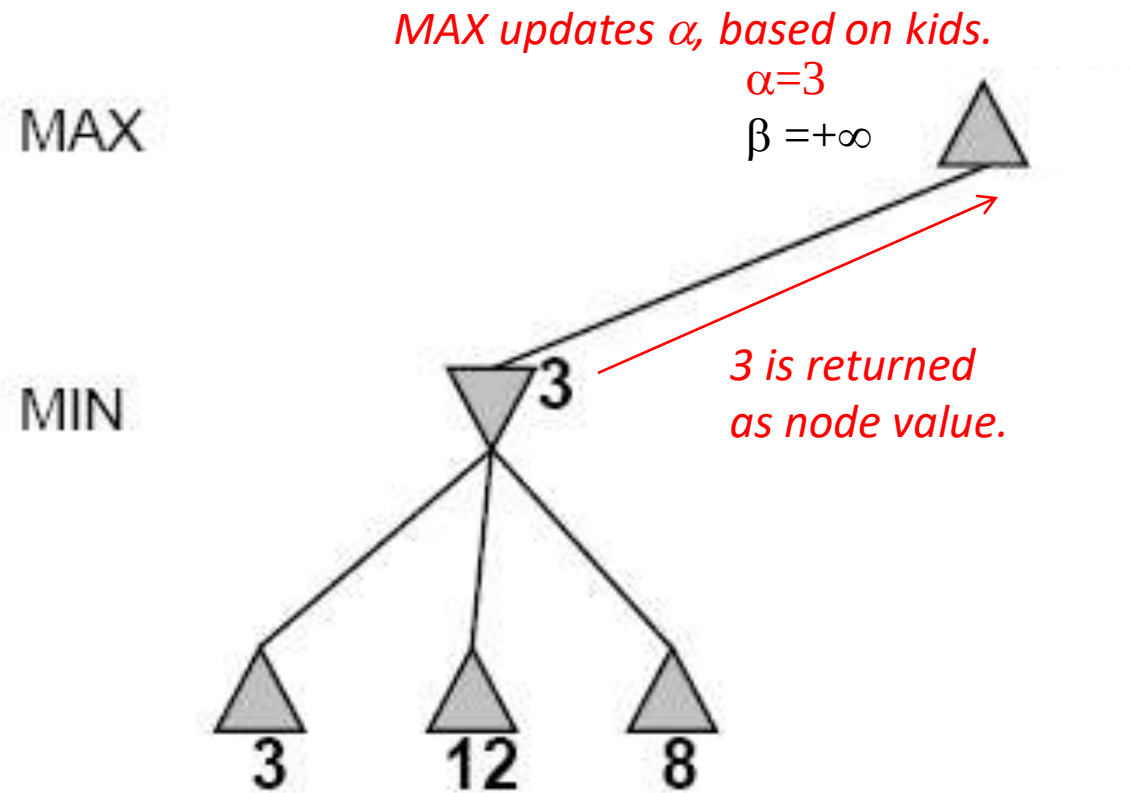
Alpha-Beta Example (continued)



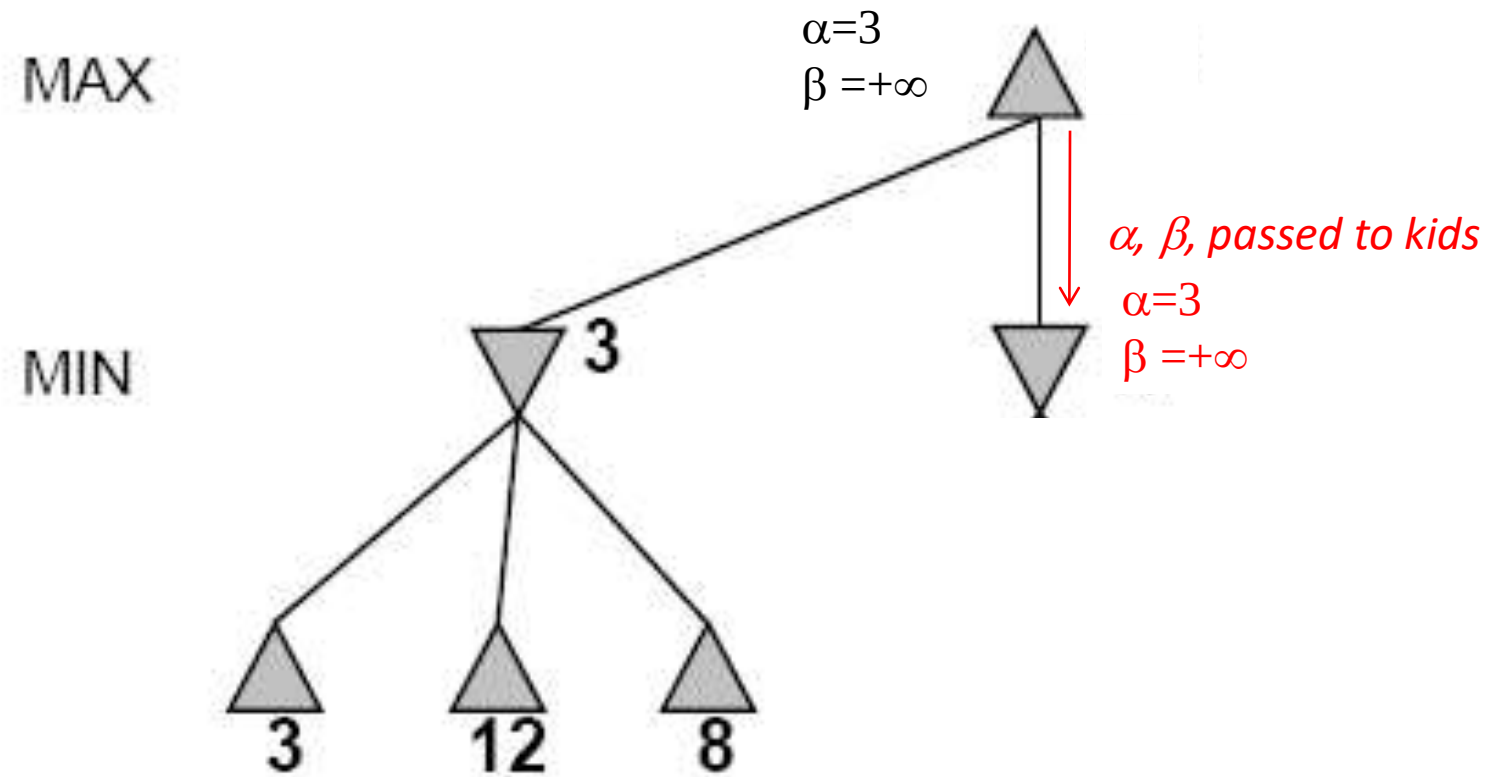
Alpha-Beta Example (continued)



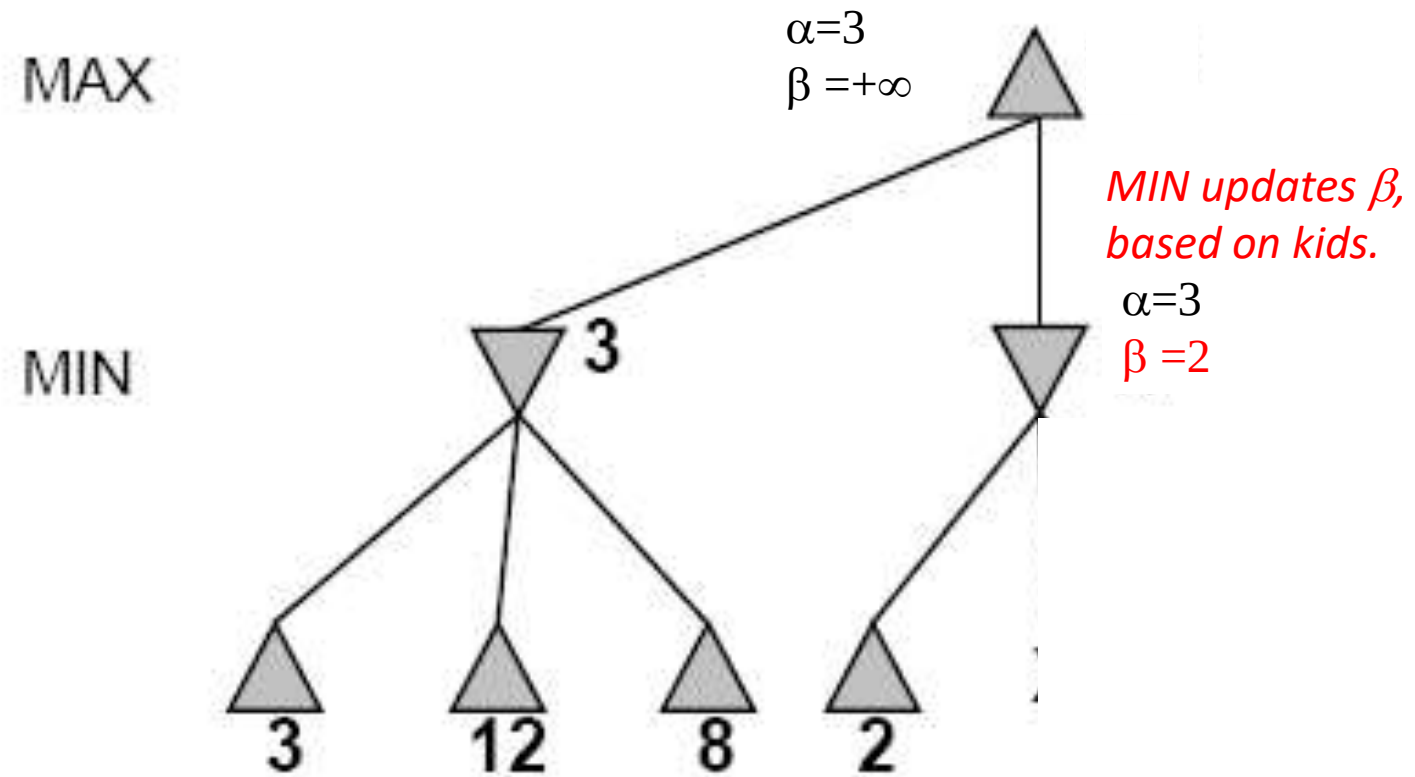
Alpha-Beta Example (continued)



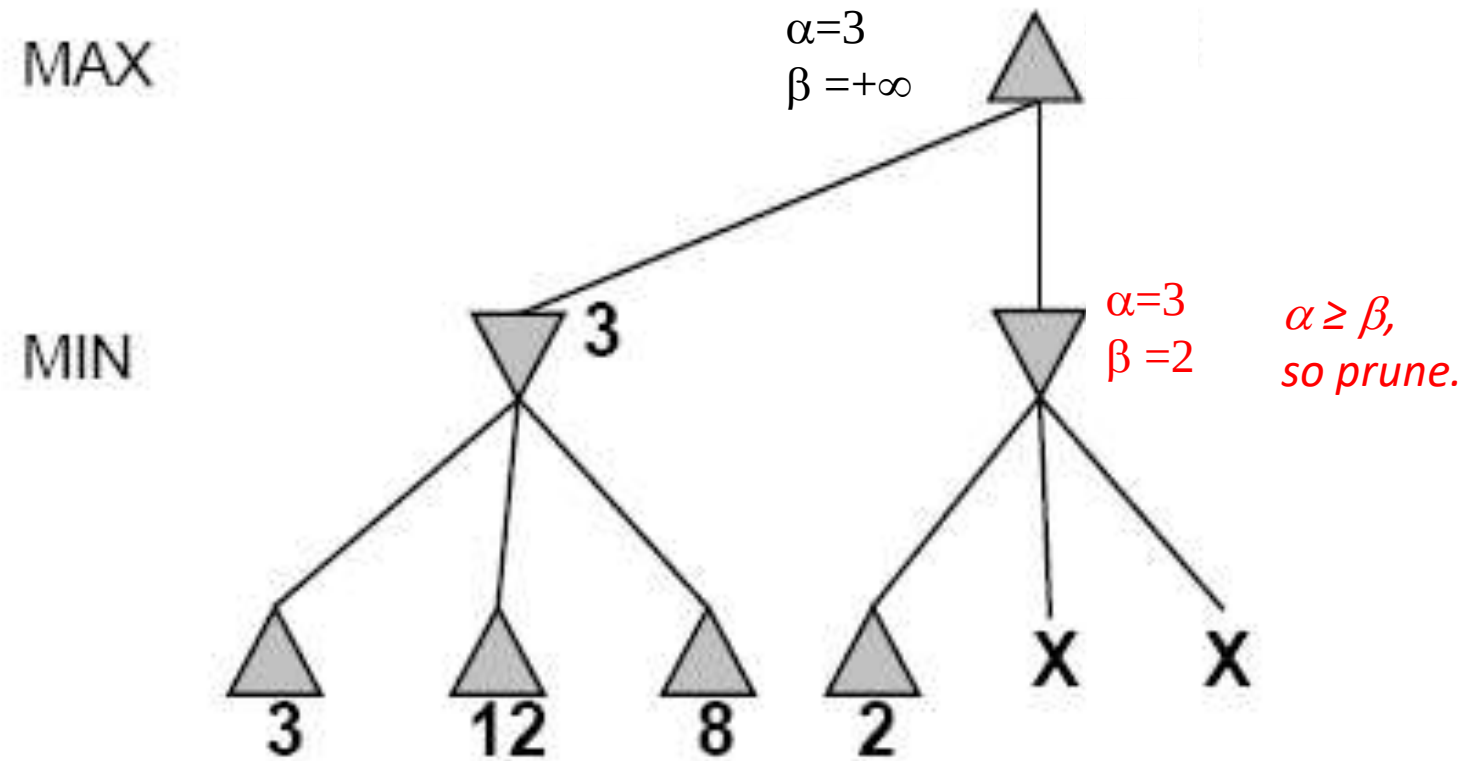
Alpha-Beta Example (continued)



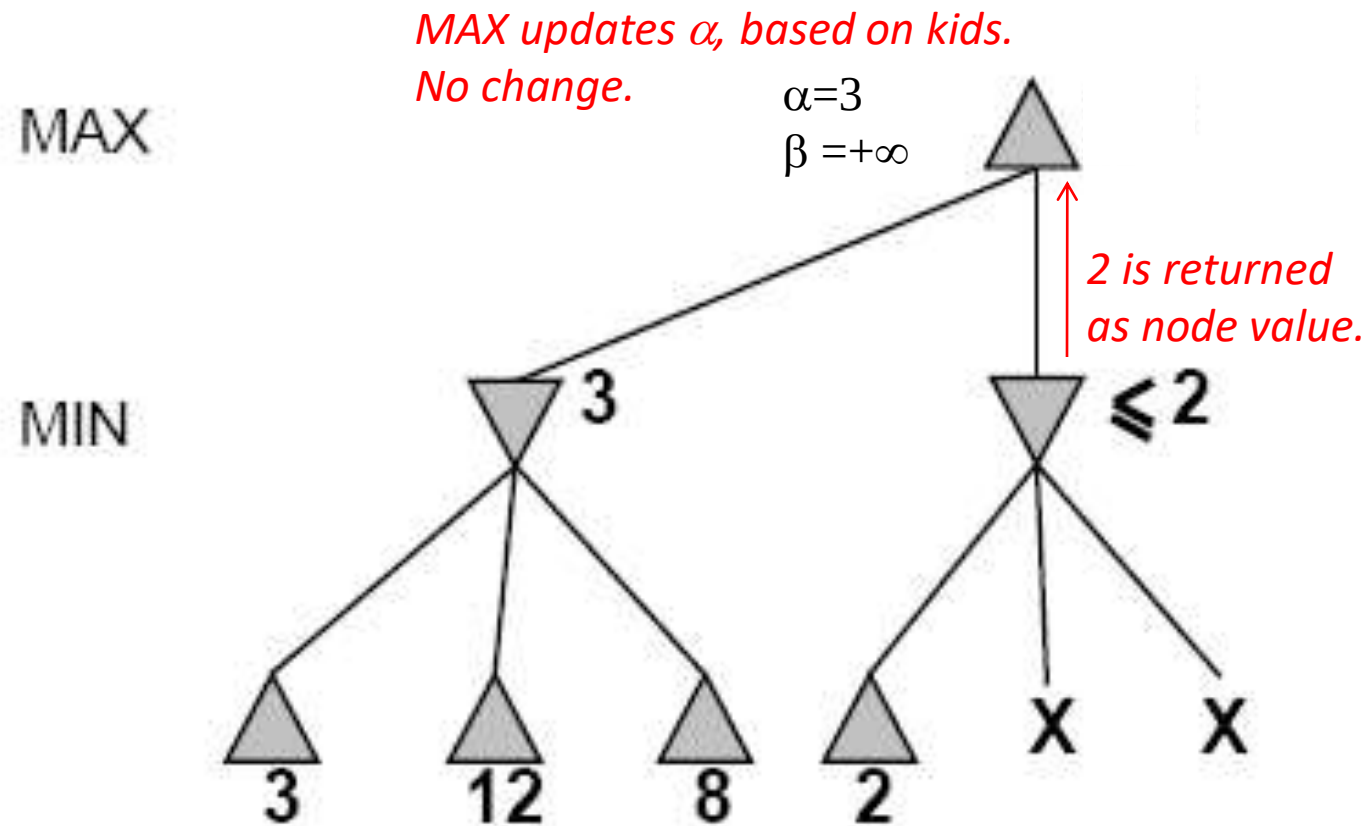
Alpha-Beta Example (continued)



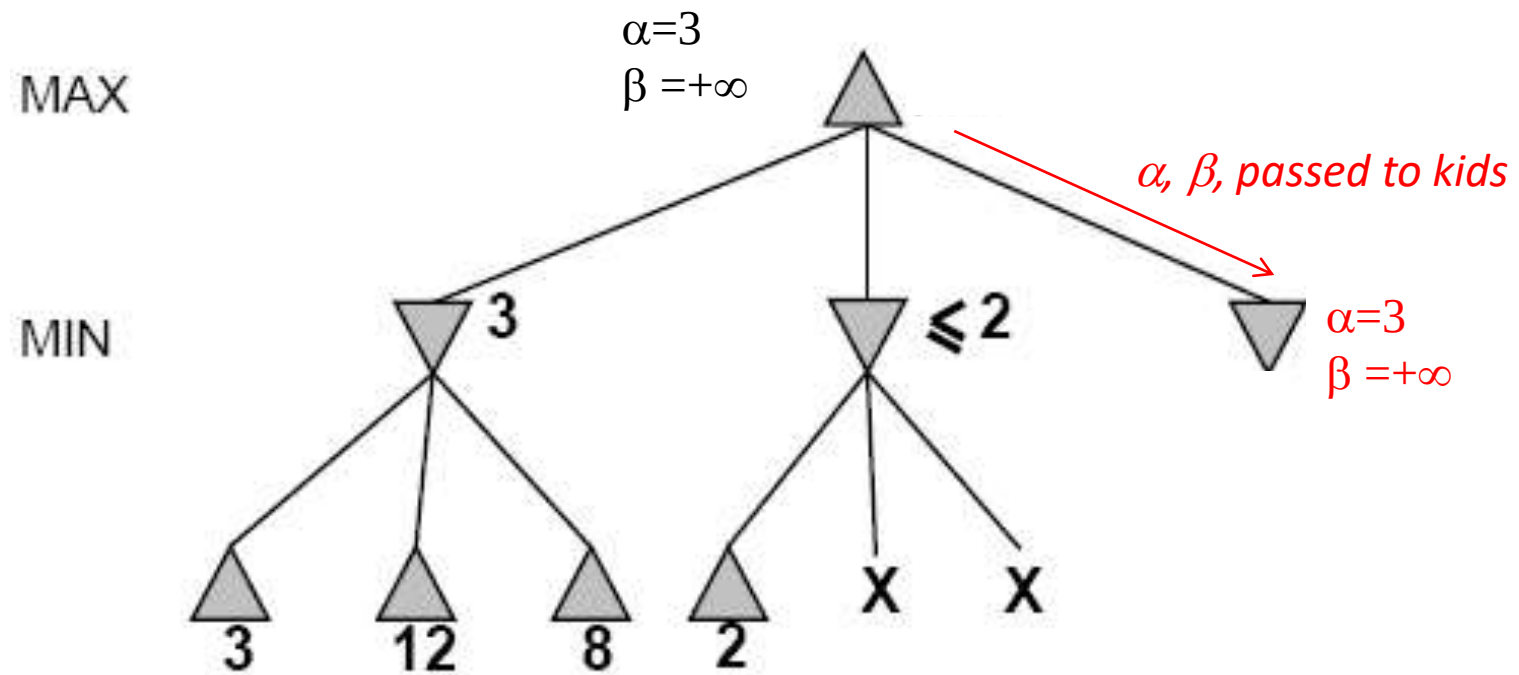
Alpha-Beta Example (continued)



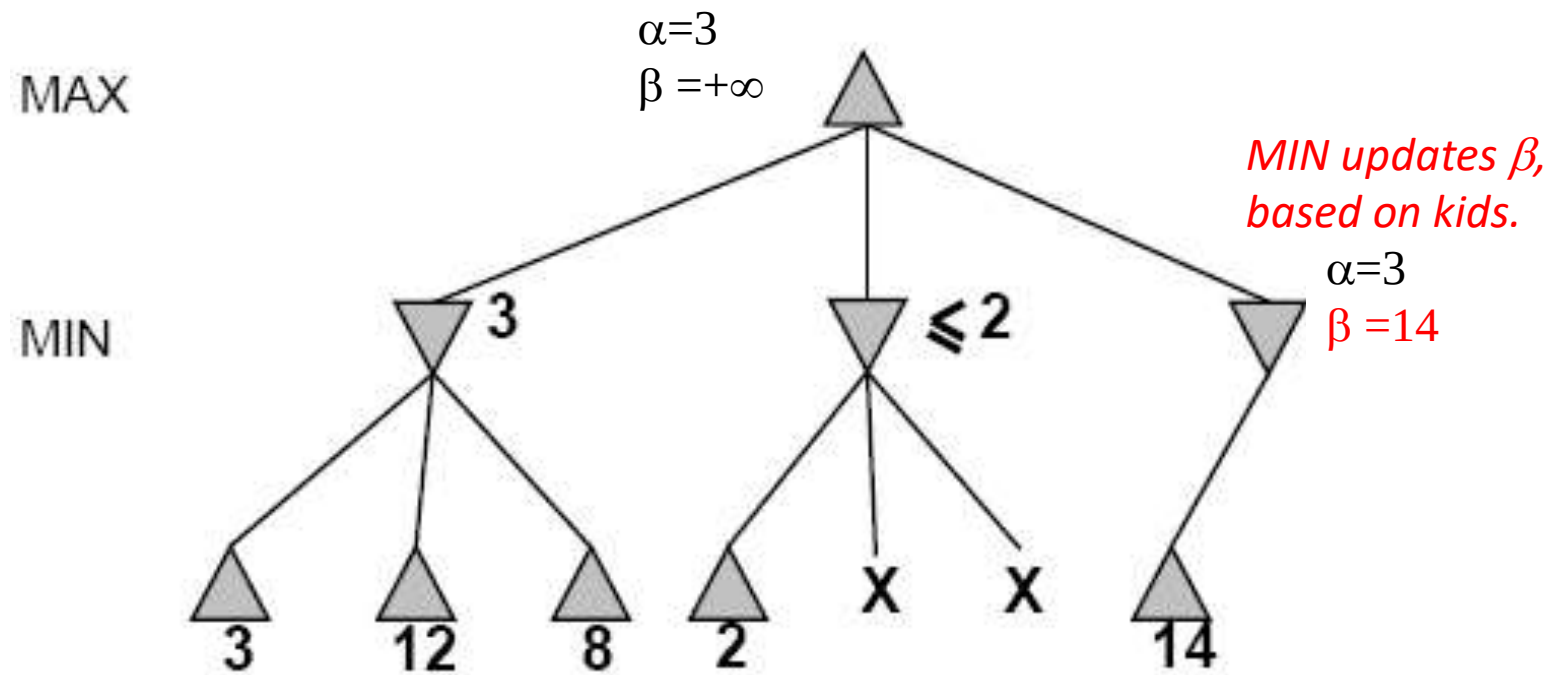
Alpha-Beta Example (continued)



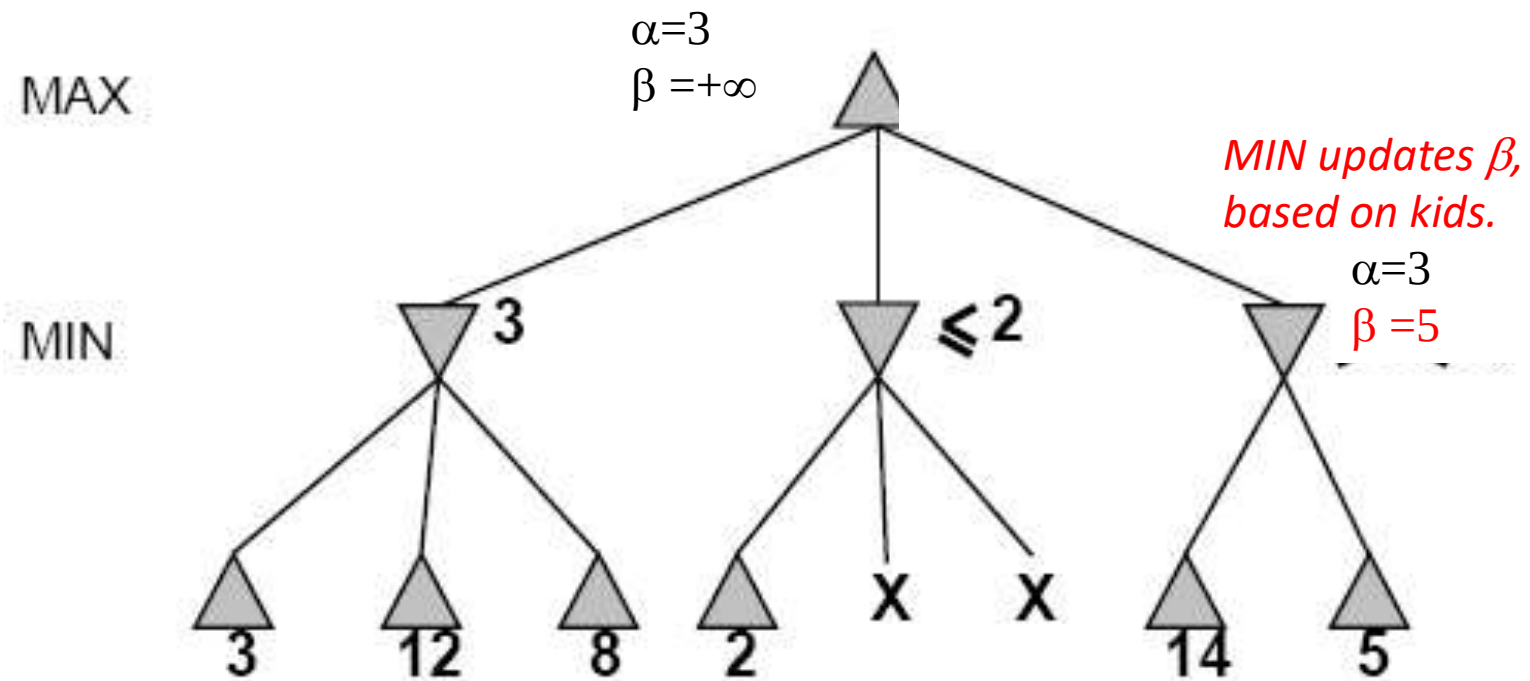
Alpha-Beta Example (continued)



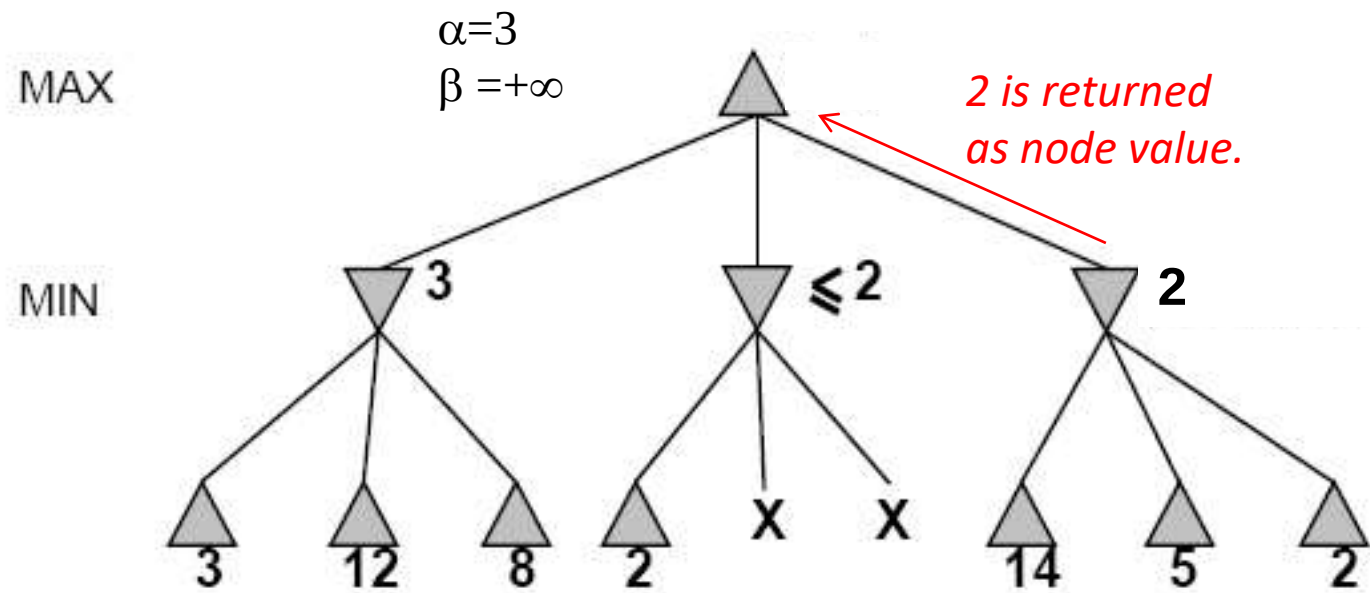
Alpha-Beta Example (continued)



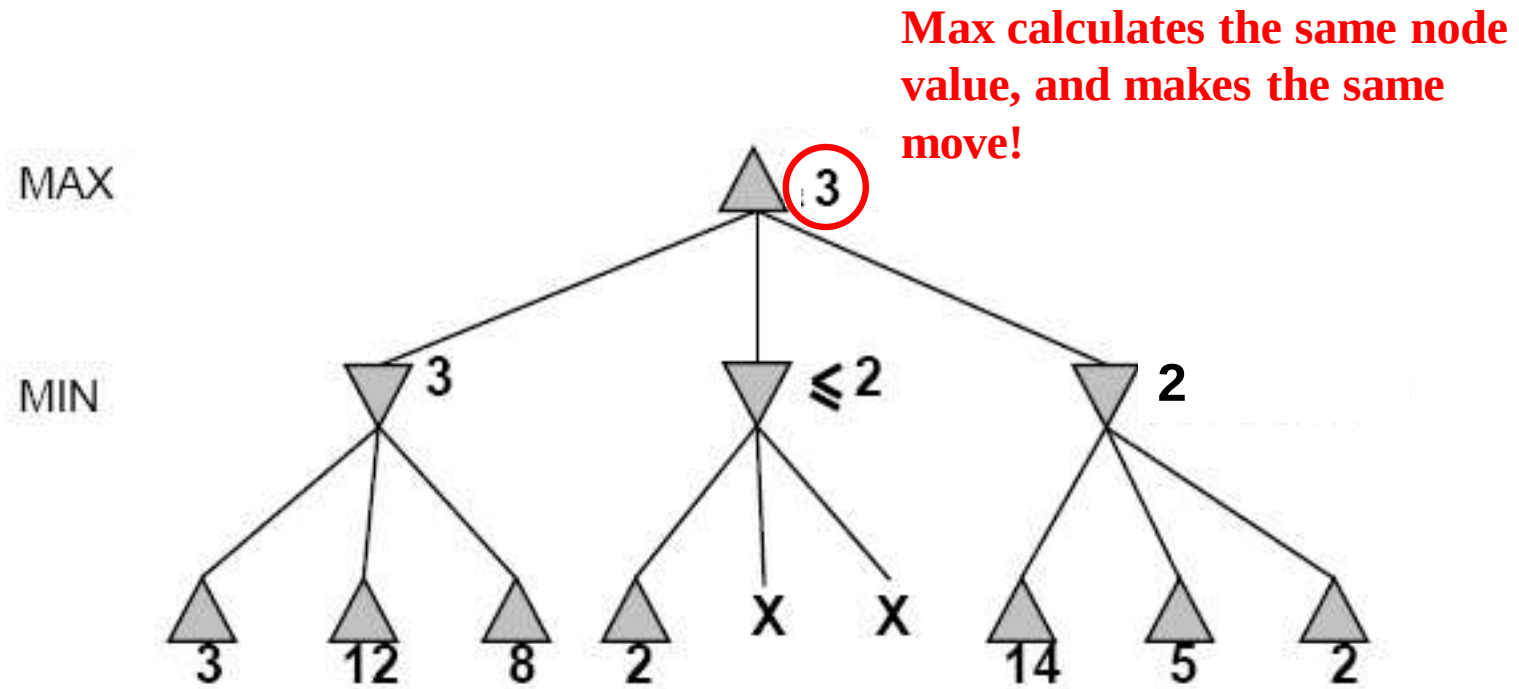
Alpha-Beta Example (continued)



Alpha-Beta Example (continued)



Alpha-Beta Example (continued)



Pseudocode for Alpha-Beta Algorithm

function ALPHA-BETA-SEARCH(*state*) **returns** *an action*

inputs: *state*, current state in game

$v \leftarrow \text{MAX-VALUE}(\textit{state}, -\infty, +\infty)$

return the *action* in $\text{ACTIONS}(\textit{state})$ with value v

function MAX-VALUE(*state*, α , β) **returns** *a utility value*

if $\text{TERMINAL-TEST}(\textit{state})$ **then return** $\text{UTILITY}(\textit{state})$

$v \leftarrow -\infty$

for a in $\text{ACTIONS}(\textit{state})$ **do**

$v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{Result}(s, a), \alpha, \beta))$

if $v \geq \beta$ **then return** v

$\alpha \leftarrow \text{MAX}(\alpha, v)$

return v

```
function MIN-VALUE( $state, \alpha, \beta$ ) returns a utility value  
  if TERMINAL-TEST( $state$ ) then return UTILITY( $state$ )  
   $v \leftarrow \infty$   
  for  $a$  in ACTIONS( $state$ ) do  
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{Result}(s, a), \alpha, \beta))$   
    if  $v \leq \alpha$  then return  $v$   
   $\beta \leftarrow \text{MIN}(v, \beta)$   
  return  $v$ 
```


Move Ordering in Alpha-Beta pruning:

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

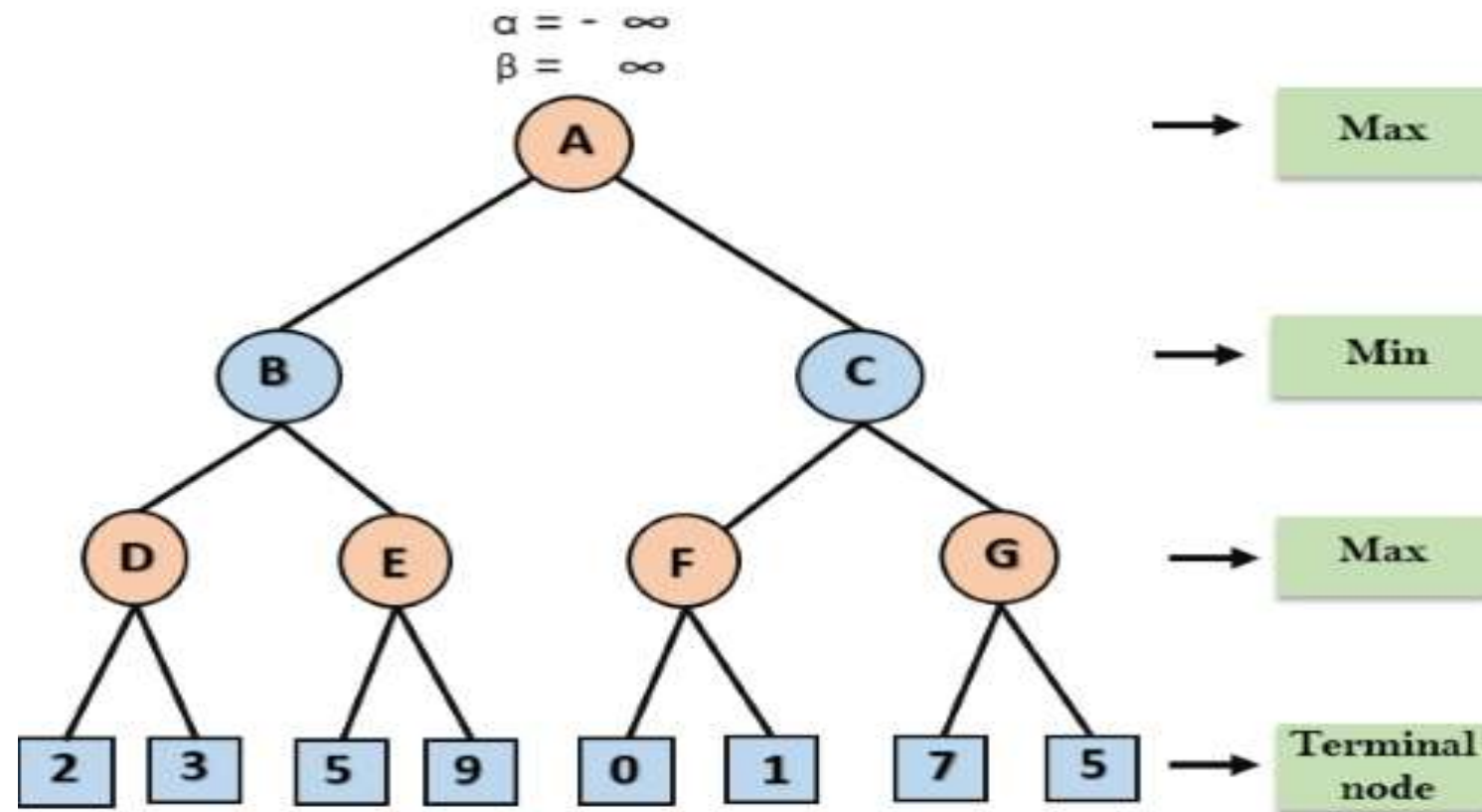
It can be of two types:

- **Worst ordering:** In some cases, the best move occurs on the right side of the tree, works exactly as minimax algorithm and it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering.
- 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 $O(b^{m/2})$.

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

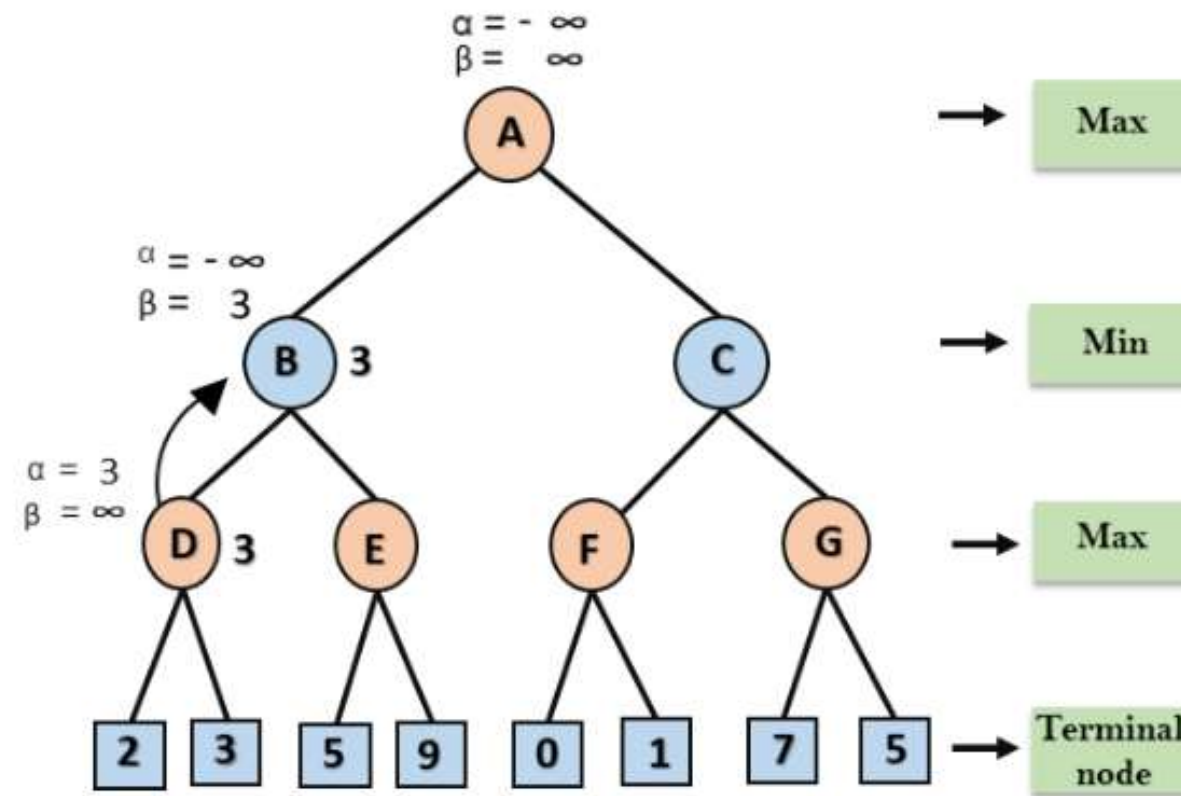
- Order the nodes in the tree such that the best nodes are checked first.
- Use domain knowledge while finding the best move. Ex: for Chess, try order: captures first, then threats, then forward moves, backward moves.
- as there is a possibility that states may repeat, store the evaluation of the resulting position in a hash table the first time it is encountered, so that we don't have to recompute it on subsequent occurrences

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



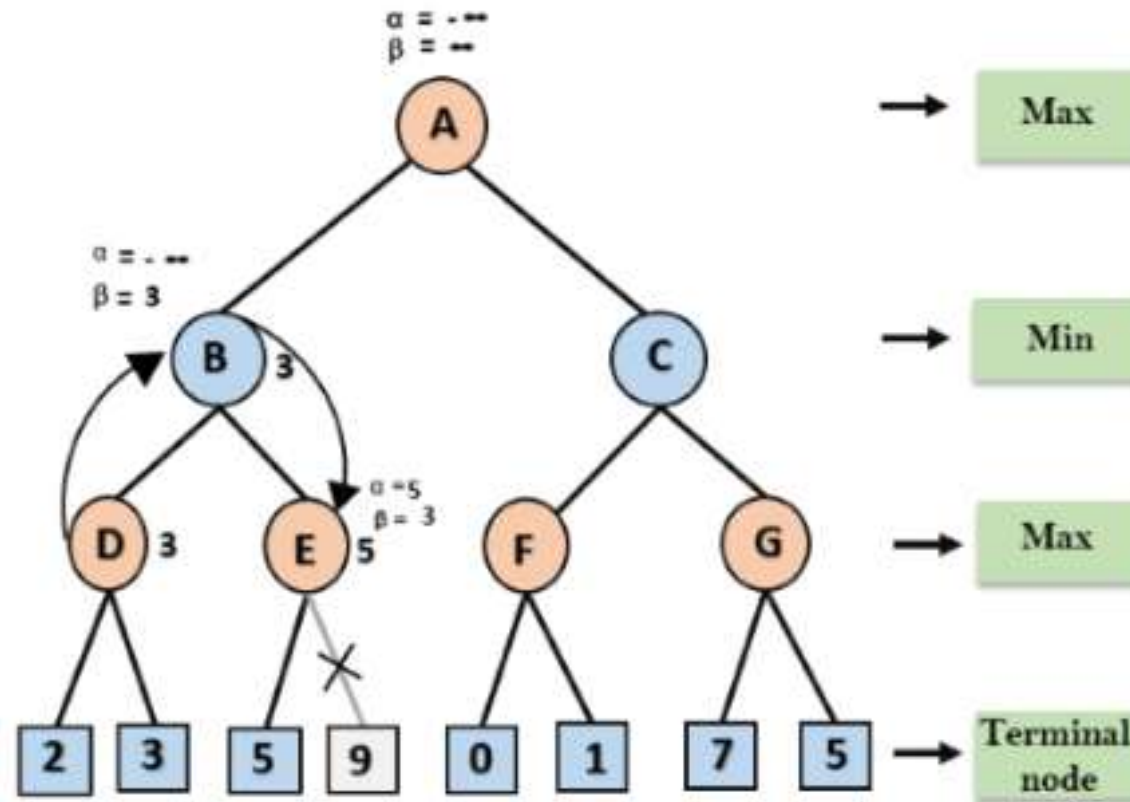
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the $\max(2, 3) = 3$ will be the value of α at node D and node value will also 3.

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



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

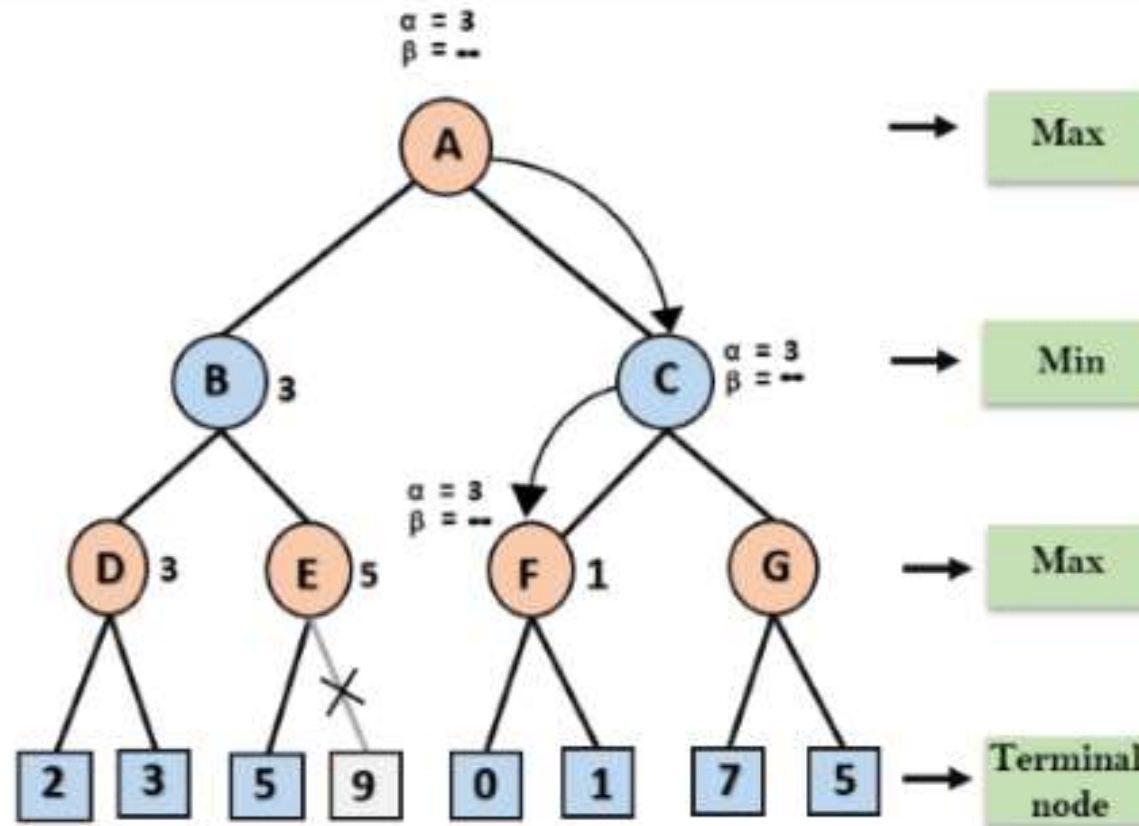
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so $\max(-\infty, 5) = 5$, hence at node E $\alpha = 5$ and $\beta = 3$, where $\alpha \geq \beta$, so the right successor of E will be pruned, and algorithm will not traverse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



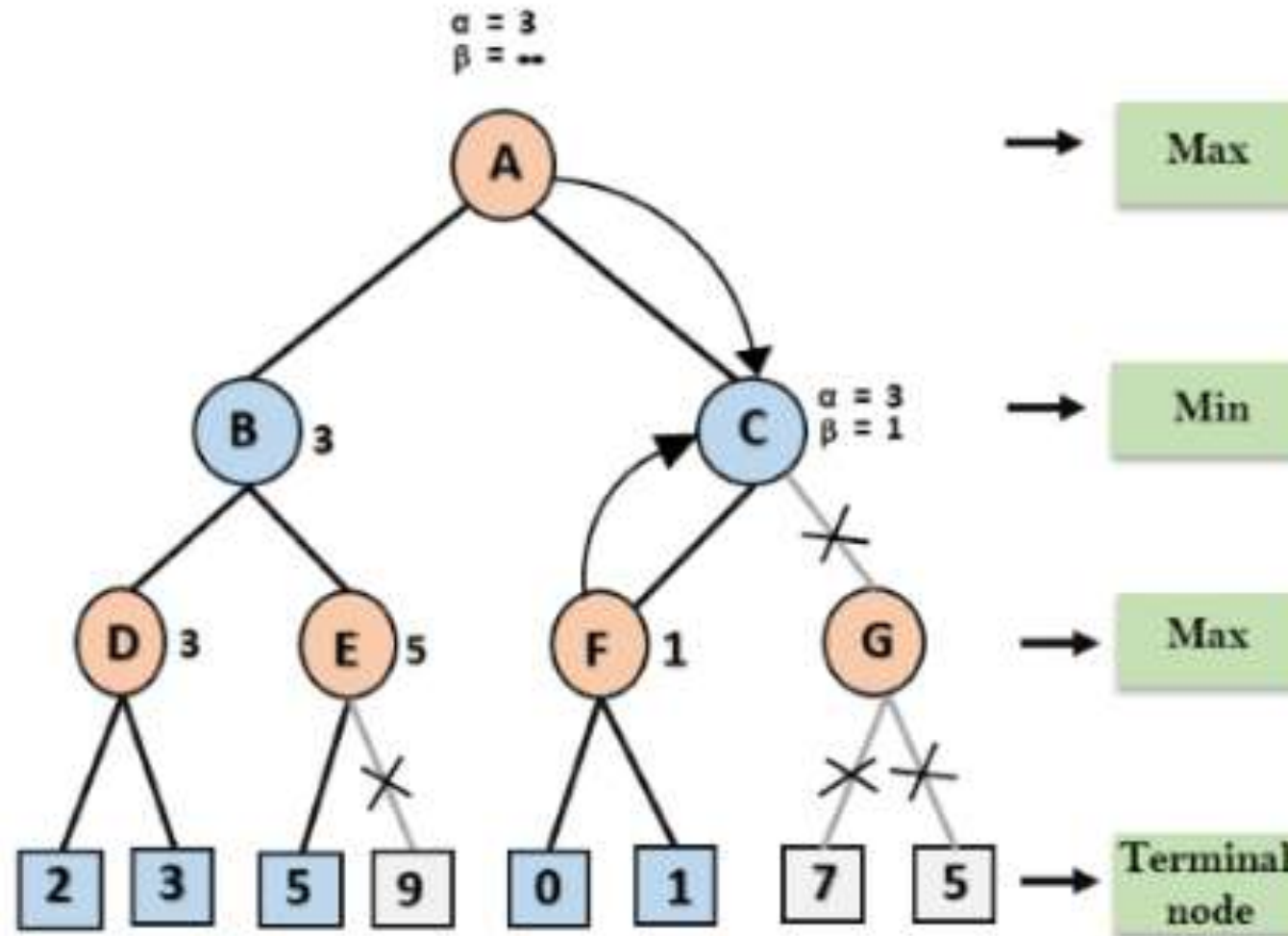
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as $\max(-\infty, 3) = 3$, and $\beta = +\infty$, these two values now passes to right successor of A which is Node C.

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

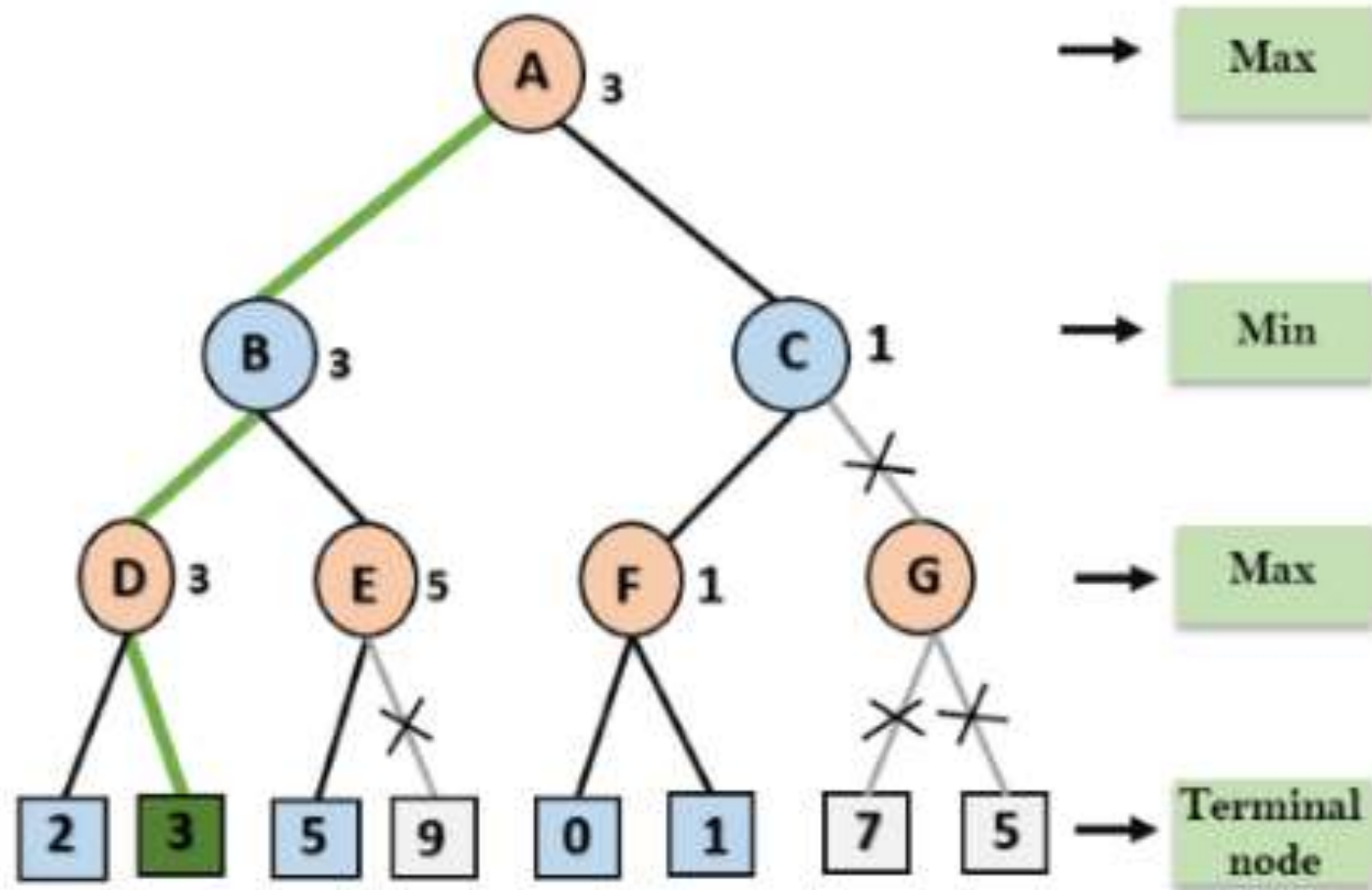
Step 6: At node F, again the value of α will be compared with left child which is 0, and $\max(3,0) = 3$, and then compared with right child which is 1, and $\max(3,1) = 3$ still α remains 3, but the node value of F will become 1



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



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



Optimal decisions in multiplayer games

- Many popular games allow more than two players. Let us examine how to extend the minimax idea to multiplayer games.
- First, we need to replace the single value for each node with a *vector* of values. For example, in a three-player game with players A, B, and C, a vector $\langle v_A, v_B, v_C \rangle$ is associated with each node.
- For terminal states, this vector gives the utility of the state from each player's viewpoint.
- The simplest way to implement this is to have the UTILITY function return a vector of utilities.

to move
A

B

C

A

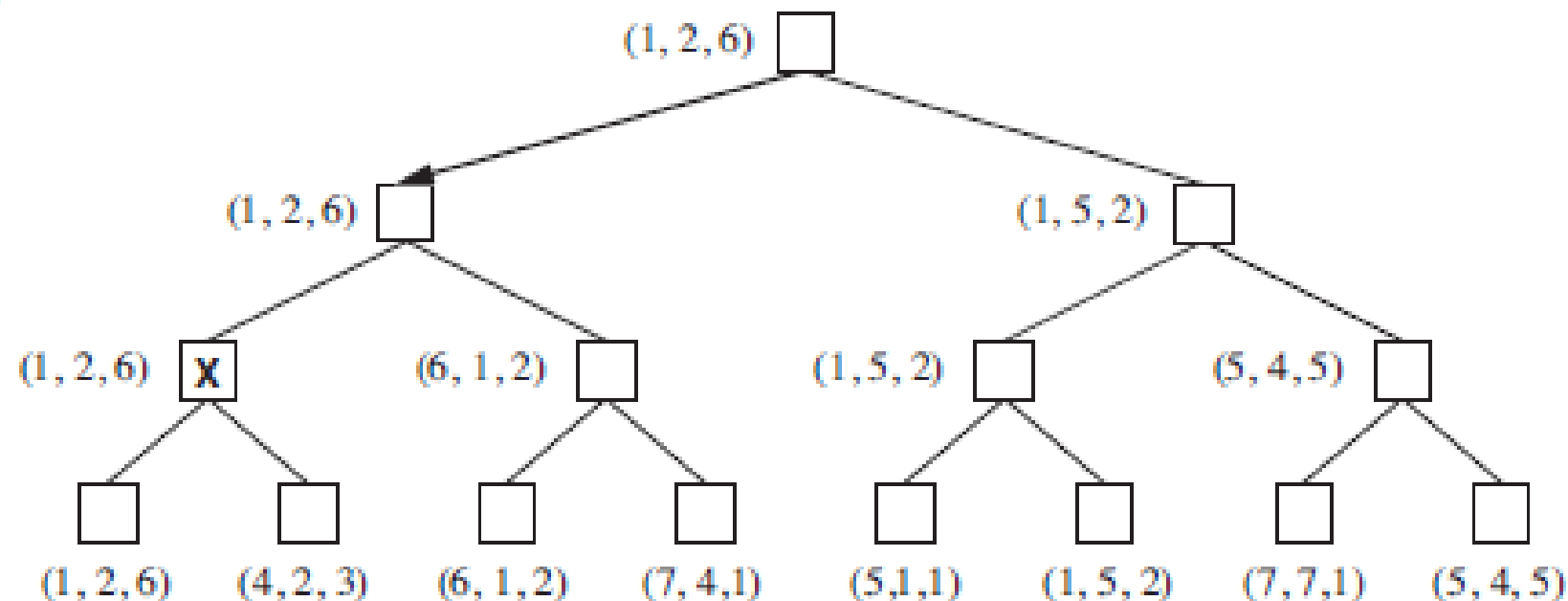


Figure 5.4 The first three plies of a game tree with three players (A , B , C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

- Now we have to consider nonterminal states. Consider the node marked X in the game tree shown in Figure 5.4. In that state, player C chooses what to do.
- The two choices lead to terminal states with utility vectors $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$ and $\langle v_A = 4, v_B = 2, v_C = 3 \rangle$.
- Since 6 is bigger than 3, C should choose the first move. This means that if state X is reached, subsequent play will lead to a terminal state with utilities $v_A = 1, v_B = 2, v_C = 6$.
- Hence, the backed-up value of X is this vector. The backed-up value of a node n is always the utility vector of the successor state with the highest value for the player choosing at n.

Problem in Game playing

- **Limited scope:** The techniques and algorithms developed for game playing may not be well-suited for other types of applications and may need to be adapted or modified for different domains.
- **Computational cost:** Game playing can be computationally expensive, especially for complex games such as chess or Go, and may require powerful computers to achieve real-time performance.

Evaluation functions

- The minimax algorithm generates the entire game search space, whereas the alpha–beta algorithm allows us to prune large parts of it. However, alpha–beta still has to search all the way to terminal states for at least a portion of the search space.
- It is possible to cut off the search earlier and apply a heuristic **evaluation function** to states in the search, effectively turning nonterminal nodes into terminal leaves.
- In other words, the suggestion is to alter minimax or alpha–beta in two ways: replace the utility function by a heuristic evaluation function EVAL, which estimates the position's utility, and replace the terminal test by a **cutoff test** that decides when to apply EVAL. That gives us the following for heuristic minimax for state s and maximum depth d :

H-MINIMAX(s, d) =

$$\begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s, d) \\ \max_{a \in \text{Actions}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if PLAYER}(s) = \text{MIN}. \end{cases}$$

R23-Unit 3

REPRESENTATION OF KNOWLEDGE

Knowledge Representation Issues

Introduction

- For the purpose of solving complex problems encountered in AI, we need both a large amount of knowledge and some mechanism for manipulating that knowledge to create solutions to new problems.
- A variety of ways of representing knowledge (facts) have been exploited in AI programs. In all variety of knowledge representations , we deal with two kinds of entities.
 - A. Facts: Truths in some relevant world. These are the things we want to represent.
 - B. Representations of facts in some chosen formalism . these are things we will actually be able to manipulate.

- One way to think of structuring these entities is at two levels :
 - (a) the knowledge level, at which facts are described, and
 - (b) the symbol level, at which representations of objects at the knowledge level are defined in terms of symbols that can be manipulated by programs.
- The facts and representations are linked with two-way mappings.
- This link is called representation mappings.
- The forward representation mapping maps from facts to representations.
- The backward representation mapping goes the other way, from representations to facts.

Representations and Mappings

- In order to solve complex problems encountered in artificial intelligence, one needs both a large amount of knowledge and some mechanism for manipulating that knowledge to create solutions.
- Knowledge and Representation are two distinct entities. They play central but distinguishable roles in the intelligent system.
 - ✓ Knowledge is a description of the world. It determines a system's competence by what it knows.
 - ✓ Moreover, Representation is the way knowledge is encoded. It defines a system's performance in doing something.
 - ✓ Different types of knowledge require different kinds of representation.

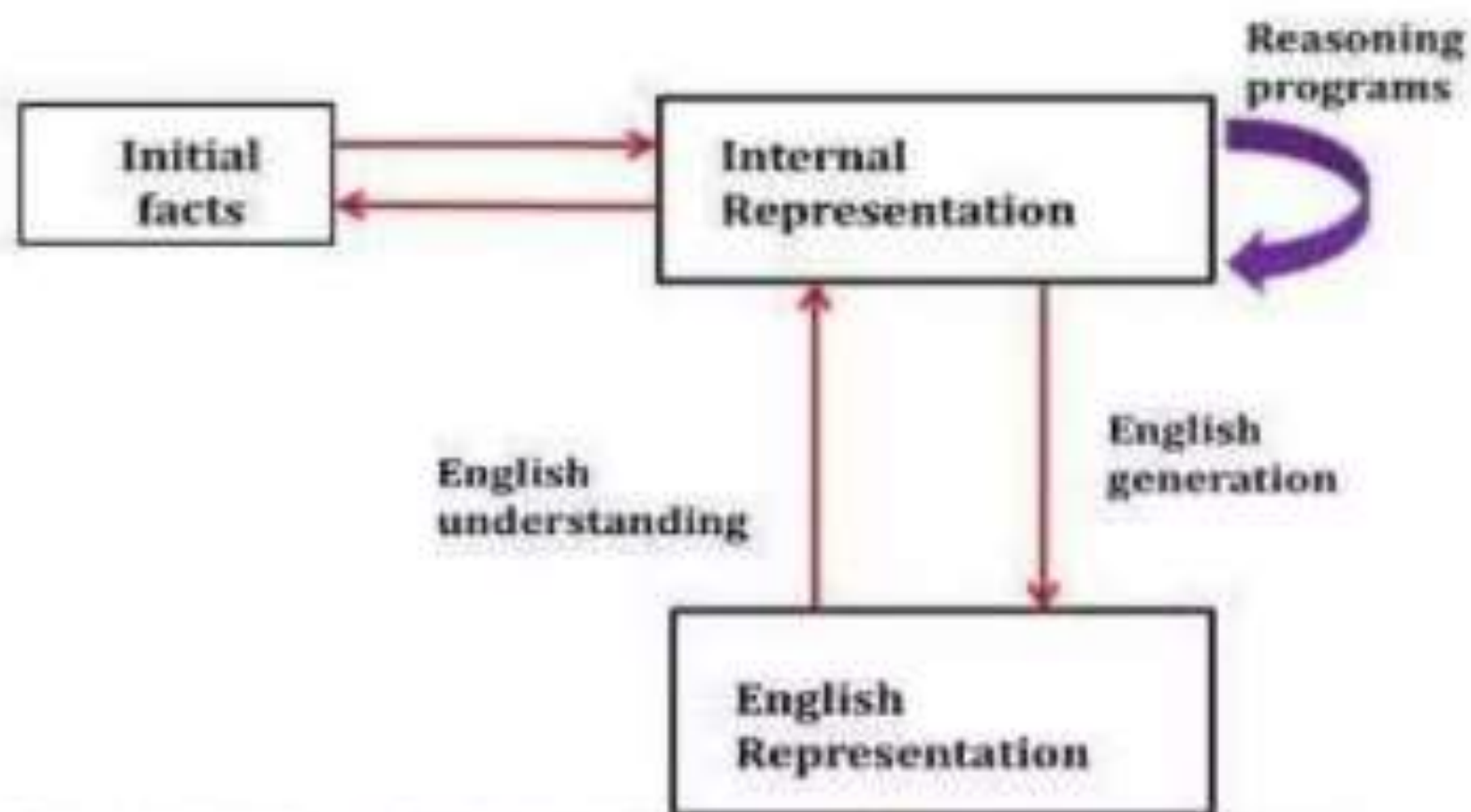


Fig: Mapping between Facts and Representations

Let's look at a simple example using mathematical logic as the representational formalism. Consider the English sentence:

Spot is a dog.

- The fact represented by that English sentence can also be represented in logic as:

$\text{dog}(\text{Spot})$

- Suppose that we also have a logical representation of the fact that all dogs havetails:
- For all x : $\text{dog}(x) \rightarrow \text{hastail}\{x\}$

- Then, using the deductive mechanisms of logic, we may generate the new representation object:

hastail(Spot)

- Using an appropriate backward mapping function, we could then generate the English sentence:

Spot has a tail.

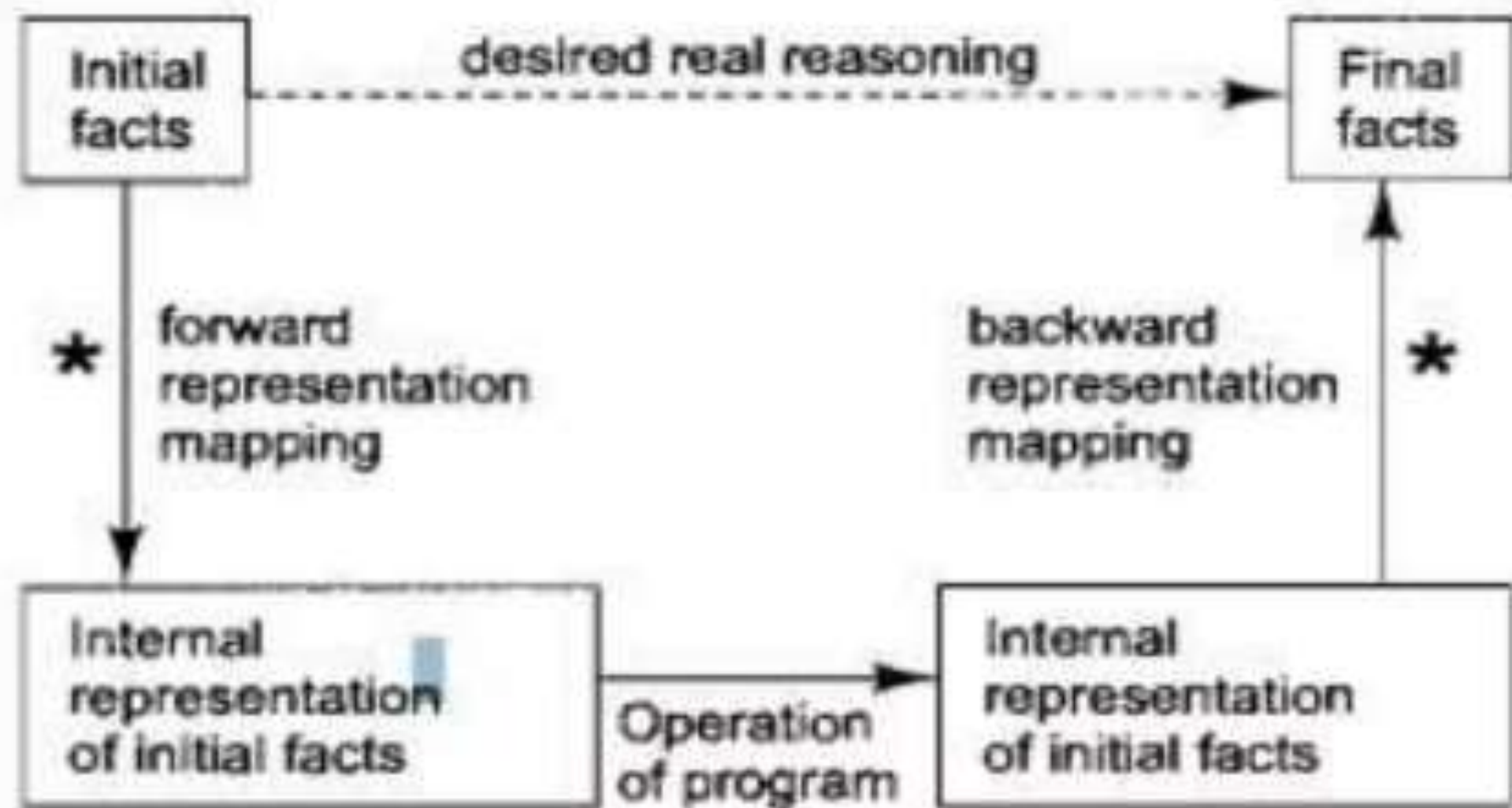


Fig. 4.3 *Representation of Facts*

Approaches to knowledge Representation

- Representational adequacy:- the ability to represent all of the kinds of knowledge that are needed in that domain.
- Inferential Adequacy:- the ability to manipulate the representation structures in such a way as to derive new structures corresponding to new knowledge inferred from old.
- Inferential Efficiency:- the ability to incorporate into the knowledge structure additional information that can be used to focus the attention of the inference mechanism in the most promising directions.
- Acquisitional Efficiency:- the ability to acquire new information easily. The simplest case involves direct insertion by a person, of new knowledge into the database.

Approaches to knowledge representation:

- There are mainly four approaches to knowledge representation, which are given below:

1. Simple relational knowledge:

- It is the simplest way of storing facts which uses the relational method, and each fact about a set of the object is set out systematically in columns.
- This approach of knowledge representation is famous in database systems where the relationship between different entities is represented.
- This approach has little opportunity for inference.

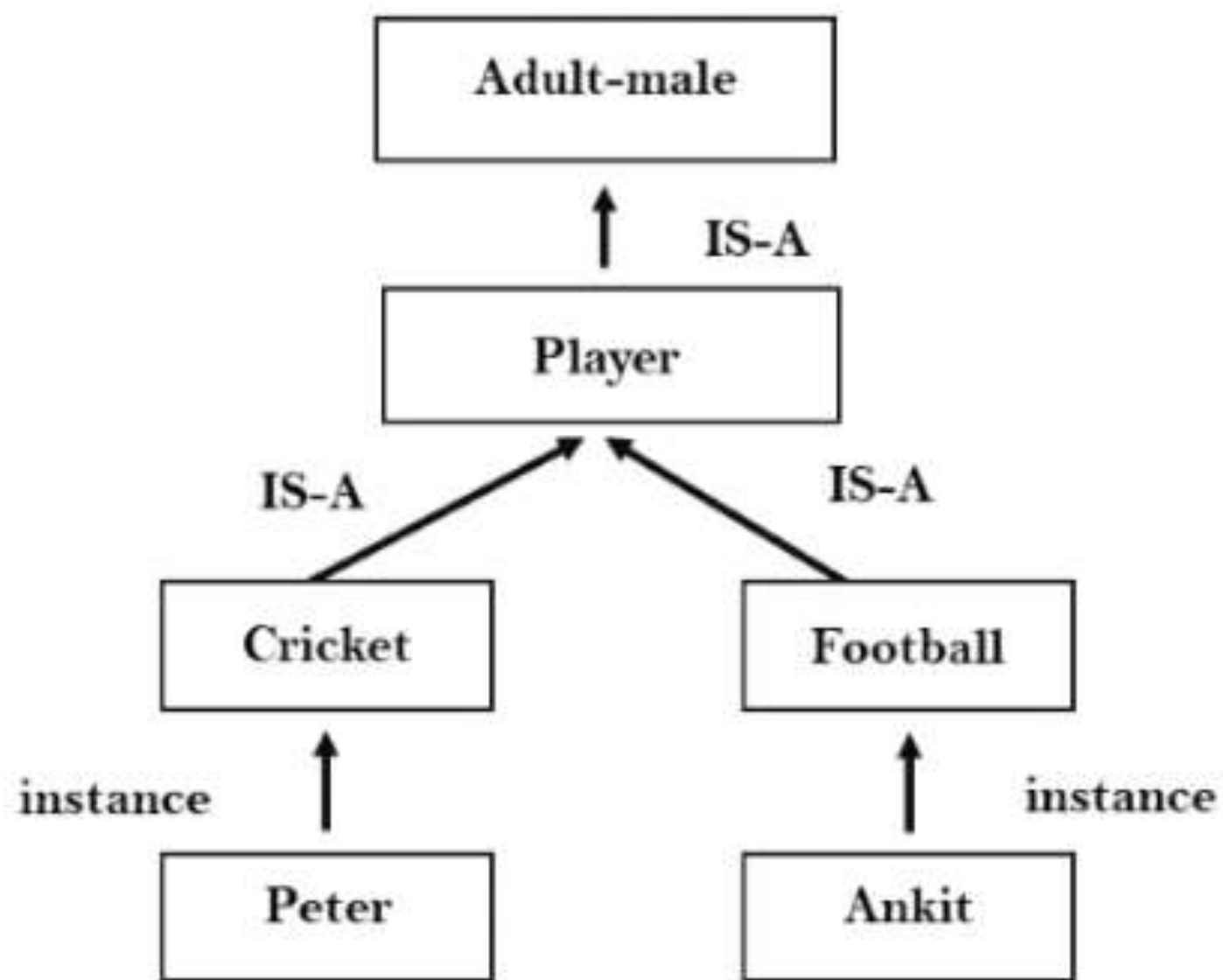
- **Example: The following is the simple relational knowledge representation.**

Player	Weight	Age
Player1	65	23
Player2	58	18
Player3	75	24

2. Inheritable knowledge:

- In the inheritable knowledge approach, all data must be stored into a hierarchy of classes.
- All classes should be arranged in a generalized form or a hierarchal manner.
- In this approach, we apply inheritance property.
- Elements inherit values from other members of a class.
- This approach contains inheritable knowledge which shows a relation between instance and class, and it is called instance relation.
- Every individual frame can represent the collection of attributes and its value.
- In this approach, objects and values are represented in Boxed nodes.
- We use Arrows which point from objects to their values.

◦ Example:



3. Inferential knowledge:

- Inferential knowledge approach represents knowledge in the form of formal logics.
- This approach can be used to derive more facts.
- It guaranteed correctness.
- **Example:** suppose there are two statements:
 - Marcus is a man
 - All men are mortalThen it can represent as;

man(Marcus)

$\forall x = \text{man}(x) \text{ -----} \rightarrow \text{mortal}(x)$

4. Procedural knowledge:

- Procedural knowledge approach uses small programs and codes which describes how to do specific things, and how to proceed.
- In this approach, one important rule is used which is **If-Then rule**.
- In this knowledge, we can use various coding languages such as **LISP language** and **Prolog language**.
- We can easily represent heuristic or domain-specific knowledge using this approach.
- But it is not necessary that we can represent all cases in this approach.

Issues in Knowledge Representation

- The fundamental goal of knowledge Representation is to facilitate inference (conclusions) from knowledge.
- Issues that arises while KR techniques:
 1. Important Attributes
 2. Relationships among Attributes.
 3. Choosing the granularity of Representation.
 4. Representing Sets of Objects.
 5. Finding the Right structures as Needed

Baseball knowledge

- *isa* : show class inclusion
- *instance* : show class membership



Fig. Inheritable knowledge representation (KR)

1. Important Attributed:

- Any attribute of objects so basic that they occur in almost every problem domain?
- There are two attributed “instance” and “isa”, that are general significance. These attributes are important because they support property inheritance.

2. Relationship among attributes:

- Any important relationship that exists among object attributed?
- The attributes we use to describe objects are themselves entities that we represent.

- The relationship between the attributes of an object, independent of specific knowledge they encode, may hold properties like:
- **Inverse** — This is about consistency check, while a value is added to one attribute. The entities are related to each other in many different ways.
- **Existence in an isa hierarchy** — This is about generalization-specification, like, classes of objects and specialized subsets of those classes, there are attributes and specialization of attributes. For example, the attribute height is a specialization of general attribute physical-size which is, in turn, a specialization of physical-attribute. These generalization-specialization relationships are important for attributes because they support inheritance.

- **Technique for reasoning about values** — This is about reasoning values of attributes not given explicitly. Several kinds of information are used in reasoning, like, height: must be in a unit of length, Age: of a person cannot be greater than the age of person's parents. The values are often specified when a knowledge base is created.
- **Single valued attributes** — This is about a specific attribute that is guaranteed to take a unique value. For example, a baseball player can at time have only a single height and be a member of only one team. KR systems take different approaches to provide support for single valued attributes.

3. Choosing Granularity:

- At what level of detail should the knowledge be represented?
- Regardless of the KR formalism, it is necessary to know:
 - At what level should the knowledge be represented and what are the primitives?
 - Should there be a small number or should there be a large number of low-level primitives or High-level facts.
- High-level facts may not be adequate for inference while Low-level primitives may require a lot of storage.

Example of Granularity:

- Suppose we are interested in following facts:

John spotted Sue.

This could be represented as
Spotted (agent(John),object (Sue))

Such a representation would make it easy to answer questions such are:

- Who spotted Sue?

Suppose we want to know:

- Did John see Sue?

Given only one fact, we cannot discover that answer.

We can add other facts, such as

Spotted(x, y) -> saw(x, y)

We can now infer the answer to the question.

4. Set of objects:

- There are certain properties of objects that are true as member of a set but not as individual;

Example: Consider the assertion made in the sentences:

*“there are more sheep than people in Australia”, and
“English speakers can be found all over the world.”*

To describe these facts, the only way is to attach assertion to the sets representing people, sheep, and English.

The reason to represent sets of objects is: if a property is true for all or most elements of a set, then it is more efficient to associate it once with the set rather than to associate it explicitly with every elements of the set.

This is done,

- in logical representation through the use of universal quantifier, and
- in hierarchical structure where node represent sets and inheritance propagate set level assertion down to individual.

5. Finding Right structure:

- Given a large amount of knowledge stored in a database, how can relevant parts be accessed when they are needed?
- This is about access to right structure for describing a particular situation.
- This requires, selecting an initial structure and then revising the choice.

Using Predicate Logic

Logic

Logic is concerned with the truth of statements about the world. Generally each statement is either *TRUE* or *FALSE*.

Logic includes :*Syntax* , *Semantics* and *Inference Procedure*.

◇ **Syntax** :

Specifies the *symbols* in the language about how they can be combined to form sentences. The **facts about the world are represented** as sentences in logic.

◇ **Semantic** :

Specifies how to assign a truth value to a sentence based on its *meaning* in the world. It Specifies what facts a sentence refers to. A fact is a claim about the world, and it may be *TRUE* or *FALSE*.

◇ **Inference Procedure** :

Specifies *methods* for computing new sentences from the existing sentences.

- Logic symbols used in predicate logic:

$\forall$ – *For all*

$\exists$ – *There exists*

$\rightarrow$ – *Implies*

$\neg$ – *Not*

$\vee$ – *OR*

$\wedge$ – *AND*

Representing Simple Facts in Logic

Q Using propositional logic

–Real-world facts are represented as logical propositions written as well-formed formulas (wff's)

–Example 1:

It is raining	RAINING
It is sunny	SUNNY
It is Windy	WINDY
If it is raining, then it is not sunny	$\text{RAINING} \rightarrow \neg \text{SUNNY}$

Representing Simple Facts in Logic

– Example 2:

Socrates is a man	SOCRATESMAN
Plato is a man	PLATOMAN
All men are mortal	MORTALMAN

=> Since the assertions are separate, it is not possible to draw any conclusion about similarities between Socrates and Plato.

Representing Simple Facts in Logic

It would be much better to represent these facts as

Socrates is a man	man(socrates)
Plato is a man	man(plato)
All men are mortal	mortalman

- This fails to capture the relationship between any individual being a man and that individual being a mortal.
- Therefore it is necessary to move to first order predicate logic as a way of representing knowledge because it permits representation of things that cannot reasonably be represented in propositional logic.
- In predicate logic, real world facts are represented as statements written as wff's.

Representing Simple Facts in Logic

Consider the following set of sentences.

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeians were Romans.
4. Caesar was a ruler.
5. All Romans were either loyal to Caesar or hated him.
6. Every one is loyal to someone.
7. People only try to assassinate rulers they are not loyal to.
8. Marcus tried to assassinate Caesar

1. Marcus was a man.

man(Marcus)

2. Marcus was a Pompeian.

Pompeian(Marcus)

3. All Pompeians were Romans.

$\forall x : \text{Pompeian}(x) \rightarrow \text{Roman}(x)$

4. Caesar was a ruler.

ruler(Caesar)

5. All Romans were either loyal to Caesar or hated him.

$\forall x : \text{Roman}(x) \rightarrow \text{loyalto}(x, \text{Caesar}) \vee \text{hate}(x, \text{Caesar})$

6. Everyone is loyal to someone.

$$\forall x : \rightarrow y : \textit{loyalto}(x,y)$$

7. People only try to assassinate rulers they are not loyal to.

$$\forall x : \forall y : \textit{person}(x) \wedge \textit{ruler}(y) \wedge \textit{tryassassinate}(x,y) \rightarrow \neg \textit{loyalto}(x,y)$$

8. Marcus tried to assassinate Caesar.

$$\textit{tryassassinate}(\textit{Marcus}, \textit{Caesar})$$

Now suppose that we want to use these statements to answer the question

Was Marcus loyal to Caesar?

It seems that using 7 and 8, we should be able to prove that Marcus was not loyal to Caesar (again ignoring the distinction between past and present tense). Now let's try to produce a formal proof, reasoning backward from the desired goal:

$$\neg \textit{loyalto}(\textit{Marcus}, \textit{Caesar})$$

9. All men are people.

$$\forall : \textit{man}(x) \rightarrow \textit{person}(x)$$

REPRESENTING KNOWLEDGE USING RULES

- Procedural and Declarative Knowledge
- Forward Reasoning
- Backward Reasoning

Representing knowledge using rules:

- Consider a set of rules that represent,
 1. Knowledge about relationships in the world and
 2. Knowledge about how to solve the problem using the content of the rules.

Procedural vs Declarative Knowledge

- Procedural Knowledge
 - A representation in which the control information that is necessary to use the knowledge is embedded in the knowledge itself for e.g. computer programs, directions, and recipes; these indicate specific use or implementation;
 - The real difference between declarative and procedural views of knowledge lies in where control information reside.

- For example, consider the following
- Man (Marcus)
- Man (Caesar)
- Person (Cleopatra)
- $\forall x: \text{Man}(x) \rightarrow \text{Person}(x)$
- Now, try to answer the question. ?Person(y)
- The knowledge base justifies any of the following answers.
- Y=Marcus
- Y=Caesar
- Y=Cleopatra

- We get more than one value that satisfies the predicate.
- If only one value needed, then the answer to the question will depend on the order in which the assertions examined during the search for a response.
- If the assertions declarative then they do not themselves say anything about how they will be examined. In case of procedural representation, they say how they will examine.
- **Declarative Knowledge**
- A statement in which knowledge specified, but the use to which that knowledge is to be put is not given.
- For example, laws, people's name; these are the facts which can stand alone, not dependent on other knowledge;
- So to use declarative representation, we must have a program that explains what is to do with the knowledge and how.

- For example, a set of logical assertions can combine with a resolution theorem prover to give a complete program for solving problems but in some cases, the logical assertions can view as a program rather than data to a program.
- Hence the implication statements define the legitimate reasoning paths and automatic assertions provide the starting points of those paths.
- These paths define the execution paths which is similar to the ‘if then else “in traditional programming.
- So logical assertions can view as a procedural representation of knowledge.

Difference the Procedural and Declarative Knowledge

PROCEDURAL KNOWLEDGE	DECLARATIVE KNOWLEDGE
It is also known as Interpretive knowledge.	It is also known as Descriptive knowledge.
Procedural Knowledge means how a particular thing can be accomplished	While Declarative Knowledge means basic knowledge about something.
Procedural Knowledge is generally not used means it is not more popular.	Declarative Knowledge is more popular.
Procedural Knowledge can't be easily communicate.	Declarative Knowledge can be easily communicate
Procedural Knowledge is generally process oriented in nature	Declarative Knowledge is data oriented in nature.
In Procedural Knowledge debugging and validation is not easy.	In Declarative Knowledge debugging and validation is easy.

Forward Reasoning

- The solution of a problem generally includes the initial data and facts in order to arrive at the solution. These unknown facts and information is used to deduce the result
- For example, while diagnosing a patient the doctor first check the symptoms and medical condition of the body such as temperature, blood pressure, pulse, eye colour, blood, etcetera. After that, the patient symptoms are analysed and compared against the predetermined symptoms. Then the doctor is able to provide the medicines according to the symptoms of the patient. So, when a solution employs this manner of reasoning, it is known as **forward reasoning**.

Steps that are followed in the forward reasoning

1. In the first step, the system is given one or more than one constraints.
2. Then the rules are searched in the knowledge base for each constraint. The rules that fulfill the condition are selected(i.e., IF part).
3. Now each rule is able to produce new conditions from the conclusion of the invoked one. As a result, THEN part is again included in the existing one.
4. The added conditions are processed again by repeating step 2. The process will end if there is no new conditions exist.

Backward Reasoning

- The **backward reasoning** is inverse of forward reasoning in which goal is analysed in order to deduce the rules, initial facts and data.
- We can understand the concept by the similar example given in the above definition, where the doctor is trying to diagnose the patient with the help of the inceptive data such as symptoms. However, in this case, the patient is experiencing a problem in his body, on the basis of which the doctor is going to prove the symptoms. This kind of reasoning comes under backward reasoning.

Steps that are followed in the backward reasoning

1. Firstly, the goal state and the rules are selected where the goal state reside in the THEN part as the conclusion.
2. From the IF part of the selected rule the sub goals are made to be satisfied for the goal state to be true.
3. Set initial conditions important to satisfy all the sub goals.
4. Verify whether the provided initial state matches with the established states. If it fulfills the condition then the goal is the solution otherwise other goal state is selected.

Difference between backward chaining and forward chaining

Forward Chaining	Backward Chaining
It starts from known facts and applies inference rule to extract more data unit it reaches to the goal.	It starts from the goal and works backward through inference rules to find the required facts that support the goal.
It is a bottom-up approach	It is a top-down approach
It is known as data-driven inference technique as we reach to the goal using the available data.	It is known as goal-driven technique as we start from the goal and divide into sub-goal to extract the facts.

It applies a breadth-first search strategy.	It applies a depth-first search strategy.
It tests for all the available rules	It tests only for few required rules.
It is suitable for the planning, monitoring, control, and interpretation application.	It is suitable for diagnostic, prescription, and debugging application.
It can generate an infinite number of possible conclusions.	It generates a finite number of possible conclusions.
It operates in the forward direction.	It operates in the backward direction.

2.Logic programming

- Logic Programming – Representing Knowledge Using Rules
- ❖ Logic programming is a programming paradigm in which logical assertions viewed as programs.
- ❖ These are several logic programming systems, PROLOG is one of them.
- ❖ A PROLOG program consists of several logical assertions where each is a horn clause i.e. a clause with at most one positive literal.
- ❖ Ex : $P, P \vee Q, P \rightarrow Q, \neg P \vee Q$
- ❖ The facts are represented on Horn Clause for two reasons.
 - 1. Because of a uniform representation, a simple and efficient interpreter can write.
 - 2. The logic of Horn Clause decidable.

- ❖ Also, The first two differences are the fact that PROLOG programs are actually sets of Horn clause that have been transformed as follows:-
- 1. If the Horn Clause contains no negative literal then leave it as it is.
- 2. Also, Otherwise rewrite the Horn clauses as an implication, combining all of the negative literals into the antecedent of the implications and the single positive literal into the consequent.
- ❖ Moreover, This procedure causes a clause which originally consisted of a disjunction of literals (one of them was positive) to be transformed into a single implication whose antecedent is a conjunction universally quantified.
- ❖ But when we apply this transformation, any variables that occurred in negative literals and so now occur in the antecedent become existentially quantified, while the variables in the consequent are still universally quantified.

- For example the PROLOG clause $P(x) :- Q(x, y)$ is equal to logical expression $\forall x: \exists y: Q(x, y) \rightarrow P(x)$.
- ❖ The difference between the logic and PROLOG representation is that the PROLOG interpretation has a fixed control strategy. And so, the assertions in the PROLOG program define a particular search path to answer any question.
- ❖ But, the logical assertions define only the set of answers but not about how to choose among those answers if there is more than one.

$\forall x : \text{pet}(x) \wedge \text{small}(x) \rightarrow \text{apartmentpet}(x)$
 $\forall x : \text{cat}(x) \vee \text{dog}(x) \rightarrow \text{pet}(x)$
 $\forall x : \text{poodle}(x) \rightarrow \text{dog}(x) \wedge \text{small}(x)$
 $\text{poodle}(\text{fluffy})$

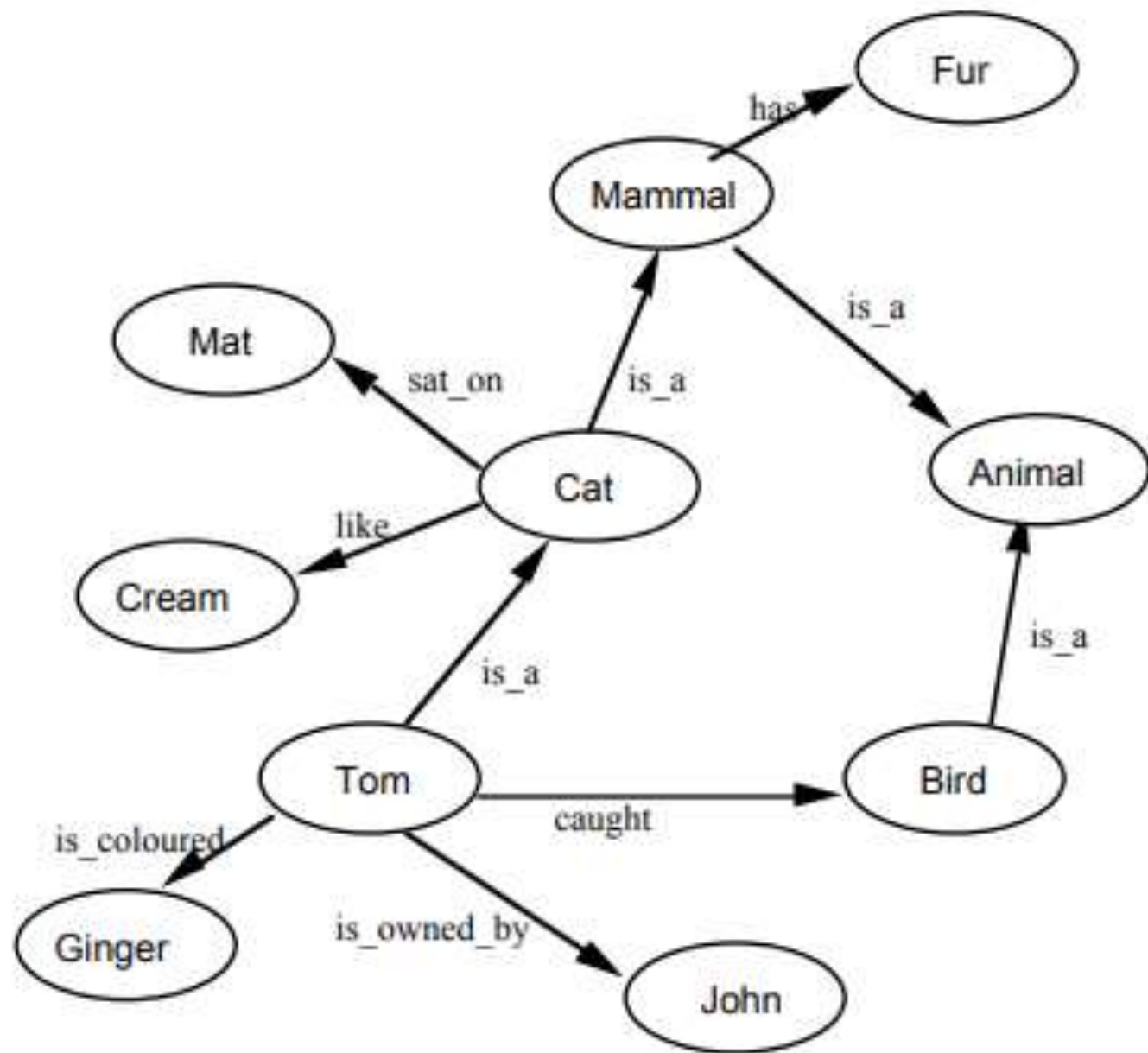
A Representation in Logic

```
apartmentpet(X) :- pet(X), small(X).  
pet(X) :- cat(X).  
pet(X) :- dog(X).  
dog(X) :- poodle(X).  
small(X) :- poodle(X).  
poodle(fluffy).
```

A Representation in PROLOG

Semantic Nets and Frames

- Semantic Nets
- Semantic networks are an alternative to predicate logic as a form of knowledge representation.
- The idea is that we can store our knowledge in the form of a graph, with nodes representing objects in the world, and arcs representing relationships between those objects. For example, the following:

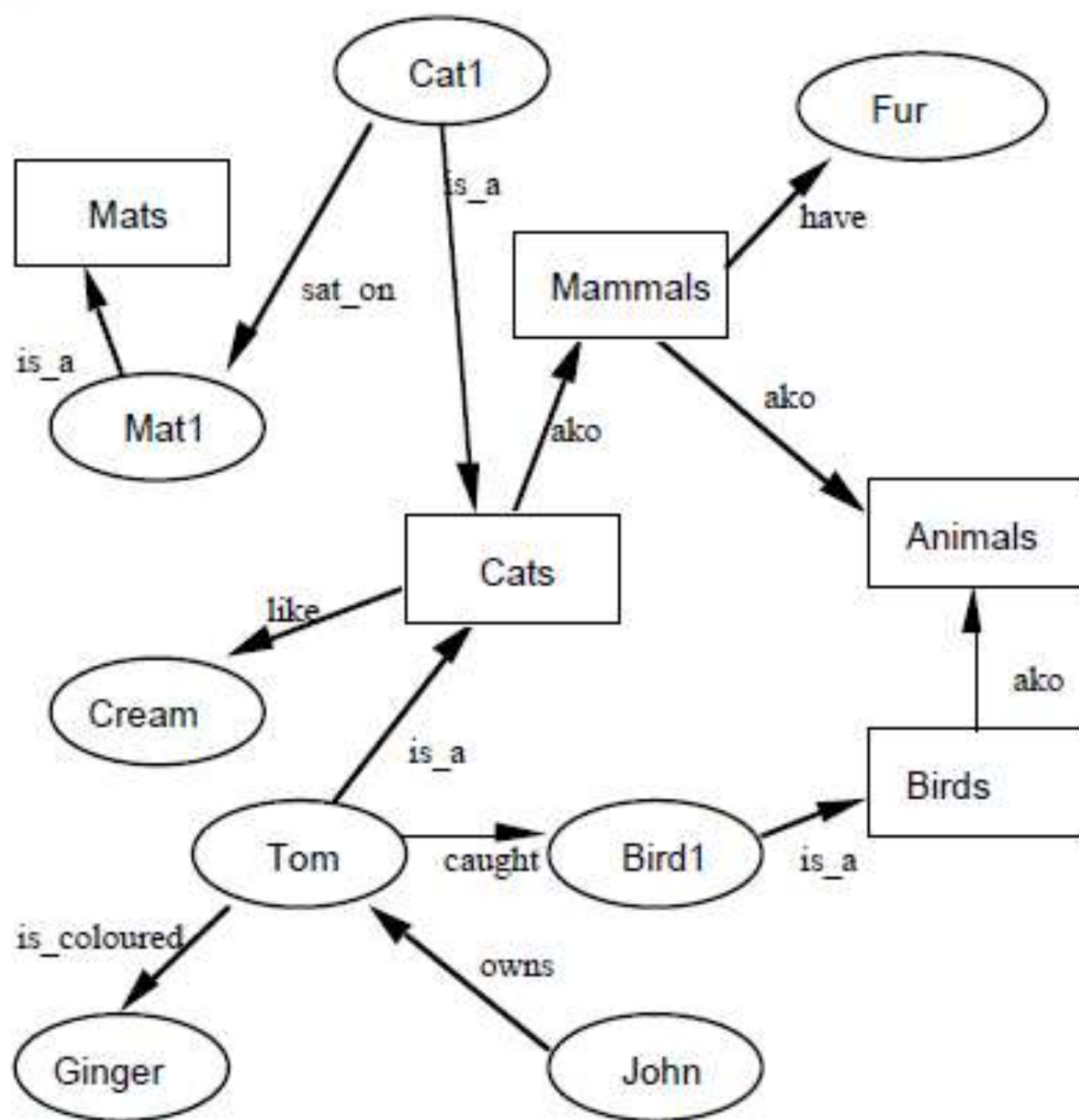


- is intended to represent the data:
- Tom is a cat.
- Tom caught a bird.
- Tom is owned by John.
- Tom is ginger in colour.
- Cats like cream.
- The cat sat on the mat.
- A cat is a mammal.
- A bird is an animal.
- All mammals are animals.
- Mammals have fur.

- we haven't distinguished between nodes representing classes of things, and nodes representing individual objects.
- So, for example, the node labelled Cat represents both the single (nameless) cat who sat on the mat, and the whole class of cats to which Tom belongs, which are mammals and which like cream.
- The is_a link has two different meanings – it can mean that one object is an individual item from a class, for example Tom is a member of the class of cats, or that one class is a subset of another, for example, the class of cats is a subset of the class of mammals.

- We can clean up the representation by distinguishing between nodes representing individual or instances, and nodes representing *classes*.
- The is_a link will only be used to show an individual belonging to a class. The link representing one class being a subset of another will be labelled a_kind_of, or ako for short.
- The names instance and subclass are often used in the place of is_a and ako, but we will use these terms with a slightly different meaning in the section on Frames below.
- Note also the modification

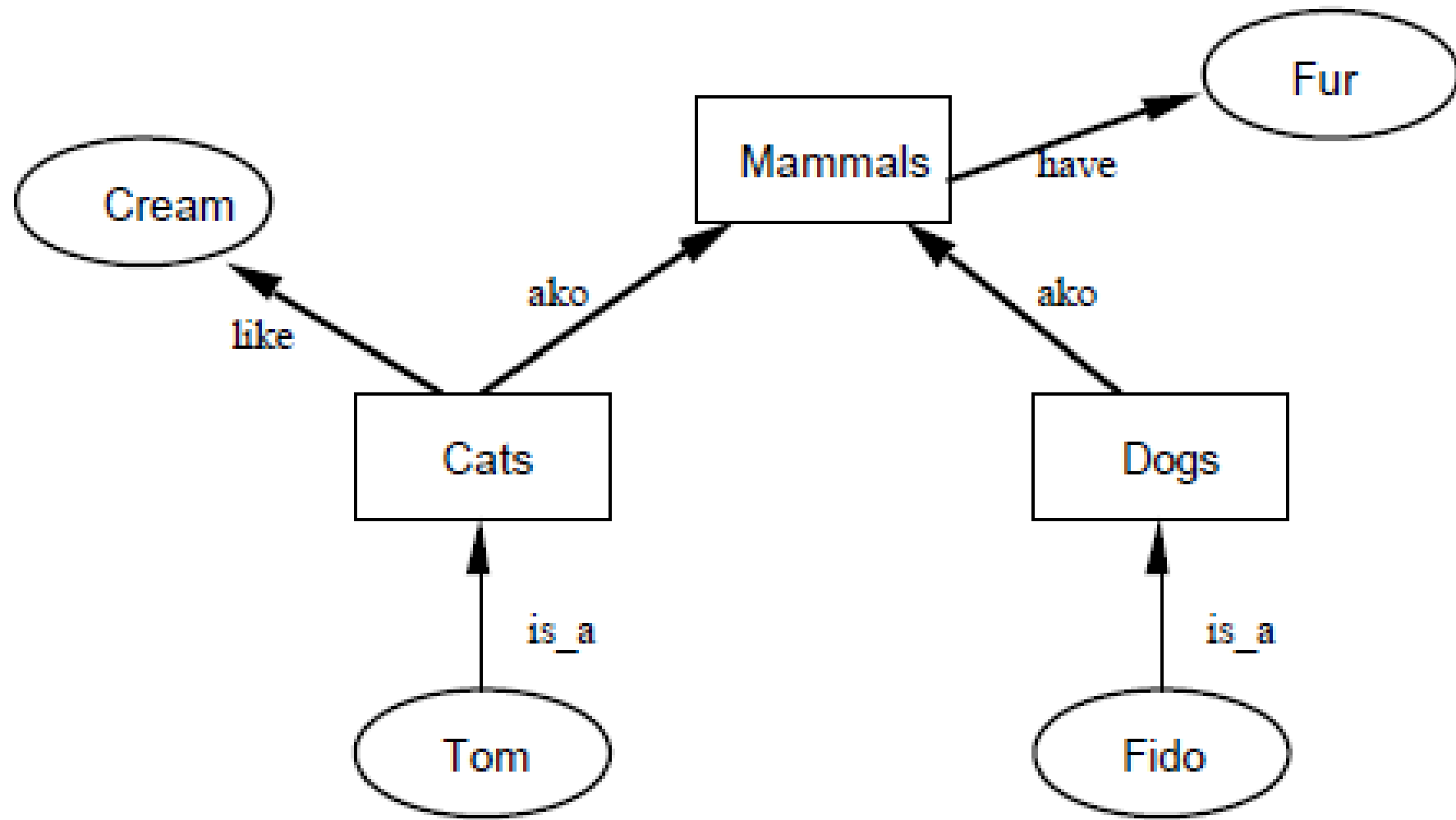
The revised semantic net is:



Inheritance

- This Prolog equivalent captures an important property of semantic nets, that they may be used for a form of inference known as *inheritance*.
- The idea of this is that if an object belongs to a class (indicated by an *is_a* link) it *inherits* all the properties of that class.
- So, for example as we have a *likes* link between cats and cream, meaning “all cats like cream”, we can infer that any object which has an *is_a* link to cats will like cream.
- So both Tom and Cat1 like cream. However, the *is_coloured* link is between Tom and ginger, not between cats and ginger, indicating that being ginger is a property of Tom as an individual, and not of all cats.

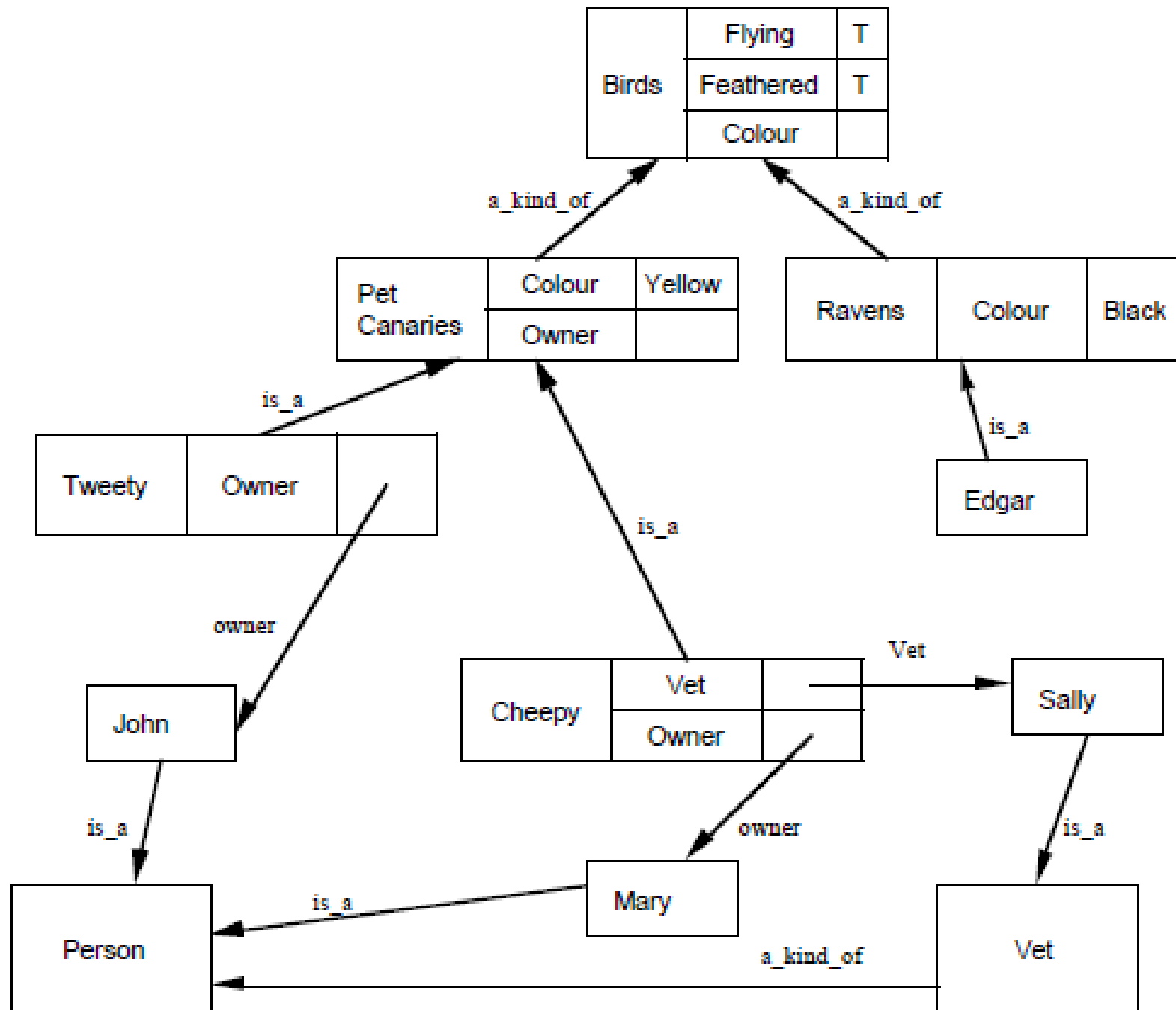
- Inheritance also applies across the a_kind_of links.
- For example, any property of mammals or animals will automatically be a property of cats.
- So we can infer, for example, that Tom has fur, since Tom is a cat, a cat is a kind of mammal, and mammals have fur.
- If, for example, we had another subclass of mammals, say dogs, and we had, say, Fido is_a dog, Fido would inherit the property has fur from mammals, but not the property likes cream, which is specific to cats.
- This situation is shown in the diagram below:



Frames, Slots and Fillers

- the *frame* notation is used to tighten up on the semantic net notation to give something which is more consistent.
- In the place of an arbitrary number of arcs leading from a node there are a fixed number of slots representing attributes of an object.
- Every object is a member or *instance* of a class, which it may be thought of as linking to with an is_a link as we saw before.
- The class indicates the number of slots that an object has, and the name of each slot.

- The idea of inheritance is used, with some slots being filled at class level, and some at instance level.
- Where a slot is filled at class level the idea is that this represents attributes which are common to all members of that class.
- Where it is filled at instance level, it indicates that the value of that attribute varies among members of that class.
- Slots may be filled with values or with pointers to other objects.





Common raven

Temporal range:

Middle Pleistocene–Recent

PreЄ Є S D C P T J K P_oN



Near the Tower of London, England

CONSTRAINT SATISFACTION PROBLEM, CSP

- **Constraint Satisfaction Problem**, or CSP use a **factored representation** for each state: a set of variables, each of which has a value. A problem is solved when each variable has a value that satisfies all the constraints on the variable.
- Search algorithms take advantage of the structure of states and use *general-purpose* rather than *problem-specific* heuristics to enable the solution of complex problems.
- The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints.

Constraint satisfaction

constraint satisfaction means *solving a problem under certain constraints or rules*.

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

Constraint satisfaction depends on three components, namely:

X: It is a set of variables.

D: It is a set of domains where the variables reside. There is a specific domain for each variable.

C: It is a set of constraints which are followed by the set of variables.

Solving Constraint Satisfaction Problems

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

A state-space

The notion of the solution.

A state in state-space is defined by assigning values to some or all variables such as

$\{X_1=v_1, X_2=v_2, \text{ and so on...}\}$.

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

Consistent or Legal Assignment: An assignment which does not violate any constraint or rule is called Consistent or legal assignment.

Complete Assignment: An assignment where every variable is assigned with a value, and the solution to the CSP remains consistent. Such assignment is known as Complete assignment.

Partial Assignment: An assignment which assigns values to some of the variables only. Such type of assignments are called Partial assignments.

Example: Map-Coloring



- **Variables** WA, NT, Q, NSW, V, SA, T
- **Domains** $D_i = \{\text{red}, \text{green}, \text{blue}\}$
- **Constraints**: adjacent regions must have different colors
e.g., $WA \neq NT$, or (WA, NT) in $\{(\text{red}, \text{green}), (\text{red}, \text{blue}), (\text{green}, \text{red}), (\text{green}, \text{blue}), (\text{blue}, \text{red}), (\text{blue}, \text{green})\}$

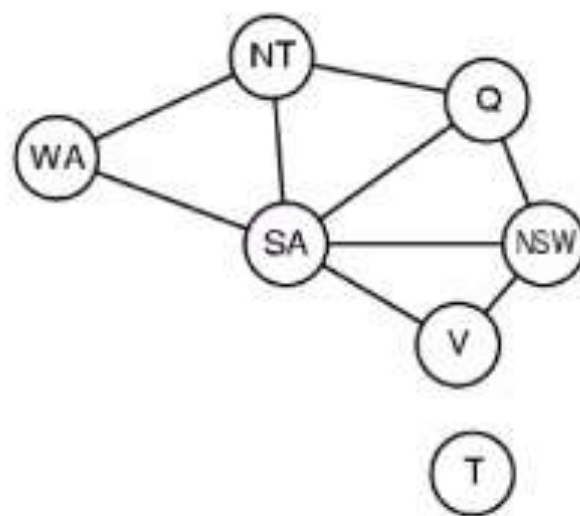
Example: Map-Coloring



- Solutions are **complete** and **consistent** assignments
- e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

Constraint graph

- **Binary CSP:** each constraint relates two variables
- **Constraint graph:** nodes are variables, arcs are constraints



Varieties of constraints

- **Unary** constraints involve a single variable,
 - e.g., $SA \neq \text{green}$
- **Binary** constraints involve pairs of variables,
 - e.g., $SA \neq WA$
- **Higher-order** constraints involve 3 or more variables,
 - e.g., cryptarithmic column constraints

Constraint Propagation

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

We can search for a solution or

We can perform a special type of inference called **constraint propagation**.

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

In local consistency, variables are treated as **nodes**, and each binary constraint is treated as an **arc** in the given problem.

Constraint Propagation

There are following local consistencies which are discussed below:

Node Consistency: A single variable is said to be node consistent if all the values in the variable's domain satisfy the unary constraints on the variables.

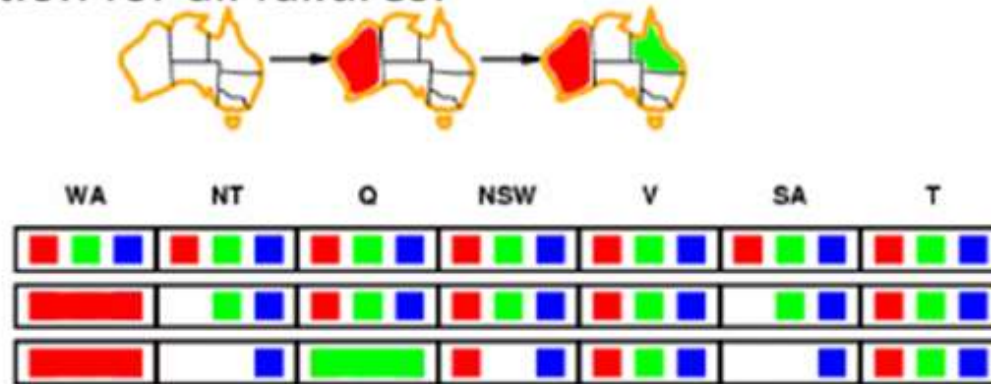
Arc Consistency: A variable is arc consistent if every value in its domain satisfies the binary constraints of the variables.

Path Consistency: When the evaluation of a set of two variable with respect to a third variable can be extended over another variable, satisfying all the binary constraints. It is similar to arc consistency.

k-consistency: This type of consistency is used to define the notion of stronger forms of propagation. Here, we examine the k-consistency of the variables.

Constraint propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



- NT and SA cannot both be blue!
- Constraint propagation algorithms repeatedly enforce constraints locally...

k-consistency

- A CSP is *k-consistent* if, for any set of $k-1$ variables, and for any consistent assignment to those variables, a consistent value can always be assigned to any k th variable
- 1-consistency is node consistency
- 2-consistency is arc consistency
- For binary constraint networks, 3-consistency is the same as *path consistency*
- Getting k -consistency requires time and space exponential in k
- *Strong k-consistency* means k' -consistency for all k' from 1 to k
 - Once strong k -consistency for $k=\text{\#variables}$ has been obtained, solution can be constructed trivially
- Tradeoff between propagation and branching
- Practitioners usually use 2-consistency and less commonly 3-consistency

Applications of Constraint Propagation

Constraint propagation is widely used in various AI applications. Some notable areas include:

- **Scheduling**
 - In scheduling problems, tasks must be assigned to time slots without conflicts. Constraint propagation helps by reducing the possible time slots for each task based on constraints like availability and dependencies.
- **Planning**
 - AI planning involves creating a sequence of actions to achieve a goal. Constraint propagation simplifies the planning process by reducing the possible actions at each step, ensuring that the resulting plan satisfies all constraints.
- **Resource Allocation**
 - In resource allocation problems, resources must be assigned to tasks in a way that meets all constraints, such as capacity limits and priority rules. Constraint propagation helps by narrowing down the possible assignments, making the search for an optimal allocation more efficient.

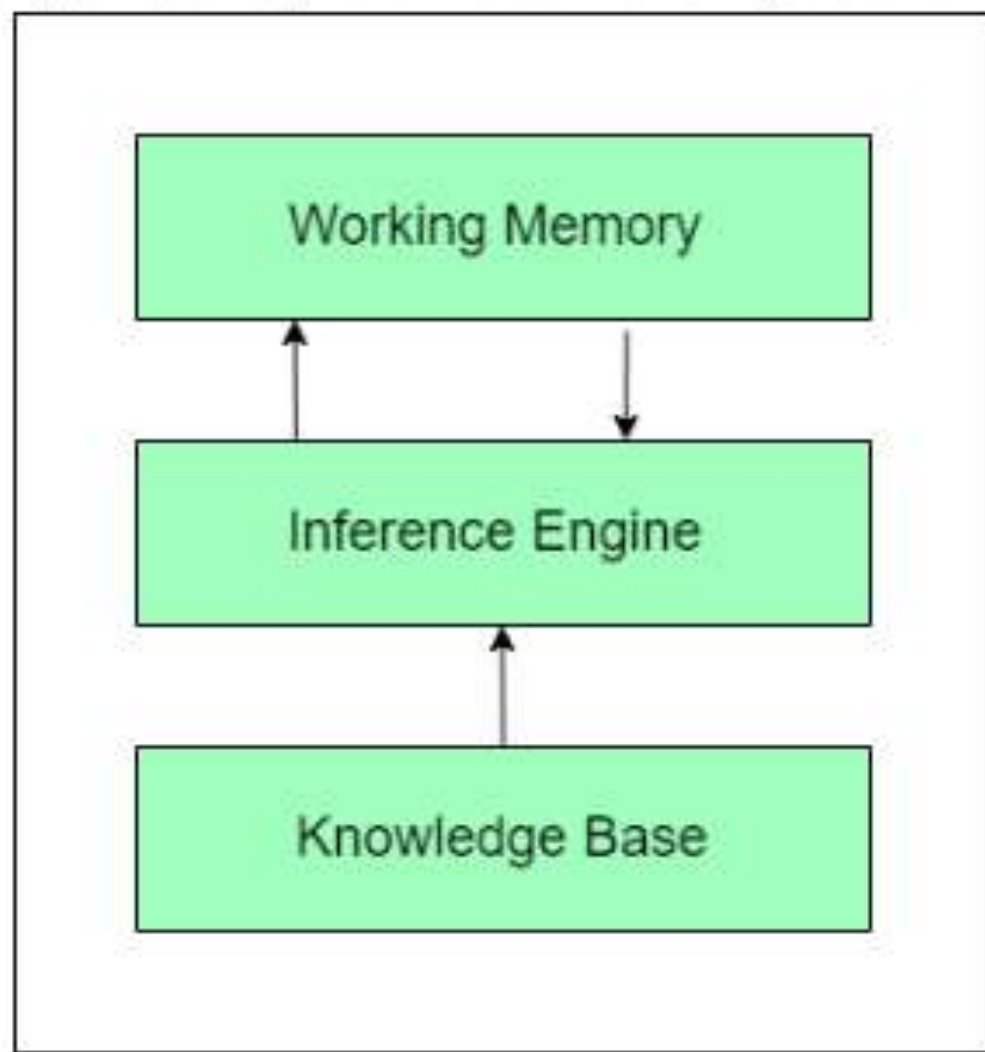
Rule-Based System

- Rule-based systems, a foundational technology in artificial intelligence (AI), have long been instrumental in decision-making and problem-solving across various domains.
- These systems operate on a set of predefined rules and logic to make decisions, perform tasks, or derive conclusions.
- Despite the rise of more advanced AI methodologies, such as machine learning and neural networks, rule-based systems remain crucial due to their transparency, ease of use, and interpretability.

Components of a Rule-Based System

- A typical rule-based system comprises several key components:
- **Rules:** The core of the system, these are conditional statements that define the system's behavior. A rule generally follows the format "IF condition THEN action." For example, in an expert system for medical diagnosis, a rule might be "IF patient has fever AND cough THEN consider flu."
- **Knowledge Base:** This is the repository where all the rules and facts are stored. The knowledge base is built from domain-specific knowledge and can be manually curated or derived from expert input.
- **Inference Engine:** The inference engine is the component that applies the rules to the knowledge base to derive conclusions or make decisions. It interprets the rules, processes them against the current facts or data, and determines the appropriate actions or outputs.

- **Working Memory:** This is a dynamic component that holds the current facts or data being processed by the system. It is updated as the inference engine applies rules and new information becomes available.
- **User Interface:** In many rule-based systems, the user interface allows users to interact with the system, input data, and receive outputs or recommendations.



Rule Based System in AI

- The operation of a rule-based system involves several stages:
- **Data Input:** The system receives input data from the user or another source. This data can range from simple numerical values to complex information like patient symptoms or transaction records.
- **Rule Matching:** The inference engine examines the input data against the rules stored in the knowledge base. It looks for rules whose conditions match the input data.
- **Rule Execution:** Once a rule is matched, the inference engine executes the corresponding action. This might involve updating the working memory, deriving new facts, or generating an output.

- **Conflict Resolution:** In cases where multiple rules are triggered simultaneously, the inference engine uses conflict resolution strategies to determine which rule to apply first. Common strategies include prioritizing rules based on specificity or order of entry.
- **Output Generation:** The system generates an output based on the executed rules. This output can be a decision, recommendation, or another form of response. For example, in a medical diagnosis system, the output might be a suggested treatment plan.

- Consider a simplified example of a rule-based system used in a customer service chatbot:
- **Rule 1:** *If the customer asks about their account balance, then retrieve and display the current balance.*
- **Rule 2:** *If the customer asks about recent transactions, then retrieve and display the last five transactions.*
- **Rule 3:** *If the customer asks to speak to a human agent, then transfer the chat to a human operator.*

- There are several types of rule-based systems, each tailored to different applications:
- **Forward Chaining Systems:** These systems start with the available data and apply rules to infer new data until a goal is reached. Forward chaining is often used in problem-solving and diagnostic systems.
- **Backward Chaining Systems:** These systems start with a goal and work backward to determine which rules and data can achieve that goal. Backward chaining is commonly used in expert systems where the goal is to reach a specific diagnosis or conclusion.
- **Hybrid Systems:** Some systems combine forward and backward chaining to leverage the strengths of both approaches. Hybrid systems are useful in complex scenarios where both data-driven and goal-driven reasoning are required.

Applications of Rule-Based Systems

- Rule-based systems have a broad range of applications, including:
- **1. Expert Systems**
- [Expert systems](#) are designed to emulate the decision-making abilities of human experts. They use a large set of rules to make inferences or recommendations.
- Applications include:
- **Medical Diagnosis:** Systems like MYCIN, an early expert system, assist in diagnosing diseases and recommending treatments based on patient data and medical knowledge.
- **Financial Services:** Expert systems can assess credit risks, detect fraud, and provide investment advice.

- **2. Decision Support Systems**

- These systems aid decision-making processes by providing relevant information and recommendations based on predefined rules.
Examples include:
- **Customer Support:** Automated customer support systems use rules to handle common queries and problems, providing quick and consistent responses.
- **Manufacturing:** Rule-based systems help in managing production schedules, inventory control, and quality assurance.

- **3. Control Systems**

- In control systems, rules govern the operation of machinery and equipment. Examples include:
- **Automated Traffic Lights:** Rules determine the timing and sequencing of traffic lights based on traffic flow and other factors.
- **Home Automation:** Smart home systems use rules to control lighting, heating, and security systems based on user preferences and sensor data.

Benefits of Rule-Based Systems

- **Transparency:** Rule-based systems are highly transparent because the rules governing their decisions are explicit and understandable. This clarity makes it easier to trace and debug the system's behavior.
- **Ease of Implementation:** For well-defined problems with clear rules, rule-based systems are relatively easy to implement. They do not require extensive data for training, unlike machine learning models.
- **Consistency:** Rule-based systems provide consistent responses and decisions as they follow predefined rules. This consistency is crucial in applications where uniformity is essential.
- **Ease of Updating:** Rules can be updated or added to adapt to new knowledge or changes in the domain. This flexibility allows the system to evolve with the changing requirements.

Reasoning under uncertainty

Symbolic Reasoning

- The reasoning is the act of deriving a conclusion from certain properties using a given methodology
- The reasoning is a process of thinking; reasoning is logically arguing; reasoning is drawing the inference.
- When a system is required to do something, that it has not been explicitly told how to do, it must reason. It must figure out what it needs to know from what it already knows.
- Many types of Reasoning have been identified and recognized, but many questions regarding their logical and computational properties still remain controversial.

Logical Reasoning

- Logic is a language for reasoning. It is a collection of rules called Logic arguments, we use when doing logical reasoning.
- The logic reasoning is the process of drawing conclusions from premises using rules of inference.
- The study of logic divided into formal and informal logic. The formal logic is sometimes called symbolic logic.
- Symbolic logic is the study of symbolic abstractions (construct) that capture the formal features of logical inference by a formal system.
- The formal system consists of two components, a formal language plus a set of inference rules.
- The formal system has axioms. Axiom is a sentence that is always true within the system.
- Sentence derived using the system's axioms and rules of derivation called theorems.

Symbolic Reasoning

□ The basis for intelligent mathematical software is the integration of the “power of symbolic mathematical tools” with the suitable “proof technology”.

□ Mathematical reasoning enjoys a property called monotonicity, that says, “If a conclusion follows from given premises A, B, C... then it also follows from any larger set of premises, as long as the original premises A, B, C.. included.”

□ Moreover, Human reasoning is not monotonic.

□ People arrive at conclusions only tentatively; based on partial or incomplete information reserve the right to retract those conclusions while they learn new facts. Such reasoning non-monotonic, precisely because the set of accepted conclusions have become smaller when the set of premises expanded.

Formal Logic

Moreover, The Formal logic is the study of inference with purely formal content, i.e. where content made explicit.

Examples – Propositional logic and Predicate logic.

□ Here the logical arguments are a set of rules for manipulating symbols. The rules are of two types,

1. Syntax rules: say how to build meaningful expressions.
2. Inference rules: say how to obtain true formulas from other true formulas.

□ Moreover, Logic also needs semantics, which says how to assign meaning to expressions.

Uncertainty in Reasoning

- The world is an uncertain place; often the Knowledge is imperfect which causes uncertainty.
- So, Therefore reasoning must be able to operate under uncertainty.
- Also, AI systems must have the ability to reason under conditions of uncertainty rule.

Monotonic Reasoning

- A reasoning process that moves in one direction only.
- Moreover, The number of facts in the knowledge base is always increasing.
- The conclusions derived are valid deductions and they remain so.

A monotonic logic cannot handle

1. Reasoning by default: because consequences may derive only because of lack of evidence to the contrary.
2. Abductive reasoning: because consequences only deduced as most likely explanations.
3. Belief revision: because new knowledge may contradict old beliefs.

Non-monotonic Reasoning

A logic is non-monotonic if some conclusions can be invalidated by adding more knowledge. The logic of definite clauses with negation as failure is non-monotonic. Non-monotonic reasoning is useful for representing defaults. A default is a rule that can be used unless it is overridden by an exception.

For example, to say that b is normally true if c is true, a knowledge base designer can write a rule of the form

$$b \leftarrow c \wedge \sim aba.$$

where aba is an atom that means abnormal with respect to some aspect a . Given c , the agent can infer b unless it is told aba . Adding aba to the knowledge base can prevent the conclusion of b . Rules that imply aba can be used to prevent the default under the conditions of the body of the rule

“ABNORMALITY”

- $p(X) \wedge \text{unless ab } p(X) \rightarrow q(X)$
- ab: abnormal
- Examples: If X is a bird, it will fly unless it is abnormal.
 - (abnormal: broken wing, sick, trapped, ostrich, ...)
- If X is a car, it will run unless it is abnormal.
 - (abnormal: flat tire, broken engine, no gas, ...)

Review of Probability

Probability

- Probabilities are used to compute the truth of given statement, written as numbers between 0 and 1, that describes how likely an event is to occur.
- 0 indicates impossibility and 1 indicates certainly.
 - 1. Tossing a coin 2. Trolling a dice
- Probability based reasoning
 - understanding from knowledge
 - how much of uncertainty present in that event.

Probability

- *Probability provides a way of **summarizing** the uncertainty, that comes from our **laziness** and **ignorance**.*
- Toothache problem - an 80% chance , a probability of 0.8 that the patient has a cavity if he or she has a toothache.
- The 80% summarizes those cases, but both toothache and cavity are unconnected.
- The missing 20% summarizes, all other possible causes of toothache, that we are **too lazy** or **ignorant to confirm** or **deny**.

- Probabilities between 0 and 1 correspond to intermediate degrees of belief in the **truth of the sentence**.
- The sentence itself is in **fact** either **true or false**.
- It is important to note that a **degree of belief** is different from a degree of truth.
- A probability of 0.8 does not mean "80% true" but rather an 80% degree of belief-that is, a fairly strong expectation.
- Thus, probability theory makes the same **ontological commitment** as logic-namely, that facts either do or do not hold in the world.
- Degree of truth, as opposed to degree of belief, is the subject of **fuzzy logic**

- In probability theory, a sentence such as
- "The probability that the patient has a cavity is 0.8",
- is about the agent's beliefs, not directly about the world.
- These percepts create the **evidence**, which are based on probability statements.

- All probability statements must indicate the evidence with respect to that probability is being assessed.
- If an agent receives new percepts, its probability assessments are updated to reflect the new evidence.

Random Variable

- Referring to a "part" of the world, whose "status" is initially unknown
- We will use lowercase for the names of values
 - $P(a) = 1 - P(\neg a)$
- Tossing coin : $P(h) = 1 - P(\neg h) : (0.5 = 1 - 0.5)$
- Rolling dice : $P(n) = 1 - P(\neg n) : (0.16 = 1 - 0.84)$

Types of random variables

- Boolean random variables
 - Cavity domain (true,false), if Cavity = true then cavity, or
 - if Cavity = false then $\neg$ cavity
- Discrete random variables – countable domain
 - *Weather* might be (sunny, rainy, cloudy, snow)
- Continuous random variables – finite set real numbers with equal intervals e.g. interval(0.1)

Atomic events

- The concept of an **atomic event** is useful in understanding the foundations of probability theory.
- It is a **complete specification** of the state of the world about which the agent is uncertain.
- It can be an **assignment of particular values**, to all the variables of which the world is composed

Prior Probability

- The **unconditional** or **prior probability** associated with a proposition a , is the **degree of belief** according to the absence of any other information;
- it is written as $P(a)$.
- For example, if the prior probability that one have a cavity is 0.1, then we would write
 - $P(\text{Cavity} = \text{true}) = 0.1$ or $P(\text{cavity}) = 0.1$.
- It is important to remember that $P(a)$ can be used only when **there is no other information**.

Prior Probability...

- we will use an expression $P(\textit{Weather})$, which denotes a **vector** of values, for the probabilities of each individual state of the weather.
 - $P(\textit{Weather} = \textit{sunny}) = 0.7$
 - $P(\textit{Weather} = \textit{rain}) = 0.2$
 - $P(\textit{Weather} = \textit{cloudy}) = 0.08$
 - $P(\textit{Weather} = \textit{snow}) = 0.02$.
- we may simply write
- $P(\textit{Weather}) = (0.7, 0.2, 0.08, 0.02)$.
- This statement defines a **prior probability distribution** for the random variable *Weather*.

Conditional Probability

- The **conditional** or **posterior** probabilities notation is $P(a|b)$,
- where a and b are any proposition.
- This is read as "the probability of a , given that *all* we know is b ."
- For example, $P(\text{cavity} | \text{toothache}) = 0.8$
- if a patient is observed to have a toothache and no other information is yet available, then the probability of the patient's having a cavity will be 0.8.

Conditional probabilities...

- Conditional probabilities can be defined in terms of unconditional probabilities.
- The equation is

$$P(a|b) = \frac{P(a \wedge b)}{P(b)}$$

- whenever $P(b) > 0$.
- This equation can also be written as
- $P(a \wedge b) = P(a/b) P(b)$ which is called the **product rule**.

Basic Axioms of Probability

- All probabilities are between 0 and 1. For any proposition a ,

$$0 \leq P(a) \leq 1$$

- Necessarily true (i.e., valid) propositions have probability 1, and necessarily false (i.e., unsatisfiable) propositions have probability 0.

$$P(\text{true}) = 1 \qquad P(\text{false}) = 0 .$$

- The probability of a disjunction is given by

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b) .$$

Start with the joint distribution:

	<i>toothache</i>		$\neg$ <i>toothache</i>	
	<i>catch</i>	$\neg$ <i>catch</i>	<i>catch</i>	$\neg$ <i>catch</i>
<i>cavity</i>	.108	.012	.072	.008
$\neg$ <i>cavity</i>	.016	.064	.144	.576

For any proposition ϕ , sum the atomic events where it is true:

$$P(\phi) = \sum_{\omega: \omega \models \phi} P(\omega)$$


$$\begin{aligned} P(\text{toothache}) &= .108 + .012 + .016 + .064 \\ &= .20 \text{ or } 20\% \end{aligned}$$

Joint probability distribution

- The **full joint probability distribution** specifies the probability of values to random variables.
- It is usually **too large** to create or use in its explicit form.
- Joint probability distribution of two variables **X** and **Y** are

Joint probabilities	X	X'
Y	0.20	0.12
Y'	0.65	0.03

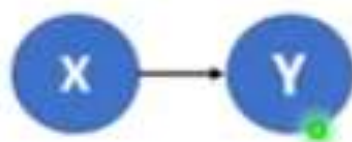
- Joint probability distribution for **n** variables require **2^n** entries with all possible combination.

Drawbacks of joint Probability Distribution

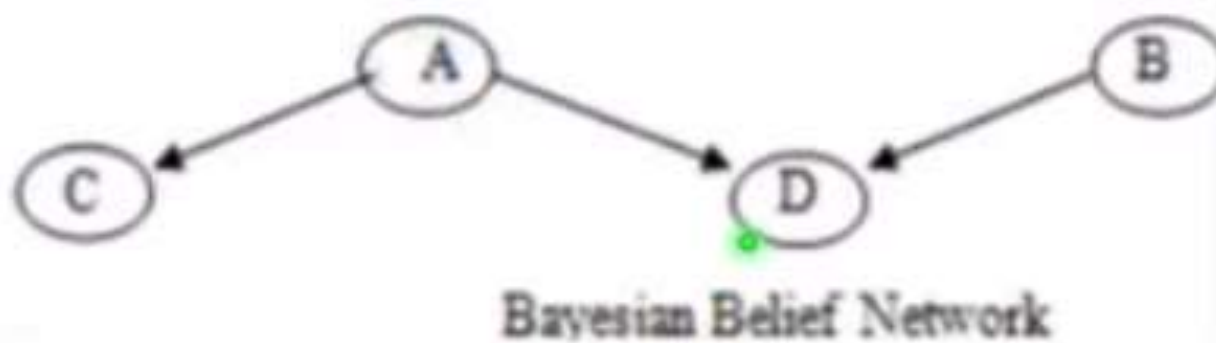
- Large number of variables and grows rapidly
- Time and space complexity are huge
- Statistical estimation with probability is difficult
- Human tends signal out few propositions
- The alternative to this is Bayesian Networks.

Bayesian Networks

- Bayesian Network is to represent the dependencies among variables and to give a brief specification of any full joint probability distribution.
- A Bayesian network is a directed graph in which each nodes are variables and edges are relations.
- The full specification is as follows:
 - 1. A set of random variables makes up the nodes of the network. Variables may be discrete or continuous.
 - 2. A set of directed links or arrows connects pairs of nodes. If there is an arrow from node X to node Y , X is said to be a parent of Y .
 - 3. Each node X , has a conditional probability distribution $P(X, (Parents(X,)))$ that quantifies the effect of the parents on the node. (X is parent of Y)
 - 4. The graph has no directed cycles (and hence is a directed, acyclic graph, or DAG).

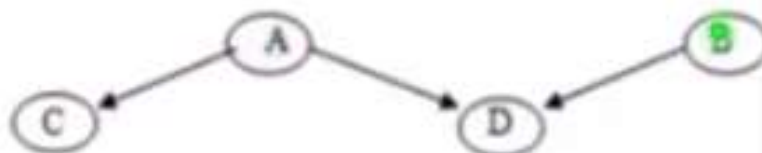


Example



- A & B are unconditional, independent, evidence and parent nodes
- C & D are conditional, dependent, hypothesis and child nodes.

Conditional Probability Table



$P(A)$	=	0.3
$P(B)$	=	0.6
$P(C A)$	=	0.4
$P(C \sim A)$	=	0.2
$P(D A, B)$	=	0.7
$P(D A, \sim B)$	=	0.4
$P(D \sim A, B)$	=	0.2
$P(D \sim A, \sim B)$	=	0.01

Conditional Probability Tables						
P(A)	P(B)	A	P(C)	A	B	P(D)
0.3	0.6	T	0.4	T	T	0.7
		F	0.2	T	F	0.4
				F	T	0.2
				F	F	0.01

$$P(A,B,C,D) = P(D|A,B) \cdot P(C|A) \cdot P(B) \cdot P(A)$$



Example 2

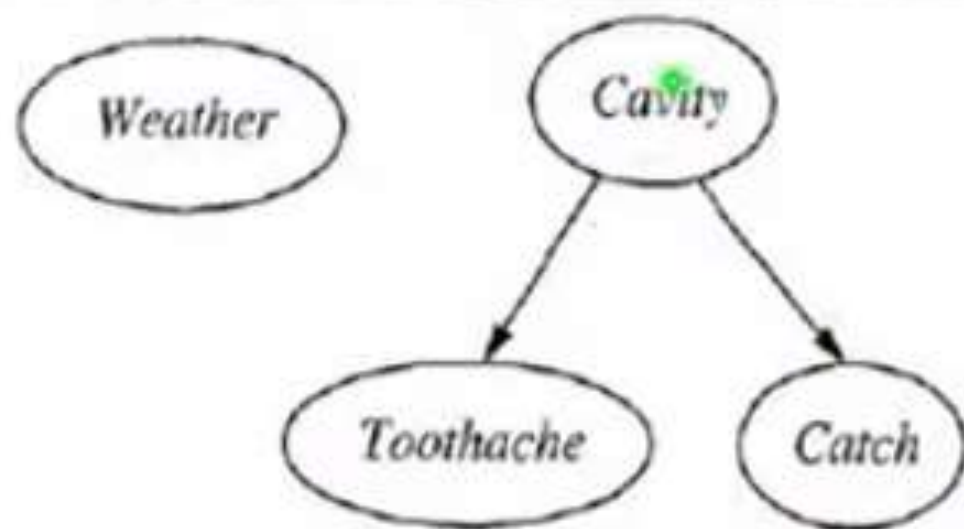
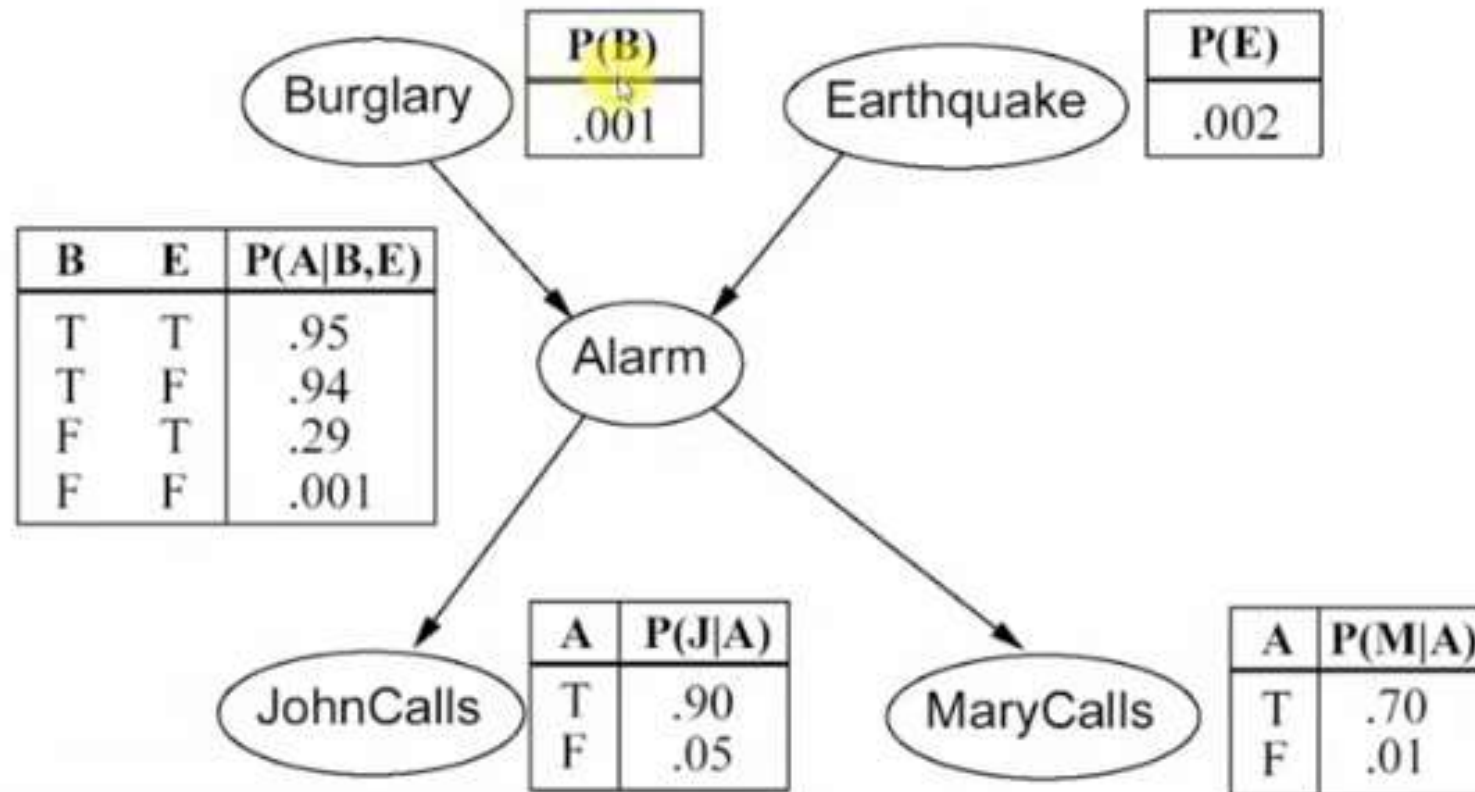


Figure 14.1 A simple Bayesian network in which *Weather* is independent of the other three variables and *Toothache* and *Catch* are conditionally independent, given *Cavity*.

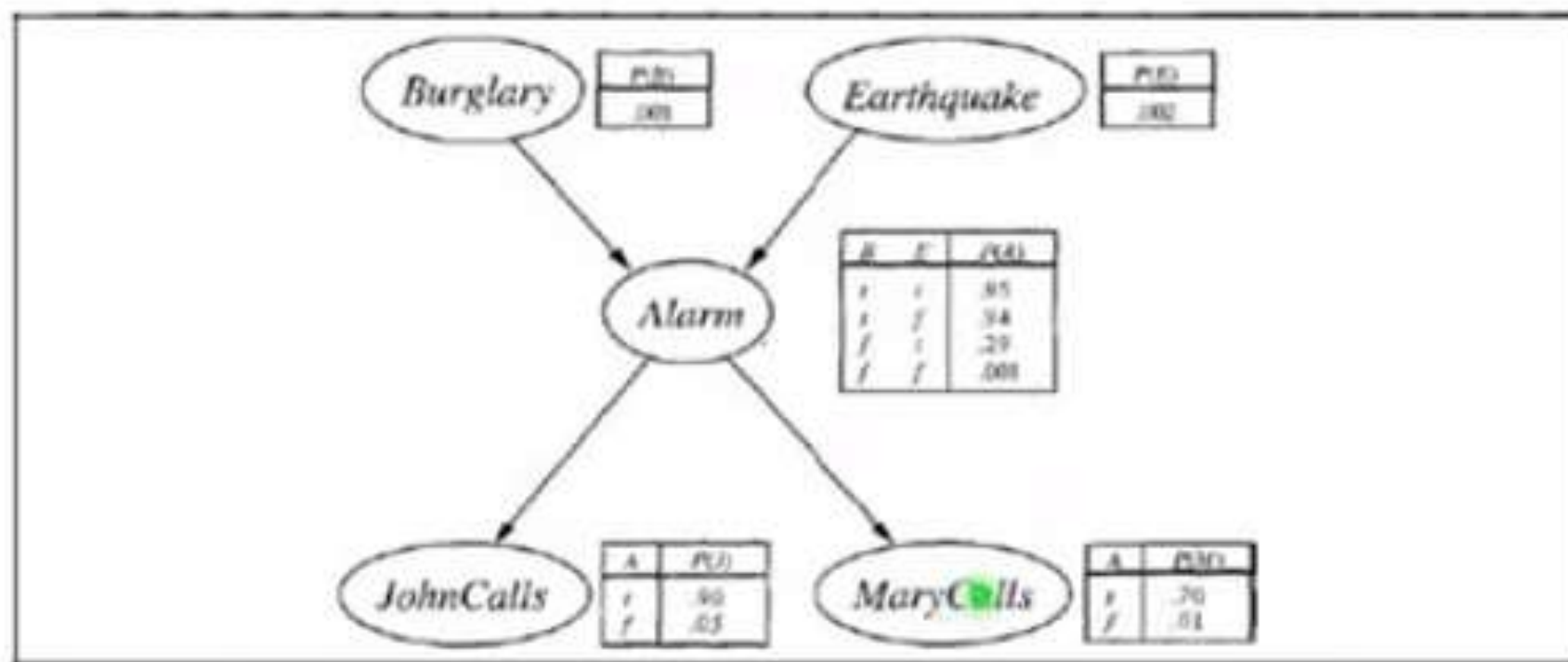
BAYESIAN BELIEF NETWORKS – EXAMPLE – 1



Example - Burglar alarm

- You have a new burglar alarm installed at home.
- It also responds on occasion to minor earthquakes.
- You also have two neighbors, John and Mary, they promised to call you when they hear the alarm.
- John always calls when he hears the alarm, but sometimes confuses the telephone ringing with the alarm, and calls then, too.
- Mary likes rather loud music, and sometimes, she misses the alarm altogether.
- Given the evidence of who has or has not called, we would like to estimate the probability of a burglary.

- The burglary and earthquakes directly affect the probability of the alarm's going off,
- But, John and Mary call depends only on the alarm.
- The network does not have nodes for Mary's currently listening to loud music or the telephone ringing and confusing John.



Example

- We can calculate the probability that the alarm has sounded, but neither a burglary nor an earthquake has occurred, and both John and Mary call.

$$\begin{aligned}P(j \wedge m \wedge a \wedge \neg b \wedge \neg e) \\&= P(j|a)P(m|a)P(a|\neg b \wedge \neg e)P(\neg b)P(\neg e) \\&= 0.90 \times 0.70 \times 0.001 \times 0.999 \times 0.998 = 0.0006'2.\end{aligned}$$

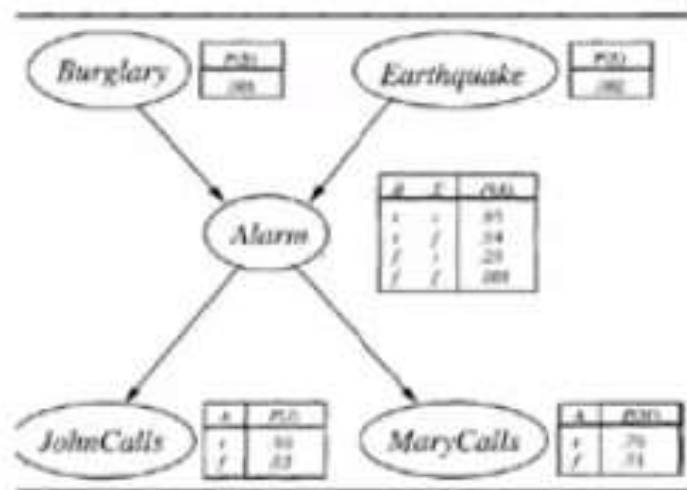
Inferences in Bayesian Network - Purpose

- Probabilistic Inference System is to compute Posterior Probability Distribution for a set of query variables, given some observed events.
- That is, some assignment of values to a set of evidence variables.

Inference in Bayesian Networks - Notations

- X – denotes the query variable
- E - set of evidence variables $\{E_1, \dots, E_m\}$
- e – particular observed event
- Y – non-evidence, non-query variables, $Y_1, \dots, Y_n$. (Called the hidden variables)
- The complete set of variables - $X = \{X\} \cup E \cup Y$
- A typical query asks for the Posterior Probability Distribution $P(X|e)$

- In the burglary network, we might observe the event in which
 - *JohnCalls = true* and *MaryCalls = true*.
- We could then ask for, say, the probability that a burglary has occurred:
 - $P(\text{Burglary} \mid \text{JohnCalls} = \text{true}, \text{MaryCalls} = \text{true}) = (0.284, 0.716)$



Types of Inferences

- Inference by Enumeration.
 - (inference by listing or recording all variables)
- Inference by Variable Elimination.
 - (inference by variable removal)

Inference by Enumeration

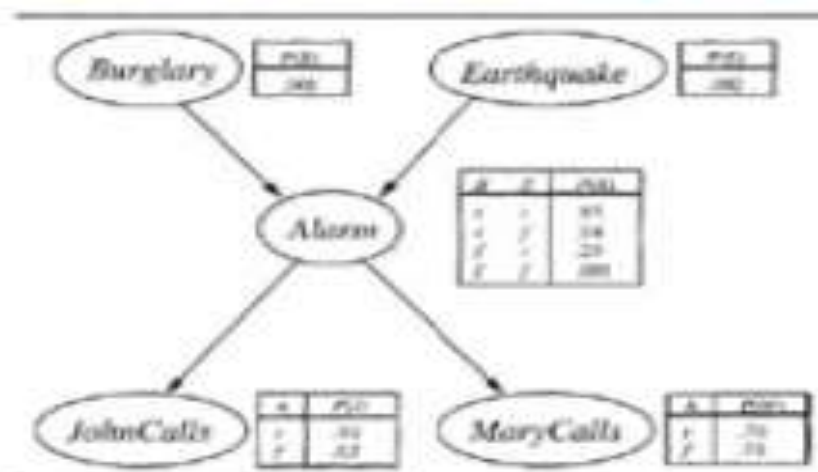
- Any conditional probability can be computed by summing terms from the full joint distribution.
- More specifically, a query $P(X|e)$ can be answered using equation:

$$P(X|e) = \alpha P(X, e) = \alpha \sum_y P(X, e, y)$$

- Where α is normalized constant
- X – Query Variable
- e – event
- y – number of terms

Inference by Enumeration...

- Consider $P(\text{Burglary} \mid \text{JohnCalls} = \text{true}, \text{MaryCalls} = \text{true})$
- Burglary – query variable (X)
- JohnCalls – Evidence variable 1 (E1)
- MaryCalls – Evidence Variable 2 (E2)
- The hidden variables of this query are earthquake and alarm



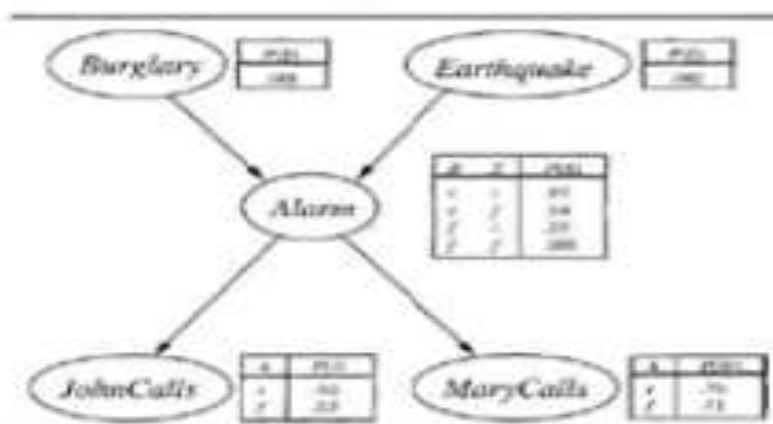
Inference by Enumeration...

- Using initial letter for the variables to shorten the expression we have

$$\mathbf{P}(B|j, m) = \alpha \mathbf{P}(B, j, m) = \alpha \sum_e \sum_a \mathbf{P}(B, e, a, j, m)$$

- The semantic of Bayesian network give us an expression, in terms of CPT entries, for simplicity we do this just for **Burglary = true**

$$P(b|j, m) = \alpha \sum_e \sum_a P(b)P(e)P(a|b, e)P(j|a)P(m|a)$$



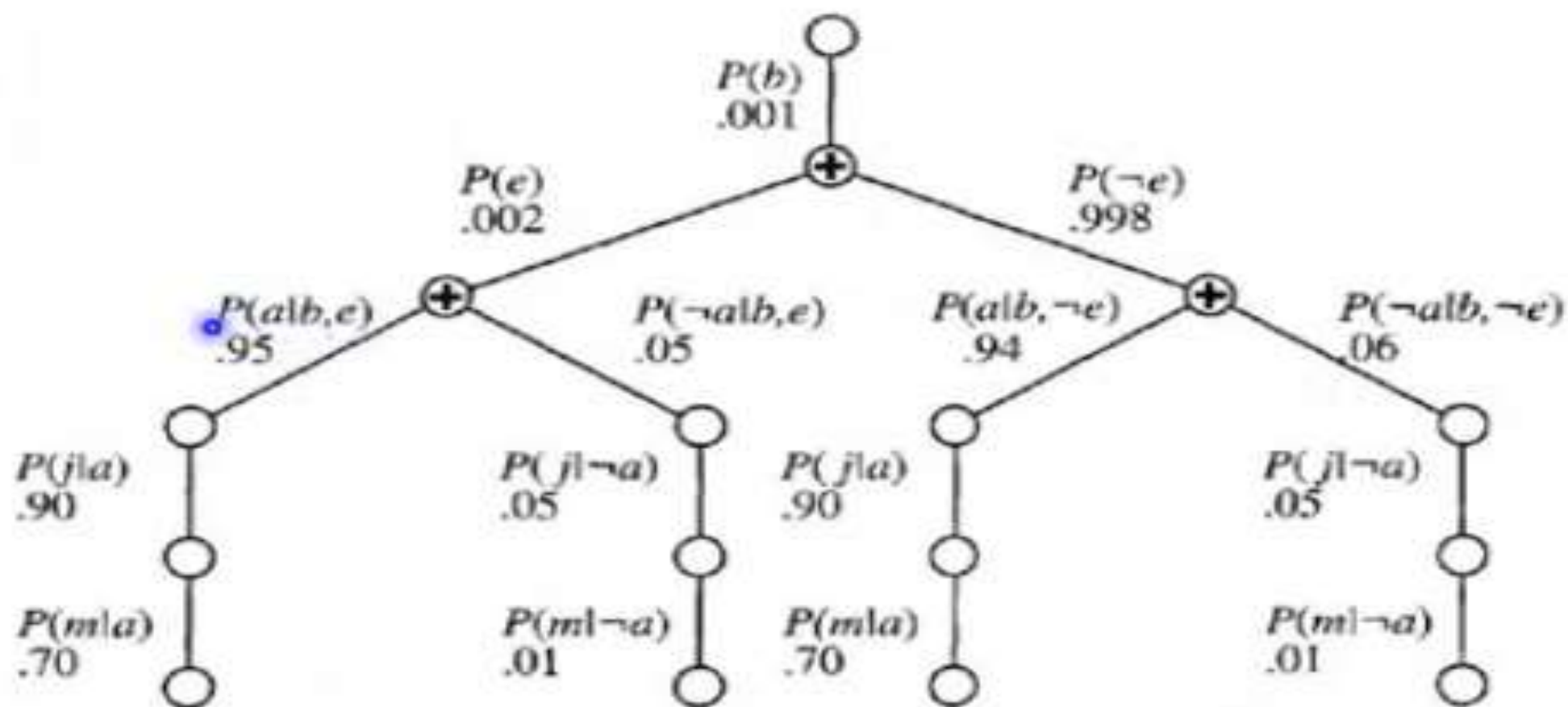
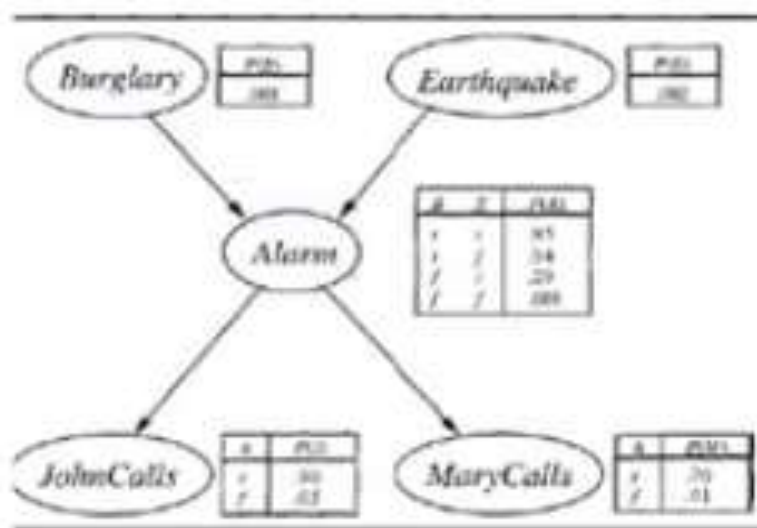


Figure 14.8 The structure of the expression shown in Equation (14.3). The evaluation proceeds top-down, multiplying values along each path and summing at the "+" nodes. Notice the repetition of the paths for j and m .

- In the burglary network, we might observe the event in which
 - *JohnCalls = true and MaryCalls = true.*
- We could then ask for, say, the probability that a burglary has occurred:
 - *$P(\text{Burglary} \mid \text{JohnCalls} = \text{true}, \text{MaryCalls} = \text{true}) = (0.284, 0.716)$*



Inference by Variable Elimination

- The enumeration algorithm can be improved substantially by elimination repeated calculations.
- The idea is simple: do the calculation once and solve the result for later use. This is a form of dynamic programming.

Inference by Variable Elimination ...

- Variable elimination works by evaluating expressions,
- previous equation (derived in inference by enumeration)

$$P(b|j, m) = \alpha \sum_e \sum_a P(b)P(e)P(a|b, e)P(j|a)P(m|a)$$

- From this the repeated variables are separated

$$P(b|j, m) = \alpha P(b) \sum_e P(e) \sum_a P(a|b, e)P(j|a)P(m|a)$$

Inference by Variable Elimination ...

- Intermediate results are stored, and summations of each variable are done, for only those portion of the expression, that depends on the variable.
- Let us illustrate this process for the burglary network.
- We evaluate the expression

$$\mathbf{P}(B|j, m) = \alpha \underbrace{\mathbf{P}(B)}_B \sum_e \underbrace{P(e)}_E \sum_a \underbrace{\mathbf{P}(a|B, e)}_A \underbrace{P(j|a)}_J \underbrace{P(m|a)}_M$$

- We have annotated each part of the expression with the name of the associated variable; these parts are called factors

Dempster-Shafer Theory

- **Dempster-Shafer Theory** was given by Arthur P. Dempster in 1967 and was later developed by his student Glenn Shafer in 1976.
- This theory was released because of the following reason:-
- Bayesian theory is only concerned about single evidence.
- Bayesian probability cannot describe ignorance.

- Dempster Shafer Theory(DST) is an evidence theory, it combines all possible outcomes of the problem.
- Hence it is used to solve problems where there may be a chance that a piece of different evidence will lead to some different result.
- The **uncertainty in this model** is given by:-
- Consider all possible outcomes.
- **Belief will lead to belief in some possibility by bringing out some evidence.**
- Plausibility will make evidence compatible with possible outcomes.

- **Example:**
- Let us consider a room where four people are present, A, B, C, and D. Suddenly the lights go out and when the lights come back, B has been stabbed in the back by a knife, leading to his death. No one came into the room and no one left the room. We know that B has not committed suicide. Now we have to find out who the murderer is.
- To solve these there are the **following possibilities:**
- Either {A} or {C} or {D} has killed him.
- Either {A, C} or {C, D} or {A, D} have killed him.
- Or the three of them have killed him i.e.; {A, C, D}
- None of them have killed him {o} (let's say).

- There will be possible evidence by which we can find the murderer by the measure of plausibility.
- Using the above example we can say:
- Set of possible conclusion (P): $\{p_1, p_2, \dots, p_n\}$
- where P is a set of possible conclusions and cannot be exhaustive, i.e. at least one (p) must be true.
- (p) must be mutually exclusive.
- Power Set will contain 2^n elements where n is the number of elements in the possible set.
- For e.g.:-
If $P = \{a, b, c\}$, then Power set is given as
 $\{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\} = 2^3$ elements

- Mass Function: The mass function, denoted as $m(K)$, quantifies the belief assigned to a set of hypotheses, denoted as K . It provides a measure of uncertainty by allocating probabilities to various hypotheses, reflecting the degree of support each hypothesis has from the available evidence.

- **Mass function $m(K)$:** It is an interpretation of $m(\{K \text{ or } B\})$ i.e; it means there is evidence for $\{K \text{ or } B\}$ which cannot be divided among more specific beliefs for K and B .
- **Belief in K :** The belief in element K of Power Set is the sum of masses of the element which are subsets of K . This can be explained through an example
 Lets say $K = \{a, d, c\}$

$$\text{Bel}(K) = m(a) + m(d) + m(c) + m(a, d) + m(a, c) + m(d, c) + m(a, d, c)$$
- **Plausibility in K :** It is the sum of masses of the set that intersects with K .
 i.e.;
$$\text{Pl}(K) = m(a) + m(d) + m(c) + m(a, d) + m(d, c) + m(a, c) + m(a, d, c)$$

Characteristics of Dempster Shafer Theory:

- **Uncertainty Representation** : The DST is designed to handle situations where there is uncertainty of information and it provides a way to represent and reason incomplete evidence.
- **Conflict of Evidence** : The DST allows for the combination of multiple sources of evidence. It provides a rule, Dempster's rule of combination, to combine belief functions from different sources.
- **Decision-Making Ability** : By deriving measures such as belief, probability and plausibility from the combined belief function it helps in decision making.

Advantages of Dempster Shafer Theory:

- As we add more information, the uncertainty interval reduces.
- DST has a much lower level of ignorance.
- Diagnose hierarchies can be represented using this.
- Person dealing with such problems is free to think about evidence.

Disadvantages of Dempster Shafer Theory:

- In this, computation effort is high, as we have to deal with 2^n sets

Artificial Intelligence

UNIT -4

- The intelligence of humans is achieved—not by purely reflex mechanisms but by processes of **reasoning** that operate on internal **representations** of knowledge.
- In AI, this approach to intelligence is embodied in **knowledge-based agents**.

KNOWLEDGE-BASED AGENTS

- The central component of a knowledge-based agent is its **knowledge base**, or KB. A knowledge base is a set of **sentences**. (Here “sentence” is used as a technical term. It is related but not identical to the sentences of English and other natural languages.)
- Each sentence is expressed in a language called a **knowledge representation language** and represents some assertion about the world.
- a sentence with the name **axiom**, when the sentence is taken as given without being derived from other sentences.

- There must be a way to add new sentences to the knowledge base and a way to query what is known.
- The standard names for these operations are TELL and ASK, respectively.
- Both operations may involve **inference**—that is, deriving new sentences from old.
- Inference must obey the requirement that when one ASKs a question of the knowledge base, the answer should follow from what has been told (or TELLED) to the knowledge base previously.

function KB-AGENT(*percept*) **returns** an *action*

persistent: *KB*, a knowledge base

t, a counter, initially 0, indicating time

TELL(*KB*, MAKE-PERCEPT-SENTENCE(*percept*, *t*))

action $\leftarrow$ ASK(*KB*, MAKE-ACTION-QUERY(*t*))

TELL(*KB*, MAKE-ACTION-SENTENCE(*action*, *t*))

t $\leftarrow$ *t* + 1

return *action*

Figure 7.1 A generic knowledge-based agent. Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

- Figure 7.1 shows the outline of a knowledge-based agent program. Like all our agents,
- it takes a percept as input and returns an action. The agent maintains a knowledge base, KB, which may initially contain some **background knowledge**.

- Each time the agent program is called, it does three things. First, it TELLS the knowledge base what it perceives.
- Second, it ASKS the knowledge base what action it should perform.
- In the process of answering this query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences, and so on.
- Third, the agent program TELLS the knowledge base which action was chosen, and the agent executes the action.

- A knowledge-based agent can be built simply by TELLing it what it needs to know.
- Starting with an empty knowledge base, the agent designer can TELL sentences one by one until the agent knows how to operate in its environment.
- This is called the **declarative** approach to system building.

- In contrast, the **procedural** approach encodes desired behaviors directly as program code.
- a successful agent often combines both declarative and procedural elements in its design, and that declarative knowledge can often be compiled into more efficient procedural code.
- We can also provide a knowledge-based agent with mechanisms that allow it to learn for itself. These mechanisms create general knowledge about the environment from a series of percepts. A learning agent can be fully autonomous.

THE WUMPUS WORLD

- The **wumpus world** is a cave consisting of rooms connected by passageways.
- Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room.
- The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in).
- The only mitigating feature of this bleak environment is the possibility of finding a heap of gold.

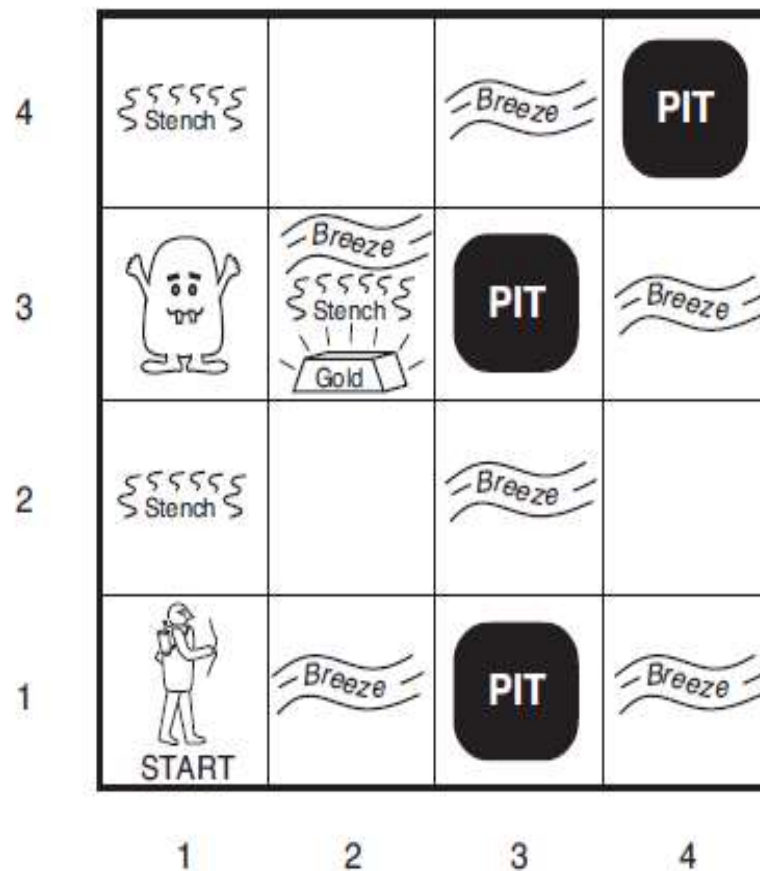


Figure 7.2 A typical wumpus world. The agent is in the bottom left corner, facing right.

- A sample wumpus world is shown in Figure 7.2.
- The precise definition of the task environment is given by the PEAS description:

- **Performance measure:**
- +1000 for climbing out of the cave with the gold,
- −1000 for falling into a pit or being eaten by the wumpus,
- −1 for each action taken and
- −10 for using up the arrow.
- The game ends either when the agent dies or when the agent climbs out of the cave.

- **Environment:**
- A 4×4 grid of rooms.
- The agent always starts in the square labeled [1,1], facing to the right.
- The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square.
- In addition, each square other than the start can be a pit, with probability 0.2.

- **Actuators:**

- The agent can move *Forward*, *TurnLeft* by 90°, or *TurnRight* by 90°.
- The agent dies a miserable death if it enters a square containing a pit or a live wumpus. (It is safe, although smelly, to enter a square with a dead wumpus.)
- If an agent tries to move forward and bumps into a wall, then the agent does not move.
- The action *Grab* can be used to pick up the gold if it is in the same square as the agent.
- The action *Shoot* can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits (and hence kills) the wumpus or hits a wall. The agent has only one arrow, so only the first *Shoot* action has any effect.
- Finally, the action *Climb* can be used to climb out of the cave, but only from square [1,1].

- **Sensors:** The agent has five sensors, each of which gives a single bit of information:
 - In the square containing the wumpus and in the directly (not diagonally) adjacent squares, the agent will perceive a *Stench*.
 - In the squares directly adjacent to a pit, the agent will perceive a *Breeze*.
 - In the square where the gold is, the agent will perceive a *Glitter*.
 - When an agent walks into a wall, it will perceive a *Bump*.
 - When the wumpus is killed, it emits a woeful *Scream* that can be perceived anywhere in the cave.

- The percepts will be given to the agent program in the form of a list of five symbols;
- for example, if there is a stench and a breeze, but no glitter, bump, or scream, the agent program will get [Stench, Breeze, None, None, None].

Wumpus World...

- Characterization of Wumpus World Environment
- Observable
 - partial, only local perception
- Deterministic
 - Yes, outcomes are specified
- Episodic
 - No, sequential at the level of actions
- Static
 - Yes, Wumpus and pits do not move
- Discrete
 - Yes
- Single Agent
 - Yes

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2	3,2	4,2
OK			
1,1	2,1	3,1	4,1
A			
OK	OK		

A = Agent
B = Breeze
G = Glitter, Gold
OK = Safe square
P = Pit
S = Stench
V = Visited
W = Wumpus

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2	3,2	4,2
OK	P?		
1,1	2,1	3,1	4,1
V	A	P?	
OK	B		
	OK		

(a)

(b)

Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept *[None, None, None, None, None]*. (b) After one move, with percept *[None, Breeze, None, None, None]*.

1,4	2,4	3,4	4,4
1,3 W!	2,3	3,3	4,3
1,2 A S OK	2,2 OK	3,2	4,2
1,1 V OK	2,1 B V OK	3,1 P!	4,1

A = Agent
 B = Breeze
 G = Glitter, Gold
 OK = Safe square
 P = Pit
 S = Stench
 V = Visited
 W = Wumpus

1,4	2,4 P?	3,4	4,4
1,3 W!	2,3 A S G B	3,3 P?	4,3
1,2 S V OK	2,2 V OK	3,2	4,2
1,1 V OK	2,1 B V OK	3,1 P!	4,1

(a)

(b)

Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept [*Stench, None, None, None, None*]. (b) After the fifth move, with percept [*Stench, Breeze, Glitter, None, None*].

Wumpus World...

OK			
OK A	OK		

- Variables used...
- A – Agent
- B – Breeze
- G – Glitter, Gold
- OK – Safe Square
- P – Pit
- S – Stench
- V – Visited Square
- W – Wumpus

Exploring the Wumpus World

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2 OK	2,2	3,2	4,2
1,1 A OK	2,1 OK	3,1	4,1

A = Agent
B = Breeze
G = Glitter, Gold
OK = Safe square
P = Pit
S = Stench
V = Visited
W = Wumpus

Initial situation:

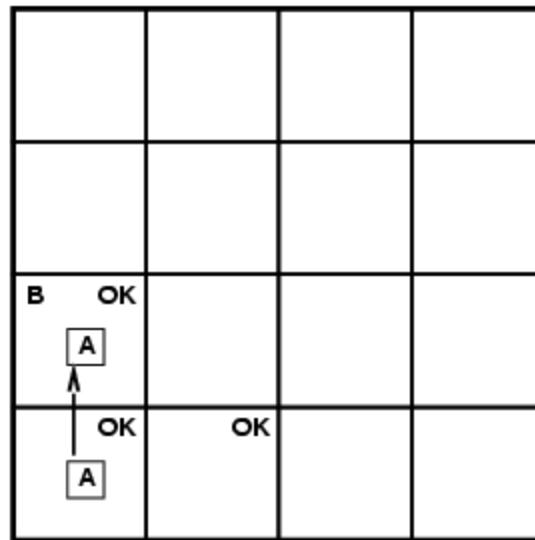
**Agent in 1,1 and percept is
[None, None, None, None,
None]**

**From this the agent can
infer the neighboring
squares are safe (otherwise
there would be a breeze or a
stench)**

Exploring a wumpus world

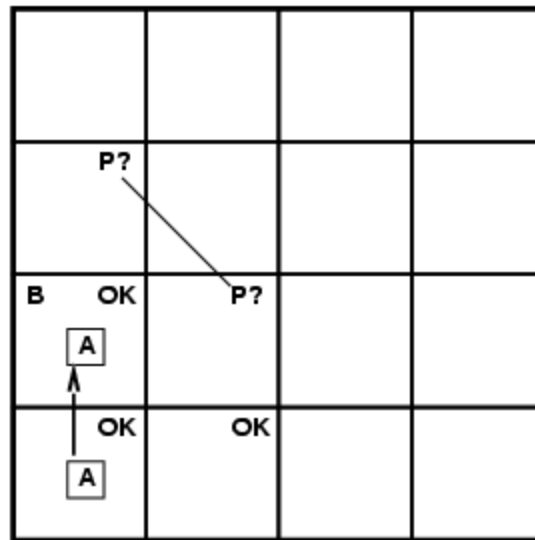
OK			
OK A	OK		

Exploring a Wumpus world

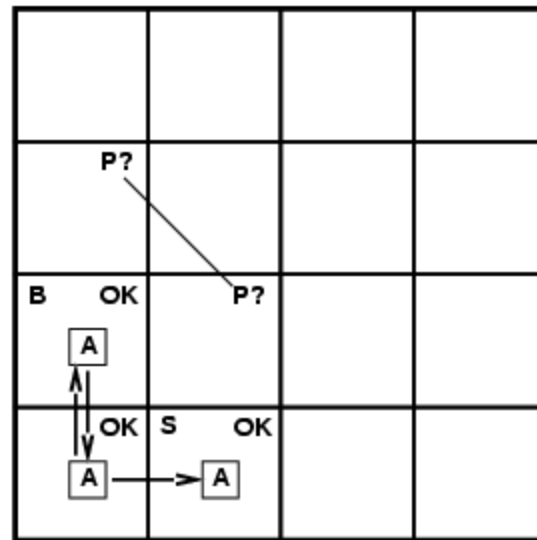


After one move, the percept
[None, Breeze, None, None, None].

Exploring a wumpus world

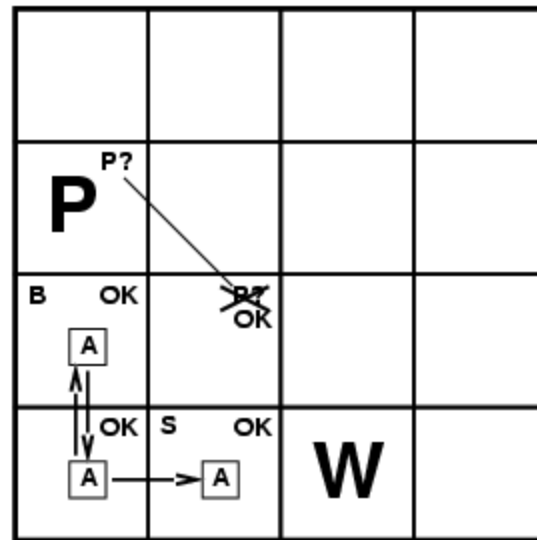


Exploring a wumpus world

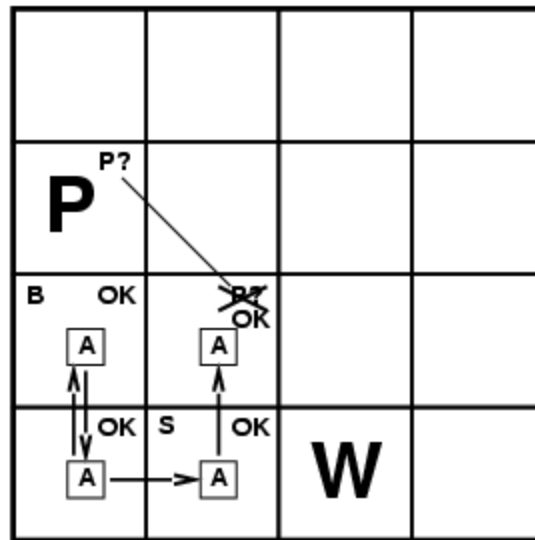


After the third move,
The percept [Stench, None, None, None, None].

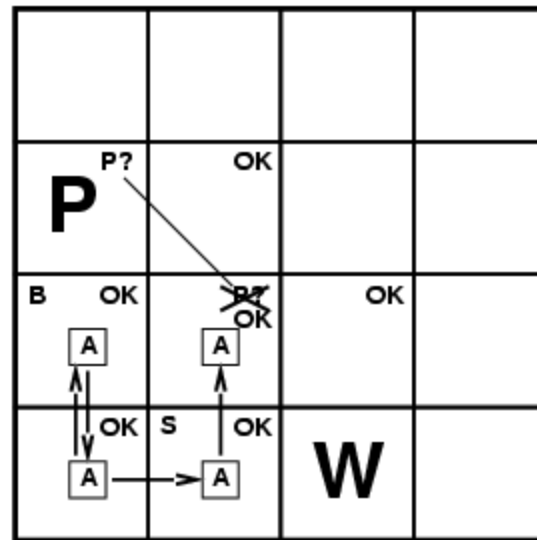
Exploring a wumpus world



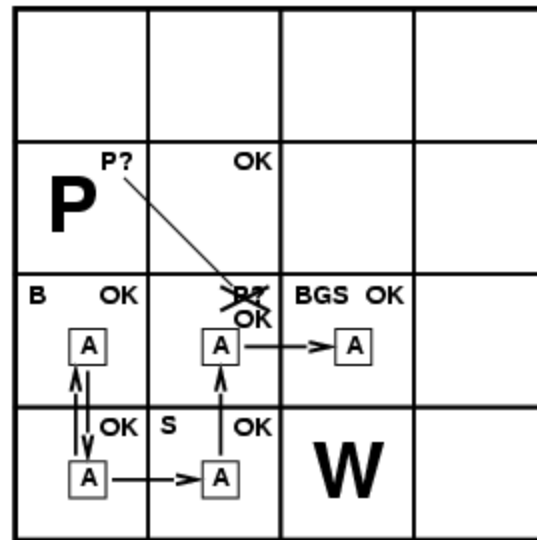
Exploring a wumpus world



Exploring a wumpus world

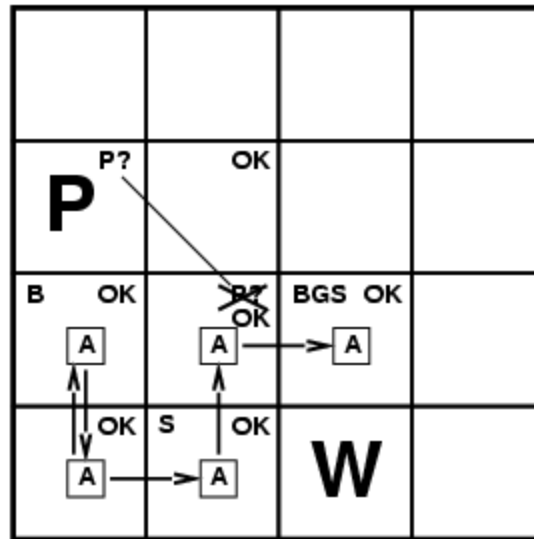


Exploring a wumpus world



Now the percept is [Stench, Breeze, Glitter, None, None].

Exploring a wumpus world

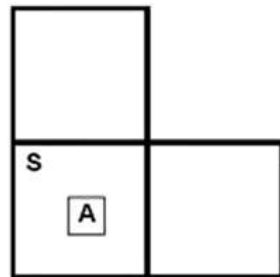
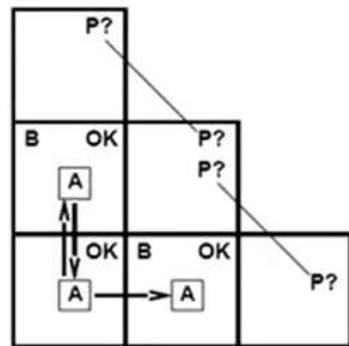


In each case where the agent draws a conclusion from the available Information, that conclusion is guaranteed to be correct if the available Information is correct...

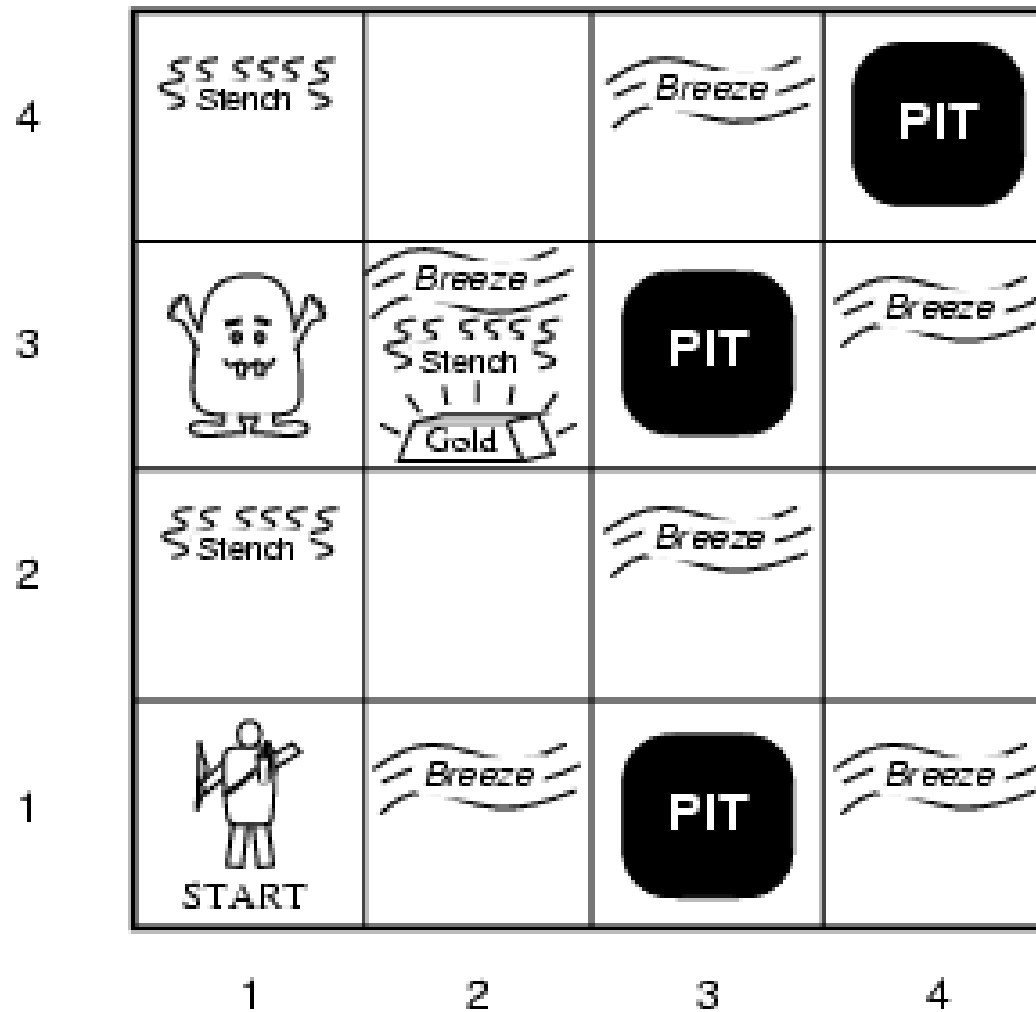
This is a fundamental property of logical reasoning

Other Sticky Situations

- Breeze in (1,2) and (2,1)
 - No safe actions
- Smell in (1,1)
 - Cannot move



try it



Logic

- ✓ In AI, Logic is used for knowledge representation.
- ✓ Logic is a notation or method used to represent the knowledge.
- ✓ Logic defines the **syntax** and **semantics** of representation language.
- ✓ **Syntax:**
 - ✓ Syntax defines the sentences in the language.
 - ✓ We know that The central component of a knowledge-based agent is its **knowledge base**, or KB. knowledge base is a collection of sentences. **These sentences are expressed according to the syntax of the representation language.**
 - ✓ The syntax specifies that all sentences are well formed.
 - ✓ Example: In arithmetic
 - ✓ $x + y = 4$ is a well formed sentence
 - ✓ $x \ 4 \ y \ + \ =$ is not a sentence.

✓ **Semantics:**

- A logic must also define the **semantics** or meaning of sentences.
- The semantics defines the **truth** of each sentence with respect to each **possible world**.
- For example, the semantics for arithmetic specifies that the sentence “ $x + y = 4$ ” is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1.
- In standard logics, every sentence must be either true or false in each possible world—there is no “in between.”

✓ **Model:**

- ✓ We use **model** in place of Possible world
- ✓ Possible worlds are real environments that the agent may or may not be in.
- ✓ Models are mathematical abstractions, each of which simply fixes the truth or false of every relevant sentence.
- ✓ Possible world as, for example, having x men and y women sitting at a table playing carroms, and the sentence $x + y = 4$ is true when there are four people in total.
- ✓ the possible models are just all possible assignments of real numbers to the variables x and y .
- ✓ If a sentence α is true in model m , we say that m satisfies α or sometimes m is a model of α .
- ✓ We use the notation **$M(\alpha)$** to refer the set of all models of α .

✓ Entailment:

- ✓ Entailment means a notion of truth in logical reasoning
- ✓ This involves the relation of logical entailment between sentences—the idea that a sentence follows logically from another sentence.
- ✓ In mathematical notation, entailment is represented as

$$\alpha \models \beta$$

- ✓ The formal definition of entailment is this:

$$\alpha \models \beta$$

if and only if, in every model in which α is true, β is also true.

- ✓ Using the notation just introduced, we can write

- ✓ $\alpha \models \beta$ if and only if $M(\alpha) \subseteq M(\beta)$.

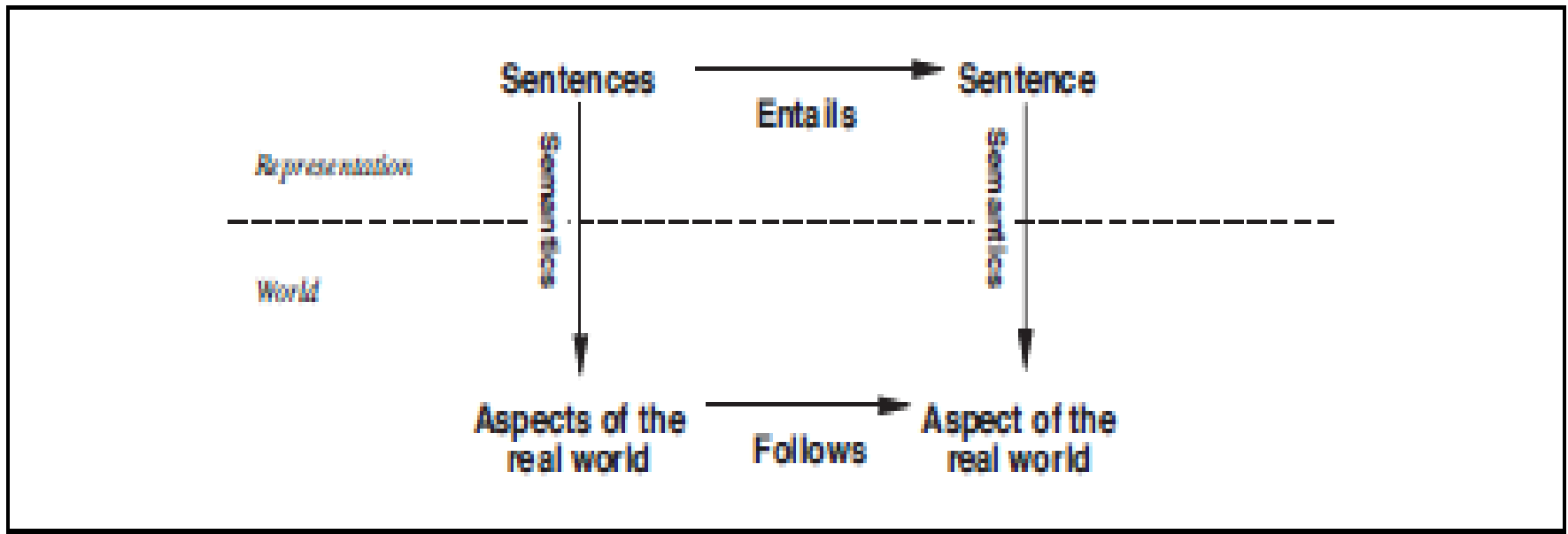
- ✓ Ex: the sentence $x = 0$ entails the sentence $xy = 0$

Logical inference

- The process of deriving new conclusions from the available information or from known facts or from KB
- entailment can be applied to carry out logical inference
- **model checking** is an example of inference algorithm
- model checking specify all possible models and check whether α is true
- if an inference algorithm i can derive α from KB we write

$$KB \vdash_i \alpha$$

- which is pronounced as α is derived from KB by i or i derives α from KB



- ✓ Properties of inference algorithms:
 - ✓ Soundness
 - ✓ an inference algorithm that derives only entailed sentence are sound
 - ✓ Completeness
 - ✓ An inference algorithm is complete if it derives any sentence that is entailed
 - ✓ Grounding
 - ✓ the connection between logical reasoning processes and the real environment in which the agent exists.

Propositional Logic

- ✓ Proposition means a statement that affirms or denies something and is either true or false.
- ✓ Propositional logic is a simple statement logic. It is also called boolean logic.
- ✓ Propositional logic is used to decide the truth value of a given sentence.
- ✓ **Syntax**
 - ✓ The syntax of propositional logic defines the allowable sentences
 - ✓ There are two types of sentences
 - Atomic sentence
 - Complex sentence
- ✓ **Atomic Sentence**
 - ✓ Atomic sentences consist of single propositional symbol. Each symbol stands for a proposition that can be true or false.
 - ✓ We use the symbols that start with an uppercase letter and may contain other letters or subscripts.
 - ✓ For example: P, Q, R, $W_{1,3}$ and North.
 - ✓ We may use $w_{1,3}$ to stand for the proposition that wumpus in [1,3].
 - ✓ There are two proposition symbols with fixed meanings:
 - True** is the always true proposition and **False** is the always-false proposition.

Propositional Logic

Complex Sentence

- ✓ Complex sentences are constructed from simpler sentences, using parentheses and logical connectives.
- ✓ There are five connectives in common use:
 1. $\neg$ (not) : Represents the negation of a given sentence.
 - ✓ Example
 - ✓ $W_{1,3}$: There is a wumpus in square [1,3]
 - ✓ $\neg W_{1,3}$: There is no wumpus in square [1,3]
 2. $\wedge$ (and) or conjunction :
 - ✓ A sentence whose main connective is $\wedge$, such as $W_{1,3} \wedge P_{3,1}$, is a conjunction. Its parts are the conjuncts.
 3. $\vee$ (or) disjunction :
 - ✓ A sentence using $\vee$, such as $W_{1,3} \vee P_{3,1}$, is a disjunction, Its parts are the disjuncts
 4. $\Rightarrow$ (implies) or implication : Implication is represented $P \Rightarrow Q$
 - ✓ Its left side part is called **premise or antecedent** and its right side part is called **conclusion or consequent**.
 - ✓ Implications are also called as **rules** or **if-then** statements

Propositional Logic

5. $\Leftrightarrow$ (if and only if) or biconditional :

- ✓ It is represented as $P \Leftrightarrow Q$
 - ✓ The sentence $W_{1,3} \Leftrightarrow \neg W_{2,2}$ is a biconditional
- ✓ A formal grammar of sentences in propositional logic is shown below:

```
Sentence   → AtomicSentence | ComplexSentence
AtomicSentence → True | False | P | Q | R | ...
ComplexSentence → ( Sentence ) | [ Sentence ]
                  |  $\neg$  Sentence |
                  | Sentence  $\wedge$  Sentence
                  | Sentence  $\vee$  Sentence
                  | Sentence  $\Rightarrow$  Sentence
                  | Sentence  $\Leftrightarrow$  Sentence
```

OPERATOR PRECEDENCE : $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$

A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

Propositional Logic

Semantics:

- ✓ The semantics defines the rules for determining the truth of a sentence with respect to a particular model.
- ✓ In propositional logic, a model simply **assigns** the truth value **true** or **false** for every proposition symbol.

For example:

- ✓ if the sentences in the knowledge base make use of the proposition symbols $P_{1,2}$, $P_{2,2}$, and $P_{3,1}$, then one possible model is
$$\mathbf{m_1 = \{P_{1,2} = false, P_{2,2} = false, P_{3,1} = true\}}$$
- ✓ The semantics for propositional logic must specify how to compute the truth value of any sentence, given a model.
- ✓ For atomic sentences:
 - ✓ **True** is true in every model and **False** is false in every model
 - ✓ The truth value of every proposition symbol must be specify directly in the model'

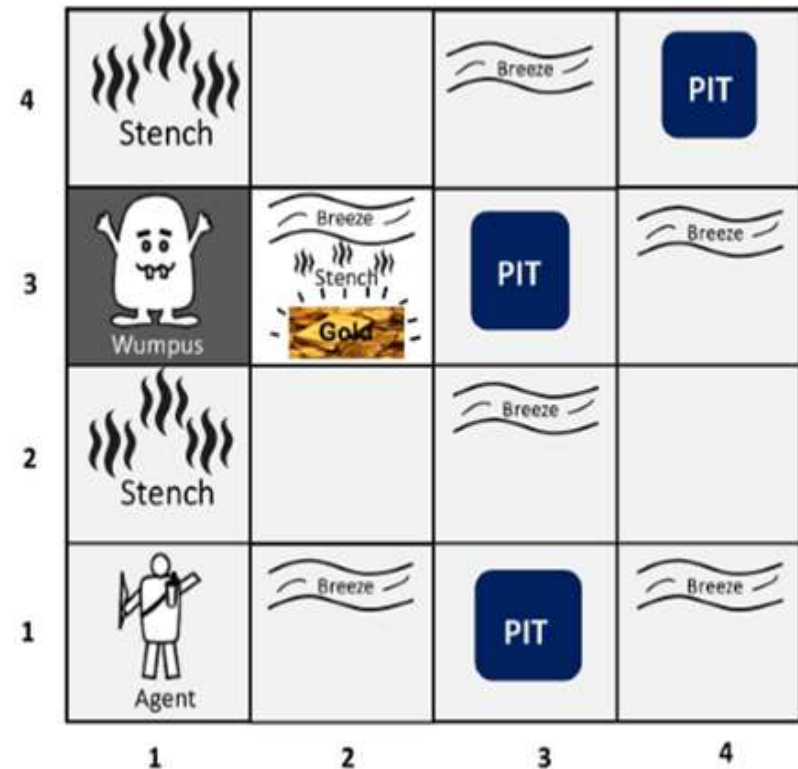
Propositional Logic

- ✓ For complex sentences
 - ✓ We have five rules, which hold for any sub-sentences P and Q in any model m .
 - $\neg P$ is true iff P is false in m .
 - $P \wedge Q$ is true iff both P and Q are true in m .
 - $P \vee Q$ is true iff either P or Q is true in m .
 - $P \Rightarrow Q$ is true unless P is true and Q is false in m .
 - $P \Leftrightarrow Q$ is true iff P and Q are both true or both false in m .
- ✓ Truth tables for the five logical connectives are given below:

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

Propositional Logic – Simple KB

- ✓ Let us consider for each i, j
 - ✓ Let $P_{i,j}$ be true if there is a Pit in $[i,j]$
 - ✓ Let $B_{i,j}$ be true if there is a breeze in $[i,j]$
- ✓ The KB includes the following sentences, each one labelled for convenience
 - ✓ $R_1 : \neg P_{1,1}$
 - ✓ $R_2 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$
 - ✓ $R_3 : B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$
 - ✓ $R_4 : \neg B_{1,1}$
 - ✓ $R_5 : B_{2,1}$



FIRST-ORDER LOGIC

- **Propositional logic** is too limited to effectively represent knowledge of complex environments. **first-order logic** is more expressive and capable of representing a significant amount of commonsense knowledge.
- First-order logic not only provides a more powerful framework for knowledge representation but also serves as the basis for many other representation languages.
- Programming languages (such as C++ or Java or Lisp) are by far the largest class of formal languages in common use.
- Programs themselves represent only computational processes.
- Data structures within programs can represent facts; for example, a program could use a 4×4 array to represent the contents of the wumpus world.
- Thus, the programming language statement $\text{World}[2,2] \leftarrow \text{Pit}$ is a fairly natural way to assert that there is a pit in square $[2,2]$.

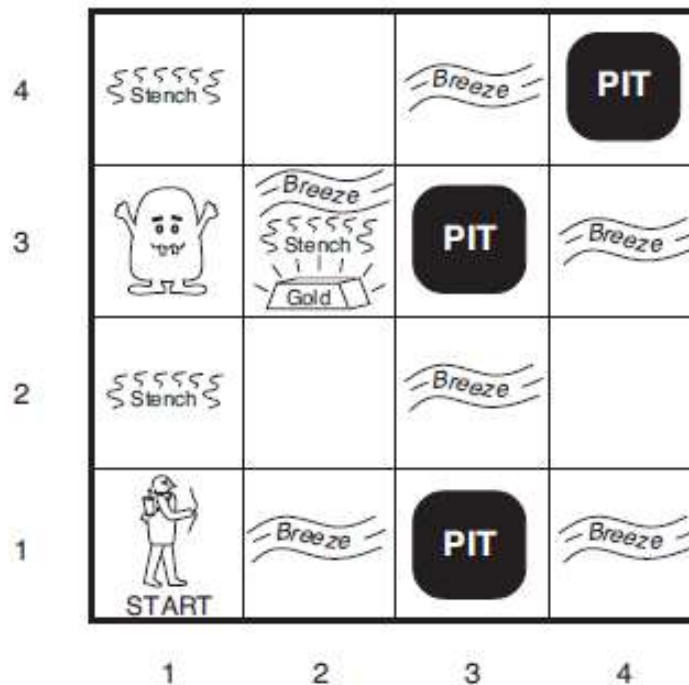


Figure 7.2 A typical wumpus world. The agent is in the bottom left corner, facing right.

- Programming languages generally lack a built-in, general-purpose mechanism for deriving facts from other facts. Instead, any updates to data structures are made through domain-specific procedures that the programmer must define based on their understanding of the problem domain.
- This procedural approach contrasts with the **declarative nature of propositional logic**, where knowledge is expressed separately from inference. In propositional logic, the process of inference is domain-independent, meaning that the rules for deriving conclusions are not tied to the specific details of the domain.

- A second drawback of data structures in programs is the lack of any easy way to say, for example, “There is a pit in [2,2] or [3,1]” or “If the wumpus is in [1,1] then he is not in [2,2].” Programs can store a single value for each variable, and some systems allow the value to be “unknown,” but they lack the expressiveness required to handle partial information.

- Propositional logic is a declarative language because its semantics is based on a truth relation between sentences and possible worlds.
- It also has sufficient expressive power to deal with partial information, using disjunction and negation.
- Propositional logic has a third property that is desirable in representation languages, namely, **compositionality**. In a compositional language, the meaning of a sentence is a function of the meaning of its parts.
- For example, the meaning of " $S_{1,4} \wedge S_{1,2}$ " is related to the meanings of " $S_{1,4}$ " and " $S_{1,2}$."
- Clearly, noncompositionality makes life much more difficult for the reasoning system.

- propositional logic lacks the expressive power to *concisely* describe an environment with many objects.
- For example, we were forced to write a separate rule about breezes and pits for each square,
such as $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$.
- In English, on the other hand, it seems easy enough to say, once and for all, “Squares adjacent to pits are breezy.”
- The syntax and semantics of English somehow make it possible to describe the environment concisely.

- Natural languages (such as English or Spanish) are very expressive indeed.
- Natural language can be viewed as a declarative knowledge representation language.
- The foundation of **propositional logic** can be used to develop a more expressive logic by incorporating representational ideas inspired by **natural language**. This approach enhances the ability to capture complex relationships and meanings, moving beyond the basic true/false propositions in propositional logic.

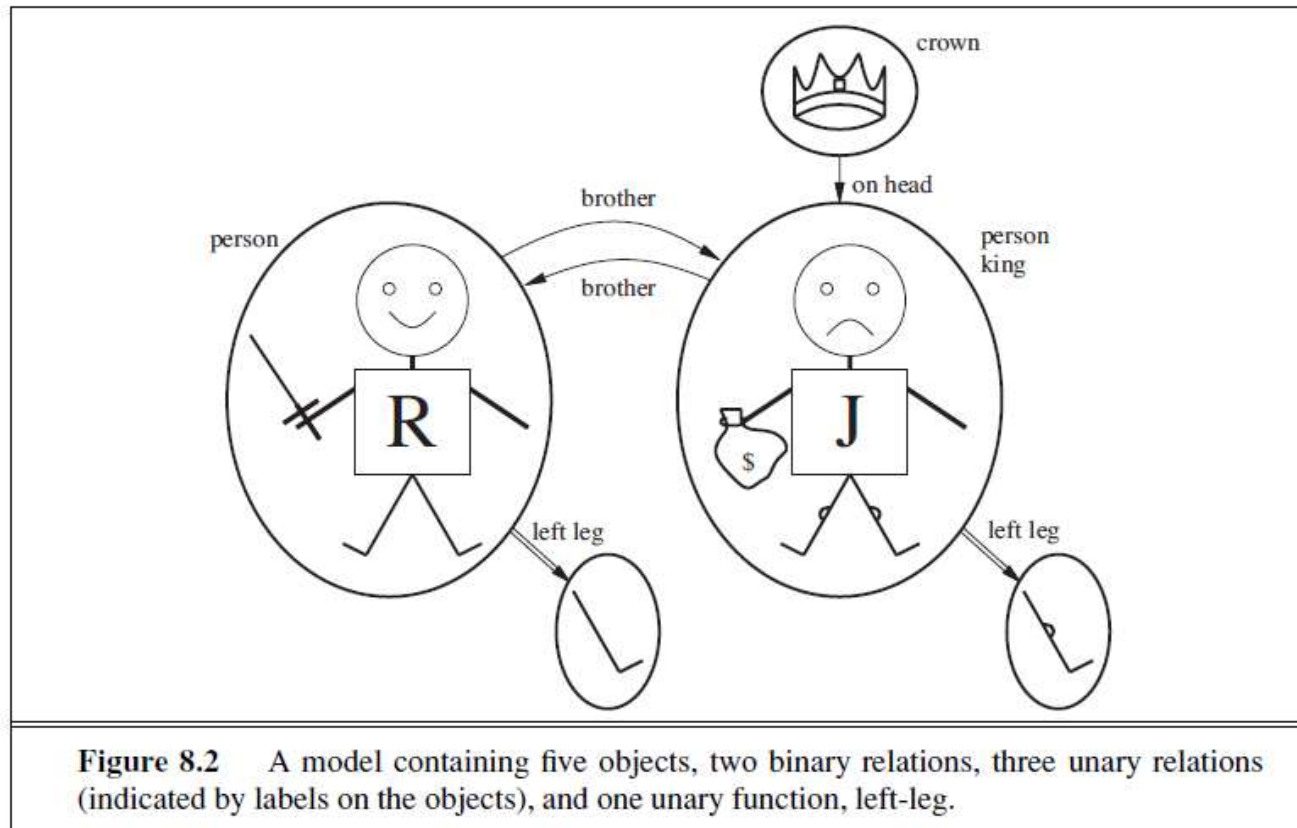
- the syntax of natural language, the most obvious elements are nouns and noun phrases that refer to **objects** (squares, pits, wumpuses) and verbs and verb phrases that refer to **relations** among objects (is breezy, is adjacent to, shoots). Some of these relations are **functions**—relations in which there is only one “value” for a given “input.”
- It is easy to start listing examples of objects, relations, and functions:
- Some examples follow:

- “One plus two equals three.”
Objects: one, two, three, one plus two;
Relation: equals;
Function: plus.
- “Squares neighboring the wumpus are smelly.”
Objects: wumpus, squares;
Property: smelly;
Relation: neighboring.
- “Evil King John ruled England in 1200.”
Objects: John, England, 1200;
Relation: ruled;
Properties: evil, king.

- The language of **first-order logic**, whose syntax and semantics is built around objects and relations.
- First-order logic can also express facts about *some* or *all* of the objects in the universe.
- This enables one to represent general laws or rules, such as the statement “Squares neighboring the wumpus are smelly.”

Language	Ontological Commitment (What exists in the world)	Epistemological Commitment (What an agent believes about facts)
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief $\in [0, 1]$
Fuzzy logic	facts with degree of truth $\in [0, 1]$	known interval value
Figure 8.1 Formal languages and their ontological and epistemological commitments.		

- **Models for first-order logic are**
- First, **they have objects in them!** The **domain** of a model is the set of objects or **domain elements** it contains.
- The domain is required to be *nonempty*—every possible world must contain at least one object.
- Figure 8.2 shows a model with five objects:
Richard the Lionheart, King of England from 1189 to 1199; his younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; and a crown.



- The objects in the model may be *related* in various ways. In the figure, Richard and John are brothers. Formally speaking, a relation is just the set of **tuples** of objects that are related.
- Thus, the brotherhood relation in this model is the set
- { <Richard the Lionheart, King John>, <King John, Richard the Lionheart> } .
- The crown is on King John's head, so the "on head" relation contains
- just one tuple, <the crown, King John>.

- The “brother” and “on head” relations are binary relations—that is, they relate pairs of objects.
- The model also contains unary relations, or properties: the “person” property is true of both Richard and John; the “king” property is true only of John (presumably because Richard is dead at this point); and the “crown” property is true only of the crown.

- Certain kinds of relationships are best considered as functions, in that a given object must be related to **exactly one object** in this way. For example, each person has one left leg, so the model has a unary “left leg” function that includes the following mappings:

Richard the Lionheart $\rightarrow$ Richard’s left leg

King John $\rightarrow$ John’s left leg .

- models in first-order logic require **total functions**, that is, there must be a value for every input tuple.

Symbols and interpretations

- The basic syntactic elements of first-order logic are the symbols that stand for objects, relations, and functions. The symbols, therefore, come in three kinds: **constant symbols**, which stand for objects; **predicate symbols**, which stand for relations; and **function symbols**, which stand for functions.
- We adopt the convention that these symbols will begin with uppercase letters.
- For example, we might use the constant symbols Richard and John;
- The predicate symbols Brother , OnHead, Person, King, and Crown; and the function symbol LeftLeg.

- each model includes an **interpretation** that specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols.
- the **intended interpretation**—is as follows:
 - Richard refers to Richard the Lionheart and John refers to the evil King John.
 - Brother refers to the brotherhood relation, that is, the set of tuples of objects; OnHead refers to the “on head” relation that holds between the crown and King John; Person, King, and Crown refer to the sets of objects that are persons, kings, and crowns.
 - LeftLeg refers to the “left leg” function, that is, the mapping.

$$\begin{aligned}
\textit{Sentence} &\rightarrow \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} &\rightarrow \textit{Predicate} \mid \textit{Predicate}(\textit{Term}, \dots) \mid \textit{Term} = \textit{Term} \\
\textit{ComplexSentence} &\rightarrow (\textit{Sentence}) \mid [\textit{Sentence}] \\
&\mid \neg \textit{Sentence} \\
&\mid \textit{Sentence} \wedge \textit{Sentence} \\
&\mid \textit{Sentence} \vee \textit{Sentence} \\
&\mid \textit{Sentence} \Rightarrow \textit{Sentence} \\
&\mid \textit{Sentence} \Leftrightarrow \textit{Sentence} \\
&\mid \textit{Quantifier Variable}, \dots \textit{Sentence} \\
\\
\textit{Term} &\rightarrow \textit{Function}(\textit{Term}, \dots) \\
&\mid \textit{Constant} \\
&\mid \textit{Variable} \\
\\
\textit{Quantifier} &\rightarrow \forall \mid \exists \\
\textit{Constant} &\rightarrow A \mid X_1 \mid \textit{John} \mid \dots \\
\textit{Variable} &\rightarrow a \mid x \mid s \mid \dots \\
\textit{Predicate} &\rightarrow \textit{True} \mid \textit{False} \mid \textit{After} \mid \textit{Loves} \mid \textit{Raining} \mid \dots \\
\textit{Function} &\rightarrow \textit{Mother} \mid \textit{LeftLeg} \mid \dots
\end{aligned}$$

OPERATOR PRECEDENCE : $\neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow$

- a model in first-order logic consists of a set of objects and an interpretation that maps constant symbols to objects, predicate symbols to relations on those objects, and function symbols to functions on those objects.

- A **term** is a logical expression that refers to an object. Constant symbols are therefore terms
- For the expression “King John’s left leg” , we use LeftLeg(John).
- a complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol. It is important to remember that a complex term is just a complicated kind of name.
- The formal semantics of terms is straightforward. Consider a term $f(t_1, \dots, t_n)$.
- The function symbol f refers to some function in the model (call it F); the argument terms refer to objects in the domain

Atomic sentence

- Now that we have both terms for referring to objects and predicate symbols for referring to relations, we can put them together to make **atomic sentences** that state facts. An **atomic sentence** (or **atom** for short) is formed from a predicate symbol optionally followed by a parenthesized list of terms, such as
- Brother (Richard , John).
- This states that Richard the Lionheart is the brother of King John.
- Atomic sentences can have complex terms as arguments. Thus,
- Married(Father (Richard),Mother (John))
- *An atomic sentence is **true** in a given model if the relation referred to by the predicate symbol holds among the objects referred to by the arguments.*

complex sentences

- We can use **logical connectives** to construct more complex sentences,

$\neg \text{Brother}(\text{LeftLeg}(\text{Richard}), \text{John})$

$\text{Brother}(\text{Richard}, \text{John}) \wedge \text{Brother}(\text{John}, \text{Richard})$

$\text{King}(\text{Richard}) \vee \text{King}(\text{John})$

$\neg \text{King}(\text{Richard}) \Rightarrow \text{King}(\text{John})$

- **Quantifiers** is used to express properties of entire collections of objects.
- First-order logic contains two standard quantifiers, called *universal* and *existential*.
- **Universal quantification ($\forall$):**
 - The rule, “All kings are persons,” is written in first-order logic as
 - $\forall x \text{ King}(x) \Rightarrow \text{Person}(x)$.
 - $\forall$ is usually pronounced “For all . . .”.
 - Thus, the sentence says, “For all x , if x is a king, then x is a person.” The symbol x is called a **variable**.
 - A Term with no variables is called a **ground term**
 - The sentence $\forall x P$, says that P is true for every object x .

- **Existential quantification ($\exists$)**
- Universal quantification makes statements about every object. Similarly, we can make a statement about *some* object in the universe without naming it, by using an existential quantifier.
- To say, for example, that King John has a crown on his head, we write

$$\exists x \text{ Crown}(x) \wedge \text{OnHead}(x, \text{John}) .$$
- $\exists x$ is pronounced “There exists an x such that . . .” or “For some x . . .”.
- The sentence $\exists x P$ says that P is true for at least one object x .
- That is, at least one of the following is true:
 Richard the Lionheart is a crown $\wedge$ Richard the Lionheart is on John’s head;
 King John is a crown $\wedge$ King John is on John’s head;
 Richard’s left leg is a crown $\wedge$ Richard’s left leg is on John’s head;
 John’s left leg is a crown $\wedge$ John’s left leg is on John’s head;
 The crown is a crown $\wedge$ the crown is on John’s head.
- The fifth assertion is true in the model,

- **Nested quantifiers**
- We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type. For example, “Brothers are siblings” can be written as
- $\forall x \forall y \text{ Brother}(x, y) \Rightarrow \text{Sibling}(x, y)$.
- Consecutive quantifiers of the same type can be written as one quantifier with several variables.
- For example, to say that siblinghood is a symmetric relationship, we can write
- $\forall x, y \text{ Sibling}(x, y) \Leftrightarrow \text{Sibling}(y, x)$.
- Other Ex: $\forall x \exists y \text{ Loves}(x, y)$.

Connections between $\forall$ and $\exists$

- The two quantifiers are actually intimately connected with each other, through negation. “Everyone likes ice cream” means that there is no one who does not like ice cream:
- $\forall x \text{ Likes}(x, \text{IceCream})$ is equivalent to $\neg \exists x \neg \text{Likes}(x, \text{IceCream})$.

- Because $\forall$ is really a conjunction over the universe of objects and $\exists$ is a disjunction, it should not be surprising that they obey De Morgan's rules. The De Morgan rules for quantified and unquantified sentences are as follows:

$$\forall x \neg P \equiv \neg \exists x P$$

$$\neg \forall x P \equiv \exists x \neg P$$

$$\forall x P \equiv \neg \exists x \neg P$$

$$\exists x P \equiv \neg \forall x \neg P$$

$$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$$

$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$$

$$P \wedge Q \equiv \neg(\neg P \vee \neg Q)$$

$$P \vee Q \equiv \neg(\neg P \wedge \neg Q) .$$

Equality

- First-order logic includes one more way to make atomic sentences by using the **equality symbol** to signify that two terms refer to the same object.
- For example,
- $\text{Father (John)} = \text{Henry}$
- says that the object referred to by Father (John) and the object referred to by Henry are the same.

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**

Inference in first order logic

- Complete inference can be achieved for propositional logic is extended to obtain algorithms that can answer any answerable question stated in first-order logic.
- Propositional vs. first order inference,
- unification & lifts
- forward chaining
- Backward chaining
- Resolution

PROPOSITIONAL VS. FIRST-ORDER INFERENCE

- **Inference rules for quantifiers**
- **Reduction to propositional inference**

Inference rules for quantifiers

- Suppose our knowledge base contains the standard axiom stating that all greedy kings are evil:

$$\forall x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$$

- Then it seems quite permissible to infer any of the following sentences:

$$\text{King}(\text{John}) \wedge \text{Greedy}(\text{John}) \Rightarrow \text{Evil}(\text{John})$$

$$\text{King}(\text{Richard}) \wedge \text{Greedy}(\text{Richard}) \Rightarrow \text{Evil}(\text{Richard})$$

$$\text{King}(\text{Father}(\text{John})) \wedge \text{Greedy}(\text{Father}(\text{John})) \Rightarrow \text{Evil}(\text{Father}(\text{John}))$$

⋮

- The rule of **Universal Instantiation** (**UI** for short) says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable. To write out the inference rule formally, we use the notion of **substitutions**.
- Let $\text{SUBST}(\theta, \alpha)$ denote the result of applying the substitution θ to the sentence α . Then the rule is written

$$\frac{\forall v \alpha}{\text{SUBST}(\{v/g\}, \alpha)}$$

- for any variable v and ground term g . For example, the three sentences given earlier are obtained with the substitutions $\{x/\text{John}\}$, $\{x/\text{Richard}\}$, and $\{x/\text{Father (John)}\}$.

- In the rule for **Existential Instantiation**, the variable is replaced by a single *new constant symbol*. The formal statement is as follows: for any sentence α , variable v , and constant symbol k that does not appear elsewhere in the knowledge base.

$$\frac{\exists v \alpha}{\text{SUBST}(\{v/k\}, \alpha)}$$

- For example, from the sentence
- $\exists x \text{Crown}(x) \wedge \text{OnHead}(x, \text{John})$
- we can infer the sentence
- $\text{Crown}(C1) \wedge \text{OnHead}(C1, \text{John})$
- as long as $C1$ does not appear elsewhere in the knowledge base. Basically, the existential sentence says there is some object satisfying a condition, and applying the existential instantiation rule just gives a name to that object.

Reduction to propositional inference

- Once we have rules for inferring nonquantified sentences from quantified sentences, it becomes possible to reduce first-order inference to propositional inference.
- The first idea is that, just as an existentially quantified sentence can be replaced by one instantiation, a universally quantified sentence can be replaced by the set of *all possible* instantiations.

- For example, suppose our knowledge base contains just the sentences

$$\forall x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$$

$$\text{King}(\text{John})$$

$$\text{Greedy}(\text{John})$$

$$\text{Brother}(\text{Richard}, \text{John}) .$$

- Then we apply UI to the first sentence using all possible ground-term substitutions from the vocabulary of the knowledge base—in this case, $\{x/\text{John}\}$ and $\{x/\text{Richard}\}$. We obtain

$$\text{King}(\text{John}) \wedge \text{Greedy}(\text{John}) \Rightarrow \text{Evil}(\text{John})$$

$$\text{King}(\text{Richard}) \wedge \text{Greedy}(\text{Richard}) \Rightarrow \text{Evil}(\text{Richard})$$

- and we discard the universally quantified sentence. Now, the knowledge base is essentially propositional if we view the ground atomic sentences— $\text{King}(\text{John})$, $\text{Greedy}(\text{John})$, and so on—as proposition symbols.
- Therefore, we can apply any of the complete propositional algorithms to obtain conclusions such as $\text{Evil}(\text{John})$.

UNIFICATION AND LIFTING

- A first-order inference rule(Lifting)
- Unification

A first-order inference rule(Lifting)

- The inference that John is evil—that is, that $\{x/\text{John}\}$ solves the query $\text{Evil}(x)$ —works like this: to use the rule that greedy kings are evil, find some x such that x is a king and x is greedy, and then infer that this x is evil.
- More generally, if there is some substitution θ that makes each of the conjuncts of the premise of the implication identical to sentences already in the knowledge base, then we can assert the conclusion of the implication, after applying θ .

- Suppose that instead of knowing Greedy(John), we know that *everyone* is greedy:
- $\forall y \text{ Greedy}(y)$.
- Then we would still like to be able to conclude that Evil(John), because we know that John is a king (given) and John is greedy (because everyone is greedy).
- What we need for this to work is to find a substitution both for the variables in the implication sentence and for the variables in the sentences that are in the knowledge base.
- In this case, applying the substitution $\{x/\text{John}, y/\text{John}\}$ to the implication premises King(x) and Greedy(x) and the knowledge-base sentences King(John) and Greedy(y) will make them identical.
- Thus, we can infer the conclusion of the implication.

- This inference process can be captured as a single inference rule that we call **Generalized Modus Ponens**: For atomic sentences p_i , p_i' , and q , where there is a substitution θ such that $\text{SUBST}(\theta, p_i') = \text{SUBST}(\theta, p_i)$, for all i ,

$$\frac{p_1', p_2', \dots, p_n', (p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)} .$$

- There are $n+1$ premises to this rule: the n atomic sentences p_i and the one implication. The conclusion is the result of applying the substitution θ to the consequent q . For our example:

$$\begin{array}{ll} p_1' \text{ is } King(John) & p_1 \text{ is } King(x) \\ p_2' \text{ is } Greedy(y) & p_2 \text{ is } Greedy(x) \\ \theta \text{ is } \{x/John, y/John\} & q \text{ is } Evil(x) \\ \text{SUBST}(\theta, q) \text{ is } Evil(John) . & \end{array}$$

- Generalized Modus Ponens is a **lifted** version of Modus Ponens—it raises Modus Ponens from ground (variable-free) propositional logic to first-order logic.

Unification

- Lifted inference rules require finding substitutions that make different logical expressions look identical. This process is called **unification** and is a key component of all first-order inference algorithms.
- The algorithm takes two sentences and returns a **unifier** for them if one exists:

$$\text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q)$$

- Let us look at some examples of how UNIFY should behave. Suppose we have a query $\text{AskVars}(\text{Knows}(\text{John}, x))$: whom does John know?
- Answers to this query can be found by finding all sentences in the knowledge base that unify with $\text{Knows}(\text{John}, x)$. Here are the results of unification with four different sentences that might be in the knowledge base:

$\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(\text{John}, \text{Jane})) = \{x/\text{Jane}\}$

$\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, \text{Bill})) = \{x/\text{Bill}, y/\text{John}\}$

$\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, \text{Mother}(y))) = \{y/\text{John}, x/\text{Mother}(\text{John})\}$

$\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(x, \text{Elizabeth})) = \text{fail}.$

- The last unification fails because x cannot take on the values John and Elizabeth at the same time. Now, remember that $\text{Knows}(x, \text{Elizabeth})$ means “Everyone knows Elizabeth,” so we *should* be able to infer that John knows Elizabeth.
- The problem arises only because the two sentences happen to use the same variable name, x . The problem can be avoided by **standardizing apart** one of the two sentences being unified, which means renaming its variables to avoid name clashes. For example, we can rename x in $\text{Knows}(x, \text{Elizabeth})$ to x_{17} (a new variable name) without changing its meaning. Now the unification will work:

- There is one more complication: we said that UNIFY should return a substitution that makes the two arguments look the same. But there could be more than one such unifier.
- For example, $\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, z))$ could return $\{y/\text{John}, x/z\}$ or $\{y/\text{John}, x/\text{John}, z/\text{John}\}$.
- The first unifier gives $\text{Knows}(\text{John}, z)$ as the result of unification, whereas the second gives $\text{Knows}(\text{John}, \text{John})$.
- The second result could be obtained from the first by an additional substitution $\{z/\text{John}\}$; we say that the first unifier is *more general* than the second, because it places fewer restrictions on the values of the variables.
- It turns out that, for every unifiable pair of expressions, there is a single **most general unifier** (or MGU) that is unique up to renaming and substitution of variables. (For example, $\{x/\text{John}\}$ and $\{y/\text{John}\}$ are considered equivalent, as are $\{x/\text{John}, y/\text{John}\}$ and $\{x/\text{John}, y/x\}$.) In this case it is $\{y/\text{John}, x/z\}$.

RESOLUTION

- first-order resolution requires that sentences be in conjunctive normal form (CNF)—that is, a conjunction of clauses, where each clause is a disjunction of literals. Literals can contain variables, which are assumed to be

$$\forall x \text{ American}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x, y, z) \wedge \text{Hostile}(z) \Rightarrow \text{Criminal}(x)$$

becomes, in CNF,

$$\neg \text{American}(x) \vee \neg \text{Weapon}(y) \vee \neg \text{Sells}(x, y, z) \vee \neg \text{Hostile}(z) \vee \text{Criminal}(x)$$

- *Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence.*

- The resolution rule for first-order clauses is simply a lifted version of the propositional resolution rule.
- Two clauses, which are assumed to be standardized apart so that they share no variables, can be resolved if they contain

$$\frac{\ell_1 \vee \dots \vee \ell_k, \quad m_1 \vee \dots \vee m_n}{\text{SUBST}(\theta, \ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n)}$$

where $\text{UNIFY}(\ell_i, \neg m_j) = \theta$. For example, we can resolve the two clauses

$$[\textit{Animal}(F(x)) \vee \textit{Loves}(G(x), x)] \quad \text{and} \quad [\neg \textit{Loves}(u, v) \vee \neg \textit{Kills}(u, v)]$$

of the DB. Thus, we have

by eliminating the complementary literals $Loves(G(x), x)$ and $\neg Loves(u, v)$, with unifier $\theta = \{u/G(x), v/x\}$, to produce the **resolvent** clause

$$[Animal(F(x)) \vee \neg Kills(G(x), x)] .$$

- This rule is called the binary resolution rule because it resolves exactly two literals.

- **Different forms of sentences:**
- Some real-world knowledge bases satisfy certain restrictions on the form of sentences they contain, which enables them to use a more restricted and efficient inference algorithm.
- **Definite clause**
- One such restricted form is the **definite clause**, which is a disjunction of literals of which *exactly one is positive*.
- For example, the clause $(\neg L_{1,1} \vee \neg \text{Breeze} \vee B_{1,1})$ is a definite clause, whereas $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1})$ is not.
- **Horn clause**
- Slightly more general is the **Horn clause**, which is a disjunction of literals of which *at most one is positive*.
- So all definite clauses are Horn clauses, as are clauses with no positive literals; these are called **goal clauses**.

FORWARD CHAINING

- forward-chaining algorithm start with the atomic sentences in the knowledge base and apply Modus Ponens rule in the forward direction, adding new atomic sentences, until no further inferences can be made.
- Modus Ponens rule can be summarized as "*P* implies *Q*. *P* is true. Therefore, *Q* must also be true."

$$(MP) \quad \frac{p \Rightarrow q, p}{\therefore q}$$

- First-order definite clauses are disjunctions of literals of which *exactly one is positive*.
- The following are first-order definite clauses:
- $\text{King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$.
- $\text{King}(\text{John})$.
- $\text{Greedy}(y)$.

- Consider the following problem:

The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

- We will prove that West is a criminal.
- First, we will represent these facts as first-order definite clauses.
- “. . . it is a crime for an American to sell weapons to hostile nations”:

$$\text{American}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x, y, z) \wedge \text{Hostile}(z) \Rightarrow \text{Criminal}(x) .$$

(9.3)

- “Nono . . . has some missiles.”
- The sentence $\exists x \text{ Owns}(\text{Nono}, x) \wedge \text{Missile}(x)$ is transformed into two definite clauses by Existential Instantiation, introducing a new constant M1:
- $\text{Owns}(\text{Nono}, \text{M1})$ (9.4)
- $\text{Missile}(\text{M1})$ (9.5)
- “All of its missiles were sold to it by Colonel West”:
- $\text{Missile}(x) \wedge \text{Owns}(\text{Nono}, x) \Rightarrow \text{Sells}(\text{West}, x, \text{Nono})$ (9.6)
- We will also need to know that missiles are weapons:
- $\text{Missile}(x) \Rightarrow \text{Weapon}(x)$ (9.7)

- and we must know that an enemy of America counts as “hostile”:
- $\text{Enemy}(x, \text{America}) \Rightarrow \text{Hostile}(x)$. (9.8)
- “West, who is American . . .”:
- $\text{American}(\text{West})$. (9.9)
- “The country Nono, an enemy of America . . .”:
- $\text{Enemy}(\text{Nono}, \text{America})$. (9.10)

- The first forward-chaining algorithm we consider is a simple one, shown in Figure 9.3.
- Starting from the known facts, it triggers all the rules whose premises are satisfied, adding their conclusions to the known facts.
- The process repeats until the query is answered or no new facts are added.

Forward-chaining algorithm.

```
function FOL-FC-ASK( $KB, \alpha$ ) returns a substitution or false
  inputs:  $KB$ , the knowledge base, a set of first-order definite clauses
            $\alpha$ , the query, an atomic sentence
  local variables:  $new$ , the new sentences inferred on each iteration

  repeat until  $new$  is empty
     $new \leftarrow \{ \}$ 
    for each  $rule$  in  $KB$  do
       $(p_1 \wedge \dots \wedge p_n \Rightarrow q) \leftarrow \text{STANDARDIZE-VARIABLES}(rule)$ 
      for each  $\theta$  such that  $\text{SUBST}(\theta, p_1 \wedge \dots \wedge p_n) = \text{SUBST}(\theta, p'_1 \wedge \dots \wedge p'_n)$ 
        for some  $p'_1, \dots, p'_n$  in  $KB$ 
           $q' \leftarrow \text{SUBST}(\theta, q)$ 
          if  $q'$  does not unify with some sentence already in  $KB$  or  $new$  then
            add  $q'$  to  $new$ 
             $\phi \leftarrow \text{UNIFY}(q', \alpha)$ 
            if  $\phi$  is not fail then return  $\phi$ 
      add  $new$  to  $KB$ 
  return false
```

- The implication sentences are (9.3), (9.6), (9.7), and (9.8).
- Two iterations are required:
- On the first iteration, rule (9.3) has unsatisfied premises.
- Rule (9.6) is satisfied with $\{x/M1\}$, and $Sells(West, M1, Nono)$ is added.
- Rule (9.7) is satisfied with $\{x/M1\}$, and $Weapon(M1)$ is added.
- Rule (9.8) is satisfied with $\{x/Nono\}$, and $Hostile(Nono)$ is added.
- On the second iteration, rule (9.3) is satisfied with $\{x/West, y/M1, z/Nono\}$, and $Criminal(West)$ is added.

$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x) . \quad (9.3)$

$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono) \quad (9.6)$

$Missile(x) \Rightarrow Weapon(x) \quad (9.7)$

$Enemy(x, America) \Rightarrow Hostile(x) . \quad (9.8)$

- Figure 9.4 shows the proof tree that is generated.
- Notice that no new inferences are possible at this point because every sentence that could be concluded by forward chaining is already contained explicitly in the KB.

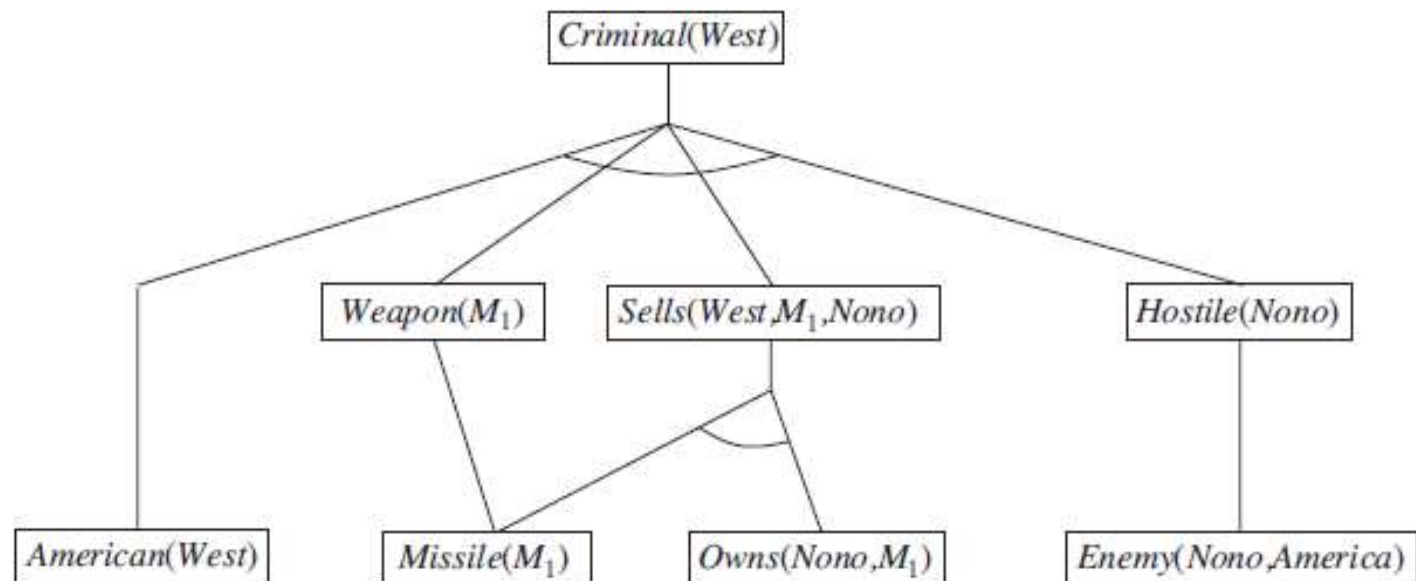


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

- FOL-FC-ASK is easy to analyze.
- First, it is **sound**, because every inference is just an application of Generalized Modus Ponens, which is sound.
- Second, it is **complete** for definite clause knowledge bases; that is, it answers every query whose answers are entailed by any knowledge base of definite clauses.

BACKWARD CHAINING

- The second major family of logical inference algorithms uses the **backward chaining** approach.
- These algorithms work backward from the goal, chaining through rules to find known facts that support the proof.

function FOL-BC-ASK($KB, query$) **returns** a generator of substitutions
 return FOL-BC-OR($KB, query, \{ \}$)

generator FOL-BC-OR($KB, goal, \theta$) **yields** a substitution
 for each rule ($lhs \Rightarrow rhs$) **in** FETCH-RULES-FOR-GOAL($KB, goal$) **do**
 (lhs, rhs) $\leftarrow$ STANDARDIZE-VARIABLES((lhs, rhs))
 for each θ' **in** FOL-BC-AND($KB, lhs, UNIFY(rhs, goal, \theta)$) **do**
 yield θ'

generator FOL-BC-AND($KB, goals, \theta$) **yields** a substitution
 if $\theta = failure$ **then return**
 else if LENGTH($goals$) = 0 **then yield** θ
 else do
 $first, rest \leftarrow$ FIRST($goals$), REST($goals$)
 for each θ' **in** FOL-BC-OR($KB, SUBST(\theta, first), \theta$) **do**
 for each θ'' **in** FOL-BC-AND($KB, rest, \theta'$) **do**
 yield θ''

Figure 9.6 A simple backward-chaining algorithm for first-order knowledge bases.

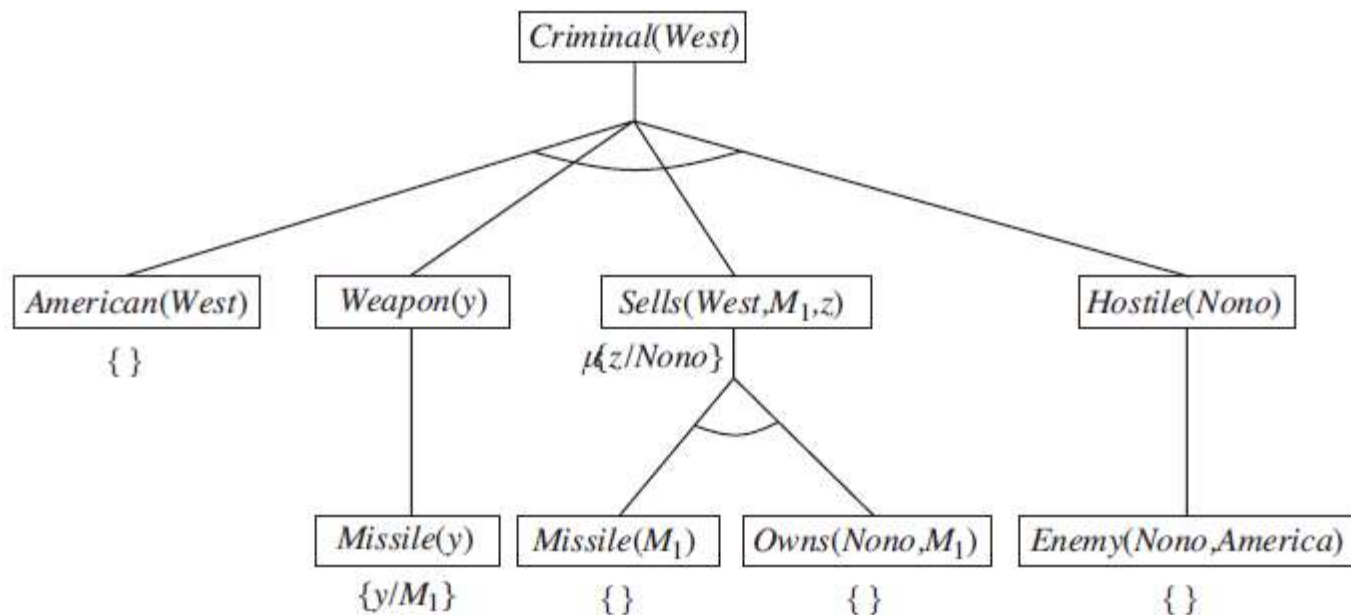


Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove $Criminal(West)$, we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally $Hostile(z)$, z is already bound to $Nono$.

- Figure 9.6 shows a backward-chaining algorithm for definite clauses.
- $\text{FOL-BC-ASK}(\text{KB}, \text{goal})$ will be proved if the knowledge base contains a clause of the form $\text{lhs} \Rightarrow \text{goal}$, where lhs (left-hand side) is a list of conjuncts.
- An atomic fact like $\text{American}(\text{West})$ is considered as a clause whose lhs is the empty list.
- Now a query that contains variables might be proved in multiple ways. For example, the query $\text{Person}(x)$ could be proved with the substitution $\{x/\text{John}\}$ as well as with $\{x/\text{Richard}\}$.
- So we implement GENERATOR FOL-BC-ASK as a **generator**—a function that returns multiple times, each time giving one possible result.

- Backward chaining is a kind of AND/OR search—the OR part because the goal query can be proved by any rule in the knowledge base, and the AND part because all the conjuncts in the lhs of a clause must be proved.
- FOL-BC-OR works by fetching all clauses that might unify with the goal, standardizing the variables in the clause to be brand-new variables, and then, if the rhs of the clause does indeed unify with the goal, proving every conjunct in the lhs, using FOL-BC-AND.
- That function in turn works by proving each of the conjuncts in turn, keeping track of the accumulated substitution as we go.
- Figure 9.7 is the proof tree for deriving Criminal (West) from sentences (9.3) through (9.10).

- Backward chaining is clearly a depth-first search algorithm.
- This means that its space requirements are linear in the size of the proof.
- It also means that backward chaining (unlike forward chaining) suffers from problems with repeated states and incompleteness.

Learning from Observation

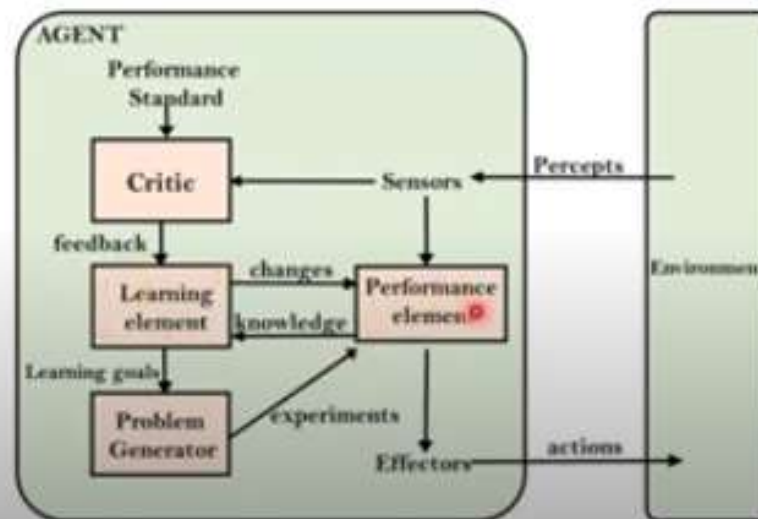
Learning



- **Learning** is Agent's percepts should be used for **acting**,
- It also used for improving the **agent's ability to act in the future**.
- Learning takes place as the agent **observes**, its interactions with the **world and its own decision-making processes**.

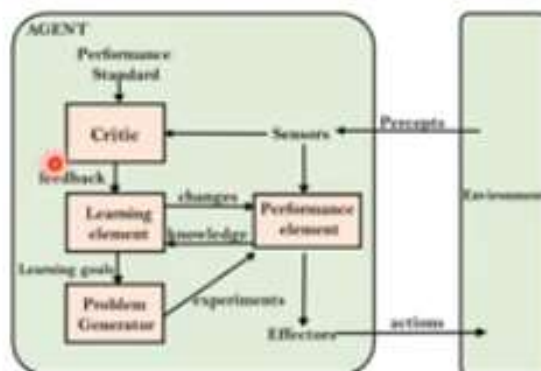
Forms of Learning

- **Learning Agent** can be thought of as containing a **Performance Element**, that decides, what actions to take, and a **learning element** that **modifies the performance element** so that it **makes better decisions**.



Learning Agent

- The design of a learning element is affected by three major issues:
 - Which *components* of the performance element are to be learned.
 - What *feedback* is available to learn these components.
 - What *representation* is used for the components.



Components of Learning Agents

- The components of learning agents include the following
 1. A **direct mapping** from conditions on the **current state** to **actions**.
 2. A **means** to infer **relevant properties** of the world from the **percept sequence**.
 3. **Information** about the way the world evolves and about the **results of possible actions** the agent can take.
 4. **Utility information** indicating the **desirability** of world states.
 5. **Action-value information** indicating the **desirability** of actions.
 6. **Goals** that describe classes of states whose achievement **maximizes the agent's utility**.

Automatic Taxi Driver



- Learning Agents' components can be learned from appropriate feedback.
- An agent training to become a **taxi driver**.
- Every time the instructor gives a command "Brake!", the agent can learn a **condition-action rule**, when we apply brake (component 1).
- By seeing many **camera images**, it can learn to recognize the objects on road (2).
- By trying **actions and observing the results**, for example, braking hard on a wet road, it can learn the effects of its actions (3).
- If there is **no tip from passengers**, but they shaken up during the trip, then it can learn a useful component of its overall utility function (4).
- The **type of feedback** available for learning is usually the most important factor in determining the nature of the learning problem that the agent faces.

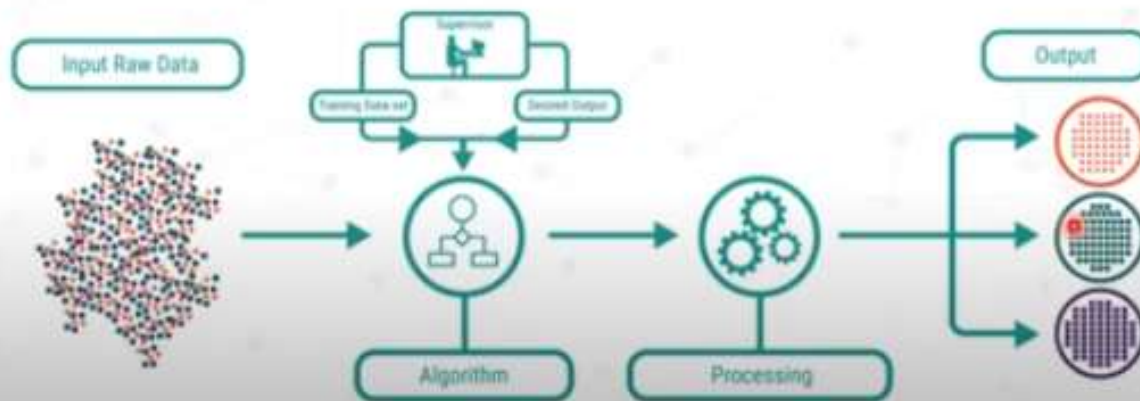
Popular Machine Learning Algorithms

- The field of machine learning usually distinguishes three cases:
 - **Supervised Learning**
 - **Unsupervised Learning**
 - **Reinforcement learning**

Supervised Learning

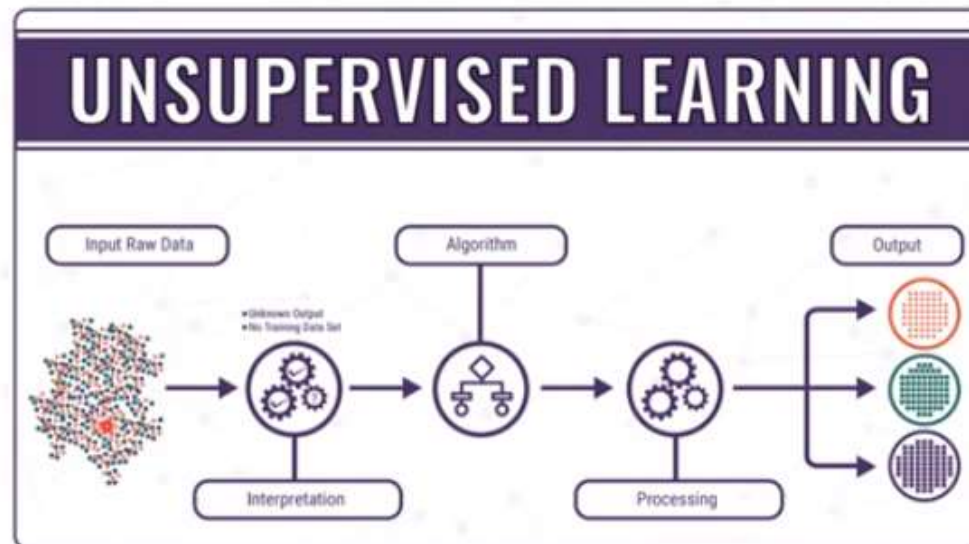
- **Supervised Learning** - the algorithm learns on a **labeled dataset**, providing an **answer key** that the algorithm can use to evaluate its **accuracy on training data**.

SUPERVISED LEARNING



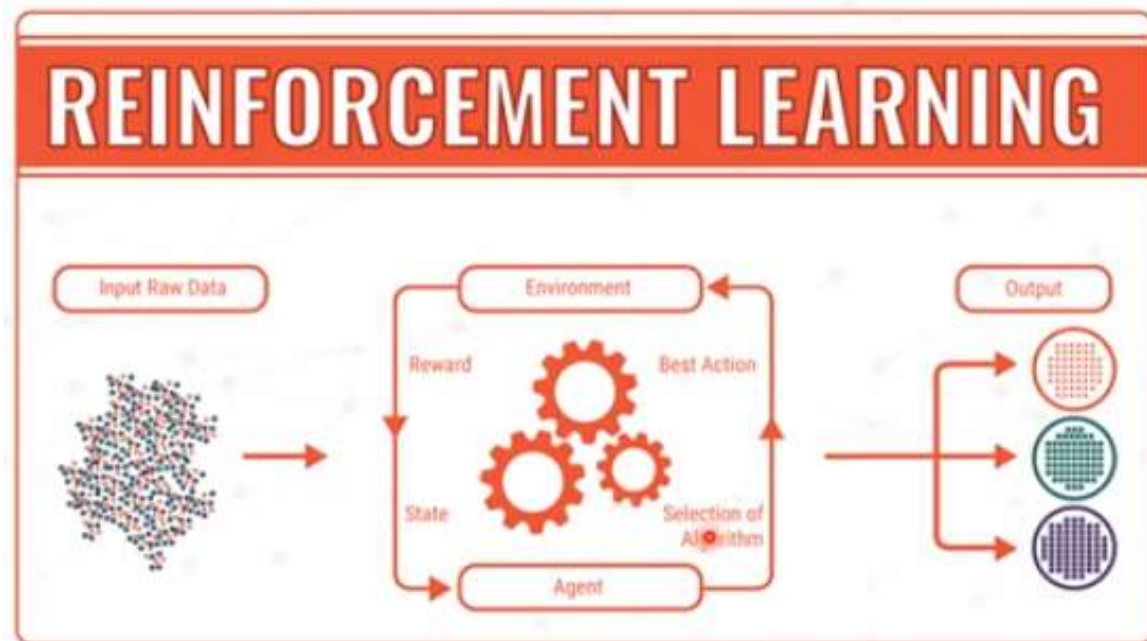
Unsupervised Learning

- **Unsupervised Learning** - provides **unlabeled data**, the algorithm tries to **make sense** of by **extracting features and patterns** on its own.



Reinforcement learning

- **Reinforcement learning** - is a type of **dynamic programming** that trains algorithms using a system of **reward** and **punishment**.



Inductive learning

- Inductive Learning Algorithm (ILA) is an iterative and inductive machine learning algorithm that is used for generating a set of classification rules, which produces rules of the form “IF-THEN”, for a set of examples, producing rules at each iteration and appending to the set of rules.

- There are basically two methods for knowledge extraction firstly from domain experts and then with machine learning.
- For a very large amount of data, the domain experts are not very useful and reliable.
- So we move towards the machine learning approach for this work.
- To use machine learning One method is to replicate the expert's logic in the form of algorithms but this work is very tedious, time taking, and expensive.
- So we move towards the inductive algorithms which generate the strategy for performing a task and need not instruct separately at each step.

Basic Requirements to Apply Inductive Learning Algorithm

- List the examples in the form of a table 'T' where each row corresponds to an example and each column contains an attribute value.
- Create a set of m training examples, each example composed of k attributes and a class attribute with n possible decisions.
- Create a rule set, R , having the initial value false.
- Initially, all rows in the table are unmarked.

Necessary Steps for Implementation

- **Step 1:** divide the table 'T' containing m examples into n sub-tables (t1, t2,.....tn). One table for each possible value of the class attribute. (repeat steps 2-8 for each sub-table)
- **Step 2:** Initialize the attribute combination count 'j' = 1.
- **Step 3:** For the sub-table on which work is going on, divide the attribute list into distinct combinations, each combination with 'j' distinct attributes.
- **Step 4:** For each combination of attributes, count the number of occurrences of attribute values that appear under the same combination of attributes in unmarked rows of the sub-table under consideration, and at the same time, not appears under the same combination of attributes of other sub-tables. Call the first combination with the maximum number of occurrences the max-combination 'MAX'.
- **Step 5:** If 'MAX' == null, increase 'j' by 1 and go to Step 3.
- **Step 6:** Mark all rows of the sub-table where working, in which the values of 'MAX' appear, as classified.
- **Step 7:** Add a rule (IF attribute = "XYZ" → THEN decision is YES/ NO) to R whose left-hand side will have attribute names of the 'MAX' with their values separated by AND, and its right-hand side contains the decision attribute value associated with the sub-table.
- **Step 8:** If all rows are marked as classified, then move on to process another sub-table and go to Step 2. Else, go to Step 4. If no sub-tables are available, exit with the set of rules obtained till then.

- **An example showing the use of ILA** suppose an example set having attributes Place type, weather, location examples, our task is to find rules that underlie the decision.

Example no.	Place type	weather	location	decision
1.	hilly	winter	kullu	Yes
2.	mountain	windy	Mumbai	No
3.	mountain	windy	Shimla	Yes
4.	beach	windy	Mumbai	No
5.	beach	warm	goa	Yes
6.	beach	windy	goa	No
7.	beach	warm	Shimla	Yes

Subset – 1

s.no	place type	weather	location	decision
1.	hilly	winter	kullu	Yes
2.	mountain	windy	Shimla	Yes
3.	beach	warm	goa	Yes
4.	beach	warm	Shimla	Yes

Subset – 2

s.no	place type	weather	location	decision
5.	mountain	windy	Mumbai	No
6.	beach	windy	Mumbai	No
7.	beach	windy	goa	No

- **At iteration 1** rows 3 & 4 column weather is selected and rows 3 & 4 are marked. the rule is added to R IF the weather is warm then a decision is yes.
- **At iteration 2** row 1 column place type is selected and row 1 is marked. the rule is added to R IF the place type is hilly then the decision is yes.
- **At iteration 3** row 2 column location is selected and row 2 is marked. the rule is added to R IF the location is Shimla then the decision is yes.
- **At iteration 4** row 5&6 column location is selected and row 5&6 are marked. the rule is added to R IF the location is Mumbai then a decision is no.
- **At iteration 5** row 7 column place type & the weather is selected and row 7 is marked. the rule is added to R IF the place type is beach AND the weather is windy then the decision is no.

- **Finally, we get the rule set:- Rule Set**
- **Rule 1:** IF the weather is warm THEN the decision is yes.
- **Rule 2:** IF the place type is hilly THEN the decision is yes.
- **Rule 3:** IF the location is Shimla THEN the decision is yes.
- **Rule 4:** IF the location is Mumbai THEN the decision is no.
- **Rule 5:** IF the place type is beach AND the weather is windy THEN the decision is no.

Advantages

- Because inductive learning models are flexible and adaptive, they are well suited for handling difficult, complex, and dynamic information.
- Finding hidden patterns and relationships in data: Inductive learning models are ideally suited for tasks like pattern recognition and classification because they can identify links and patterns in data that may not be immediately apparent to humans.
- Huge datasets – Inductive learning models are suitable for applications requiring the processing of massive quantities of data because they can efficiently handle enormous volumes of data.
- Appropriate for situations where the rules are ambiguous – Since inductive learning models may learn from examples without explicit programming, they are suitable for situations when the rules are not precisely described or understood beforehand.

Disadvantages

- May overfit to particular data – Inductive learning models that have overfit to specific training data, or that have learned the noise in the data rather than the underlying patterns, may perform badly on fresh data.
- computationally costly possible – The employment of inductive learning models in real-time applications may be constrained by their computationally costly nature, especially for complex datasets.
- Limited interpretability – Inductive learning models may be difficult to understand, making it difficult to understand how they arrive at their predictions, in applications where the decision-making process must be transparent and explicable.
- Inductive learning models are only as good as the data they are trained on, therefore if the data is inaccurate or inadequate, the model may not perform effectively.

Applications of Inductive Reasoning in AI

- Inductive reasoning is utilized in numerous AI applications, enhancing their efficiency and effectiveness:
- **Spam Filtering:** In email spam filtering, inductive learning algorithms analyze large volumes of labeled data (spam and non-spam emails) to identify patterns and features indicative of spam. These algorithms learn from the examples to classify incoming emails as either spam or non-spam, even if they have not encountered those specific emails before.
- **Image Classification:** Inductive reasoning is widely used in image classification tasks. Machine learning models can learn to recognize patterns and features associated with specific object classes by training on a labeled image dataset. For instance, we can train an AI system on thousands of images of cats to learn the common characteristics that define a cat, enabling it to classify unseen images as either cats or non-cats.
- **Natural Language Processing:** Inductive reasoning is crucial in natural language processing tasks like sentiment analysis or text classification. By analyzing large amounts of labeled text data, AI models can identify patterns in the text that indicate sentiment or classify documents into different categories. These models can then generalize from the observed patterns to classify new, unseen text.

- **Credit Risk Assessment:** The goal of credit risk assessment is to **predict the probability** that a borrower will be incapable of repaying a loan (or other kind of credit). By looking at a borrower's financial records and other important factors, lenders can use credit risk assessment to make knowledgeable choices about whether to offer credit and on what conditions.
- **Disease Diagnosis**
- With the help of inductive learning algorithms, doctors and other medical professionals can use data to make more accurate and **timely diagnoses of various diseases**.
- One of the key benefits of inductive learning in disease diagnosis is the ability to analyze large datasets of medical information, including **patient histories, lab results, and imaging data**. Inductive learning algorithms can find patterns and trends by examining these big datasets that could be challenging for medical professionals to spot on their own.

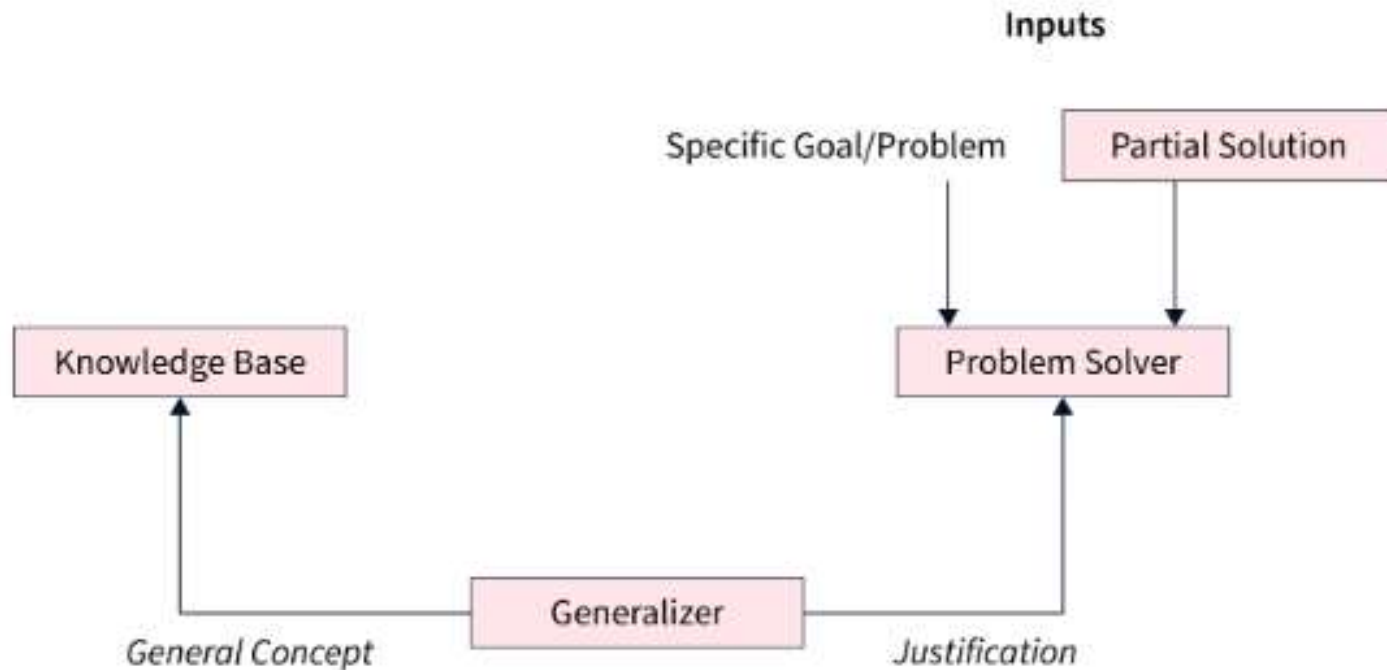
Face Recognition

- Face recognition seeks to recognize people based on their **facial features**, which can be helpful for various applications, from **social media and marketing** to **security and surveillance**.
- We can train inductive learning algorithms on large facial picture data datasets to discover patterns and **pinpoint distinctive traits for every person**. Inductive learning algorithms can properly identify people with a high degree of accuracy by analyzing face traits such as the **distance between the eyes, the curve of the nose, and the curvature of the lips**.

Explanation-based learning

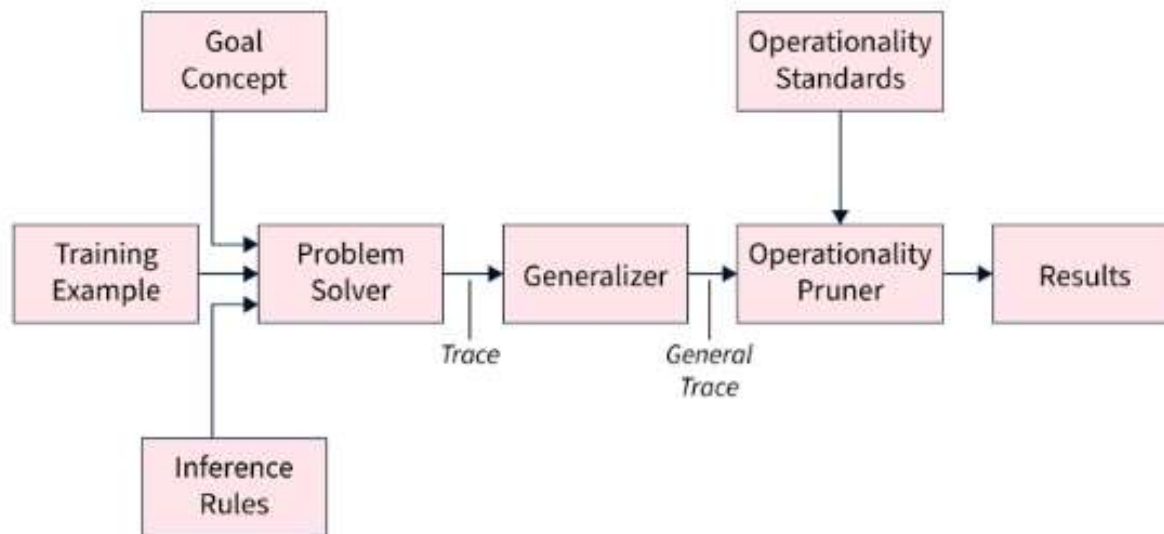
- Explanation-based learning in artificial intelligence is a problem-solving method that involves agent learning by analyzing specific situations and connecting them to previously acquired information.
- Also, the agent applies what he has learned to solve similar issues.
- Rather than relying solely on statistical analysis, EBL algorithms incorporate logical reasoning and domain knowledge to make predictions and identify patterns

- **Explanation-based learning architecture:**



- The environment provides two inputs to the EBL architecture:
- **A specific goal, and**
- **A partial solution.**
- The problem solver analyses these sources and provides reasoning to the generalizer.
- The generalizer uses general ideas from the knowledge base as input and compares them to the problem solver's reasoning to come up with an answer to the given problem.

Explanation-based learning System Representation:



- **Problem Solver:** It takes 3 kinds of external inputs: The goal idea is a complex problem statement that the agent must learn. Training instances are facts that illustrate a specific instance of a target idea. Inference rules reflect facts and procedures that demonstrate what the learner already understands.
- **Generalizer:** The problem solver's output is fed into the generalizer, which compares the problem solver's explanation to the knowledge base and outputs to the operational pruner.
- **Operational pruner:** It takes two inputs, one from generalized and the other from operationally standard. The operational standard describes the final concept and defines the format in which the learned concept should be conveyed.

a) The Explanation-Based Learning Hypothesis

- According to the Explanation based learning hypothesis, if a system has an explanation for how to tackle a comparable problem it faced previously, it will utilize that explanation to handle the current problem more efficiently. This hypothesis is founded on the concept that learning via explanations is more successful than learning through instances alone.

b) Standard Approach to Explanation-Based Learning

- The typical approach to explanation-based learning in artificial intelligence entails the following steps:
- Determine the problem to be solved
- Gather samples of previously solved problems that are comparable to the current problem.
- Identify the connections between the previously solved problems and the new problem.
- Extraction of the underlying principles and rules used to solve previously solved problems.
- Apply the extracted rules and principles to solve the new problem.

c) Examples of Explanation-Based Learning

- **Medical Diagnosis:** Explanation-based learning can be used in medical diagnosis to determine the underlying causes of a patient's symptoms. Explanation-based learning algorithms can find trends and produce more accurate diagnoses by analyzing previously diagnosed instances.
- **Robot Navigation:** Explanation-based learning may be used to educate robots on how to navigate through complicated settings. Explanation-based learning algorithms can discover the rules and principles that were utilized to navigate those settings and apply them to new scenarios by analyzing prior successful navigation efforts.
- **Fraud Detection:** Explanation-based learning may be utilized in fraud detection to discover patterns of fraudulent conduct. Explanation-based learning algorithms can find the rules and principles that were utilized to detect prior cases of fraud and apply them to new cases by analyzing previous incidents of fraud.

Advantages of Explanation-Based Learning

- **Efficiency in Learning:** EBL can learn effectively from a single example, making it efficient in situations where data is scarce or expensive to obtain.
- **Understanding and Generalization:** EBL focuses on understanding the rationale behind examples, leading to more robust generalizations that can be applied to a wide range of situations.
- **Interpretable Models:** The rules or concepts learned through EBL are often more interpretable than those learned through other methods, making it easier to understand and trust the system's decisions

Challenges and Limitations

- **Dependency on Domain Knowledge:** EBL relies heavily on accurate and comprehensive domain knowledge. If the domain knowledge is incomplete or incorrect, the system may generate flawed explanations and generalizations.
- **Limited to Well-Defined Problems:** EBL is most effective in well-defined problem domains where the rules and relationships are clear. It may struggle in more complex or ambiguous domains.
- **Complexity of Explanation Generation:** Generating explanations can be computationally intensive, especially in domains with complex relationships and a large number of features.

Statistical Learning methods

- Statistical learning is using data to make predictions or identify patterns.
- It uses statistical algorithms to analyze and model relationships between variables in a dataset. These algorithms can perform tasks such as classification, regression, and clustering.
- Statistical learning refers to a set of methods and techniques used in machine learning and statistics to analyze and make predictions or decisions based on data.
- It involves developing models that can uncover patterns, relationships, and trends within datasets, allowing for the extraction of valuable insights and the creation of predictive models.

Here are key concepts associated with statistical learning:

1. Supervised Learning:

- Definition: In supervised learning, the algorithm is trained on a labeled dataset, where both the input data and the corresponding desired output (target) are provided. The goal is to learn a mapping or relationship between the input features and the target variable.
- Examples: Regression (predicting a continuous outcome) and classification (predicting a categorical outcome) are common types of supervised learning tasks.

2. Unsupervised Learning:

- Definition: Unsupervised learning involves working with unlabeled data, where the algorithm explores the inherent structure or patterns in the input features without explicit guidance on the output. Clustering and dimensionality reduction are examples of unsupervised learning tasks.
- Examples: K-means clustering, hierarchical clustering, and principal component analysis (PCA) are unsupervised learning techniques.

3. Statistical Models:

- Definition: Statistical models are mathematical representations that describe the relationships between variables in a dataset. These models can be simple, such as linear regression, or complex, such as ensemble methods or deep neural networks.
- Examples: Linear regression, logistic regression, decision trees, support vector machines, and neural networks are common statistical models used in learning algorithms.

4. Training and Testing:

- Definition: Statistical learning models are trained on a subset of the data, and their performance is assessed on another subset not seen during training. This helps evaluate how well the model generalizes to new, unseen data.
- Examples: The dataset is typically split into a training set and a testing set, or cross-validation techniques are used to assess the model's performance.

5. Bias-Variance Tradeoff:

- Definition: The bias-variance tradeoff is a key concept in statistical learning that refers to the balance between a model's ability to capture the underlying patterns in data (low bias) and its sensitivity to variations in the training data (low variance).
- Examples: Models with high complexity may exhibit low bias but high variance, and vice versa. Finding the right balance is essential for optimal model performance.

- 6. Feature Engineering:
- Definition: Feature engineering involves selecting, transforming, or creating relevant features from the input data to improve the model's performance. It plays a crucial role in the effectiveness of statistical learning models.
- Examples: Feature scaling, one-hot encoding, and creating interaction terms are common feature engineering techniques.

- Statistical learning techniques are widely applied in various domains, including finance, healthcare, marketing, and natural language processing.
- These methods enable computers to learn from data, make predictions, and uncover patterns that may not be apparent through traditional programming approaches.

Reinforcement Learning

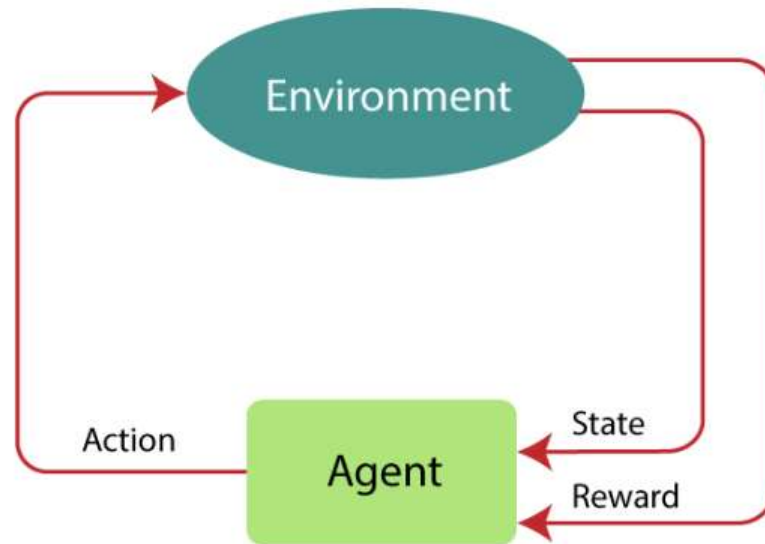
Reinforcement Learning

- Reinforcement Learning is a feedback-based Machine learning technique in which an agent learns to behave in an environment by performing the actions and seeing the results of actions. For each good action, the agent gets positive feedback, and for each bad action, the agent gets negative feedback or penalty.
- In Reinforcement Learning, the agent learns automatically using feedbacks without any labeled data

- Reinforcement learning essentially consists of the relationship between an agent, environment, and goal. Literature widely formulates this relationship in terms of the Markov decision process (MDP).

Markov decision process

- The reinforcement learning agent learns about a problem by interacting with its environment. The environment provides information on its current state. The agent then uses that information to determine which actions(s) to take. If that action obtains a reward signal from the surrounding environment, the agent is encouraged to take that action again when in a similar future state. This process repeats for every new state thereafter. Over time, the agent learns from rewards and punishments to take actions within the environment that meet a specified goal.



Key Concepts of Reinforcement Learning

- **Agent:** The learner or decision-maker.
- **Environment:** Everything the agent interacts with.
- **State:** A specific situation in which the agent finds itself.
- **Action:** All possible moves the agent can make.
- **Reward:** Feedback from the environment based on the action taken.

- RL operates on the principle of learning optimal behavior through trial and error. The agent takes actions within the environment, receives rewards or penalties, and adjusts its behavior to maximize the cumulative reward. This learning process is characterized by the following elements:
- **Policy:** A strategy used by the agent to determine the next action based on the current state.
- **Reward Function:** A function that provides a scalar feedback signal based on the state and action.
- **Value Function:** A function that estimates the expected cumulative reward from a given state.
- **Model of the Environment:** A representation of the environment that helps in planning by predicting future states and rewards.

- **Types of Reinforcement:**

- **Positive:** Positive Reinforcement is defined as when an event, occurs due to a particular behavior, increases the strength and the frequency of the behavior. In other words, it has a positive effect on behavior.

Advantages of reinforcement learning are:

- Maximizes Performance
- Sustain Change for a long period of time
- Too much Reinforcement can lead to an overload of states which can diminish the results

- **Negative:** Negative Reinforcement is defined as strengthening of behavior because a negative condition is stopped or avoided.

Advantages of reinforcement learning:

- Increases Behavior
- Provide defiance to a minimum standard of performance
- It Only provides enough to meet up the minimum behavior

Elements of Reinforcement Learning

- **i) Policy:** Defines the agent's behavior at a given time.
- **ii) Reward Function:** Defines the goal of the RL problem by providing feedback.
- **iii) Value Function:** Estimates long-term rewards from a state.
- **iv) Model of the Environment:** Helps in predicting future states and rewards for planning.

Application of Reinforcement Learnings

- **i) Robotics:** Automating tasks in structured environments like manufacturing.
- **ii) Game Playing:** Developing strategies in complex games like chess.
- **iii) Industrial Control:** Real-time adjustments in operations like refinery controls.
- **iv) Personalized Training Systems:** Customizing instruction based on individual needs.

- **Advantages:**

- 1. Reinforcement learning can be used to solve very complex problems that cannot be solved by conventional techniques.
- 2. The model can correct the errors that occurred during the training process.
- 3. In RL, training data is obtained via the direct interaction of the agent with the environment
- 4. Reinforcement learning can handle environments that are non-deterministic, meaning that the outcomes of actions are not always predictable. This is useful in real-world applications where the environment may change over time or is uncertain.
- 5. Reinforcement learning can be used to solve a wide range of problems, including those that involve decision making, control, and optimization.
- 6. Reinforcement learning is a flexible approach that can be combined with other machine learning techniques, such as deep learning, to improve performance.

Disadvantages:

- 1. Reinforcement learning is not preferable to use for solving simple problems.
- 2. Reinforcement learning needs a lot of data and a lot of computation
- 3. Reinforcement learning is highly dependent on the quality of the reward function. If the reward function is poorly designed, the agent may not learn the desired behavior.
- 4. Reinforcement learning can be difficult to debug and interpret. It is not always clear why the agent is behaving in a certain way, which can make it difficult to diagnose and fix problems.

Unit 5

Expert System

- An expert system is a computer program that is designed to solve complex problems and to provide decision-making ability like a human expert.
- It performs this by extracting knowledge from its knowledge base using the reasoning and inference rules according to the user queries.
- The expert system is a part of AI, and the first ES was developed in the year 1970, which was the first successful approach of artificial intelligence.

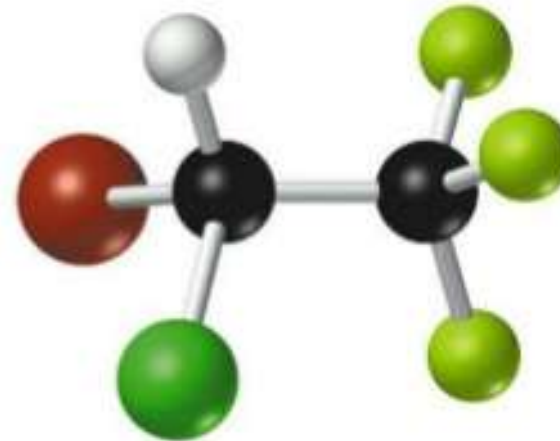
- Expert System solves the most complex issue as an expert by extracting the knowledge stored in its knowledge base.
- The system helps in decision making for complex problems using **both facts and heuristics like a human expert**.
- It is called so because it contains the expert knowledge of a specific domain and can solve any complex problem of that particular domain.
- These systems are designed for a specific domain, such as **medicine, science, etc.**

Applications of Expert Systems



LITHIAN: Gives advice to archaeologists examining stone tools

DENDRAL: Used to identify the structure of chemical compounds.
First used in 1965



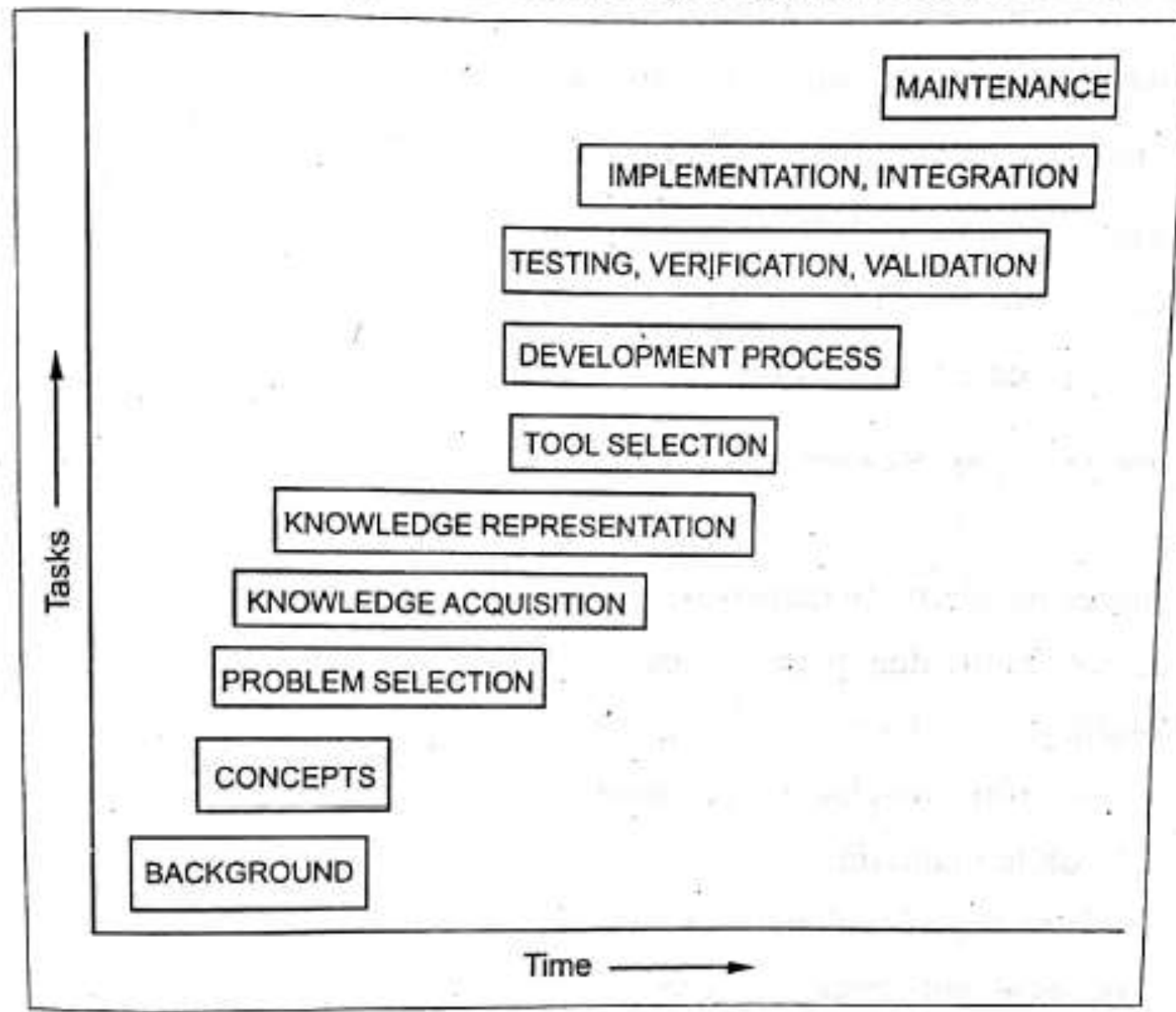
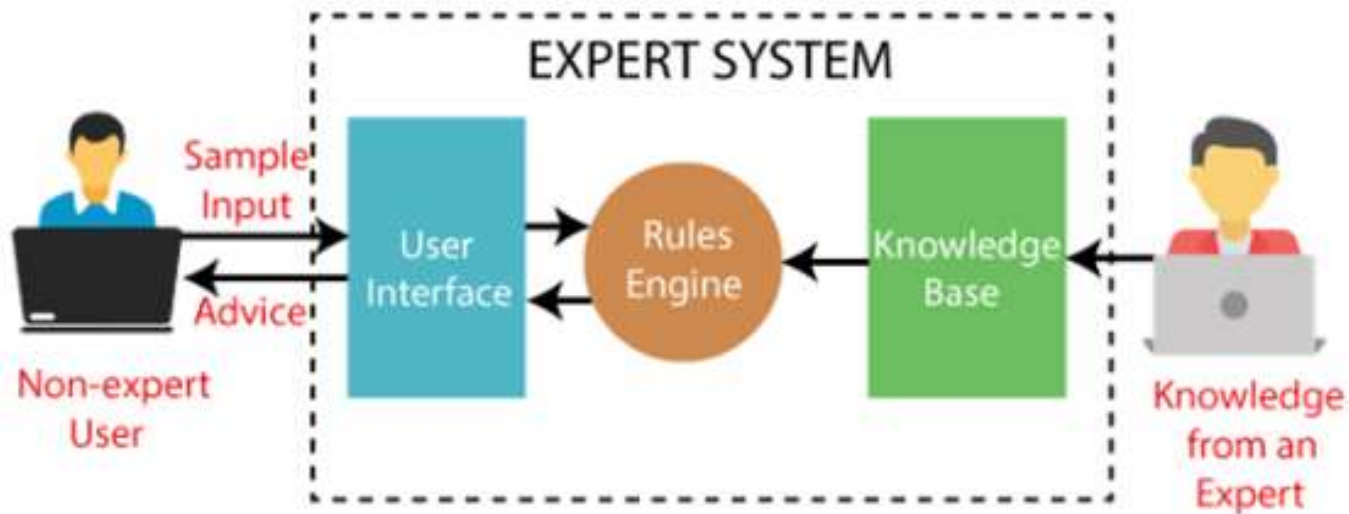


Fig. 5.1. Hierarchy of expert systems development process

- The performance of an expert system is based on the expert's knowledge stored in its knowledge base.
- The more knowledge stored in the KB, the more that system improves its performance.
- One of the common examples of an ES is a suggestion of spelling errors while typing in the Google search box.
- Below is the block diagram that represents the working of an expert system:



Characteristics of an Expert System

An expert system is usually designed to have the following general characteristics.

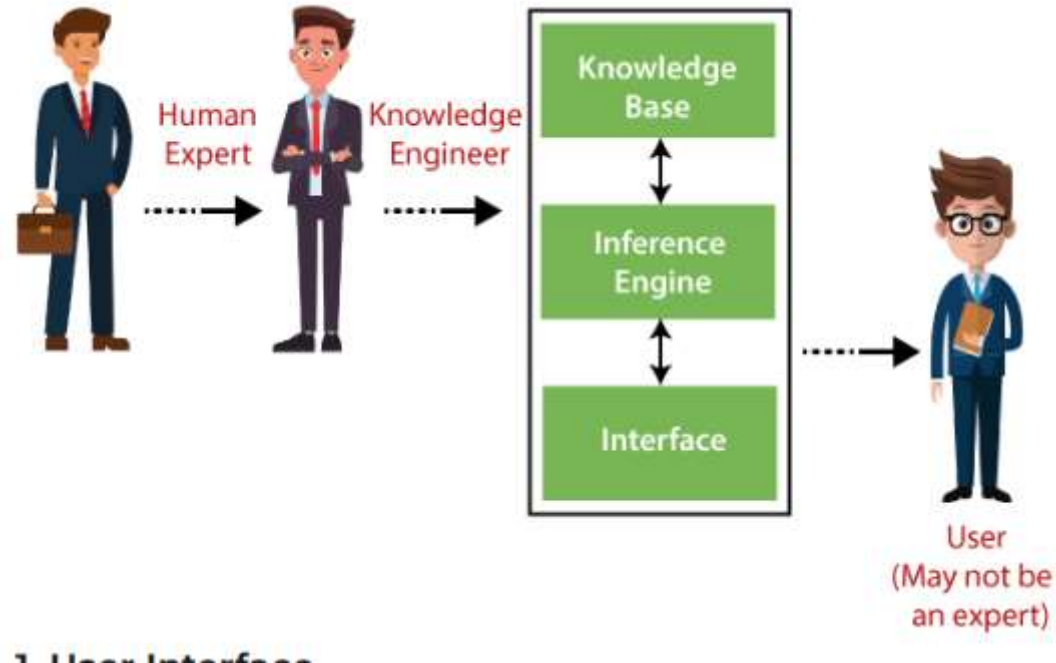
- **1. High level Performance:** The system must be capable of responding at a level of competency equal to or better than an expert system in the field. The quality of the advice given by the system should be in a high level integrity and for which the performance ratio should be also very high.
- **2. Domain Specificity:** Expert systems are typically very domain specific. For ex., a diagnostic expert system for troubleshooting computers must actually perform all the necessary data manipulation as a human expert would. The developer of such a system must limit his or her scope of the system to just what is needed to solve the target problem. Special tools or programming languages are often needed to accomplish the specific objectives of the system.
- **3. Good Reliability:** The expert system must be as reliable as a human expert.

- **4. Understandable:** The system should be understandable i.e. be able to explain the steps of reasoning while executing. The expert system should have an explanation capability similar to the reasoning ability of human experts.
- **5. Adequate Response time:** The system should be designed in such a way that it is able to perform within a small amount of time, comparable to or better than the time taken by a human expert to reach at a decision point. An expert system that takes a year to reach a decision compared to a human expert's time of one hour would not be useful.
- **6. Use symbolic representations:** Expert system use symbolic representations for knowledge (rules, networks or frames) and perform their inference through symbolic computations that closely resemble manipulations of natural language.
- **7. Linked with Metaknowledge:** Expert systems often reason with metaknowledge i.e. they reason with knowledge about themselves and their own knowledge limits and capabilities. The use of metaknowledge is quite interactive and simple for various data representations.

- **8. Expertise knowledge:** Real experts not only produce good solutions but also find them quickly. So, an expert system must be skillful in applying its knowledge to produce solutions both efficiently and effectively by using the intelligence human experts.
- **9. Justified Reasoning:** This allows the users to ask the expert system to justify the solution or advice provided by it. Normally, expert systems justify their answers or advice by explaining their reasoning. If a system is a rule based system, it provides to the user all the rules and facts it has used to achieve its answer.
- **10. Explaining capability:** Expert systems are capable of explaining how a particular conclusion was reached and why requested information is needed during a consultation. This is very important as it gives the user a chance to access and understand the system's reasoning ability, thereby improving the user's confidence in the system.
- **11. Special Programming Languages:** Expert systems are typically written in special programming languages. The use of languages like LISP and PROLOG in the development of an expert system simplifies the coding process. The major advantage of these languages, as compared to conventional programming languages is the simplicity of the addition, elimination or substitution of new rules and memory management capabilities.

Components of Expert System

- An expert system mainly consists of three components:
- **User Interface**
- **Inference Engine**
- **Knowledge Base**



1. User Interface

- With the help of a user interface, the expert system interacts with the user, takes queries as an input in a readable format, and passes it to the inference engine.
- After getting the response from the inference engine, it displays the output to the user.
- In other words, **it is an interface that helps a non-expert user to communicate with the expert system to find a solution.**

2. Inference Engine(Rules of Engine)

- The inference engine is known as the brain of the expert system as it is the main processing unit of the system.
- It applies inference rules to the knowledge base to derive a conclusion or deduce new information.
- It helps in deriving an error-free solution of queries asked by the user.
- With the help of an inference engine, the system extracts the knowledge from the knowledge base.

There are two types of inference engine:

- **Deterministic Inference engine:** The conclusions drawn from this type of inference engine are assumed to be true. It is based on **facts** and **rules**.
- **Probabilistic Inference engine:** This type of inference engine contains uncertainty in conclusions, and based on the probability.

Inference engine uses the below modes to derive the solutions:

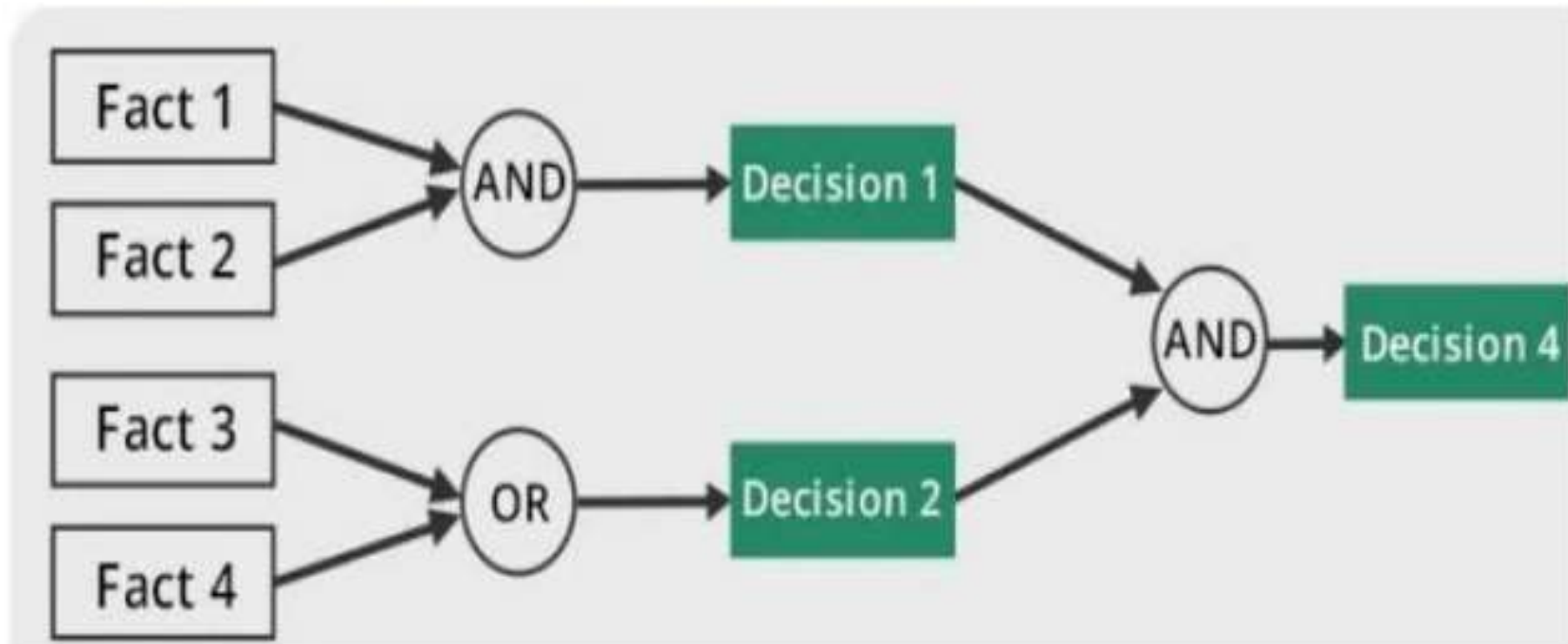
- **Forward Chaining:** It starts from the known facts and rules, and applies the inference rules to add their conclusion to the known facts.
- **Backward Chaining:** It is a backward reasoning method that starts from the goal and works backward to prove the known facts.

Forward Chaining

It is a strategy of an expert system to answer the question, **"What can happen next?"**

Here, the interface engine follows the chain of conditions and derivations and finally deduces the outcome. It considers all the facts and rules, and sorts them before concluding to a solution.

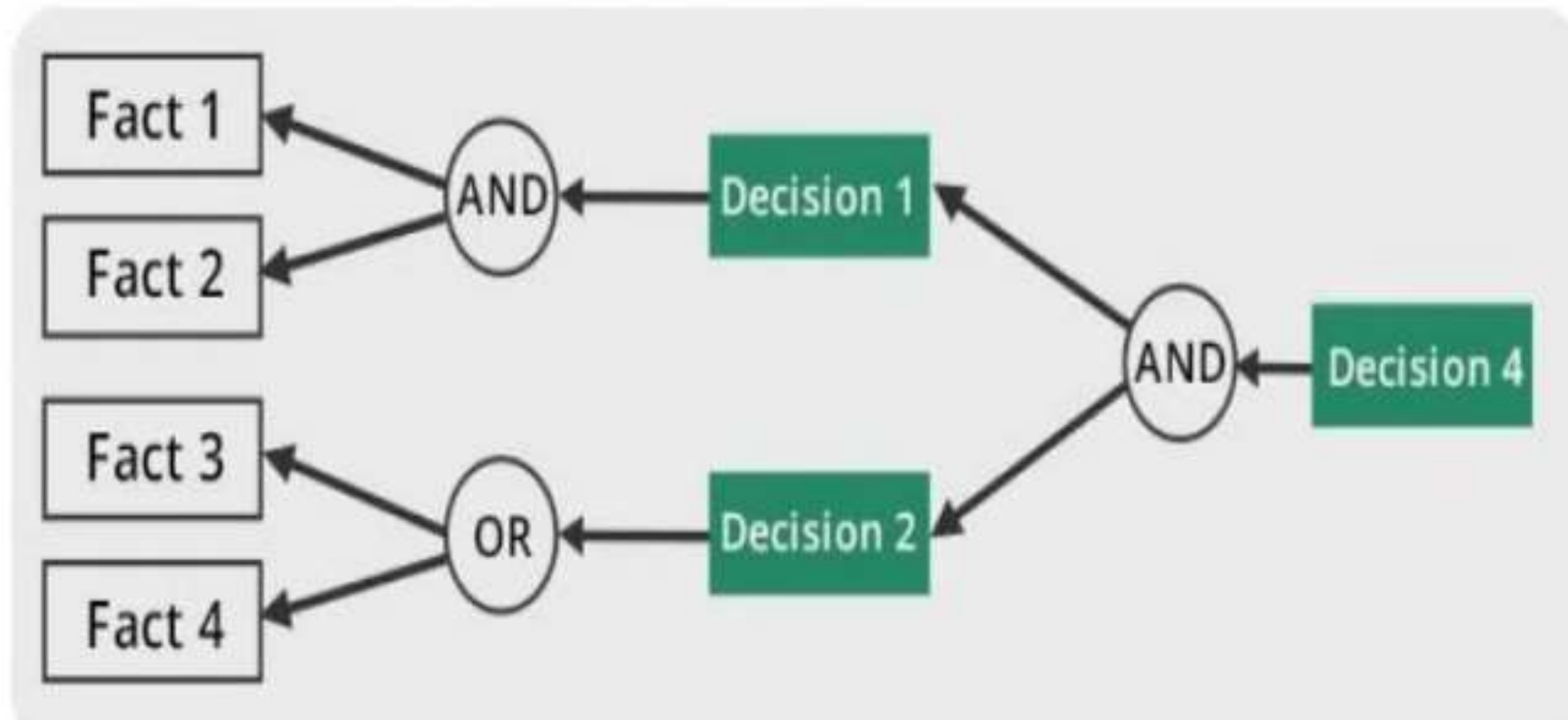
This strategy is followed for working on conclusion, result, or effect. For example, prediction of share market status as an effect of changes in interest rates.



Backward Chaining

With this strategy, an expert system finds out the answer to the question, **“Why this happened?”**

On the basis of what has already happened, the interface engine tries to find out which conditions could have happened in the past for this result. This strategy is followed for finding out cause or reason. For example, diagnosis of blood cancer in humans.



3. Knowledge Base

- The knowledgebase is a type of storage that stores knowledge acquired from the different experts of the particular domain.
- It is considered as big storage of knowledge.
- The more the knowledge base, the more precise will be the Expert System.
- It is similar to a database that contains information and rules of a particular domain or subject.
- One can also view the knowledge base as collections of objects and their attributes.
- Such as a Lion is an object and its attributes are it is a mammal, it is not a domestic animal, etc

Components of Knowledge Base

- **Factual Knowledge:** The knowledge which is based on facts and accepted by knowledge engineers comes under factual knowledge.
- **Heuristic Knowledge:** This knowledge is based on practice, the ability to guess, evaluation, and experiences.
- **Knowledge Representation:** It is used to formalize the knowledge stored in the knowledge base using the If-else rules.
- **Knowledge Acquisitions:** It is the process of extracting, organizing, and structuring the domain knowledge, specifying the rules to acquire the knowledge from various experts, and store that knowledge into the knowledge base.

5.2. ARCHITECTURE OF EXPERT SYSTEM

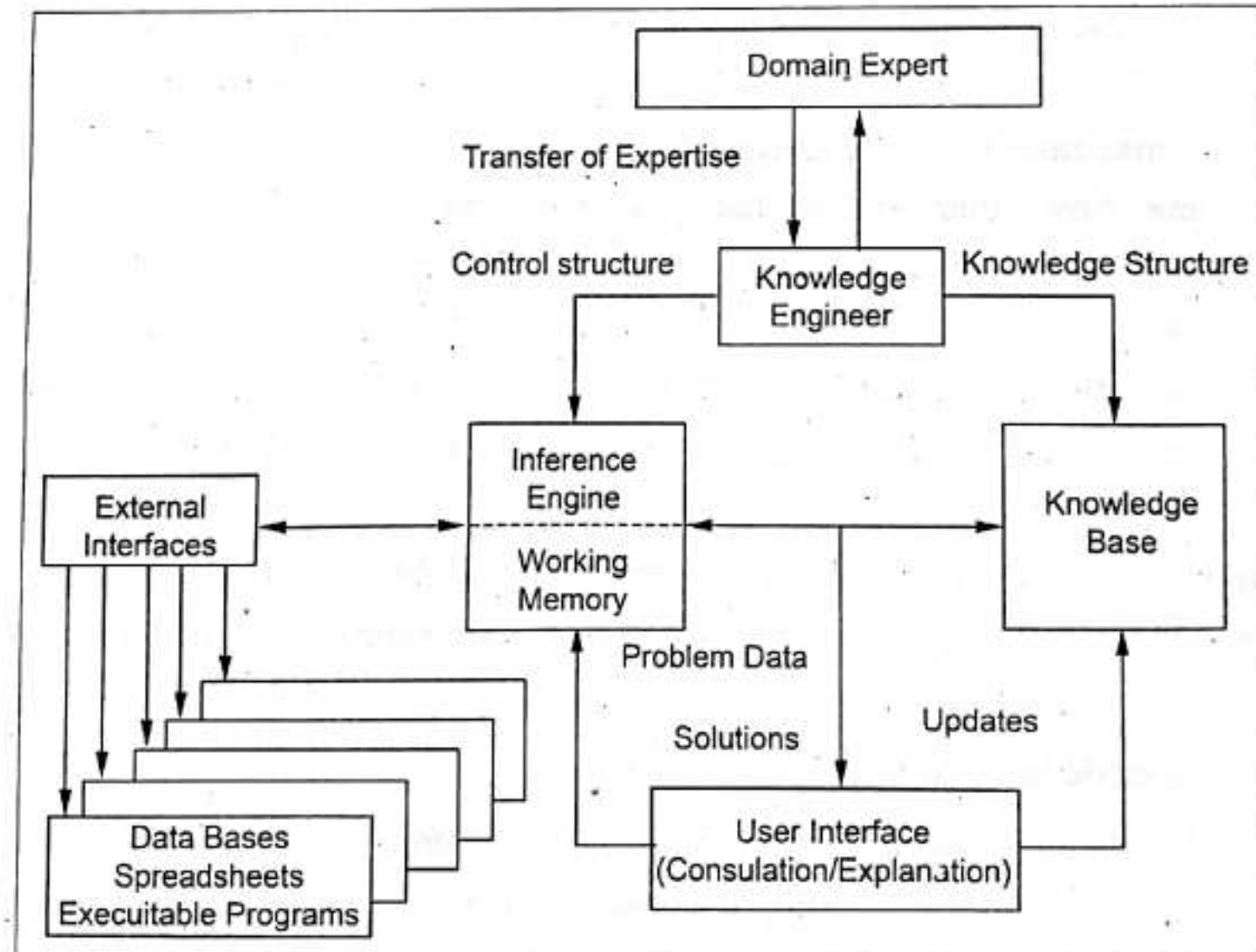
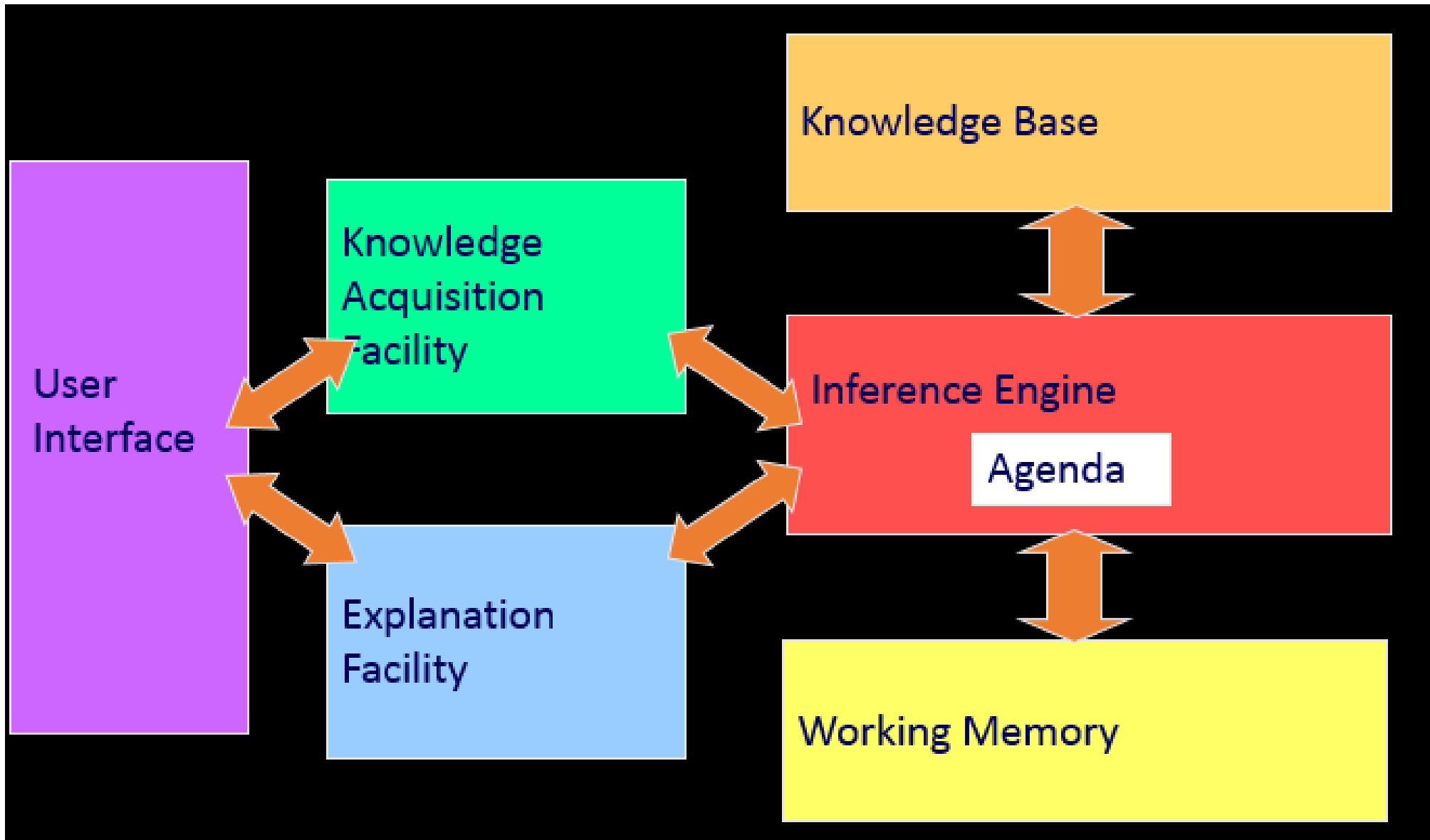


Fig. 5.4. Expert System Components



- **Knowledge-Base / Rule-Base** store expert knowledge as **condition-action-rules** (aka: **if-then-** or **premise-consequence-rules**)
- **Working Memory** stores **initial facts** and **generated facts** derived by inference engine; maybe with additional parameters like the “degree of trust” into the truth of a fact $\cong$ **certainty factor**
- **Inference Engine**
 - matches condition-part of rules against facts stored in Working Memory (pattern matching);
 - rules with satisfied condition are active rules and are placed on the agenda;
 - among the active rules on the agenda, one is selected (see conflict resolution, priorities of rules) as next rule for execution (“firing”) – consequence of rule is added as new fact(s) to Working Memory

- **Inference Engine + additional components** might be necessary for other functions, like
 - calculation of **certainty values**,
 - determining **priorities** of rules,
 - **conflict resolution** mechanisms,
 - a **truth maintenance system (TMS)** if reasoning with **defaults** and **beliefs** is requested
- **Explanation** Facility provides justification of solution to user (reasoning chain)
- **Knowledge Acquisition** Facility helps to integrate new knowledge; also automated knowledge acquisition
- **User Interface** allows user to interact with the XPS - insert facts, query the system, solution presentation

knowledge is encoded as **IF ... THEN** rules

- Condition-action pairs

the inference engine determines which rule antecedents (condition-part) are satisfied

- the left-hand condition-part must “match” facts in the working memory

matching rules are “activated”, i.e. placed on the agenda

rules on the agenda can be executed (“fired”)

- an activated rule may generate new facts and/or cause actions through its right-hand side (action-part)
- the activation of a rule may thus cause the activation of other rules through added facts based on the right-hand side of the fired rule

- **Domain expert** : The individuals who currently are experts in solving the problems; here the system is intended to solve.
- **Knowledge engineer** : The individual who encodes the expert's knowledge in a declarative form that can be used by the expert system.
- **User** : The individual who will be consulting with the system to get advice which would have been provided by the expert.
- **System engineer** : builds the user interface, designs the declarative format of the knowledge base, and implements the inference engine.
- **User interface** : The code that controls the dialog between the user and the system.
- **Inference engine** : The code at the core of the system which derives recommendations from the knowledge base and problem specific data in working storage.
- **Knowledge base** : A declarative representation of the expertise often in IF THEN rules .

Applications of Expert Systems

- **Medical Diagnosis:** Expert systems assist doctors by analyzing symptoms and medical history to suggest possible diagnoses or treatment options. For example, MYCIN, an early expert system, helped identify bacterial infections and recommend antibiotics.
- **Financial Services:** In finance, expert systems are used for credit scoring, fraud detection, and investment advice. They analyze financial data and patterns to make informed decisions.
- **Technical Support:** Expert systems can troubleshoot and provide solutions for technical issues. They guide users through problem-solving steps based on pre-defined rules and knowledge.
- **Manufacturing:** In manufacturing, expert systems help optimize production processes, perform quality control, and manage inventory by analyzing data and making recommendations.

Expert systems have many roles, including:

- **Advising:** Expert systems can advise, instruct, and assist humans in making decisions.
- **Diagnosing:** Expert systems can diagnose problems and provide solutions.
- **Predicting:** Expert systems can predict results and forecast crop damage.
- **Troubleshooting:** Expert systems can troubleshoot complex electromechanical machinery and other problems.
- **Optimizing:** Expert systems can optimize supply chains and processes.
- **Assessing:** Expert systems can assess dam safety and perform financial risk assessments.
- **Recommending:** Expert systems can provide personalized recommendations and evidence-based treatment recommendations.
- **Routing:** Expert systems can route customer requests.
- **Scheduling:** Expert systems can schedule orders and bus and train schedules.
- **Maintaining:** Expert systems can maintain existing networks and perform predictive maintenance.

KNOWLEDGE ACQUISITION

- Knowledge acquisition is the process of extracting knowledge (facts, procedures, rules) from human experts, books, documents, sensors or computer files and converting it into a form that can be stored and manipulated by the computer for purposes of problem solving. It occurs throughout the entire development process.
- Knowledge Acquisition Process
 - **i). Identification:** Identify the problem including data, criteria for solutions to meet, available resources, etc.
 - **ii). Conceptualization:** Determine the key concepts and relationships by characterizing the data, flow of information, the domain structure, etc.
 - **iii). Formalization:** Understand the underlying search space, uncertainty issues, etc.
 - **iv). Implementation:** Translate acquired knowledge into the program.
 - **v). Testing:** Validate and verify.

- **Knowledge Elicitation:** It implies that knowledge acquisition is accomplished from a human expert.
- **Sources of Knowledge :** The Knowledge Engineer acquires knowledge from a number of sources:
 - Written sources, such as books, manuals, and standard procedures. This is an important source of knowledge at the start, when the knowledge engineer learns the vocabularies and general topics of the domain.
 - Experts
 - Observations of the actual process
 - Expertise of a procedural nature, in which a sequence of tasks is performed, direct observation of the operation also can provide input to the knowledge base.
 - Experts make decisions based on an abstract and intellectual process that even experts themselves have difficulty understanding and verbalizing.

Knowledge Acquisition Methods

- Knowledge acquisition methods constitute tools used for the process of modeling knowledge.
- There are two basic strategies in knowledge engineering:
 - i) Starting from general and overall concepts, gradually leading the expert to elicit details of a topic.
 - ii) Starting from the details of specific cases and helping the expert establish and derive general concepts from the specific examples.
- Therefore, the methods of eliciting knowledge can be categorized into two broad groups:
 - Top-down (or deductive) methods
 - Bottom-up (or inductive) methods

Categories of Knowledge Acquisition Methods

Top-Down Methods

- The knowledge engineer organizes the acquisition sessions for discovering general concepts, rules, and objects, and then gradually goes into the details of each concept, rule, or object.

This can be grouped into four categories namely:

- i) Questioning methods
- ii) Object-oriented methods
- iii) Quantitative methods
- iv) Inventive methods

Bottom-Up Methods

- The knowledge engineer focuses the expert's attention on specific cases. This helps the expert abstract the decision for resolving a specific case to a more generalized rule or concept.

The methods that can be grouped in this approach are

- Example-based methods.
- Protocol analysis.
- Observation of the expert's decision-making process

Types of Knowledge in AI

1. Declarative Knowledge

- Declarative knowledge refers to facts and information that describe the world, answering the "what" type of questions.
- **Example:** *Knowing that Paris is the capital of France.*
- This knowledge is often stored in databases or knowledge bases and expressed in logical statements, forming the foundation for more complex reasoning and problem-solving in AI systems.

2. Procedural Knowledge

- Procedural knowledge is the knowledge of how to perform tasks or processes, answering the "how" type of questions.
- **Example:** *Steps to solve a mathematical problem or the procedure to start a car.*
- This knowledge is embedded in algorithms or control structures, enabling AI systems to execute tasks, perform actions, and solve problems step-by-step.

3. Meta-Knowledge

- Meta-knowledge is ‘knowledge about knowledge’, understanding which types of knowledge to apply in different situations.
- ***Example:*** *Knowing when to use a specific algorithm based on the problem at hand.*
- Crucial for systems that need to adapt or optimize their performance, meta-knowledge helps in selecting the most appropriate strategy or knowledge base for a given problem.

4. Heuristic Knowledge

- Heuristic knowledge includes rules of thumb, educated guesses, and intuitive judgments derived from experience.
- **Example:** *Using an educated guess to approximate a solution when time is limited.*
- Often used in problem-solving and decision-making processes where exact solutions are not feasible, helping AI systems to arrive at good-enough solutions quickly.

5. Structural Knowledge

- Structural knowledge refers to the understanding of how different pieces of knowledge are organized and related to each other.
- **Example:** *Understanding the hierarchy of concepts in a taxonomy or the relationships between different entities in a semantic network.*
- This knowledge is essential for organizing information within AI systems, allowing for efficient retrieval, reasoning, and inferencing based on the relationships and structures defined.

Meta-knowledge

- Meta-knowledge in AI refers to knowledge about knowledge itself.
- meta-knowledge can include understanding how knowledge is acquired, how it can be represented, and how it can be manipulated or refined.
- In AI systems, this can involve using higher-level reasoning to guide the learning process, determine the relevance of information, or even reflect on the system's own reasoning process.
- For example, a machine learning model could use meta-knowledge to determine which learning strategy to use, given the particular task and data at hand, or to decide when to seek additional information from external sources.

- In expert systems, meta-knowledge is "knowledge about knowledge" that allows the system to reason about its own knowledge base and control its reasoning process, rather than just storing facts and rules about a specific domain.
- It helps the expert system to decide when to use certain knowledge, evaluate the reliability of information, select the best problem-solving approach, and even learn from its experiences.
- Examples include knowing the order of applying rules, the relevance of different pieces of information, or when it lacks enough information to solve a problem.

- In essence, meta-knowledge provides a way for an AI system to be more intelligent and adaptable by using knowledge about its own reasoning processes and the world around it to make more informed decisions.
- This is an ongoing area of research in AI and has significant implications for the development of more robust and autonomous AI systems.
- Meta-knowledge is a term used to refer to information that provides insight into the structure, organization, and characteristics of the data and knowledge contained within these systems.
- It is not specific to any particular domain or subject matter but rather focuses on the nature of knowledge itself.

Meta-Knowledge capabilities:

- **Guides Reasoning:**

- It helps the expert system determine which rules or facts are most relevant and when to apply them, leading to more intelligent and efficient problem-solving.

- **Evaluates Knowledge:**

- The system can assess the reliability, accuracy, and completeness of the knowledge it possesses, identifying inconsistencies or uncertainty.

- **Optimizes Decisions:**

- Meta-knowledge enables the system to select the most appropriate reasoning strategies for a given situation, improving decision-making.

- **Facilitates Learning:**

- By understanding its own knowledge gaps, the system can identify areas where it needs more information or how to improve its knowledge base.

- **Controls the Inference Process:**

- It provides mechanisms for the inference engine to control the order of operations and the application of rules, making the process more systematic.

- **Meta Knowledge applications in Expert Systems**

- **Rule Prioritization:**

- A system might have meta-rules like "Use less hazardous methods before more hazardous methods" or "Use cheap materials before expensive ones" to prioritize which domain-specific rules to fire.

- **Contextual Application:**

- Knowing that a dictionary is a good source for word meanings is a form of meta-knowledge that helps the system decide which knowledge source to consult.

- **Self-Awareness:**

- The system can recognize when it lacks sufficient information to provide a confident answer and then ask for more details from the user.

- **Knowledge Source Reliability:**

- An expert system could have meta-knowledge about the credibility of different information sources or the expertise level of the source of a particular piece of knowledge.

Heuristics

- In expert systems, **heuristics are "rules of thumb" or problem-solving strategies** derived from expert experience to find good-enough solutions quickly, especially when exact solutions are impossible or too computationally expensive.
- They function as a shortcut or estimate in a search space, guiding the system toward promising paths rather than exhaustively evaluating every possibility.
- Heuristics are crucial for efficient decision-making in complex domains, allowing expert systems to handle problems that would otherwise be intractable

Use of Heuristics in Expert Systems

Guided Search:

- Heuristics help an expert system navigate complex problems by estimating the potential "cost" or proximity to a goal state, focusing the search on the most viable paths.

Speed over Perfection:

- They prioritize finding a timely, workable solution over a perfect one, making them invaluable for rapid decision-making in emergency or dynamic situations.

Problem Decomposition:

- By providing informed estimates, heuristics can break down large, complicated problems into smaller, more manageable subproblems for the system to solve.

Advantages of Heuristics

Computational Cost:

- For many real-world problems, such as playing chess or solving the Traveling Salesperson Problem, the number of possible solutions is enormous.

Intractability:

- It would take an impractical amount of time and processing power to find an exact, optimal solution using brute-force methods.

Expert Knowledge Capture:

- Heuristics embed the experience and practical knowledge of human experts into the system's knowledge base, enabling it to mimic expert reasoning.

Application of Heuristics in Expert Systems

Game Playing:

- Heuristics can evaluate the strength of a chess board position to suggest the best move, rather than analyzing every single future move.

Diagnosis:

- In medical or technical diagnosis, heuristics might link certain symptoms or observed data to a likely problem class, quickly narrowing down potential causes.

Planning:

- Heuristic methods are used in planning systems to find efficient routes or sequences of actions, such as a route for a delivery vehicle to visit multiple cities.

Heuristics

•Pros	Cons
•Quick & easy •Allows decision-making that goes beyond our cognitive capacity •Allows for snap judgments when time is limited	Often inaccurate Can lead to systemic biases or errors in judgment

Typical expert systems – MYCIN

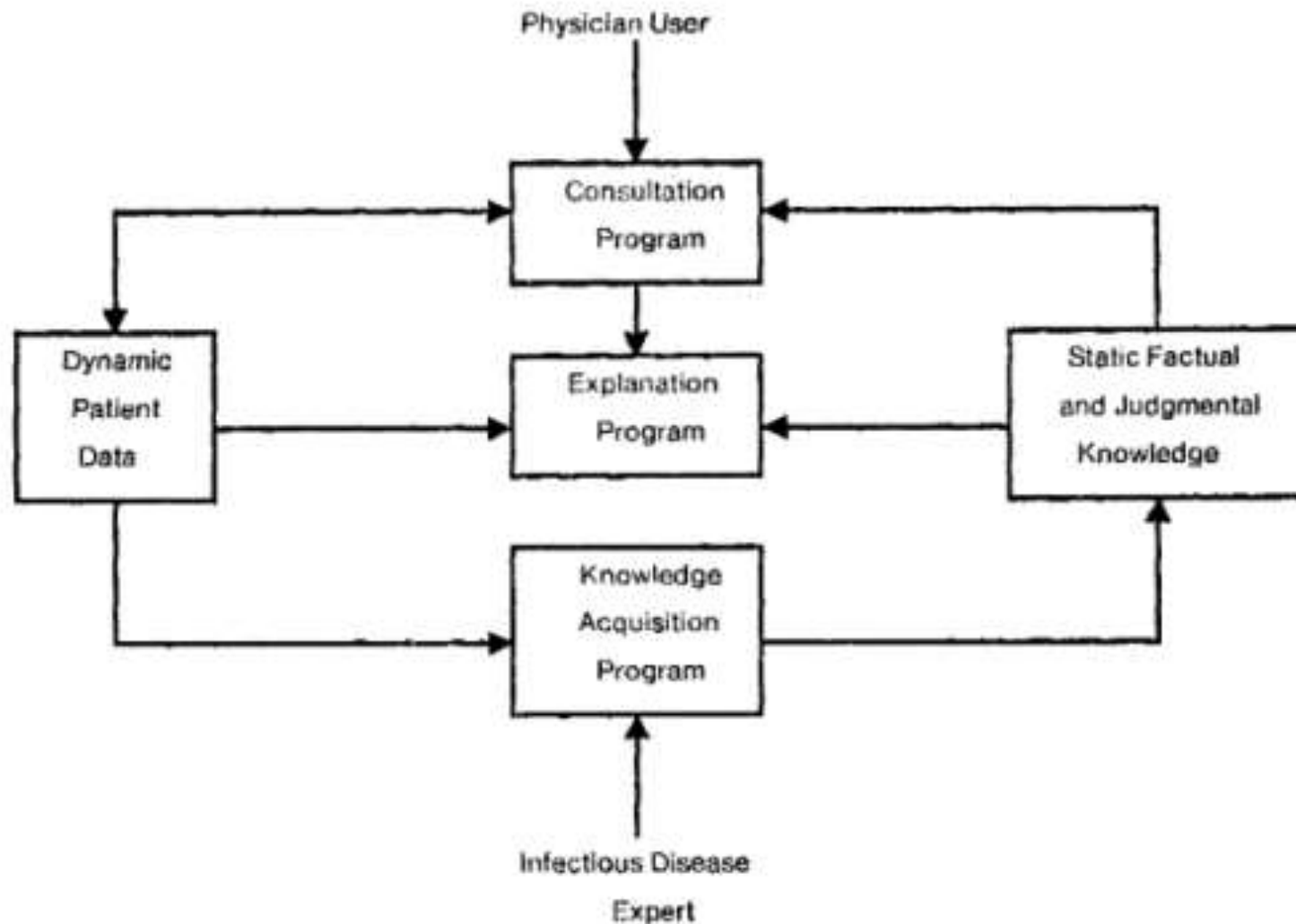
- **MYCIN**, an early [expert system](#), or [artificial intelligence](#) (AI) program, for treating [blood](#) infections.
- In 1972 work began on MYCIN at [Stanford University](#) in California.
- MYCIN would attempt to diagnose patients based on reported symptoms and medical test results. The program could request further information concerning the patient, as well as suggest additional laboratory tests, to arrive at a probable [diagnosis](#), after which it would recommend a course of treatment.
- If requested, MYCIN would explain the reasoning that led to its [diagnosis](#) and recommendation.
- Using about 600 production rules, MYCIN operated at roughly the same level of competence as human specialists in blood infections and rather better than general practitioners.

- MYCIN was an early backward chaining expert system that used artificial intelligence to identify bacteria causing severe infections and to recommend antibiotics, with the dosage adjusted for patient's body weight.
- MYCIN was developed over five or six years in the early 1970s at Stanford University.
- It was written in LISP
- MYCIN operated using fairly simple inference engine, and a knowledge base of approximately 600 rules.
- It was also used for the recommendation of antibiotics and the diagnosis of blood clotting diseases.

- MYCIN is a computer program designed to provide attending physicians with advice comparable to that which they would otherwise get from a consulting physician specializing in bacteremia and meningitis infections.
- To use MYCIN, the attending physician must sit in front of a computer terminal that is connected to a DEC-20 (one of Digital Equipment Corporation's mainframe computers) where the MYCIN program is stored.
- When the MYCIN program is evoked, it initiates a dialogue. The physician types answers in response to various questions. Eventually MYCIN provides a diagnosis and a detailed drug therapy recommendation.

Structure of Mycin Program

➤ MYCIN was an early [backward chaining expert system](#) that used [artificial intelligence](#) to identify bacteria causing severe infections. It is used in medical field.



- The MYCIN system comprises three major subprograms, as depicted in Figure above.
 - The Consultation Program
 - Explanation Program
 - Knowledge Acquisition Program
- The **Consultation Program** is the core of the system; it interacts with the physician to obtain information about the patient, generating diagnoses and therapy recommendations.
- The **Explanation Program** provides explanations and justifications for the program's actions.
- The **Knowledge-Acquisition** Program is used by experts to update the system's knowledge base.

DART – An Expert System

(Diagnostic Assistance Reference Tool)

- DART is a joint project of the Stanford University and IBM that explores the application of artificial intelligence techniques to the diagnosis of computer faults.
- It assists a technician in finding the faults in a computer system. (hardware and software)
- DART uses a device-independent language for describing devices and device-independent inference procedure for diagnosis.
- The primary goal of the DART Project is to develop programs that capture the special design knowledge and diagnostic abilities of these experts and to make them available to field engineers.
- The practical goal is the construction of an automated diagnostician that can identify which functional units are responsible for malfunctions in various system configurations.

- For DART to be useful, the field engineer must be familiar with the diagnostic equipment and software testing and tracing facilities which can be requested, and must also have access to information about the specific system hardware and software configuration of the installation.
- Finally, at the end of the consultation session, DART summarizes its findings and recommends additional tests and procedures to follow.

- Figure below depicts the major steps of the diagnostic process outlined above

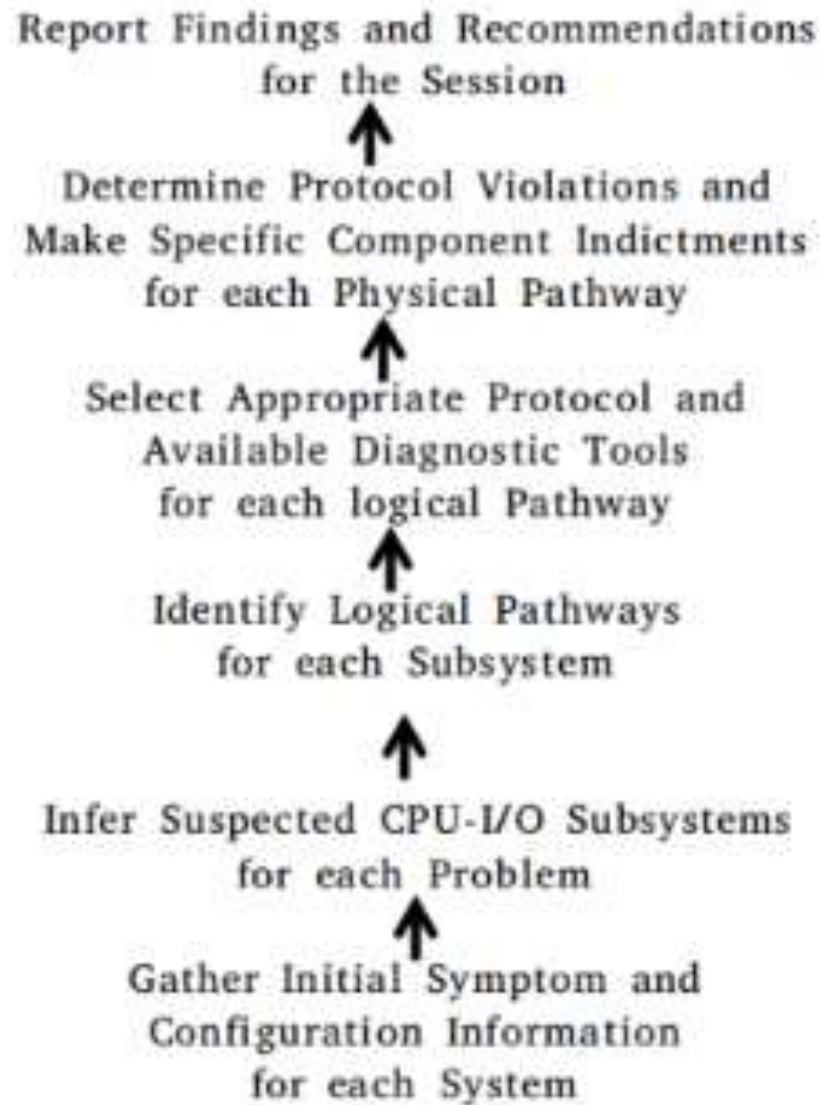


Fig: The DART Diagnostic Inference Process

DART Consultation :

- During a DART consultation session, the field engineer focuses on a particular computer system that is experiencing a problem.
- Within each such system, the user describes one or more problems by indicating a failure symptom, currently using a list of keywords.
- Using this description, the consultant makes an initial guess about which of the major subsystems might be involved in the fault.
- The user is then given the opportunity to select which of these implicated subsystems are to be pursued and in which order.
- Each subsystem serves as a focal point for tests and findings.
- These subsystems currently correspond to various input/output facilities or the CPU-complex itself.

- For each selected subsystem, the user is asked to identify one or more logical pathways which might be involved in the situation.
- Logical pathways correspond to a line of communication between a peripheral and an application program.
- On the basis of this information and details of the basic composition of the network, the appropriate communications protocol can be selected.
- Once the logical pathway and protocol have been determined, descriptions are gathered about the multiple physical pathways that actually implement the logical pathway.
- It is on this level that diagnostic test results are presented and actual component indictments occur.

XCON

- The **R1** (internally called **XCON**, for e**X**pert **CON**figurer) program was a [production-rule](#)-based system written in [OPS5](#) by [John P. McDermott](#) of [Carnegie Mellon University](#) in 1978 to assist in the ordering of [DEC](#)'s [VAX](#) computer systems by automatically selecting the computer system components based on the customer's requirements

Functions of XCON System:

- XCON is used to configure customer orders and to guide the assembly of these orders at the customer site.
- Using the customer order as input, it provides the following functionality:
- **Configures CPUs, memory, cabinets, power supplies, disk, tapes and so on.**
- **Determines and list cabling information.**
- **List components ordered with configuration related comments.**
- **Generate warning messages on issues affecting technical validity.**

Scope of XCON

- The initial purpose of XCON was to assist manufacturing plant personnel in validating the technical correctness of system orders about to be filled.
- It is now used by a broad set of users across the company's major functions like sales and marketing, manufacturing and production, field service and engineering.
- The users of these systems perform functions that span Digital's complete order flow and manufacturing cycle.

- Thus they are involved with many different business processes. Each has different needs and takes a different perspective on the configuration information provided.
- Sales uses the configuration systems as an integral part of the automated process to generate quotations for customer
- Manufacturing uses the information to verify buildability of all incoming orders.
- Field service has the perspective of assembling the order in the customer's unique environment.
- Manufacturing and engineering benefits from the configuration system's focus on system integration and can identify potential problems in system-level design and manufacturability.

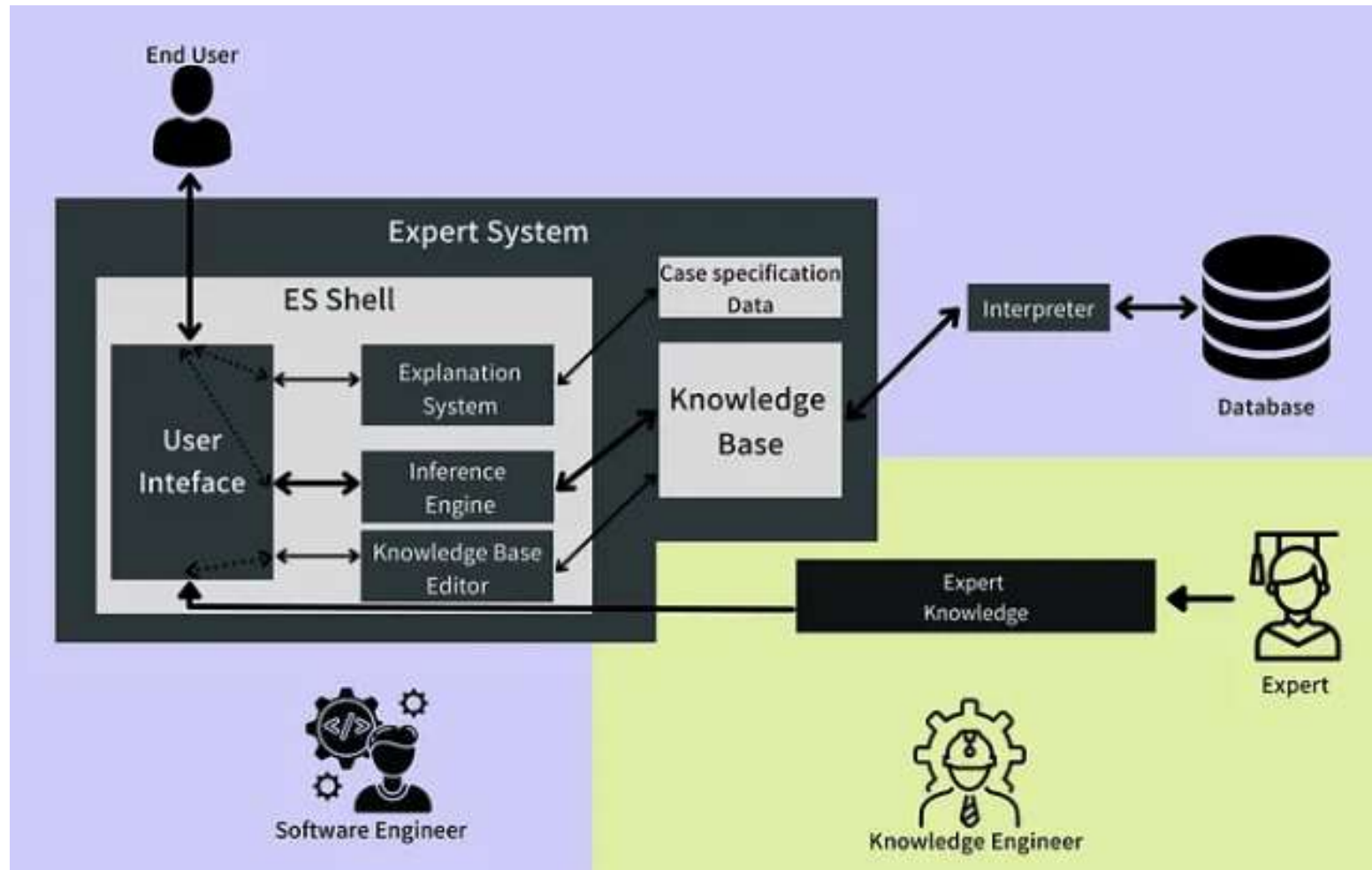
- Using XCON DEC was able to eliminate "**FA & T**" (**Faults & Time**) and to reduce average shipping time to three or four weeks, sometimes even to only two or three days.
- As the computer market changed in the 1980 a rapid turnaround of orders became a requirement for survival. Because of XCON, DEC was ready.
- These system allowed DEC to remain competitive and even to thrive in the 1980s as DEC`s sales grew as the market demanded shorter response time.
- DEC is highly dependent on its configuration system. It considers them to be a solid success for the company by contributing to customer satisfaction, lower costs and higher productivity.

Expert systems shells

- Expert system shells are toolkits that can be used to develop expert systems. They consist of some built expert system components with an empty knowledge base.
- Hence, in most cases, the knowledge engineer is left with only populating the knowledge base. It is essentially a special-purpose tool that is built-in in line with the requirements and standards of a particular domain or expert-knowledge area applications.
- It may be defined as a software package that facilitates the building of knowledge-based expert systems by providing a knowledge representation scheme and an inference engine.

- Difference ES shells offer various ways to model the knowledge into the knowledge base. They are
 1. as rules
 2. in the form of a decision tree
 3. as objects (frames) — A data structure with typical knowledge about a particular object or concept.
- The Expert System Shell refers to a software module containing an:
 - User interface (built-in)
 - Inference engine (built-in)
 - A structured skeleton of a knowledge base (in its empty state) with the suitable knowledge representation facilities

- ES Shells provide facilities for database connectivity through interpreter, web integration, and natural language processing (NLP) features.



- The user interface is the portal available for both end-users (who use the expert system to get solutions) and the knowledge engineer (who perform the knowledge engineering and modelling).
- The knowledge base can be connected with an external database (like MySQL) since the knowledge base is not optimal in storing extensive data. The knowledge base cannot directly access the database and these access features are mediated through an interpreter.
- Some ES Shells have inbuilt knowledge base editors which facilitate the Knowledge Engineer to easily update and check the knowledge base. Knowledge Engineer collects the expertise knowledge in a specific domain and models in populating the knowledge base.

- Inference engine which is the most important part of an expert system access the knowledge base and solves the problem by either backward chaining or forward chaining of facts and rules in the knowledge base. In ES Shells, the inference engine is also a built-in component that is usually programmed in ProLog.
- Most ES shells are composed of another component called **‘Explanation System’** which provides the user with reasons and explanations to provide a certain answer, by considering the ‘case specification data’ available

Popular Expert System shells

- **ES-Builder:** ES Builder stands for Expert System Builder.
- **CLIPS:** CLIPS stands for C-Language Integrated Production System
- **JESS:** JESS stands for Java Expert System Shell.
- **PyKE:** PyKE stands for Python Knowledge Engine.

Classification of various expert system shells and tools

- Expert system shells and tools can be classified by the type of knowledge they support, such as **Rule-Based**, **Frame-Based**, **Fuzzy**, and [Neural Expert Systems](#), or by their functional components, including the **Knowledge Base**, **Inference Engine**, **Knowledge Acquisition subsystem**, and **User Interface**.
- Tools provide a software framework for creating expert systems, while shells are specialized software that includes the inference engine and other essential components.

Classification by Knowledge Representation

- This classification focuses on how the system stores and represents its domain knowledge.

Rule-Based Systems:

- Store knowledge as a collection of if-then rules, which is a common and straightforward way to capture expert reasoning.

Frame-Based Systems:

- Represent knowledge using frames (or objects), which are data structures that describe concepts and their attributes and relationships, often used in natural language processing.

Fuzzy Logic Systems:

- Handle uncertainty and imprecision by allowing for "degrees of truth" rather than strict true/false values, enabling systems to work with ambiguous or incomplete information.

Neural Expert Systems:

- Incorporate artificial neural networks to learn patterns from data, with the knowledge stored as weights in the network rather than in a traditional knowledge base.

Neuro-Fuzzy Systems:

- Combine aspects of both neural networks and fuzzy logic to benefit from parallel computation, learning, and explanatory skills.

Classification by Functional Components

- This classification describes the core building blocks that expert system shells provide.

Knowledge Base:

- A repository of domain-specific facts, rules, and data, often populated by knowledge engineers using a shell's tools.

Inference Engine:

- The "brain" of the system, responsible for manipulating the knowledge in the knowledge base using reasoning methods (like forward or backward chaining) to solve problems.

Knowledge Acquisition System:

- Tools or modules that assist in extracting and encoding expert knowledge into the knowledge base, easing the development process.

User Interface:

- The means by which users interact with the expert system to provide input and receive conclusions, which can include menus, graphics, or a conversational interface.

Explanation Facilities:

- Components that can explain the reasoning process of the expert system, allowing users to understand how a conclusion was reached.

Examples of Expert System Tools and Shells

Clips:

- A powerful tool for building rule-based expert systems.

Drools:

- An open-source business rule management system often used in enterprise environments.

JESS:

- A rule-based system for building AI applications, originally developed at Sandia National Laboratories.

EMYCIN:

- An early, influential expert system shell that provided a generic framework for building diagnostic expert systems.