Course Material

(For AY 2021-22, ODD Sem)

by

S.Chandramohan
Assistant Professor-ECE
SCSVMV

ARTIFICIAL INTELLIGENCE

: **AY-2021-22 - ODD SEMESTER**

OBJECTIVES:

- The course aims to provide some fundamentals of AI and algorithms required to produce AI systems able to exhibit limited human-like abilities, particularly in the form of problem solving by search, representing and reasoning with knowledge and panning.

UNIT I (9 Hrs)

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

UNIT II (9 Hrs)

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

UNIT III (9 Hrs)

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

UNIT IV (9 Hrs)

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

UNIT V (9 Hrs)

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

OUTCOMES: Total: 45 Hrs

At the end of the course students should able to:

- Understand the fact that the computational complexity of most AI problems requires us regularly to deal with approximate techniques;
- Appreciate different perspectives on what the problems of artificial intelligence are and how different approaches are justified.
- Design basic problem solving methods based on AI-based search, knowledge representation, reasoning with knowledge and panning.

TEXT BOOK:

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

REFERENCES:

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

Unit-1

Introduction

1.1 Instructional Objectives

– Understand the definition of artificial intelligence
– Understand the different faculties involved with intelligent behavior
– Examine the different ways of approaching AI
– Look at some example systems that use AI
– Trace briefly the history of AI
– Have a fair idea of the types of problems that can be currently solved by computers and those that are as yet beyond its ability.

We will introduce the following entities:

- An agent
- An intelligent agent
- A rational agent

We will explain the notions of rationality and bounded rationality.
We will discuss different types of environment in which the agent might operate.
We will also talk about different agent architectures.

On completion of this lesson the student will be able to

- Understand what an agent is and how an agent interacts with the environment.
- Given a problem situation, the student should be able to
    - identify the percepts available to the agent and
    - the actions that the agent can execute.
- Understand the performance measures used to evaluate an agent

The student will become familiar with different agent architectures

- Stimulus response agents
- State based agents
- Deliberative / goal-directed agents
- Utility based agents

The student should be able to analyze a problem situation and be able to

- identify the characteristics of the environment
- Recommend the architecture of the desired agent

1.1.1 Definition of AI

What is AI ?

Artificial Intelligence is concerned with the design of intelligence in an artificial device. The term was coined by McCarthy in 1956.

There are two ideas in the definition.

1. Intelligence
2. artificial device

What is intelligence?

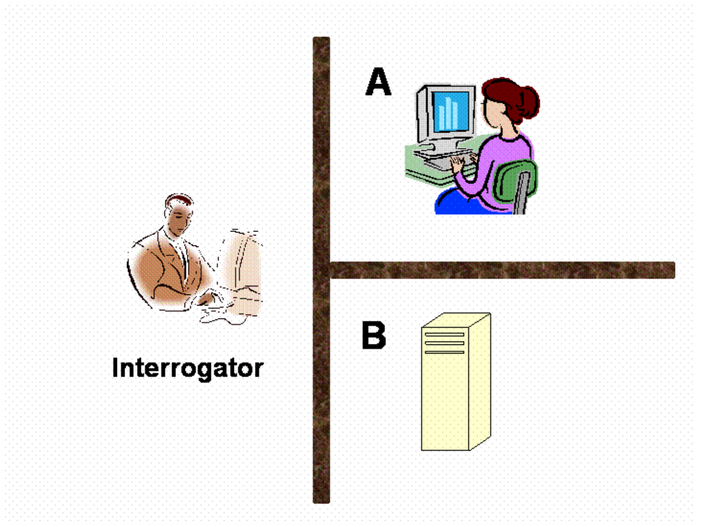
- Is it that which characterize humans? Or is there an absolute standard of judgement?
- Accordingly there are two possibilities:
  - A system with intelligence is expected to behave as intelligently as a human
  - A system with intelligence is expected to behave in the best possible manner
- Secondly what type of behavior are we talking about?
  - Are we looking at the thought process or reasoning ability of the system?
  - Or are we only interested in the final manifestations of the system in terms of its actions?

Given this scenario different interpretations have been used by different researchers as defining the scope and view of Artificial Intelligence.

1. One view is that artificial intelligence is about designing systems that are as intelligent as humans.

   This view involves trying to understand human thought and an effort to build machines that emulate the human thought process. This view is the cognitive science approach to AI.

2. The second approach is best embodied by the concept of the Turing Test. Turing held that in future computers can be programmed to acquire abilities rivaling human intelligence. As part of his argument Turing put forward the idea of an 'imitation game', in which a human being and a computer would be interrogated under conditions where the interrogator would not know which was which, the communication being entirely by textual messages. Turing argued that if the interrogator could not distinguish them by questioning, then it would be unreasonable not to call the computer intelligent. Turing's 'imitation game' is now usually called 'the Turing test' for intelligence.



Turing Test

Consider the following setting. There are two rooms, A and B. One of the rooms contains a computer. The other contains a human. The interrogator is outside and does not know which one is a computer. He can ask questions through a teletype and receives answers from both A and B. The interrogator needs to identify whether A or B are humans. To pass the Turing test, the machine has to fool the interrogator into believing that it is human. For more details on the Turing test visit the site http://cogsci.ucsd.edu/~asaygin/tt/ttest.html

3. Logic and laws of thought deals with studies of ideal or rational thought process and inference. The emphasis in this case is on the inferencing mechanism, and its properties. That is how the system arrives at a conclusion, or the reasoning behind its selection of actions is very important in this point of view. The soundness and completeness of the inference mechanisms are important here.

4. The fourth view of AI is that it is the study of rational agents. This view deals with building machines that act rationally. The focus is on how the system acts and performs, and not so much on the reasoning process. A rational agent is one that acts rationally, that is, is in the best possible manner.

1.1.2 Typical AI problems

While studying the typical range of tasks that we might expect an "intelligent entity" to perform, we need to consider both "common-place" tasks as well as expert tasks.
Examples of common-place tasks include

– *Recognizing* people, objects.
– Communicating (through *natural language*).
– *Navigating* around obstacles on the streets

These tasks are done matter of factly and routinely by people and some other animals.

Expert tasks include:

- Medical diagnosis.
- Mathematical problem solving
- Playing games like chess

These tasks cannot be done by all people, and can only be performed by skilled specialists.

Now, which of these tasks are easy and which ones are hard? Clearly tasks of the first type are easy for humans to perform, and almost all are able to master them. The second range of tasks requires skill development and/or intelligence and only some specialists can perform them well. However, when we look at what computer systems have been able to achieve to date, we see that their achievements include performing sophisticated tasks like medical diagnosis, performing symbolic integration, proving theorems and playing chess.

On the other hand it has proved to be very hard to make computer systems perform many routine tasks that all humans and a lot of animals can do. Examples of such tasks include navigating our way without running into things, catching prey and avoiding predators. Humans and animals are also capable of interpreting complex sensory information. We are able to recognize objects and people from the visual image that we receive. We are also able to perform complex social functions.

Intelligent behaviour

This discussion brings us back to the question of what constitutes intelligent behaviour. Some of these tasks and applications are:

- Perception involving image recognition and computer vision
- Reasoning
- Learning
- Understanding language involving natural language processing, speech processing
- Solving problems
- Robotics

1.1.3 Practical Impact of AI

AI components are embedded in numerous devices e.g. in copy machines for automatic correction of operation for copy quality improvement. AI systems are in everyday use for identifying credit card fraud, for advising doctors, for recognizing speech and in helping complex planning tasks. Then there are intelligent tutoring systems that provide students with personalized attention

Thus AI has increased understanding of the nature of intelligence and found many applications. It has helped in the understanding of human reasoning, and of the nature of intelligence. It has also helped us understand the complexity of modeling human reasoning.

1.1.4 Approaches to AI

Strong AI aims to build machines that can truly reason and solve problems. These machines should be self aware and their overall intellectual ability needs to be indistinguishable from that of a human being. Excessive optimism in the 1950s and 1960s concerning strong AI has given way to an appreciation of the extreme difficulty of the problem. Strong AI maintains that suitably programmed machines are capable of cognitive mental states.

Weak AI: deals with the creation of some form of computer-based artificial intelligence that cannot truly reason and solve problems, but can act as if it were intelligent. Weak AI holds that suitably programmed machines can simulate human cognition.

Applied AI: aims to produce commercially viable "smart" systems such as, for example, a security system that is able to recognise the faces of people who are permitted to enter a particular building. Applied AI has already enjoyed considerable success.

Cognitive AI: computers are used to test theories about how the human mind works--for example, theories about how we recognise faces and other objects, or about how we solve abstract problems.

1.1.5 Limits of AI Today

Today's successful AI systems operate in well-defined domains and employ narrow, specialized knowledge. Common sense knowledge is needed to function in complex, open-ended worlds. Such a system also needs to understand unconstrained natural language. However these capabilities are not yet fully present in today's intelligent systems.

What can AI systems do

Today's AI systems have been able to achieve limited success in some of these tasks.
- In Computer vision, the systems are capable of face recognition
- In Robotics, we have been able to make vehicles that are mostly autonomous.
- In Natural language processing, we have systems that are capable of simple machine translation.
- Today's Expert systems can carry out medical diagnosis in a narrow domain
- Speech understanding systems are capable of recognizing several thousand words continuous speech
- Planning and scheduling systems had been employed in scheduling experiments with

the Hubble Telescope.
- The Learning systems are capable of doing text categorization into about a 1000 topics
- In Games, AI systems can play at the Grand Master level in chess (world champion), checkers, etc.

What can AI systems NOT do yet?
- Understand natural language robustly (e.g., read and understand articles in a newspaper)
- Surf the web
- Interpret an arbitrary visual scene
- Learn a natural language
- Construct plans in dynamic real-time domains
- Exhibit true autonomy and intelligence

1.2 AI History

Intellectual roots of AI date back to the early studies of the nature of knowledge and reasoning. The dream of making a computer imitate humans also has a very early history.

The concept of intelligent machines is found in Greek mythology. There is a story in the 8th century A.D about Pygmalion Olio, the legendary king of Cyprus. He fell in love with an ivory statue he made to represent his ideal woman. The king prayed to the goddess Aphrodite, and the goddess miraculously brought the statue to life. Other myths involve human-like artifacts. As a present from Zeus to Europa, Hephaestus created Talos, a huge robot. Talos was made of bronze and his duty was to patrol the beaches of Crete.

Aristotle (384-322 BC) developed an informal system of syllogistic logic, which is the basis of the first formal deductive reasoning system.

Early in the 17th century, Descartes proposed that bodies of animals are nothing more than complex machines.

Pascal in 1642 made the first mechanical digital calculating machine.

In the 19th century, George Boole developed a binary algebra representing (some) "laws of thought."

Charles Babbage & Ada Byron worked on programmable mechanical calculating machines.

In the late 19th century and early 20th century, mathematical philosophers like Gottlob Frege, Bertram Russell, Alfred North Whitehead, and Kurt Gödel built on Boole's initial logic concepts to develop mathematical representations of logic problems.

The advent of electronic computers provided a revolutionary advance in the ability to study intelligence.

In 1943 McCulloch & Pitts developed a Boolean circuit model of brain. They wrote the paper "A Logical Calculus of Ideas Immanent in Nervous Activity", which explained how it is possible for neural networks to compute.

Marvin Minsky and Dean Edmonds built the SNARC in 1951, which is the first randomly wired neural network learning machine (SNARC stands for Stochastic Neural-Analog Reinforcement Computer).It was a neural network computer that used 3000 vacuum tubes and a network with 40 neurons.

In 1950 Turing wrote an article on "Computing Machinery and Intelligence" which articulated a complete vision of AI. For more on Alan Turing see the site http://www.turing.org.uk/turing/

Turing's paper talked of many things, of solving problems by searching through the space of possible solutions, guided by heuristics. He illustrated his ideas on machine intelligence by reference to chess. He even propounded the possibility of letting the machine alter its own instructions so that machines can learn from experience.

In 1956 a famous conference took place in Dartmouth. The conference brought together the founding fathers of artificial intelligence for the first time. In this meeting the term "Artificial Intelligence" was adopted.

Between 1952 and 1956, Samuel had developed several programs for playing checkers. In 1956, Newell & Simon's Logic Theorist was published. It is considered by many to be the first AI program. In 1959, Gelernter developed a Geometry Engine. In 1961 James Slagle (PhD dissertation, MIT) wrote a symbolic integration program, SAINT. It was written in LISP and solved calculus problems at the college freshman level. In 1963, Thomas Evan's program Analogy was developed which could solve IQ test type analogy problems.

In 1963, Edward A. Feigenbaum & Julian Feldman published Computers and Thought, the first collection of articles about artificial intelligence.

In 1965, J. Allen Robinson invented a mechanical proof procedure, the Resolution Method, which allowed programs to work efficiently with formal logic as a representation language. In 1967, the Dendral program (Feigenbaum, Lederberg, Buchanan, Sutherland at Stanford) was demonstrated which could interpret mass spectra on organic chemical compounds. This was the first successful knowledge-based program for scientific reasoning. In 1969 the SRI robot, Shakey, demonstrated combining locomotion, perception and problem solving.

The years from 1969 to 1979 marked the early development of knowledge-based systems
In 1974: MYCIN demonstrated the power of rule-based systems for knowledge representation and inference in medical diagnosis and therapy. Knowledge representation

schemes were developed. These included frames developed by Minski. Logic based languages like Prolog and Planner were developed.

In the 1980s, Lisp Machines developed and marketed.
Around 1985, neural networks return to popularity
In 1988, there was a resurgence of probabilistic and decision-theoretic methods

The early AI systems used general systems, little knowledge. AI researchers realized that specialized knowledge is required for rich tasks to focus reasoning.

The 1990's saw major advances in all areas of AI including the following:

- machine learning, data mining
- intelligent tutoring,
- case-based reasoning,
- multi-agent planning, scheduling,
- uncertain reasoning,
- natural language understanding and translation,
- vision, virtual reality, games, and other topics.

Rod Brooks' COG Project at MIT, with numerous collaborators, made significant progress in building a humanoid robot

The first official Robo-Cup soccer match featuring table-top matches with 40 teams of interacting robots was held in 1997. For details, see the site http://murray.newcastle.edu.au/users/students/2002/c3012299/bg.html

In the late 90s, Web crawlers and other AI-based information extraction programs become essential in widespread use of the world-wide-web.

Interactive robot pets ("smart toys") become commercially available, realizing the vision of the 18th century novelty toy makers.

In 2000, the Nomad robot explores remote regions of Antarctica looking for meteorite samples.

We will now look at a few famous AI system that has been developed over the years.

1. ALVINN:

Autonomous Land Vehicle In a Neural Network

In 1989, Dean Pomerleau at CMU created ALVINN. This is a system which learns to control vehicles by watching a person drive. It contains a neural network whose input is a 30x32 unit two dimensional camera image. The output layer is a representation of the direction the vehicle should travel.

The system drove a car from the East Coast of USA to the west coast, a total of about

2850 miles. Out of this about 50 miles were driven by a human, and the rest solely by the system.

2. Deep Blue
In 1997, the Deep Blue chess program created by IBM, beat the current world chess champion, Gary Kasparov.

3. Machine translation

A system capable of translations between people speaking different languages will be a remarkable achievement of enormous economic and cultural benefit. Machine translation is one of the important fields of endeavour in AI. While some translating systems have been developed, there is a lot of scope for improvement in translation quality.

4. Autonomous agents
In space exploration, robotic space probes autonomously monitor their surroundings, make decisions and act to achieve their goals.
NASA's Mars rovers successfully completed their primary three-month missions in April, 2004. The Spirit rover had been exploring a range of Martian hills that took two monthsto reach. It is finding curiously eroded rocks that may be new pieces to the puzzle of the region's past. Spirit's twin, Opportunity, had been examining exposed rock layers inside a crater.

5. Internet agents
The explosive growth of the internet has also led to growing interest in internet agents to monitor users' tasks, seek needed information, and to learn which information is most useful

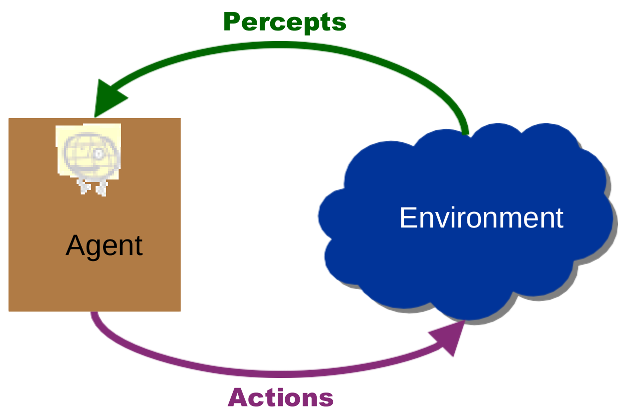
For more information the reader may consult AI in the news:
http://www.aaai.org/AITopics/html/current.html

Unit-2

Introduction to Agent

1.3.1 Introduction to Agents

An agent acts in an environment.



An agent perceives its environment through sensors. The complete set of inputs at a given time is called a percept. The current percept, or a sequence of percepts can influence the actions of an agent. The agent can change the environment through actuators or effectors. An operation involving an effector is called an action. Actions can be grouped into action sequences. The agent can have goals which it tries to achieve.

Thus, an agent can be looked upon as a system that implements a mapping from percept sequences to actions.

A performance measure has to be used in order to evaluate an agent.

An autonomous agent decides autonomously which action to take in the current situation to maximize progress towards its goals.

1.3.1.1 Agent Performance

An agent function implements a mapping from perception history to action. The behaviour and performance of intelligent agents have to be evaluated in terms of the agent function.

The **ideal mapping** specifies which actions an agent ought to take at any point in time.

The **performance measure** is a subjective measure to characterize how successful an agent is. The success can be measured in various ways. It can be measured in terms of speed or efficiency of the agent. It can be measured by the accuracy or the quality of the solutions achieved by the agent. It can also be measured by power usage, money, etc.

1.3.1.2 Examples of Agents

1. Humans can be looked upon as agents. They have eyes, ears, skin, taste buds, etc. for sensors; and hands, fingers, legs, mouth for effectors.

2. Robots are agents. Robots may have camera, sonar, infrared, bumper, etc. for sensors. They can have grippers, wheels, lights, speakers, etc. for actuators.

   Some examples of robots are Xavier from CMU, COG from MIT, etc.

Xavier Robot (CMU)

   Then we have the AIBO entertainment robot from SONY.

Aibo from SONY

3. We also have software agents or softbots that have some functions as sensors and some functions as actuators. Askjeeves.com is an example of a softbot.
4. Expert systems like the Cardiologist is an agent.
5. Autonomous spacecrafts.
6. Intelligent buildings.

1.3.1.3 Agent Faculties

The fundamental faculties of intelligence are

- Acting
- Sensing
- Understanding, reasoning, learning

Blind action is not a characterization of intelligence. In order to act intelligently, one must sense. Understanding is essential to interpret the sensory percepts and decide on an action. Many robotic agents stress sensing and acting, and do not have understanding.

1.3.1.4 Intelligent Agents

An **Intelligent Agent** must sense, must act, must be autonomous (to some extent),. It also must be rational.

AI is about building rational agents. An agent is something that perceives and acts.
A rational agent always does the right thing.

1. What are the functionalities (goals)?
2. What are the components?
3. How do we build them?

1.3.1.5 Rationality

Perfect Rationality assumes that the rational agent knows all and will take the action that maximizes her utility. Human beings do not satisfy this definition of rationality.
Rational Action is the action that maximizes the expected value of the performance measure given the percept sequence to date.

However, a rational agent is not omniscient. It does not know the actual outcome of its actions, and it may not know certain aspects of its environment. Therefore rationality must take into account the limitations of the agent. The agent has too select the best action to the best of its knowledge depending on its percept sequence, its background knowledge and its feasible actions. An agent also has to deal with the expected outcome of the actions where the action effects are not deterministic.

1.3.1.6 Bounded Rationality

"Because of the limitations of the human mind, humans must use approximate methods to handle many tasks." Herbert Simon, 1972

Evolution did not give rise to optimal agents, but to agents which are in some senses locally optimal at best. In 1957, Simon proposed the notion of Bounded Rationality:
that property of an agent that behaves in a manner that is nearly optimal with respect to its goals as its resources will allow.

Under these promises an intelligent agent will be expected to act optimally to the best of its abilities and its resource constraints.

1.3.2 Agent Environment

Environments in which agents operate can be defined in different ways. It is helpful to view the following definitions as referring to the way the environment appears from the point of view of the agent itself.

1.3.2.1 Observability

In terms of observability, an environment can be characterized as fully observable or partially observable.

In a fully observable environment all of the environment relevant to the action being considered is observable. In such environments, the agent does not need to keep track of the changes in the environment. A chess playing system is an example of a system that operates in a fully observable environment.

In a partially observable environment, the relevant features of the environment are only partially observable. A bridge playing program is an example of a system operating in a partially observable environment.

1.3.2.2 Determinism

In deterministic environments, the next state of the environment is completely described by the current state and the agent's action. Image analysis systems are examples of this kind of situation. The processed image is determined completely by the current image and the processing operations.

If an element of interference or uncertainty occurs then the environment is stochastic. Note that a deterministic yet partially observable environment will *appear* to be stochastic to the agent. Examples of this are the automatic vehicles that navigate a terrain, say, the Mars rovers robot. The new environment in which the vehicle is in is stochastic in nature.

If the environment state is wholly determined by the preceding state and the actions of *multiple* agents, then the environment is said to be strategic. Example: Chess. There are two agents, the players and the next state of the board is strategically determined by the players' actions.

1.3.2.3 Episodicity

An **episodic** environment means that subsequent episodes do not depend on what actions occurred in previous episodes.

In a **sequential** environment, the agent engages in a series of connected episodes.

1.3.2.4 Dynamism

Static Environment: does not change from one state to the next while the agent is considering its course of action. The only changes to the environment are those caused by the agent itself.

- A **static** environment does not change while the agent is thinking.
- The passage of time as an agent deliberates is irrelevant.
- The agent doesn't need to observe the world during deliberation.

A Dynamic Environment changes over time independent of the actions of the agent -- and thus if an agent does not respond in a timely manner, this counts as a choice to do nothing

1.3.2.5 Continuity

If the number of distinct percepts and actions is limited, the environment is **discrete**, otherwise it is **continuous**.

1.3.2.6 Presence of Other agents

Single agent/ Multi-agent

A multi-agent environment has other agents. If the environment contains other intelligent agents, the agent needs to be concerned about strategic, game-theoretic aspects of the environment (for either cooperative *or* competitive agents)

Most engineering environments do not have multi-agent properties, whereas most social and economic systems get their complexity from the interactions of (more or less) rational agents.

1.3.3 Agent architectures

We will next discuss various agent architectures.

1.3.3.1 Table based agent

In table based agent the action is looked up from a table based on information about the agent's percepts. A table is simple way to specify a mapping from percepts to actions. The mapping is implicitly defined by a program. The mapping may be implemented by a rule based system, by a neural network or by a procedure.

There are several disadvantages to a table based system. The tables may become very large. Learning a table may take a very long time, especially if the table is large. Such systems usually have little autonomy, as all actions are pre-determined.

1.3.3.2. Percept based agent or reflex agent

In percept based agents,

1. information comes from **sensors - percepts**
2. changes the agents current **state of the world**

3. triggers **actions** through the **effectors**

Such agents are called reactive agents or stimulus-response agents. Reactive agents have no notion of history. The current state is as the sensors see it right now. The action is based on the current percepts only.

The following are some of the characteristics of percept-based agents.

- Efficient
- No internal representation for reasoning, inference.
- No strategic planning, learning.
- Percept-based agents are not good for multiple, opposing, goals.

1.3.3.3 Subsumption Architecture

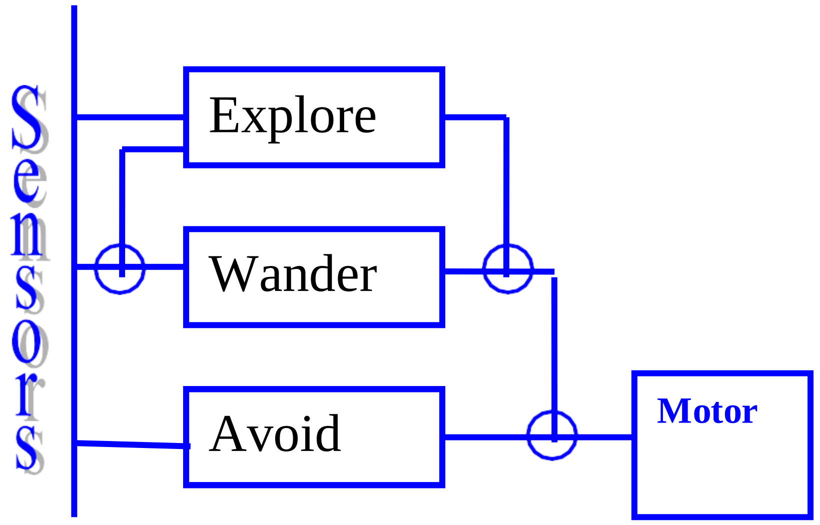
We will now briefly describe the subsumption architecture (Rodney Brooks, 1986). This architecture is based on reactive systems. Brooks notes that in lower animals there is no deliberation and the actions are based on sensory inputs. But even lower animals are capable of many complex tasks. His argument is to follow the evolutionary path and build simple agents for complex worlds.

The main features of Brooks' architecture are.

- There is no explicit knowledge representation
- Behaviour is distributed, not centralized
- Response to stimuli is reflexive
- The design is bottom up, and complex behaviours are fashioned from the combination of simpler underlying ones.
- Individual agents are simple

The Subsumption Architecture built in layers. There are different layers of behaviour. The higher layers can override lower layers. Each activity is modeled by a finite state machine.

The subsumption architecture can be illustrated by Brooks' Mobile Robot example.



Subsumption Architecture

The system is built in three layers.

1. Layer 0: Avoid Obstacles
2. Layer1: Wander behaviour
3. Layer 2: Exploration behaviour

Layer 0 (Avoid Obstacles) has the following capabilities:

- Sonar: generate sonar scan
- Collide: send HALT message to forward
- Feel force: signal sent to run-away, turn

Layer1 (Wander behaviour)

- Generates a random heading
- Avoid reads repulsive force, generates new heading, feeds to turn and forward

Layer2 (Exploration behaviour)

- Whenlook notices idle time and looks for an interesting place.
- Pathplan sends new direction to avoid.
- Integrate monitors path and sends them to the path plan.

1.3.3.4 State-based Agent or model-based reflex agent

State based agents differ from percept based agents in that such agents maintain some sort of state based on the percept sequence received so far. The state is updated regularly based on what the agent senses, and the agent’s actions. Keeping track of the state requires that the agent has knowledge about how the world evolves, and how the agent's actions affect the world.

Thus a state based agent works as follows:

- information comes from **sensors - percepts**
- based on this, the agent changes the current **state of the world**
- based on **state of the world** and **knowledge (memory)**, it triggers **actions** through the **effectors**

1.3.3.5 Goal-based Agent

The goal based agent has some goal which forms a basis of its actions.
Such agents work as follows:

- information comes from **sensors - percepts**
- changes the agents current **state of the world**
- based on **state of the world** and **knowledge (memory)** and **goals/intentions**, it chooses **actions** and does them through the **effectors**.

Goal formulation based on the current situation is a way of solving many problems and search is a universal problem solving mechanism in AI. The sequence of steps required to solve a problem is not known a priori and must be determined by a systematic exploration of the alternatives.

1.3.3.6 Utility-based Agent

Utility based agents provides a more general agent framework. In case that the agent has multiple goals, this framework can accommodate different preferences for the different goals.
Such systems are characterized by a utility function that maps a state or a sequence of states to a real valued utility. The agent acts so as to maximize expected utility

1.3.3.7 Learning Agent

Learning allows an agent to operate in initially unknown environments. The learning element modifies the performance element. Learning is required for true autonomy

1.4 Conclusion

In conclusion AI is a truly fascinating field. It deals with exciting but hard problems. A goal of AI is to build intelligent agents that act so as to optimize performance.

- An **agent** perceives and acts in an environment, has an architecture, and is implemented by an agent program.
- An **ideal agent** always chooses the action which maximizes its expected performance, given its percept sequence so far.
- An **autonomous agent** uses its own experience rather than built-in knowledge of the

environment by the designer.

- An agent program maps from percept to action and updates its internal state.

    - Reflex agents respond immediately to percepts.
    - Goal-based agents act in order to achieve their goal(s).
    - Utility-based agents maximize their own utility function.
- Representing knowledge is important for successful agent design.
- The most challenging environments are partially observable, stochastic, sequential, dynamic, and continuous, and contain multiple intelligent agents.

Questions

1. Define intelligence.
2. What are the different approaches in defining artificial intelligence?
3. Suppose you design a machine to pass the Turing test. What are the capabilities such a machine must have?
4. Design ten questions to pose to a man/machine that is taking the Turing test.
5. Do you think that building an artificially intelligent computer automatically shed light on the nature of natural intelligence?
6. List 5 tasks that you will like a computer to be able to do within the next 5 years.
7. List 5 tasks that computers are unlikely to be able to do in the next 10 years.
8. Define an agent.
9. What is a rational agent ?
10. What is bounded rationality ?
11. What is an autonomous agent ?
12. Describe the salient features of an agent.
13. Find out about the Mars rover.
    1. What are the percepts for this agent ?
    2. Characterize the operating environment.
    3. What are the actions the agent can take ?
    4. How can one evaluate the performance of the agent ?
    5. What sort of agent architecture do you think is most suitable for this agent ?
14. Answer the same questions as above for an Internet shopping agent.

Answers

1. Intelligence is a rather hard to define term.
   Intelligence is often defined in terms of what we understand as intelligence in humans.
   Allen Newell defines *intelligence* as the *ability to bring all the knowledge a system has at its disposal to bear in the solution of a problem*.
   A more practical definition that has been used in the context of building artificial

systems with intelligence is *to perform better on tasks that humans currently do better*.

2.

- Thinking rationally
- Acting rationally
- Thinking like a human
- Acting like a human

3.

- Natural language processing
- Knowledge representation
- Automated reasoning
- Machine Learning
- Computer vision
- Robotics

4-7 : Use your own imagination

8. An agent is anything that can be viewed as perceiving its environment through sensors and executing actions using actuators.
9. A rational agent always selects an action based on the percept sequence it has received so as to maximize its (expected) performance measure given the percepts it has received and the knowledge possessed by it.
10. A rational agent that can use only bounded resources cannot exhibit the optimal behaviour. A bounded rational agent does the best possible job of selecting good actions given its goal, and given its bounded resources.
11. Autonomous agents are software entities that are capable of independent action in dynamic, unpredictable environments. An autonomous agent can learn and adapt to a new environment.

12.12.

- An agent perceives its environment using sensors
- An agent takes actions in the environment using actuators
- A rational agent acts so as to reach its goal, or to maximize its utility
- Reactive agents decide their action on the basis of their current state and the percepts. Deliberative agents reason about their goals to decide their action.

13. Mars Rover
    a. Spirit's sensor include
        i. panoramic and microscopic cameras,
        ii. a radio receiver,
        iii. spectrometers for studying rock samples including an alpha particle x-ray spectrometer, M¨ossbauer spectrometer, and miniature thermal emission spectrometer

    b. The environment (the Martian surface)
        i. partially observable,
        ii. non-deterministic,
        iii. sequential,
        iv. dynamic,
        v. continuous, and
        vi. may be single-agent. If a rover must cooperate with its mother ship or other rovers, or if mischievous Martians tamper with its progress, then the environment gains additional agents

c. The **rover** Spirit has
    i. motor-driven wheels for locomotion
    ii. along with a robotic arm to bring sensors close to interesting rocks and a
    iii. rock abrasion tool (RAT) capable of efficiently drilling 45mm holes in hardvolcanic rock.
    iv. Spirit also has a radio transmitter for communication.

d. Performance measure: A **Mars rover** may be tasked with
    i. maximizing the distance or variety of terrain it traverses,
    ii. or with collecting as many samples as possible,
    iii. or with finding life (for which it receives 1 point if it succeeds, and 0 pointsif it fails).

    Criteria such as maximizing lifetime or minimizing power consumption are (at best)derived from more fundamental goals; e.g., if it crashes or runs out of power in the field, then it can't explore.

e. A model-based reflex agent is suitable for low level navigation.
    For route planning, experimentation etc, some combination of goal-based, and utility-based would be needed.

14. Internet book shopping agent

f. Sensors: Ability to parse Web pages, interface for user requests
g. Environment: Internet. Partially observable, partly deterministic, sequential, partlystatic, discrete, single-agent (exception: auctions)
h. Actuators: Ability to follow links, fill in forms, display info to user
i. Performance Measure: Obtains requested books, minimizes cost/time
j. Agent architecture: goal based agent with utilities fro open-ended situations

UNIT-3

ONLINE SEARCH AGENTS AND UNKNOWN ENVIRONMENTS

An online search agent operatesby interleaving computation and action: first it takes an action and then it observes the environment and computes the next action. Online search is a good idea in dynamic or semi dynamic domains-domains where there is a penalty for sitting around and computing too long. Online search is an even better idea for stochastic domains.

(The term "online" is commonly used in computer science to refer to algorithms that must process input data as they are received, rather than waiting for the entire input data set to become available.)

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

For example,
A chess playing agent is well-advised to make its first move long before it has figured out the complete course of the game. Online search is a necessary idea for an exploration problem, where the states and actions are unknown to the agent. An agent in this state of Ignorance must use its actions as experiments to determine what to do next, and hence must interleave computation and action.

The canonical example of online search is a robot that is placed in a new building and must explore it to build a map that it can use for getting from A to B. Methods for escaping from labyrinths-required knowledge for aspiring heroes of antiquity-are also examples of online search algorithms. Spatial exploration is not the only form of exploration, however.

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

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

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

GOAL-TEST(S).

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

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

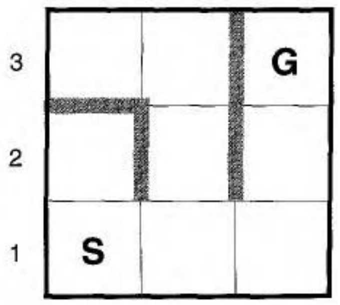
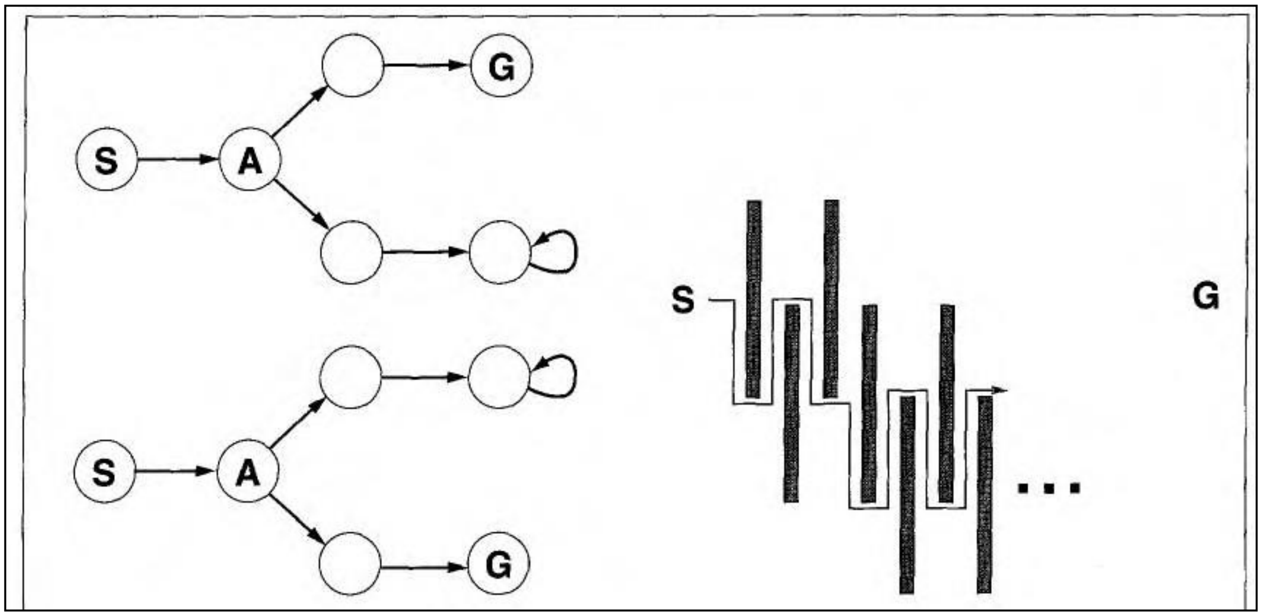


Figure: A simpIe maze problem.

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



(a) (b)

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

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

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

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

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

Online search agents

After each action, an online agent receives a percept telling it what state it has reached; from this information, it can augment its map of the environment. The current map is used to decide where to go next. This interleaving of planning and action means that online search algorithms are quite different from the offline search algorithms we have seen previously.

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

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

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

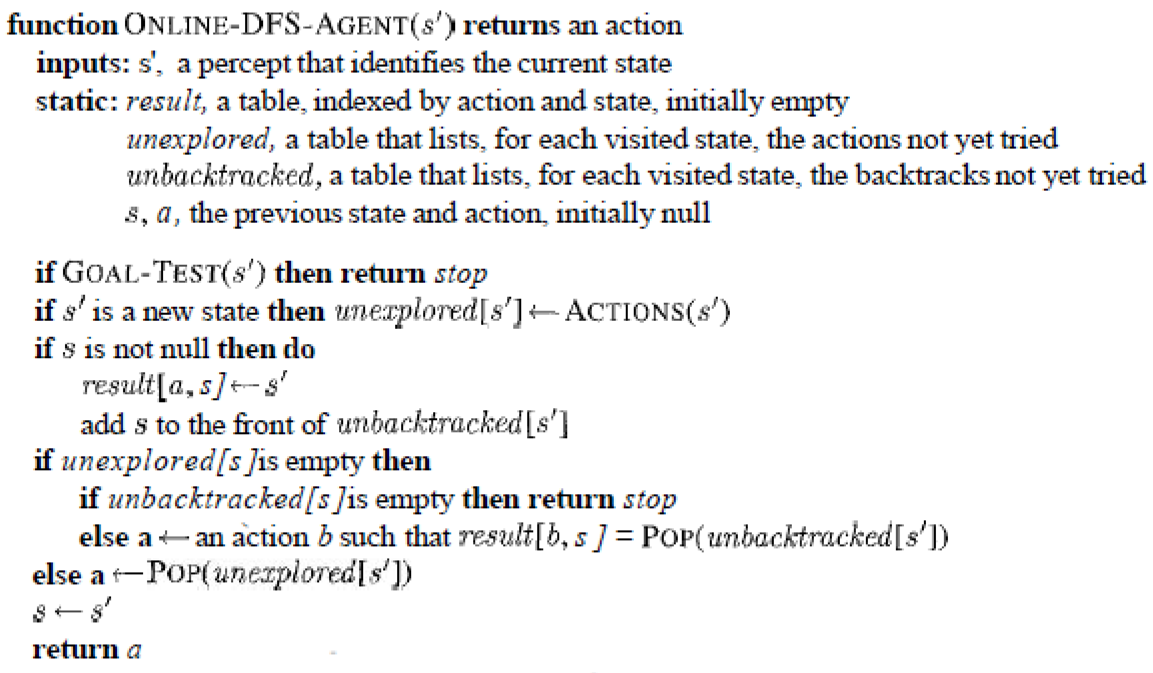
The difficulty comes when the agent has tried all the actions in a state. In offline depth-first search, the state is simply dropped from the queue; in an online search, the agent has to backtrack physically.

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

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

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

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



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

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

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

The agent is applicable only in bidirected search spaces.

Online local search

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

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

The example is contrived, of course, but there are many real-world state spaces whose topology causes these kinds of "traps" for random walks.

Augmenting hill climbing with memory rather than randomness turns out to be a more effective approach. The basic idea is to store a "current best estimate" H(s) of the cost to reach the goal from each state that has been visited. H(s) starts out being just the heuristic

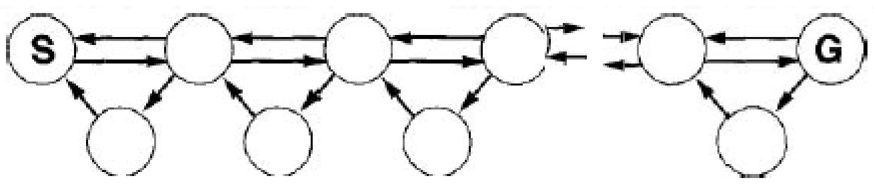


Figure - An environment in which a random walk will take exponentially many steps to find the goal.

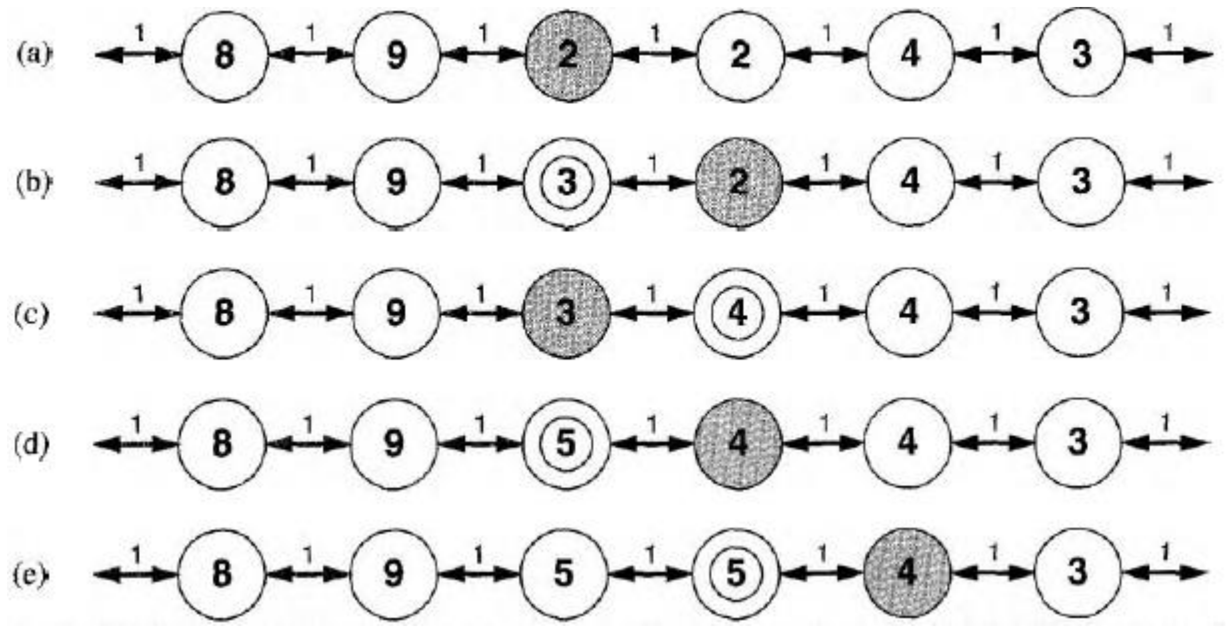
estimate h(s) and is updated as the agent gains experience in the state space. Figure shows a simple example in a one-dimensional state space. In (a), the agent seems to be stuck in a flat local minimum at the shaded state. Rather than staying where it is, the agent should follow what seems to be the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

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

An agent implementing this scheme, which is called learning real-time A* (LRTA*), is shown in Figure. Like ONLINE-DFS-AGENT, it builds a map of the environment using the result table. It updates the cost estimate for the state it has just left and then chooses the "apparently best" move according to its current cost estimates. One important detail is that actions that have not yet been tried in a state s are always assumed to lead immediately to the goal with the least possible cost, namely h(s). This optimism under uncertainty encourages the agent to explore new, possibly promising paths.

Learning in online search

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



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

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

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

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

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

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

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

Constraint satisfaction problem (CSP**)**

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

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

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

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

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

To formulate this as a CSP, we define the variables to be the regions: WA, NT, Q, NSW, V, SA, and T. The domain of each variable is the set {red, green, blue). The constraints require neighboring regions to have distinct colors; for example, the allowable combinations for WA and NT are the pairs {(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)) .
(The constraint can also be represented more succinctly as the inequality WA #

NT, provided the constraint satisfaction algorithm has some way to evaluate such expressions.) There are many possible solutions, such as {WA= red, NT = green, Q = red, NSW = green, V= red, SA= blue, T= red).

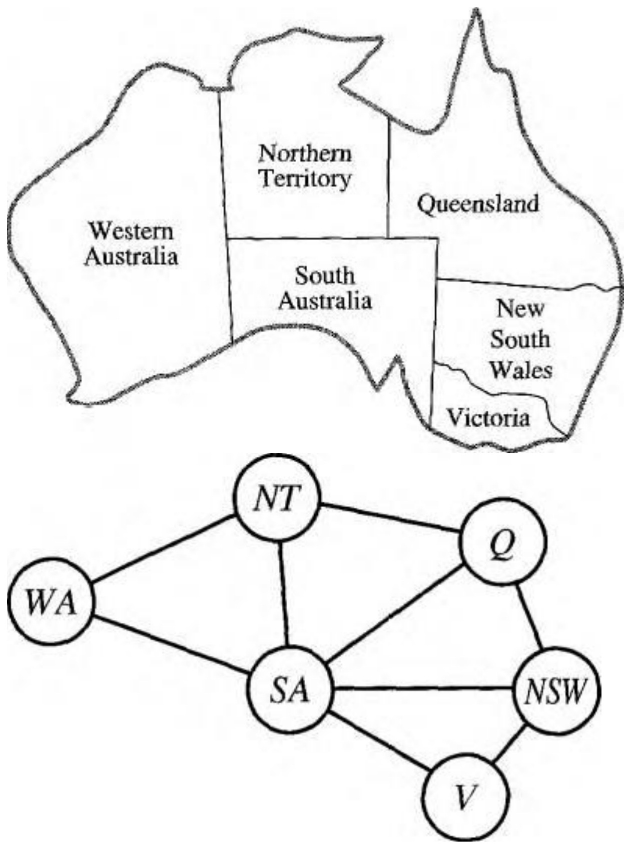


Figure (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color.
(b) The map-coloring problem represented as a constraint graph.

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

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

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

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

Goal test: the current assignment is complete.

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

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

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

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

Special solution algorithms (which we will not discuss here) exist for linear constraints on integer variables-that is, constraints, such as the one just given, in which each variable appears only in linear form. It can be shown that no algorithm exists for solving general nonlinear constraints on integer variables. In some cases, we can reduce integer constraint problems to finite-domain problems simply by bounding the values of all the variables.

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

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

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

BACKTRACKING SEARCH FOR CSP

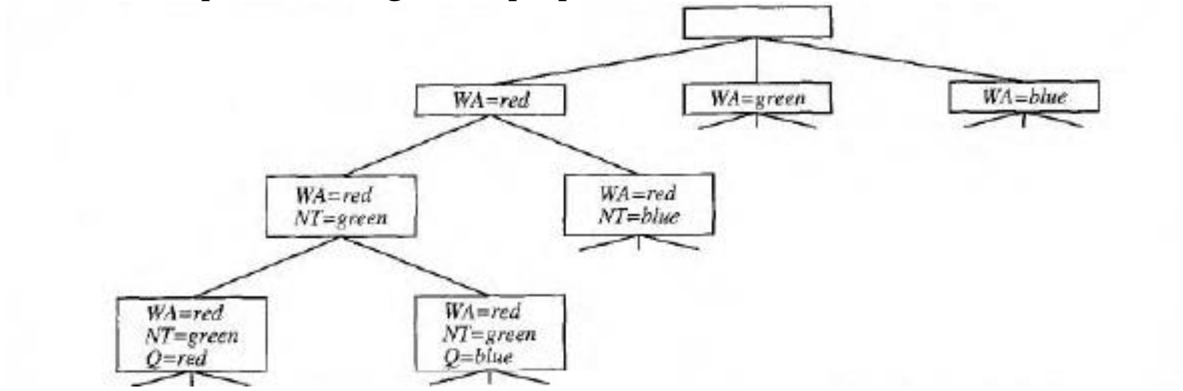
A problem is commutative if the order of application of any given set of actions has no effect on the outcome. This is the case for CSPs because, when assigning values to variables, we reach the same partial assignment, regardless of order. Therefore, all CSP search algorithms generate successors by considering possible assignments for only a single variable at each node in the search tree. For example, at the root node of a search tree for coloring the map of Australia, we might have a choice between SA = red, SA = green, and SA =

blue, but we would never choose between SA = red and WA = blue. With this restriction, the number of leaves is d^n.

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

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

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



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

The term backtracking search is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown in Figure. Notice that it uses, in effect, the one-at-a-time method of incremental successor generation described. Also, it extends the current assignment to generate a successor, rather than copying it. Because the representation of CSPs is standardized, there is no need to supply Backtracking-Search with a domain-specific initial state, successor function, or goal test. Part of the search tree for the Australia problem is shown in Figure, where we have assigned variables in the order WA, NT, Q, . . …. Plain backtracking is an uninformed algorithm, so we do not expect it to be very effective for large problems. The results for some sample problems are shown in the first column and confirm our expectations. It turns out that we can solve CSPs efficiently without such domain-specific knowledge.

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

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

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

The backtracking algorithm contains the line

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

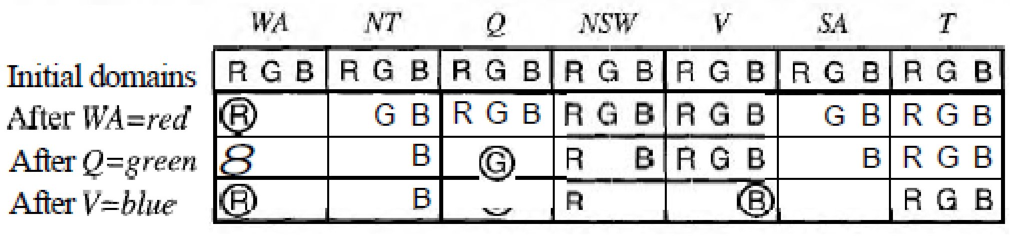
By default, SELECT-UNASSIGNED-VARIABLE simply selects the next unassigned variable in the order given by the list VARIABLES [csp. This static variable ordering seldom results in the most efficient search. For example, after the assignments for WA = red and NT = green, there is only one possible value for SA, so it makes sense to assign SA = blue next rather than assigning Q. In fact, after SA is assigned, the choices for Q, NS W, and V are all forced.
This intuitive idea-choosing the variable with the fewest "legal" values-is called the minimum remaining values (MRV) heuristic. It also has been called the "most constrained variable" or "fail-first" heuristic, the latter because it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree. If there is a variable X with zero legal values remaining, the MRV heuristic will select X and failure will be detected immediately-avoiding pointless searches through other variables which always will fail when X is finally selected.

Propagating information through constraints

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

Forward checking

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



	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	8	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B		R	Ⓑ		R G B

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

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

Constraint propagation

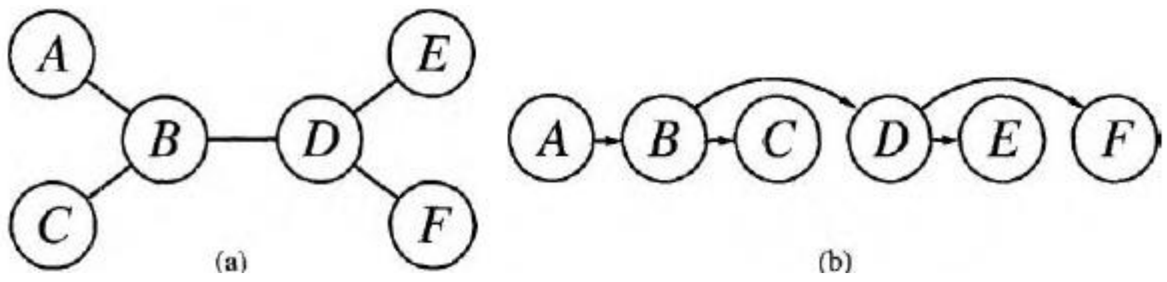
Although forward checking detects many inconsistencies, it does not detect all of them. For example, consider the third row of Figure. It shows that when WA is red and Q is green, both NT and SA are forced to be blue. But they are, adjacent and so cannot have the same value. Forward checking does not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

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

Structure Of Problems

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

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



(a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with A as the root.

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

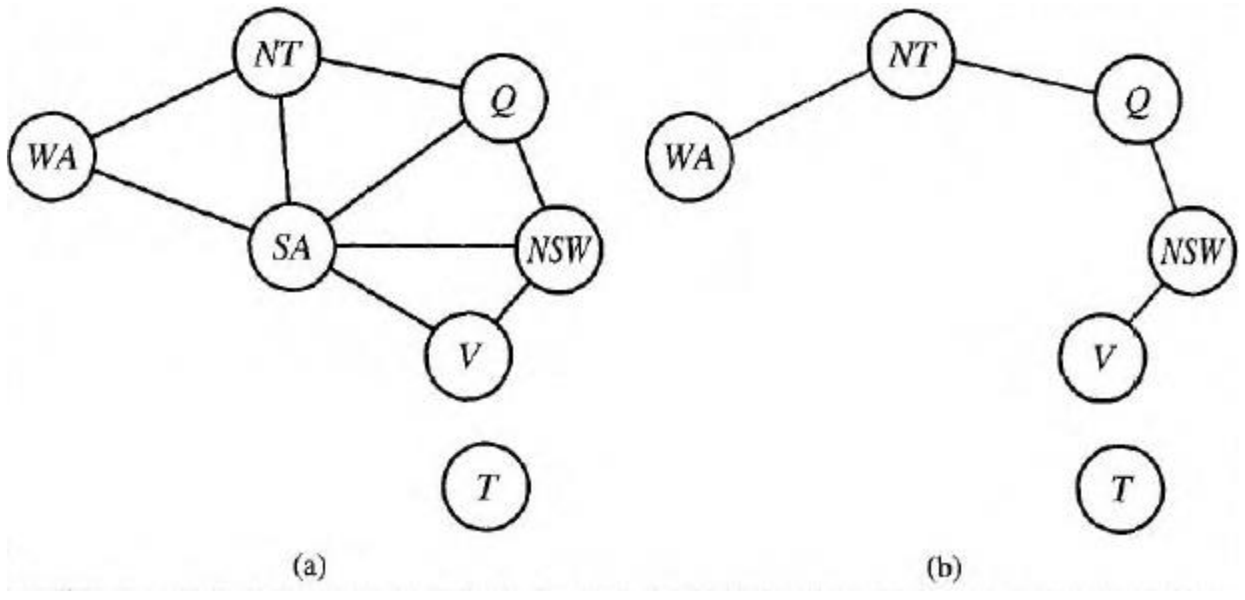
The algorithm has the following steps:

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

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

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

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



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

Now, any solution for the CSP after SA and its constraints are removed will be consistent with the value chosen for SA. (This works for binary CSPs; the situation is more complicated with higher-order constraints.) Therefore, we can solve the remaining tree with the algorithm given above and thus solve the whole problem. Of course, in the general case (as opposed to map coloring) the value chosen for SA could be the wrong one, so we would need to try each of them.
The general algorithm is as follows:

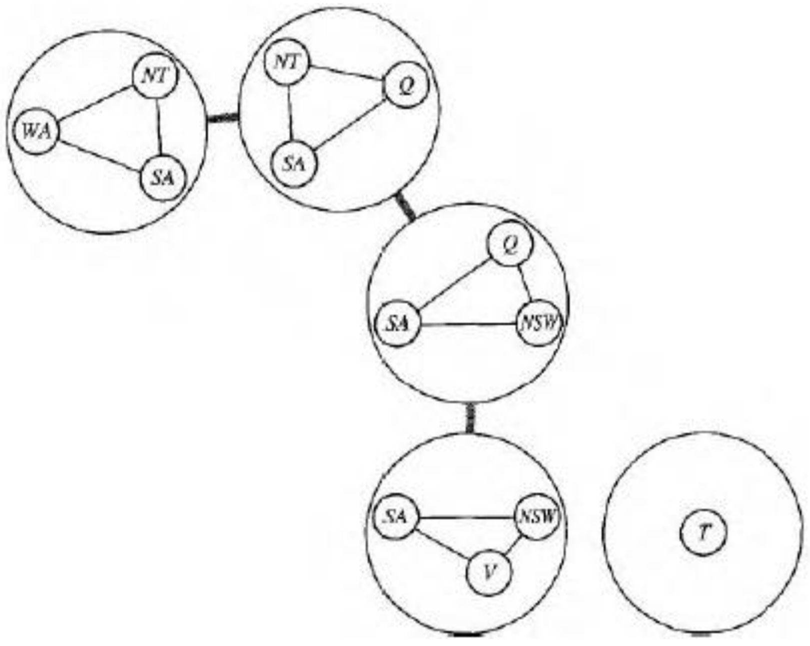
1. Choose a subset S from variables [csp] such that the constraint graph becomes a tree after removal of S. S is called a cycle cutset.
2. For each possible assignment to the variables in S that satisfies all constraints on S,

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

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

The second approach is based on constructing a tree decomposition of the constraint graph into a set of connected subproblems. Each subproblem is solved independently, and the resulting solutions are then combined. Like most divide-and-conquer algorithms, this works well if no subproblem is too large. Figure 5.12 shows a tree decomposition of the map coloring problem into five subproblems.

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



A tree decomposition of the constraint graph

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

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

Adversarial Search

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

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

Types of Games in AI:

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

are called the game with imperfect information, such as tic-tac-toe, Battleship, blind, Bridge, etc.

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

Zero-Sum Game

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

Zero-sum game: Embedded thinking

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

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

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

Optimal decision in games

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

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

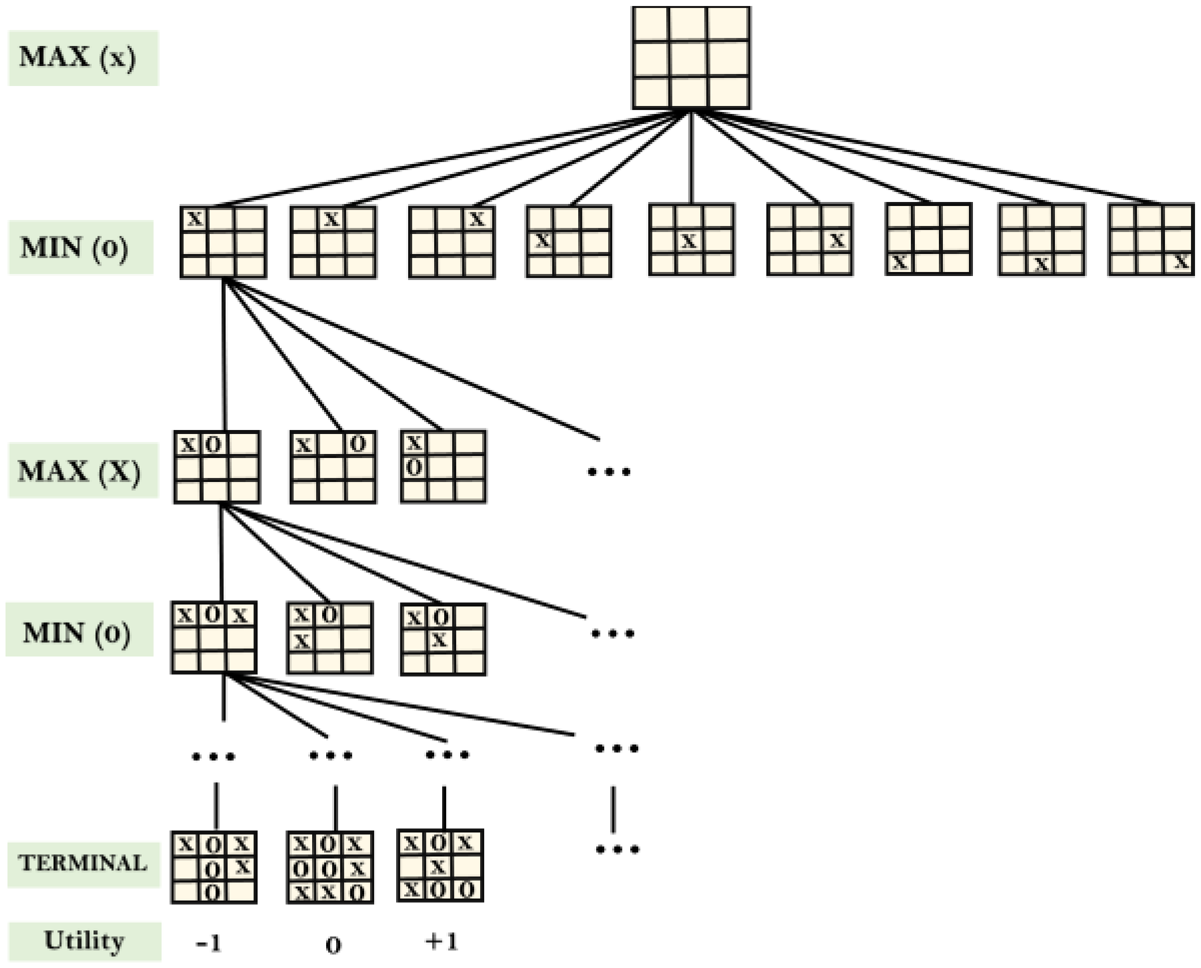
Game tree:

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

Example: Tic-Tac-Toe game tree:

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

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



Example Explanation:

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

- So in the game tree, we have a layer of Max, a layer of MIN, and each layer is called as Ply. Max place x, then MIN puts o to prevent Max from winning, and this game continues until the terminal node.
- In this either MIN wins, MAX wins, or it's a draw. This game-tree is the whole search space of possibilities that MIN and MAX are playing tic-tac-toe and taking turns alternately.

Hence adversarial Search for the minimax procedure works as follows:

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

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

For a state S MINIMAX(s) =

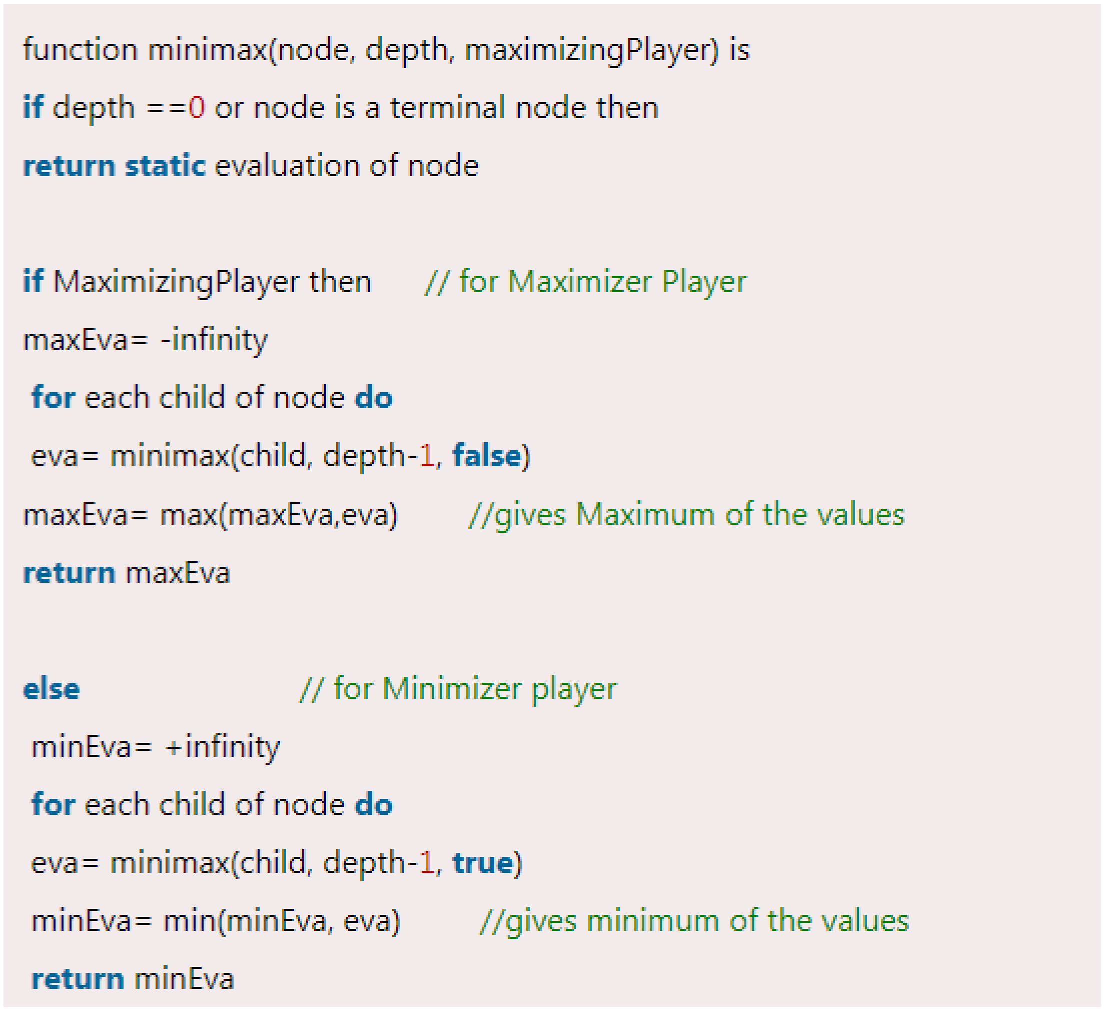
$$\begin{cases} \text{UTILITY}(s) & \text{If TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MIN.} \end{cases}$$

Mini-Max Algorithm in Artificial Intelligence

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

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

Pseudo-code for MinMax Algorithm:



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

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

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

Initial call:

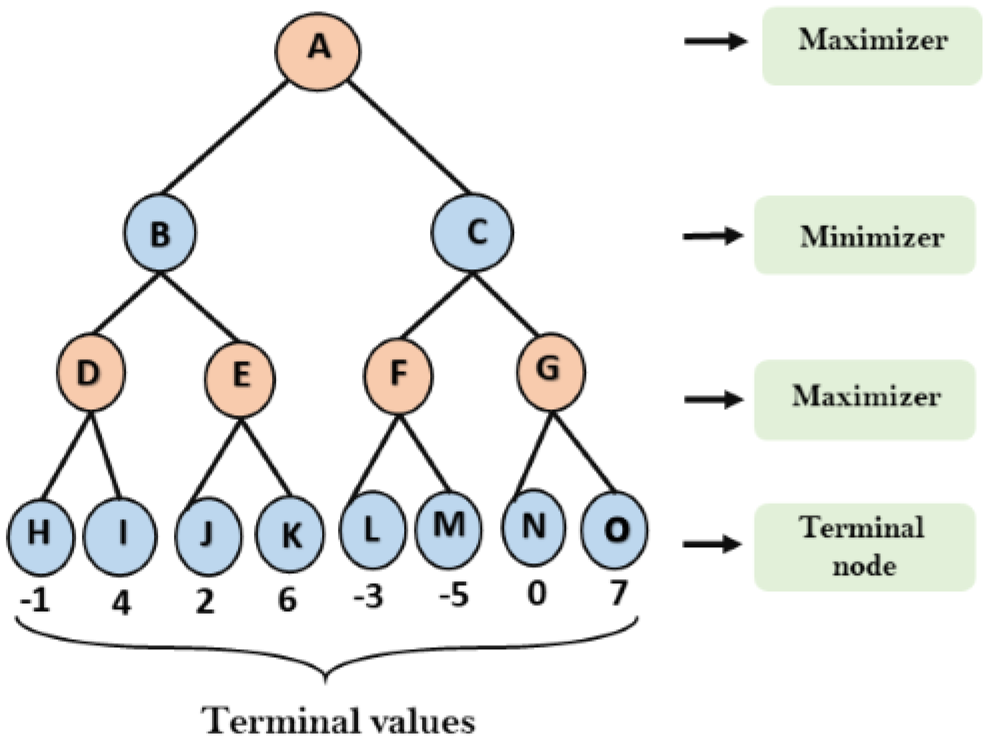
Minimax (node, 3, true)

Working of Min-Max Algorithm:

- The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.

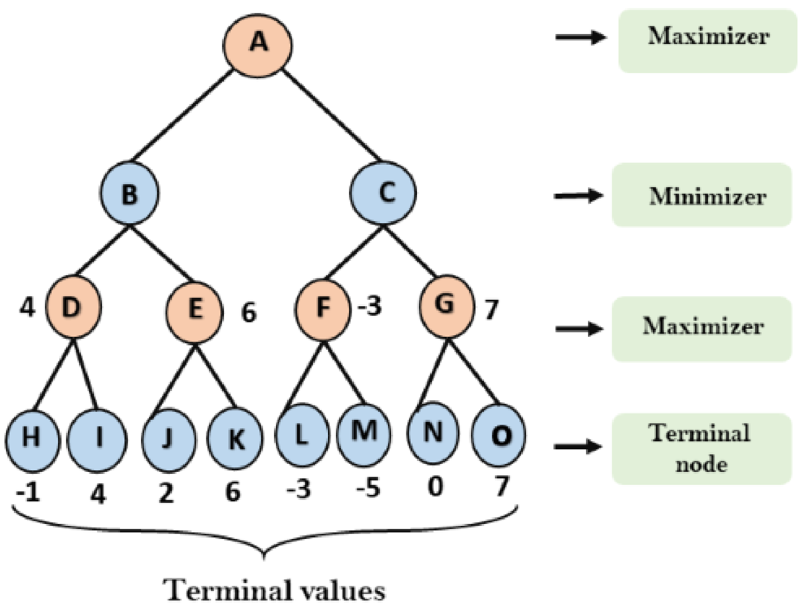
- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.
- At the terminal node, the terminal values are given so we will compare those values and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

Step-1: In the first step, the algorithm generates the entire game-tree and applies the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimize will take next turn which has worst-case initial value = +infinity.



Step 2: Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

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

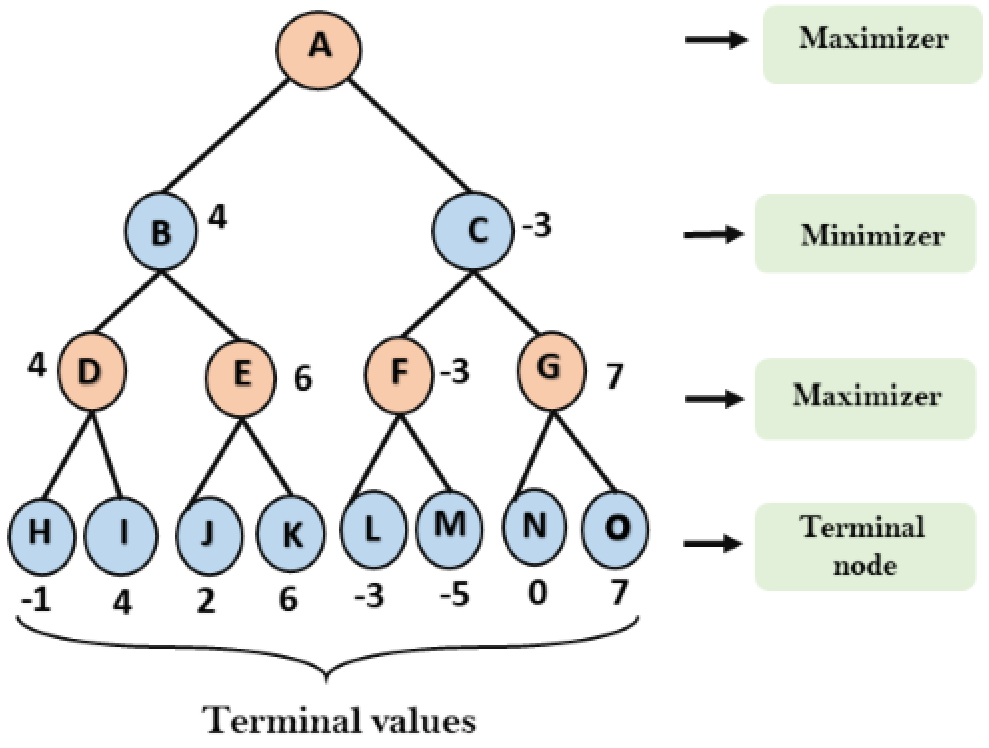


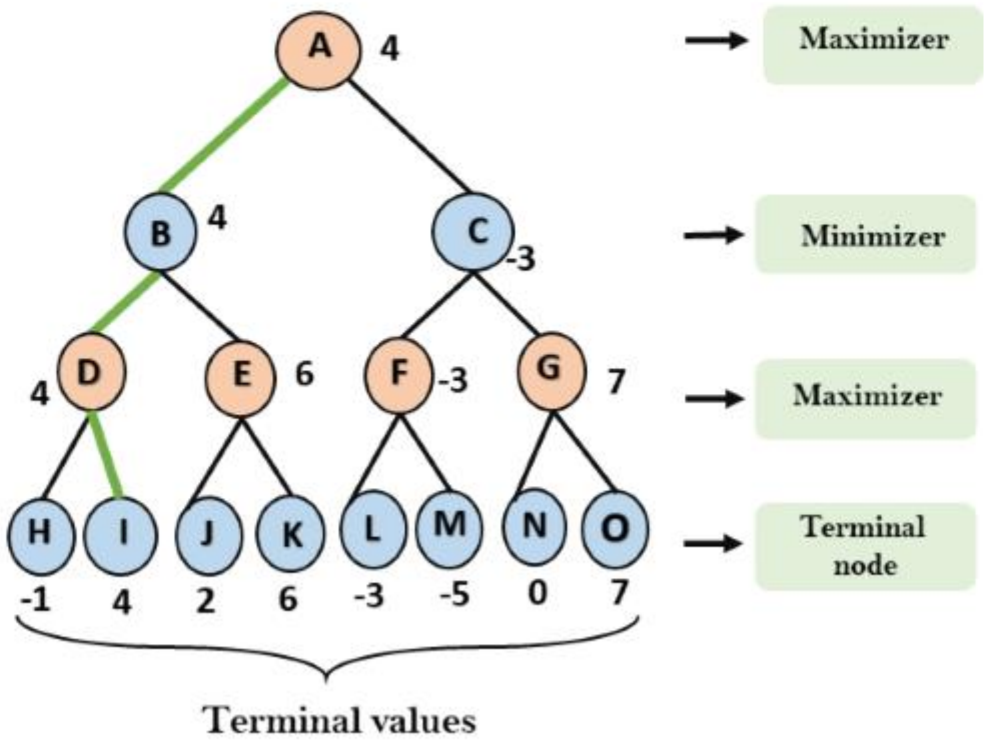
Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

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

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

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


A
B 4
C -3
4 D
E 6
F -3
G 7
H
I
J
K
L
M
N
O
-1 4 2 6 -3 -5 0 7
Terminal values
Maximizer
Minimizer
Maximizer
Terminal node



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

Properties of Mini-Max algorithm:

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

Limitation of the minimax Algorithm:

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

Alpha-Beta Pruning

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called pruning. This involves two threshold parameter Alpha and beta for future expansion, so it is called alpha-beta pruning. It is also called as Alpha-Beta Algorithm.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.
- The two-parameter can be defined as:

a. Alpha: The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞.

b. Beta: The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.

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

Condition for Alpha-beta pruning:

The main condition which required for alpha-beta pruning is $\alpha >= \beta$

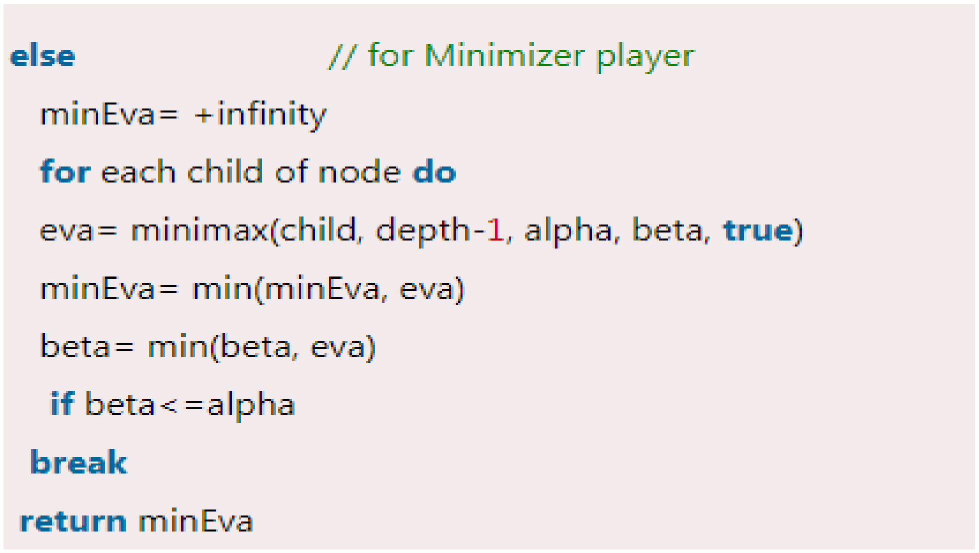
Key points about alpha-beta pruning:

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

Pseudo-code for Alpha-beta Pruning:

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

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

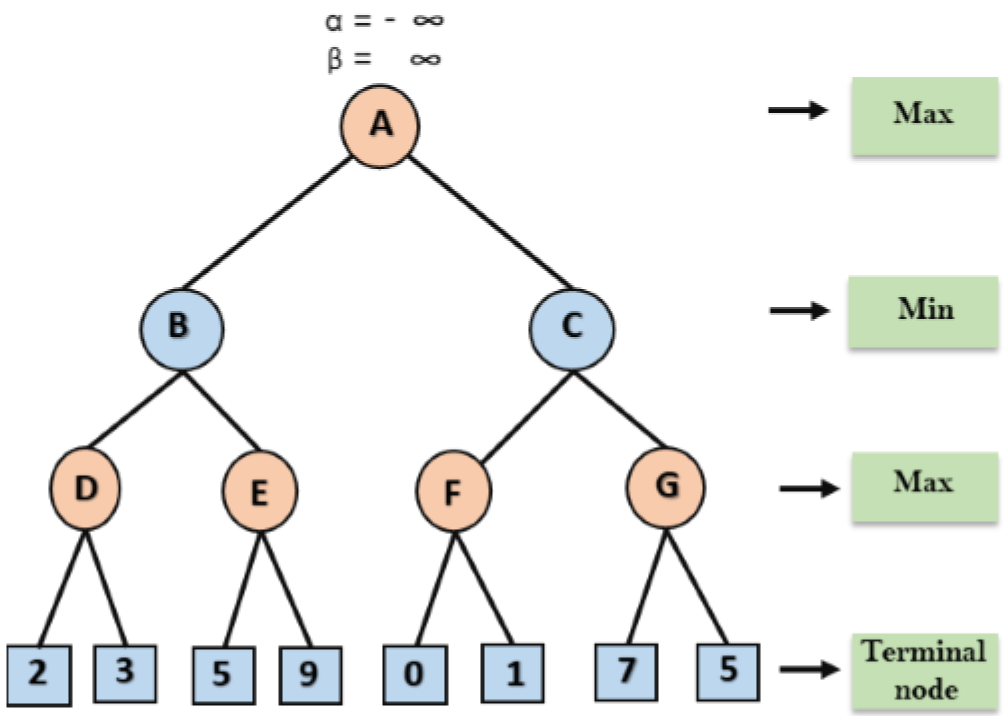


```
else                         // for Minimizer player
   minEva= +infinity
   for each child of node do
   eva= minimax(child, depth-1, alpha, beta, true)
   minEva= min(minEva, eva)
   beta= min(beta, eva)
    if beta<=alpha
  break
 return minEva
```

Working of Alpha-Beta Pruning:

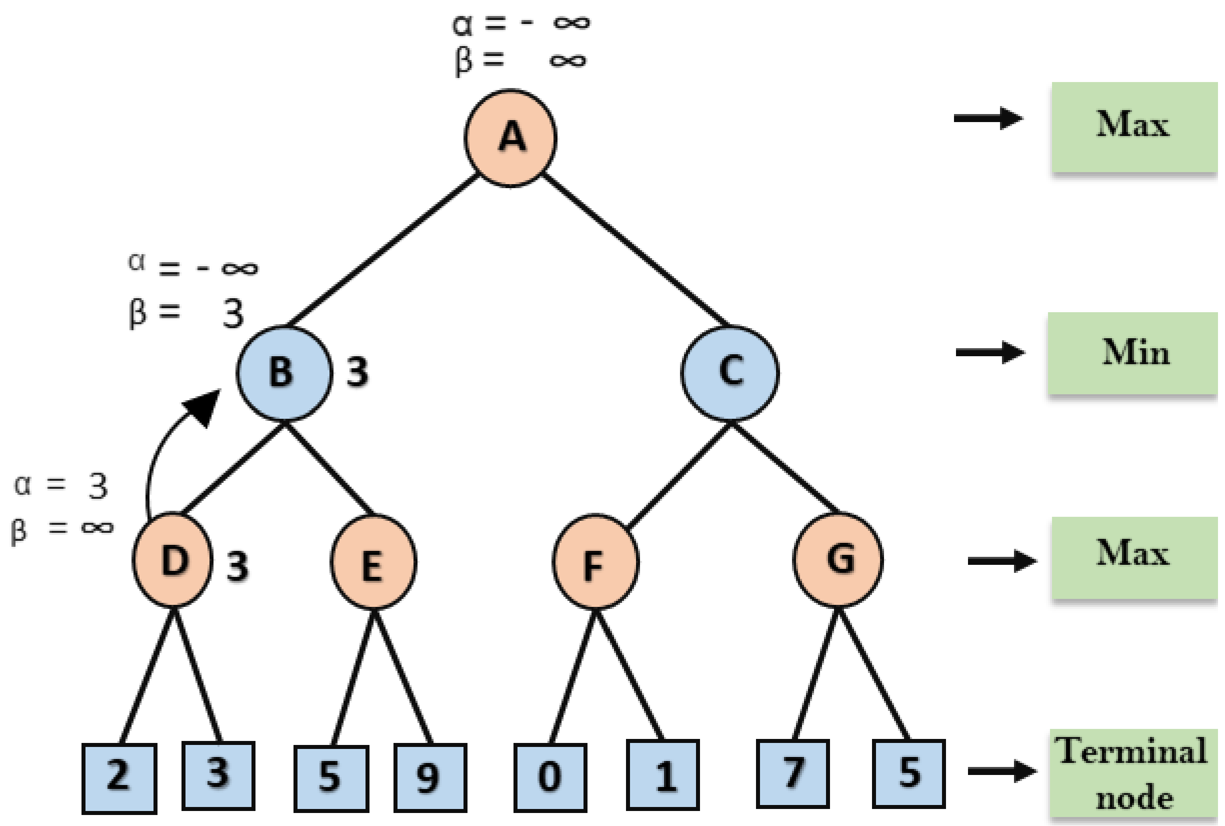
Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

Step 1: At the first step the, Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, 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 β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.

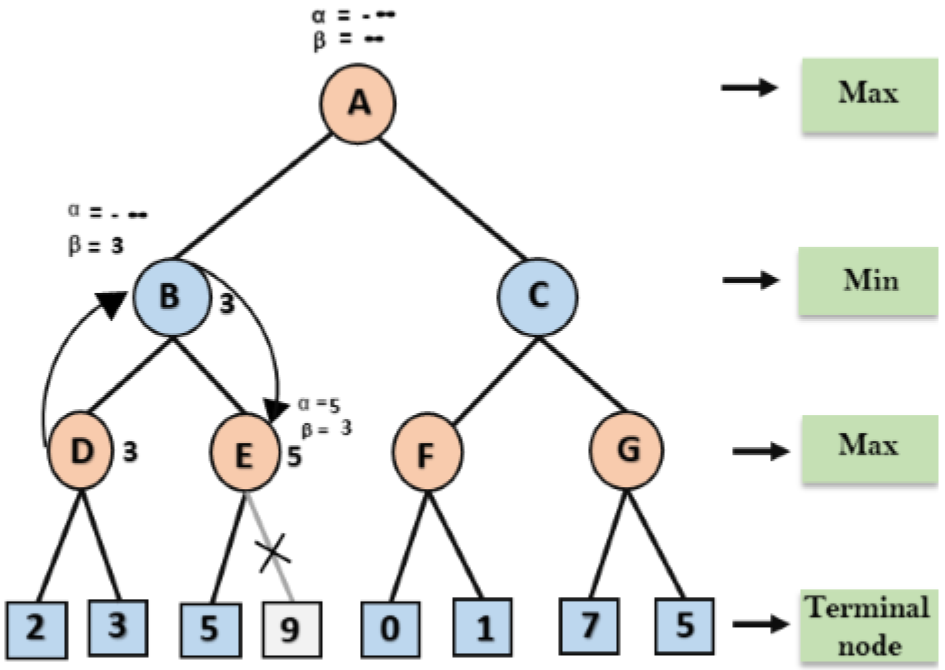


In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

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

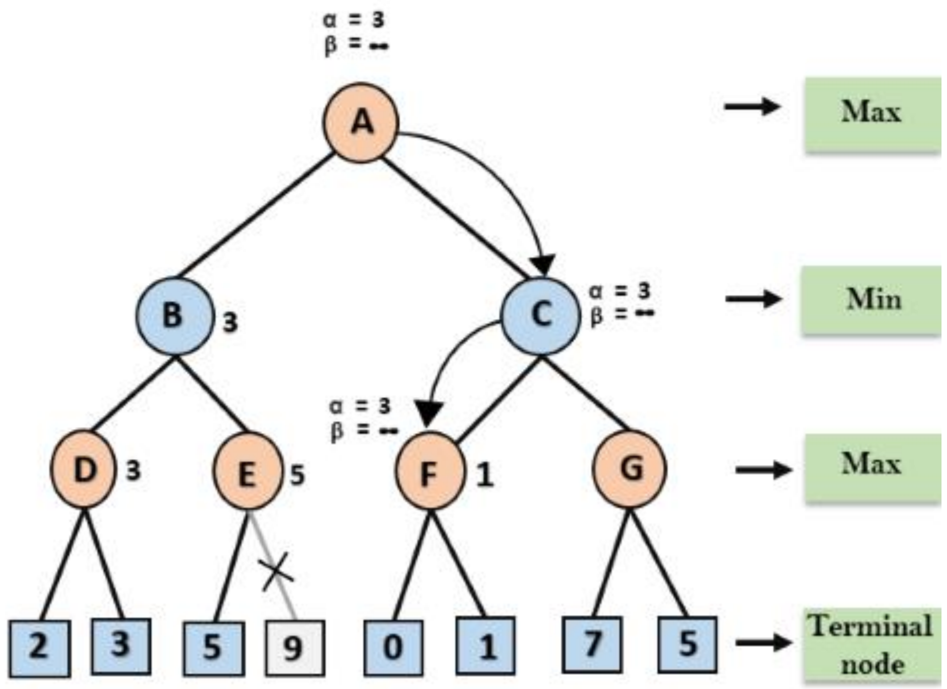
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

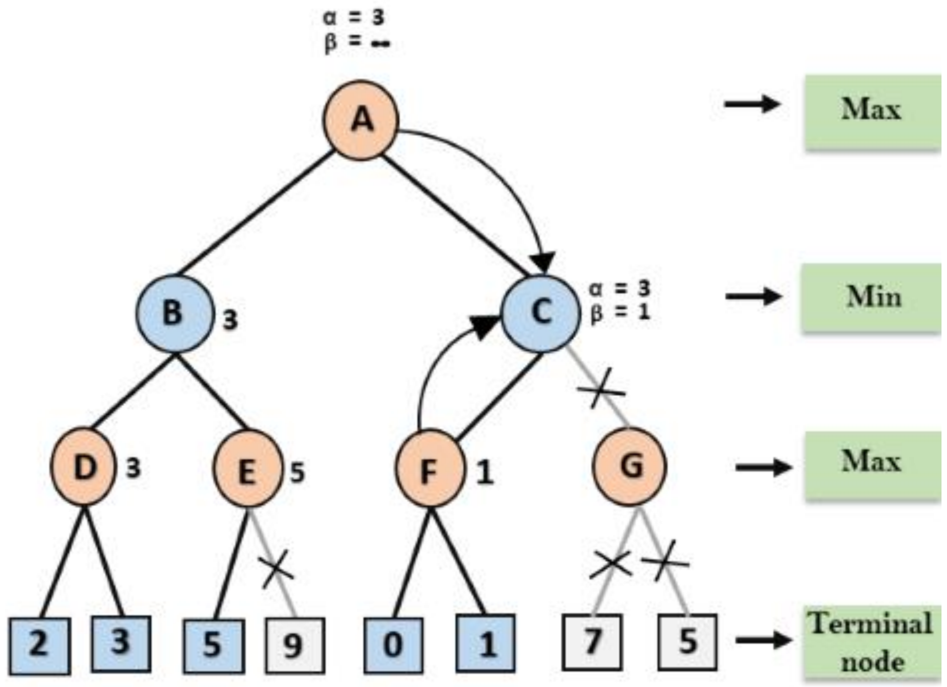
At node C, α=3 and β= +∞, and the same values will be passed on to node F.



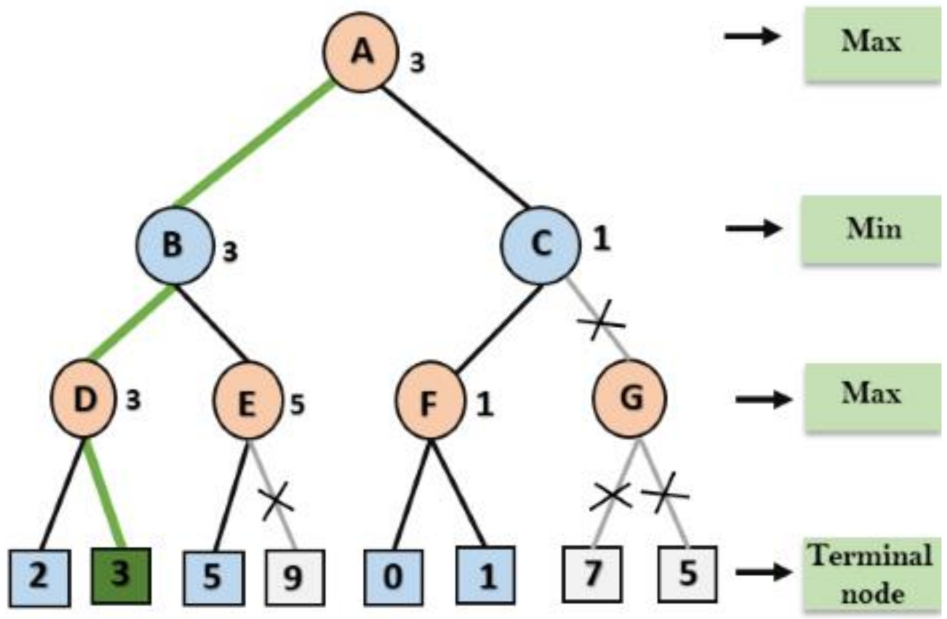
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.

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


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



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



Move Ordering in Alpha-Beta pruning:

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

It can be of two types:

- Worst ordering: In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is O(b^m).
- Ideal ordering: The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side

of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is $O(b^{m/2})$.

Rules to find good ordering:

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

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