What is Artificial Intelligence?

Artificial Intelligence is the development of computer systems that are able to perform tasks that would require human intelligence.

Examples of these tasks are visual perception, speech recognition, decision-making, and translation between languages.

Benefits of A.I.

The most important purpose of A.I. is to reduce human casualties in

→ Wars

→ Dangerous Workspaces

→ Car Accidents

→ Natural Disasters

Or to just make everyday life easier by helping with tasks such as:

→ Cleaning

→ Shopping

→ Transportation

What is Intelligence

Intelligence:

- ``the capacity to learn and solve problems'' (Webster dictionary)

- the ability to think and act rationally

Goal in Artificial Intelligence:

- build and understand intelligent systems/agents

- synergy between

• philosophy,

• psychology, and cognitive science

• computer science and engineering

• mathematics and physics

History of Artificial Intelligence

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

Maturation of Artificial Intelligence (1943-1952)

- **Year 1943:** The first work which is now recognized as AI was done by Warren McCulloch and Walter pits in 1943. They proposed a model of **artificial neurons.**

- **Year 1949:** Donald Hebb demonstrated an updating rule for modifying the connection strength between neurons. His rule is now called **Hebbian learning**.
- **Year 1950:** The Alan Turing who was an English mathematician and pioneered Machine learning in 1950. Alan Turing publishes **"Computing Machinery and Intelligence"** in which he proposed a test. The test can check the machine's ability to exhibit intelligent behavior equivalent to human intelligence, called a **Turing test**.

The birth of Artificial Intelligence (1952-1956)

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

The golden years-Early enthusiasm (1956-1974)

- **Year 1966:** The researchers emphasized developing algorithms which can solve mathematical problems. Joseph Weizenbaum created the first chatbot in 1966, which was named as ELIZA.
- **Year 1972:** The first intelligent humanoid robot was built in Japan which was named as WABOT-1.

The first AI winter (1974-1980)

- The duration between years 1974 to 1980 was the first AI winter duration. AI winter refers to the time period where computer scientist dealt with a severe shortage of funding from government for AI researches.
- During AI winters, an interest of publicity on artificial intelligence was decreased.

The emergence of intelligent agents (1993-2011)

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

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

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

- **Year 2012:** Google has launched an Android app feature "Google now", which w able to provide information to the user as a prediction.
- **Year 2014:** In the year 2014, Chatbot "Eugene Goostman" won a competition in t infamous "Turing test."
- **Year 2018:** The "Project Debater" from IBM debated on complex topics with tw master debaters and also performed extremely well.

Intelligent Agents:

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

Following are the main four rules for an AI agent:

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

What is an Agent?

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

- **Human-Agent:** A human agent has eyes, ears, and other organs which work for sensors and hand, legs, vocal tract work for actuators.
- **Robotic Agent:** A robotic agent can have cameras, infrared range finder, NLP for sensors and various motors for actuators.
- **Software Agent:** Software agent can have keystrokes, file contents as sensory input and act on those inputs and display output on the screen.

Environment

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

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

Features of Environment

1. Fully observable vs Partially Observable:

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

2. Deterministic vs Stochastic:

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

3. Single-agent vs Multi-agent

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

4. Static vs Dynamic:

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

5. Discrete vs Continuous:

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

Good behaviour: the concept of rationality

An agent should act as a Rational Agent. A rational agent is one that does the right thing that is the right actions will cause the agent to be most successful in the environment.

Performance measures

A performance measures embodies the criterion for success of an agent's behavior. As a general rule, it is better to design performance measures according to what one actually wants in the environment, rather than according to how one thinks the agent should behave.

Rationality

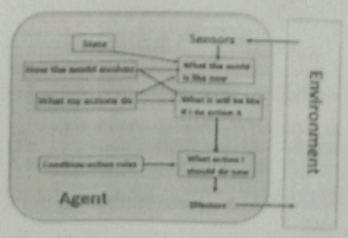
What is rational at any given time depends on four things:

The performance measure that defines the criterion of success.
The agent's prior knowledge of the environment.
The actions that the agent can perform.
The agent's percept sequence to date.

Nature of environment

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

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



Structure of an AI Agent

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

1. Agent = Architecture + Agent program

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

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

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

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

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

Types of search algorithms

Now let's see the types of the search algorithm.

Based on the search problems, we can classify the search algorithm as

- Uninformed search
- Informed search

Uninformed search algorithms

The uninformed search algorithm does not have any domain knowledge such as closeness, location of the goal state, etc. it behaves in a brute-force way.

It only knows the information about how to traverse the given tree and how to find the goal state. This algorithm is also known as the Blind search algorithm or Brute -Force algorithm.

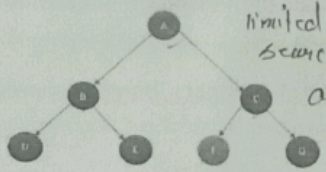
The uninformed search strategies are of six types.

They are-

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

1. Breadth-first search

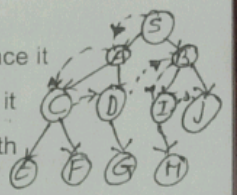
It is of the most common search strategies. It generally starts from the root node and examines the neighbor nodes and then moves to the next level. It uses First-in First-out (FIFO) strategy as it gives the shortest path to achieving the solution.



③. Depth limited search → Depth limited search is the new search algo. for uninformed search. The unbounded tree problem happens to appear in the depth first search algo. and it can be fixed by imposing a boundary or limited to the depth of the search domain.

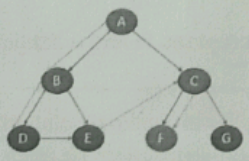
2. Depth-first search

The depth-first search uses Last-in, First-out (LIFO) strategy and hence it can be implemented by using stack. DFS uses backtracking. That is, it starts from the initial state and explores each path to its greatest depth before it moves to the next path.

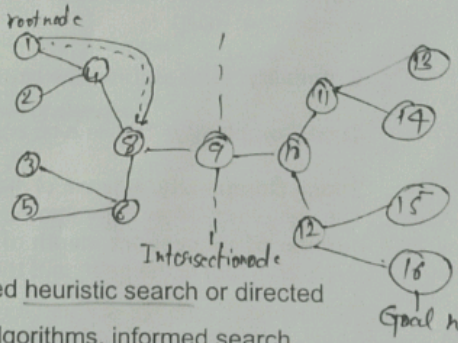


DFS will follow

Root node —> Left node —> Right node



4. Bidirectional search → Bdir search algo. runs two simultaneous searches one form initial state called as forward search and other from goal node called as backward-search to find the goal node.



Informed search algorithms

The informed search algorithm is also called heuristic search or directed search. In contrast to uninformed search algorithms, informed search algorithms require details such as distance to reach the goal, steps to reach the goal, cost of the paths which makes this algorithm more efficient.

Here, the goal state can be achieved by using the heuristic function.

The heuristic function is used to achieve the goal state with the lowest cost possible. This function estimates how close a state is to the goal.

Let's discuss some of the informed search strategies.

1. Greedy best-first search algorithm

Greedy best-first search uses the properties of both depth-first search and breadth-first search. Greedy best-first search traverses the node by selecting the path which appears best at the moment. The closest path is selected by using the heuristic function.

A search algorithm*

A* search algorithm is a combination of both uniform cost search and greedy best-first search algorithms. It uses the advantages of both with better memory usage. It uses a heuristic function to find the shortest path. A* search algorithm uses the sum of both the cost and heuristic of the node to find the best path.

Measuring problem-solving performance

the **performance measure** of an algorithm should be measured. Consequently, There are four ways to measure the performance of an algorithm:
Completeness: It measures if the algorithm guarantees to find a solution (if any solution exist).

Optimality: It measures if the strategy searches for an optimal solution.

Time Complexity: The time taken by the algorithm to find a solution.

Space Complexity: Amount of memory required to perform a search.
The complexity of an algorithm depends on **branching factor** or **maximum number of successors**, **depth of the shallowest goal node** (i.e., number of steps from root to the path) and **the maximum length of any path in a state space.**

Local Search Algorithms and Optimization Problem

"local search algorithms" **where the path cost does not matters, and** only focus on solution-state needed to reach the goal node.

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

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

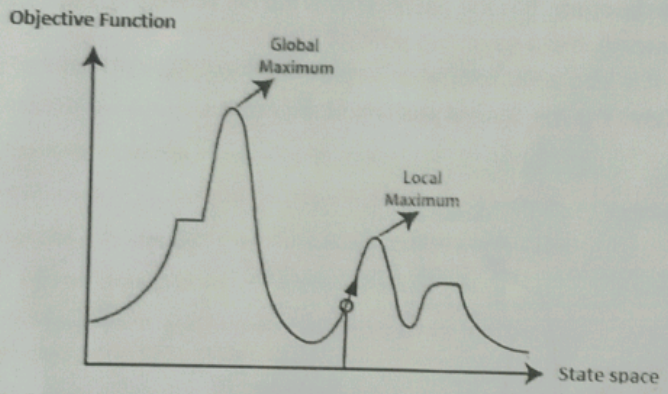
- Local search algorithms use a very little or constant amount of memory as they operate only on a single path.
- Most often, they find a reasonable solution in large or infinite state spaces where the classical or systematic algorithms do not work.

Working of a Local search algorithm

Let's understand the working of a local search algorithm with the help of an example:

Consider the below state-space landscape having both:

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



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

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

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

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

Below are some different types of local searches:

- Hill-climbing Search

- Simulated Annealing
- Local Beam Search

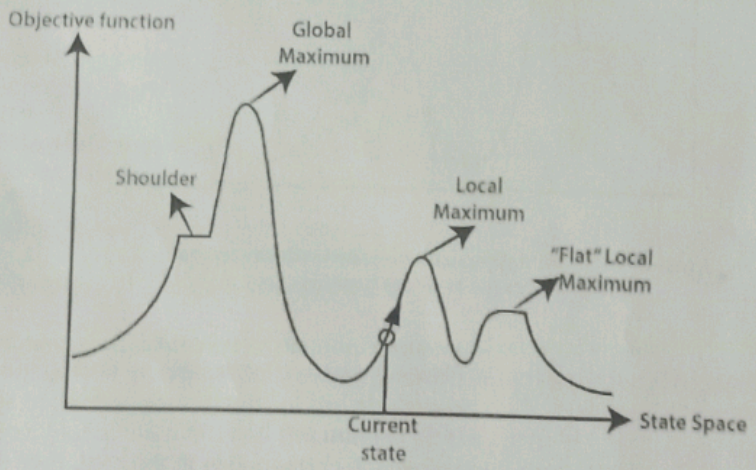
Hill Climbing Algorithm in AI

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

State-space Landscape of Hill climbing algorithm

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

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



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

Searching with Non-deterministic actions

- So far: fully-observable, deterministic worlds.

– Agent knows exact state. All actions always produce one outcome.

– Unrealistic?

• Real world = partially observable, non-deterministic

– Percepts become useful: can tell agent which action occurred

– Goal: not a simple action sequence, but contingency plan

• Example: Vacuum world, v2.0

– Suck(p1, dirty)= (p1,clean) and sometimes (p2, clean)

– Suck(p1, clean)= sometimes (p1,dirty)

– If start state=1, solution= [Suck, if(state=5) then [right,suck]]

Searching with Partial Observations

• Previously: Percept gives full picture of state

– eg. Whole chess board, whole boggle board, entire robot maze

• Partial Observation: incomplete glimpse of current state

– Agent's percept: zero <= percept < full state

– Consequence: we don't always know exactly what state we're in.

• Concept of believe state – set of all possible states agent could be in.

• Find a solution (action sequence) that the leads to goal

– Actions applied to a believe state à new believe state based on union of that action applied to all real states within believe state

Known vs Unknown Environment

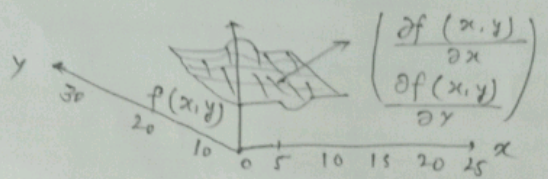
- Known and unknown are not actually a feature of an environment, but it is an agent's state of knowledge to perform an action.
- In a known environment, the results for all actions are known to the agent. While in unknown environment, agent needs to learn how it works in order to perform an action.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

Online search agents ⟶ An online search agent interleaves computation and action: first it takes action, then it observes the environment and computes the next action. Online search is a good idea in dynamic & semidynamic domains — domains where there is a penalty for sitting around and computing too long. Online search is also helpful in non deterministic domains because it allows the agent to focus its computational efforts on the contingenc that actually arise rather than those that might happen but probabal won't.

Online search is neccessary idea for unknown environments, where the agent doesn't know what state exist or what its action do.

⟶ Searching in Continuous state space ——— Write down on B3

Observation: so, far, states have been discrete "moves" apart
- Each "move" corresponds to an "atomic action"
- But the real world is generally a continuous space.
- What if we want to plain in real world space, rather than logical space?



Example: Suppose we want to cite three airports in Romaniya:

Ab
- 6-D state space defined by $(x_1, y_1), (x_2, y_2), (x_3, y_3)$
- objective function $f(x_1, y_1, x_2, y_2, x_3, y_3)$ = sum of squared distance from each city to nearest airport (six dimensional search space)

Approaches:

Discretization methods turn continuous space into discrete space
- e.g. empirical gradient search consider $\pm\delta$ change in each co-ordinate
- if you make δ small enough, you get needed accuracy.

Games: Adversarial search problems. In AI The most Common games are deterministic, fully observable environment in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite.

→ A Game can be formally defined as a kind of search Problem with the following elements:-

→ S_0: The initial state, which specifies how the game is set up the start.

→ Player (s) ⇒ Defines which player has the move in a state.

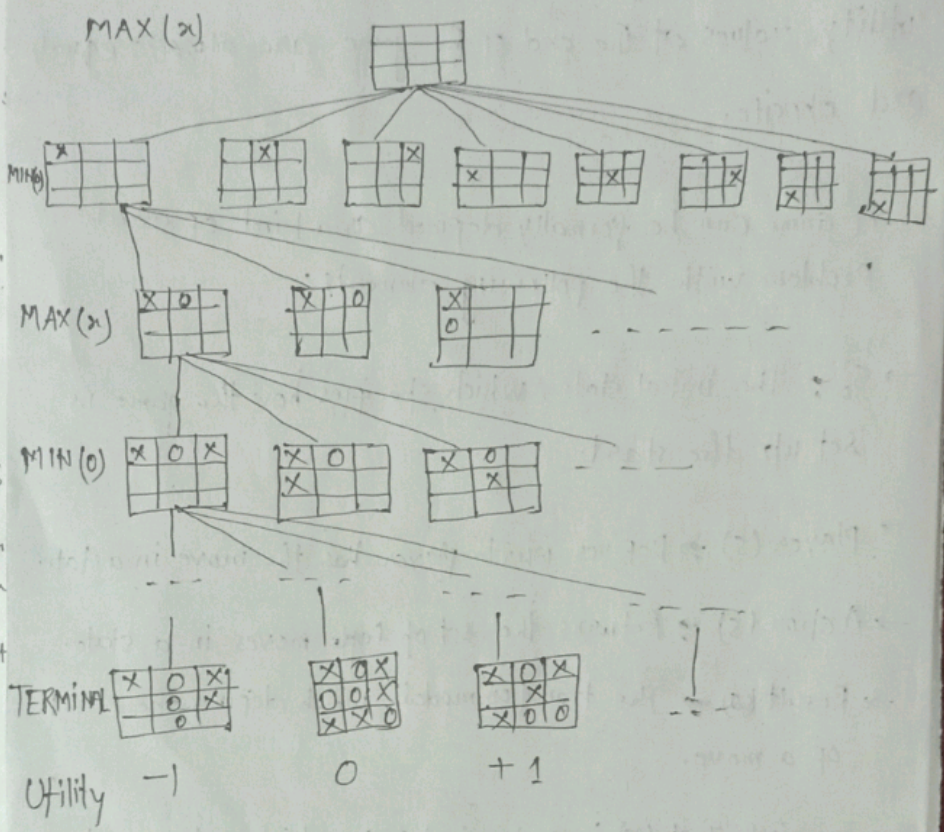
→ Action (s) ⇒ Returns the set of legal moves in a state.

→ Result (s,a) ⇒ The transition model, which defines the result of a move.

→ Terminal-Test (s): A terminal test, which is true, when the game is over and false otherwise. State ~~whether~~ where the game has ended are called terminal states.

→ Utility (s, p): A utility function applied to terminal states defines the final numeric value for a game that ends in terminal state s for a player p.

Game tree :→ Game tree defined by the initial state, Action and result function. a tree where the nodes are game states and edges are moves.



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

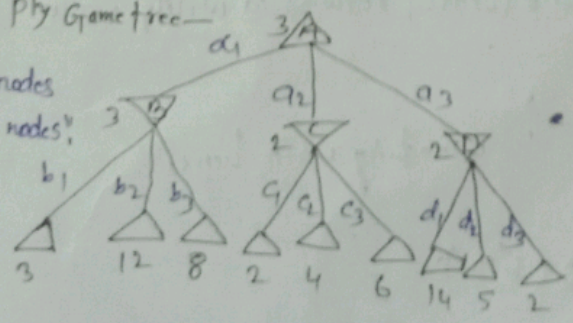
Optimal decision in games ——

optimal solution — In adversarial search, The optimal solution is a contigent strategy, which specifies MAX (The player on our side)'s move in the initial state, Then MAX's move in the states resulting from every possible response by MIN (The oponent), Then MAX's moves in the states resulting from every possible response by MIN to those moves, and so on.

Minimax value :—— The minimax value of a node is the utility (for MAX) of being in the corresponding state assuming that both players play optimally from there to the end of the game. The minimax value is just its utility.

$$\text{MINMAX}(s) = \begin{cases} \text{utility}(s) & \text{if terminal-Tst}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINMAX}(\text{RESULT}(s,a)) & \text{if Player}(s) = \text{max} \\ \min_{a \in \text{Actions}(s)} \text{MINMAX}(\text{RESULT}(s,a)) & \text{if Player}(s) = \text{min} \end{cases}$$

A Two ply Game tree—



The △ nodes are "MAX nodes", in which it is MAX's turn to move. and the ▽ nodes are "min nodes" The terminal nodes show the utility values for max.

The other nodes are labled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value and MIN's best reply is b_1, because it leads to the state with the lowest minimax values.

The minimax algorithm ⟶ The minimax algorithm computes the minimax decision from the current state. The recursion proceeds all the way down to the leaves of the tree and then the minmax value are backed up through the tree as the recursion unwinds.

function MINMAX-DECISION(state) return an action.

return $\arg\max_{a \in \text{Actions}(s)}$ MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns an action a utility value.

if Terminal-Test(state) then return Utility(state)

v ← −∞

for each a in Actions(state) do

v ← Max(v, MIN-VALUE(RESULT(s,a)))

return v

function MIN-VALUE(state) returns a utility value

: Same thing write here.

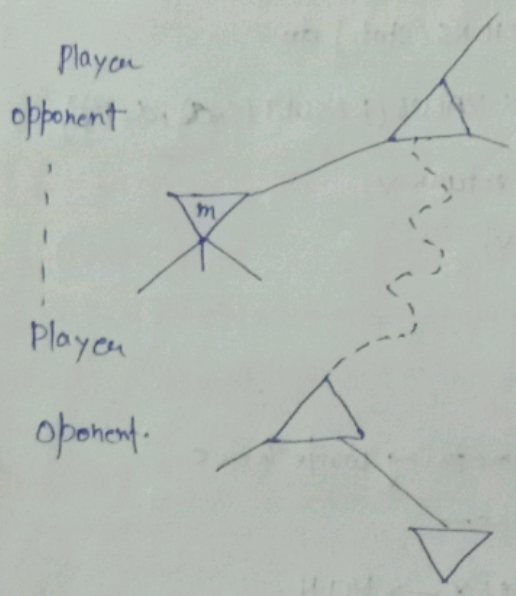
Time Complexity $O(b^m)$

Space Complexity $O(bm)$, $O(m)$

Alpha-beta pruning — The alpha beta search algorithm compute the same optimal move as minimax, but achieves much greater efficiency by eleminating subtrees that are probabily irrelevent.

When applied to a standaad minimax tree, alpha beta pruning retuins the same move as minimax would. but. prunes away branches that cannot possibly influence the final decision.

The general principle: Conciden a node somewhere in the tree such that player has a choice of moving to that node. if player has a better choice 'm' either at the parrent node OF 'n' or any choice point further up, Then n will never be reached in actual play. so once we have found out enough about n to reach this conclusion, we can prune it.



Alpha-beta search update the values of α and β as it goes along and prunes the remaining branches at a node as soon as the current node is known to be worse than the current α and β value for MAX and MIN respectively.

α = The value of the best choice we have found so far at any choice point along the path for MAX.

β = The value of the best choice we have found so far at any choice point along the path for MIN.

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

$v \leftarrow$ MAX-VALUE(state, $-\infty$, $+\infty$)

return the action in ACTIONS(state) with value v

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

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

$v \leftarrow -\infty$

for each a in ACTIONS(state) do

$v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))

if $v \geq \beta$ then return v

$\beta \leftarrow$ MIN(β, v)

return v

funct . . . something write there

some changes like—

MAX → MIN

— → +

> → <

$\alpha \rightarrow \beta$

Imperfect real time decisions ——

Because moves must be made in a reasonable amount of time, usually it is not feasible to consider the whole game tree, so program should cut the search off at some point earlier and apply a heuristic evalution function to states in the search, effectively turning nonterminal nodes into terminal ~~nodes~~ leaves.

i.e. Alter minimax or alpha-beta in 2 ways:

1) Replace the utility function by heuristic evalution function Eval which stimates the position's utility.

2) Replace the terminal test by a cutoff test that decides when to apply EVAL.

H-MINIMAX(s, d) =

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

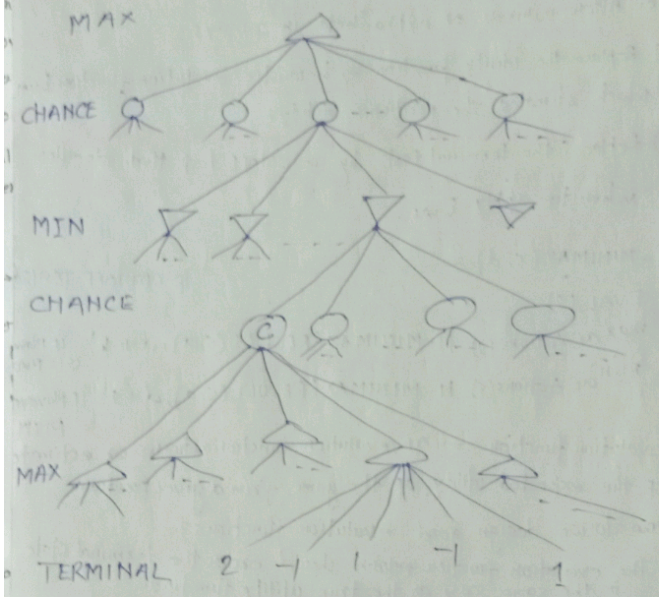
Evalution function — An evalution function returns an estimate of the expected utility of the game from a given position.

How do we design good evalution functions?

1. The evalution function ~~return~~ should order the terminal state in the same way as the true utility function.
2. The computation must not take too long.
3. For nonterminal states, the evalution function should be strongly correlated with the actual chances of wining.

Stochastic games——

stochastic games: Games mirror unpredictability by including a random element, such as the throwing of dice. Game of chance can be handled by an extension to the minimax algorithm that evaluates a chance node by taking the average utility of all its children, weighted by the probability of each node.



Schematic game tree for a backgammon position.

Partially observable games:—
optimal play in games of imperfect information, such as kriegspiel and bridge, requires reasoning about the current and future belief state of each player. A simple aproximation can be obtained by averaging the value of an action over each possible configuration of missing info.

Kriegspiel :— A partially observeable varient of chess in which piecess can move but are completly invisible to the oponent.

Belief state :— The set of all logically possible board stat given the complete histry of percepts to date.

keeping track of the belief state as the game processes is exact the problem of state estimation —

Strategy :— Instead of specifying a move to make for each possible move the oponent might make we need a move for every possible percept sequence that might be recieved.

Guaranteed checkmate/wining strategy :— One that for each possible percept sequence, leads to an actual checkmate for eve possible board state in the current belief state, regardless of how the oponent moves.

Probabilistic checkmates :— Such checkmates are still required to work in every board state in the belief state, They are probabilistic with respect to randomization of the winning play moves.

Alternative approach ——

Standard approach : based on minimax, evaluation function and alpha-beta.

Alternatives:

1. Heuristic minimax :— select an optimal move in a given search tree provided that the leaf node evaluations are exactly correct but in reality, evaluation are usually crude estimate of the value of a position and can be considerd to have large errors associated with them.

2. The search algorithm that generates the tree :— A good search algorithm should select node expansions of high utility — ones that are likely to lead to the discovery of a significantly better move. if there are no node expansions whose utility is higher than their cost, then the algorithm should stop searching and make a move.

3. The nature of search itself :— To combine two kinds of algo. into a robust and efficient system. A algorithms for heuristic search and for game playing generate sequences of concrete states; b. goal-directed reasoning or plaining used by humans, sometime eliminate combinatorial search altogether.

Adversarial Search in Artificial Intelligence

AI Adversarial search: Adversarial search is a **game-playing** technique where the agents are surrounded by a competitive environment. A conflicting goal is given to the agents (multiagent). These agents compete with one another and try to defeat one another in order to win the game. Such conflicting goals give rise to the adversarial search. Here, game-playing means discussing those games where **human intelligence** and **logic factor** is used, excluding other factors such as **luck factor**. **Tic-tac-toe, chess, checkers**, etc., are such type of games where no luck factor works, only mind works. Mathematically, this search is based on the concept of **'Game Theory.'** ***According to game theory, a game is played between two players. To complete the game, one has to win the game and the other looses automatically.'***

There are following types of adversarial search:

- **Minmax Algorithm** 2/10
- **Alpha-beta Pruning.** 3/10

Constraint Satisfaction Problems in Artificial Intelligence

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

- **X:** It is a set of variables.
- **D:** It is a set of domains where the variables reside. There is a specific domain for each variable.
- **C:** It is a set of constraints which are followed by the set of variables.

In constraint satisfaction, domains are the spaces where the variables reside, following the problem specific constraints. These are the three main elements of a constraint satisfaction technique. The constraint value consists of a pair of **{scope, rel}**. The **scope** is a tuple of variables which participate in the constraint and **rel** is a relation which includes a list of values which the variables can take to satisfy the constraints of the problem.

Solving Constraint Satisfaction Problems

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

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

A state in state-space is defined by assigning values to some or all variables such as
$\{X_1=v_1, X_2=v_2,$ and so on…$\}$.
An assignment of values to a variable can be done in three ways:

- **Consistent or Legal Assignment:** An assignment which does not violate any constraint or rule is called Consistent or legal assignment.
- **Complete Assignment:** An assignment where every variable is assigned with a value, and the solution to the CSP remains consistent. Such assignment is known as Complete assignment.
- **Partial Assignment:** An assignment which assigns values to some of the variables only. Such type of assignments are called Partial assignments.
- **Define CSP**
-
- CSPs represent a state with a set of variable/value pairs and represent the conditions for a solution by a set of constraints on the variables. Many important real-world problems can be described as CSPs.
-
- **CSP (constraint satisfaction problem):** Use a factored representation (a set of variables, each of which has a value) for each state, a problem that is solved when each variable has a value that satisfies all the constraints on the variable is called a CSP.
-
- A CSP consists of 3 components:
-
- X is a set of variables, $\{X_1, ..., X_n\}$.
-
- D is a set of domains, $\{D_1, ..., D_n\}$, one for each variable.
-
- Each domain D_i consists of a set of allowable values, $\{v_1, ..., v_k\}$ for variable X_i.
-
- C is a set of constraints that specify allowable combination of values.
- **Constraint propagation: Inference in CSPs**
-
- A number of inference techniques use the constraints to infer which variable/value pairs are consistent and which are not. These include node, arc, path, and k-consistent.
-
- **constraint propagation**: Using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on.
-
- **local consistency**: If we treat each variable as a node in a graph and each binary constraint as an arc, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph.
- There are different types of local consistency:
-
- **Node consistency**
- A single variable (a node in the CSP network) is **node-consistent** if all the values in the variable's domain satisfy the variable's unary constraint.
- We say that a network is node-consistent if every variable in the network is node-consistent.
-
- **Arc consistency**
- A variable in a CSP is **arc-consistent** if every value in its domain satisfies the variable's binary constraints.
- X_i is arc-consistent with respect to another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j).
- A network is arc-consistent if every variable is arc-consistent with every other variable.

Path consistency

Path consistency: A two-variable set $\{X_i, X_j\}$ is path-consistent with respect to a third variable X_m if, for every assignment $\{X_i = a, X_j = b\}$ consistent with the constraint on $\{X_i, X_j\}$, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$.
Path consistency tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables.

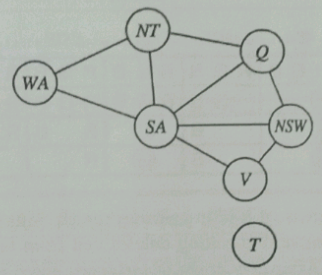
K-consistency

K-consistency: A CSP is k-consistent if, for any set of k-1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable.

Backtracking search for CSPs

Backtracking search, a form of depth-first search, is commonly used for solving CSPs. Inference can be interwoven with search.
Commutativity: CSPs are all commutative. A problem is commutative if the order of application of any given set of actions has no effect on the outcome.
Backtracking search: A depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.
Backtracking algorithm repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value.
There is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, action function, transition model, or goal test.
BACKTRACKING-SARCH keeps only a single representation of a state and alters that representation rather than creating a new ones.

Variable and value ordering



SELECT-UNASSIGNED-VARIABLE

Variable selection—fail-first

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

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

[Powerful guide]

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

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

ORDER-DOMAIN-VALUES

Value selection—fail-last

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

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

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

The **minimum-remaining-values** and **degree** heuristic are domain-independent methods for deciding which variable to choose next in a backtracking search. The **least-constraining-value** heuristic helps in deciding which value to try first for a given variable.

2. Interleaving search and inference

INFERENCE

forward checking: [One of the simplest forms of inference.] Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for X.
There is no reason to do forward checking if we have already done arc consistency as a preprocessing step.

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

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

Advantage: For many problems the search will be more effective if we combine the MRV heuristic with forward checking.
Disadvantage: Forward checking only makes the current variable arc-consistent, but doesn't look ahead and make all the other variables arc-consistent.

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

Intelligent backtracking

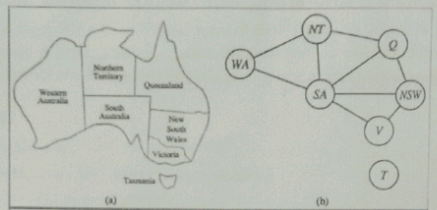


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

chronological backtracking: The BACKGRACKING-SEARCH in Fig 6.5. When a branch of the search fails, back up to the preceding variable and try a different value for it. (The most recent decision point is revisited.)
e.g.
Suppose we have generated the partial assignment {Q=red, NSW=green, V=blue, T=red}.
When we try the next variable SA, we see every value violates a constraint.
We back up to T and try a new color, it cannot resolve the problem
.

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

Conflict set for a variable: A set of assignments that are in conflict with some value for that variable.
(e.g. The set {Q=red, NSW=green, V=blue} is the conflict set for SA.)

backjumping method: Backtracks to the most recent assignment in the conflict set.
(e.g. backjumping would jump over T and try a new value for V.)

Local search for CSPs

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

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

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

Motivation of Knowledge-Based Agents

- Humans know things and do reasoning, which are important for artificial agents

- Knowledge-based agents benefit from knowledge expressed in general forms, combining information to suit various purposes
- Knowledge and reasoning is important when dealing with partially observable environments
- Understanding natural language requires inferring hidden states
- Flexibility for accepting new tasks

Logical Reasoning

- In each case where the agent draws a conclusion from the available information, that conclusion is guaranteed to be correct if the available information is correct
- The above is the fundamental of logical reasoning

Inference in First-Order Logic

Inference in First-Order Logic is used to deduce new facts or sentences from existing sentences. Before understanding the FOL inference rule, let's understand some basic terminologies used in FOL.

Substitution:

Substitution is a fundamental operation performed on terms and formulas. It occurs in all inference systems in first-order logic. The substitution is complex in the presence of quantifiers in FOL. If we write **F[a/x]**, so it refers to substitute a constant "**a**" in place of variable "**x**".

Equality:

First-Order logic does not only use predicate and terms for making atomic sentences but also uses another way, which is equality in FOL. For this, we can use **equality symbols** which specify that the two terms refer to the same object.

Example: ¬(x=y) which is equivalent to x ≠y.

Propositional logic in Artificial intelligence

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

Example:

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

Following are some basic facts about propositional logic:

- Propositional logic is also called Boolean logic as it works on 0 and 1.
- In propositional logic, we use symbolic variables to represent the logic, and we can use any symbol for a representing a proposition, such A, B, C, P, Q, R, etc.
- Propositions can be either true or false, but it cannot be both.
- Propositional logic consists of an object, relations or function, and **logical connectives**
- These connectives are also called logical operators.

Syntax of propositional logic:

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

a. **Atomic Propositions**

b. **Compound propositions**

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

Example:

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

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

Example:

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

What is Unification?

Unification is **a key component of all first-order inference algorithms.** It returns fail if the expressions do not match with each other. The substitution variables are called Most General Unifier or MGU.

- Unification is a process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process.
- It takes two literals as input and makes them identical using substitution.

Conditions for Unification:

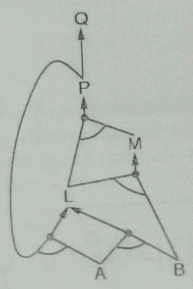
Following are some basic conditions for unification:

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

Forward Chaining

- Idea:
  - fire any rule whose premises are satisfied in the *KB*
  - add its conclusion to the *KB*, until query is found

$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
A
B



and-or graph

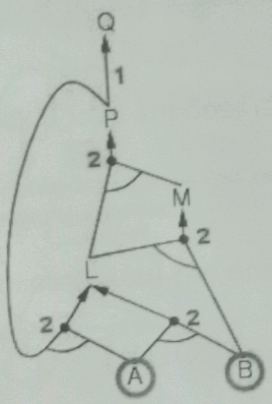
Forward Chaining Algorithm

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  local variables: count, a table, indexed by clause, initially the number of premises
                   inferred, a table, indexed by symbol, each entry initially false
                   agenda, a list of symbols, initially the symbols known to be true

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

  return false
```

- Forward chaining is sound and complete for Horn KB

Forward Chaining Example

$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
A
B

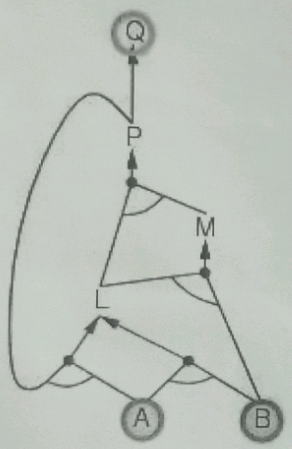


Backward Chaining

- Idea: work backwards from the query q:
  - to prove q by BC
    - check if q is known already, or
    - prove by BC all premises of some rule concluding q
- Avoid loops: check if new subgoal is already on the goal stack
- Avoid repeated work: check if new subgoal
  - has already been proved true, or
  - has already failed

Backward Chaining Example

$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
A
B



Forward vs. Backward Chaining

- FC is data-driven, automatic, unconscious processing
  - e.g., object recognition, routine decisions
- May do lots of work that is irrelevant to the goal
- BC is goal-driven, appropriate for problem-solving,
  - e.g., Where are my keys? How do I get into a PhD program?
- Complexity of BC can be much less than linear in size of KB