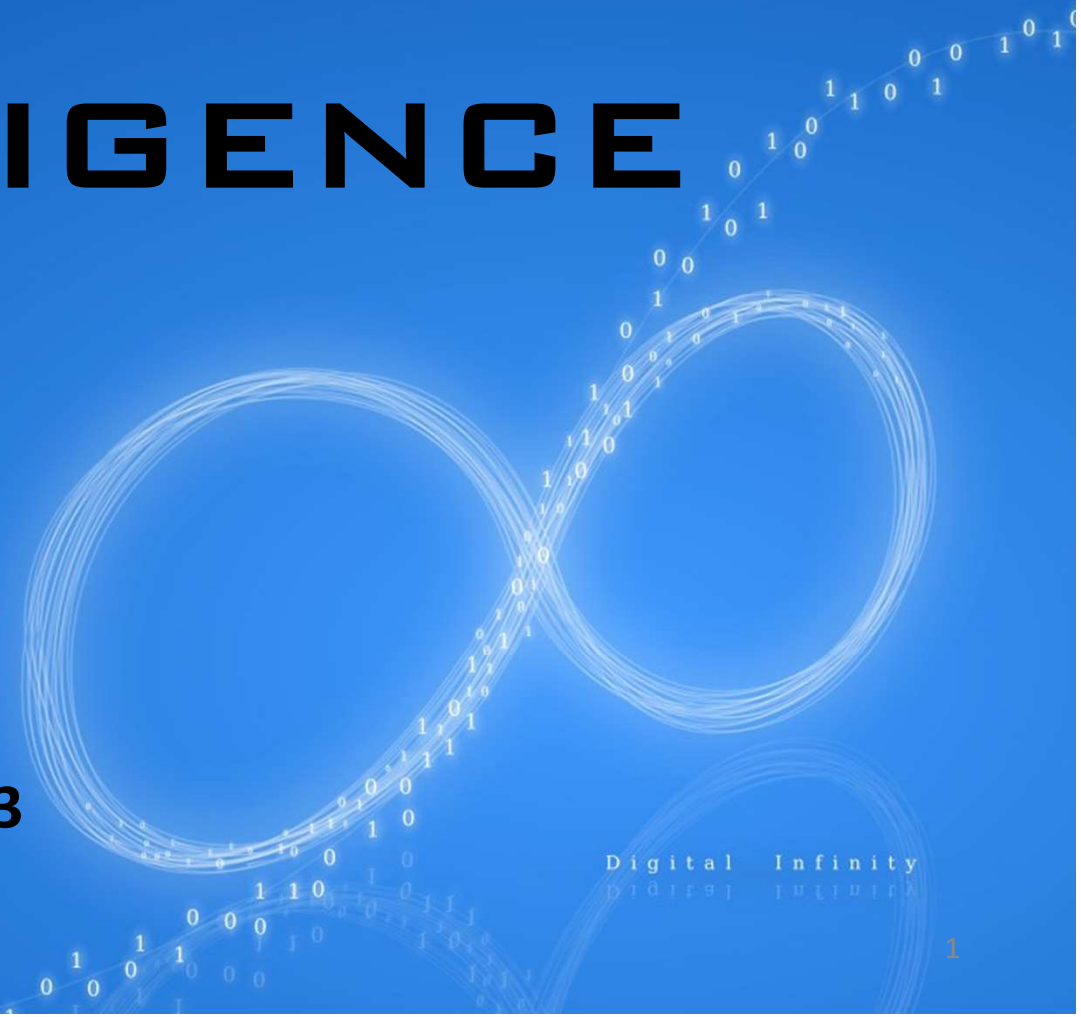
# ARTIFICIAL INTELLIGENCE

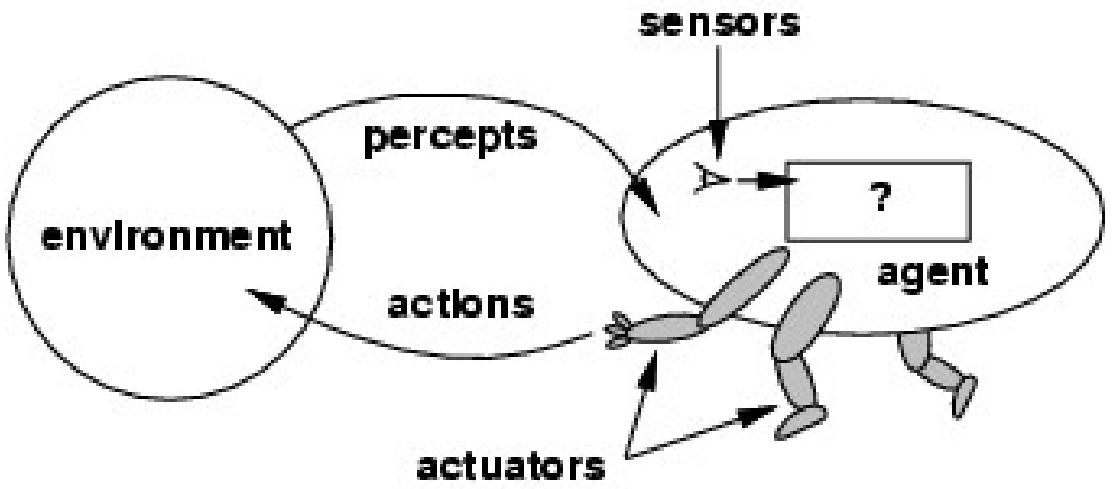



**2nd Year, 4th Semester, 2022-23**

**AI & ML**

# What is an Agent??

- An **agent** perceives its environment via **sensors** and acts upon that environment with its **actuators.**




- Also agent is a system that implements a **mapping** from percepts sequence to actions. The agent can have goal also.

# Sample Agents…

## Examples of Agents

- **Agent**: automatic car.
- **Environment**: streets, other vehicles, pedestrians, traffic signals/lights/signs.
- **Goals**: safe, fast, legal trip.
- **Percepts**: camera, GPS signals, speedometer, sonar.
- **Actions**: steer, accelerate, brake.

## Examples of Agents

- **Agent**: intelligent house
- **Environment**:
  - occupants enter and leave house,
  - occupants enter and leave rooms;
  - daily variation in outside light and temperature
- **Goals**: occupants warm, room lights are on when room is occupied, house energy efficient
- **Percepts**: signals from temperature sensor, movement sensor, clock, sound sensor
- **Actions**: room heaters on/off, lights on/off

**iRobot Roomba® 400**
Vacuum Cleaning Robot


iRobot Corporation

under Rodney Brooks (MIT)

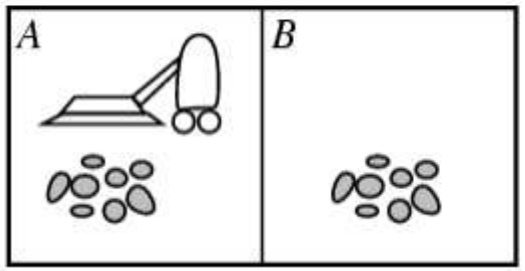
- Powerful suction and rotating brushes
- Automatically navigates for best cleaning coverage
- Cleans under and around furniture, into corners and along wall edges
- Self-adjusts from carpets to hard floors and back again
- Automatically avoids stairs, drop-offs and off-limit areas
- Simple to use— just press the Clean button and Roomba does the rest

Examples:

Human agent
Robotic agent
Software agent

# Sample Agent…

## Vacuum-cleaner world




- Two locations: A and B
- Percepts: location and contents, e.g., [A,Dirty]
- Actions: *Left*, *Right*, *Suck*, *NoOp*

**Table**

| Percept sequence | Action |
|---|---|
| $[A, Clean]$ | $Right$ |
| $[A, Dirty]$ | $Suck$ |
| $[B, Clean]$ | $Left$ |
| $[B, Dirty]$ | $Suck$ |
| $[A, Clean], [A, Clean]$ | $Right$ |
| $[A, Clean], [A, Dirty]$ | $Suck$ |
| ⋮ | ⋮ |

**Function**

**Input:** location, status
**Output:** action

```
1: if status = Dirty then
2:     return Suck
3: end if
4: if location = A then
5:     return Right
6: end if
7: if location = B then
8:     return Left
9: end if
```

# Agent Performance

- The ***input data history or perception history*** is used by the agent to map and take action accordingly.
- Agent function maps any given percept sequence to an action [f: p*--> A]**.**
- The agent program runs on the physical architecture to produce 'f'
- Agent = architecture + program
- The performance measure is a subjective measure to ***characterize how successful an agent is***. The success can be measured in various ways –
  - **In terms of speed and efficiency of the agent**
  - **Accuracy of the quality of the solution achieved by the agent**
  - **Power/memory usage, money etc.**

# Rational Agent

- An agent should "do the right thing", based on which it can perceive and the actions it can perform.
- A rational agent is suppose to have a healthy and ***reach store house of knowledge*** and so as proper mapping is done. Hence a rational agent is suppose to take the actions which increases its efficiency.

  In other words **rational action** increases/ maximizes the performance efficiency.
- For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

**Example** - performance measure of a vacuum-cleaner agent could be amount of ***dirt cleaned up***, ***amount of time taken***, amount of electricity ***consumed***, **amount of noise generated**, etc.

# PEAS

- Specifying the ***task and environment*** are always the first step in designing agent.
- PEAS: ***Performance, Environment, Actuators, Sensors***

## Taxi Driver Example

| Performance Measure | Environment | Actuators | Sensors |
|---|---|---|---|
| safe, fast, legal, comfortable trip, maximize profits | roads, other traffic, pedestrians, customers | steering, accelerator, brake, signal, horn, display | camera, sonar, speedometer, GPS, odometer, engine sensors, keyboard, accelerator |

DARPA urban challenge 07:

http://www.youtube.com/watch?v=SQFEmR50HAk

# Environment of an Agent

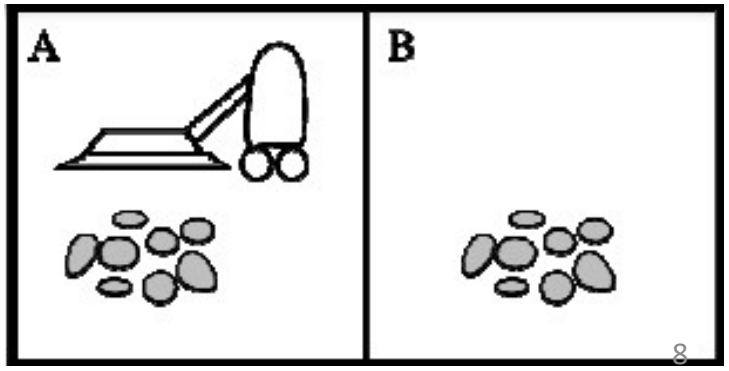
After every action of the agent the environment is observed to guide the next phase of action by the agent.

- ***Fully Observable Environment :***

  An environment is fully observable if an agent's sensors give it ***access to the complete state of the environment*** at each point in time.

  Fully observable environments are convenient, because the agent ***need not maintain any internal state*** to keep track of the world.

**Example : If the Vacuum Cleaner can sense the dirt both in Room A&B then the environment is fully observable for the agent**

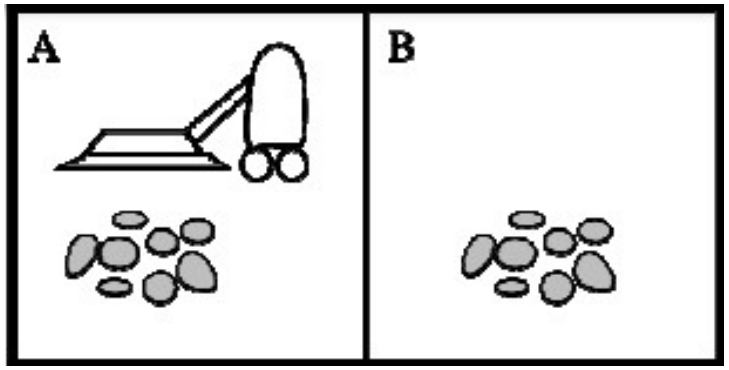


# Environment of an Agent

- ***Partially Observable Environment :***

  An environment might be partially observable because of ***noisy*** *and* ***inaccurate sensors*** or because ***parts of the state are simply missing*** from the sensor data.

**Example :** **If the Vacuum Cleaner can sense the dirt in either of the Room A or B then the environment is fully observable for the agent**



# Environment of an Agent

- ***Deterministic Environment :***

  In this environment ***next state*** of environment is ***completely described*** by the current state and agent's action.

  **Examples : Image analysis : processed image is completely determined by current image and the processing operation.**

- ***Stochastic Environment :***

  When an ***element of uncertainty*** occurs and there are multiple, unpredictable environments.

  **Examples : Automatic navigation robots. In new environment the vehicle is always in matter of uncertainty and the input is from multiple sensors.**

**In a fully observable, deterministic environment, the agent need not deal with uncertainty.**

# Environment of an Agent

- ***Strategic Environment :***

  When the environment is determined by the ***preceding state*** and ***actions of multiple agents***, then the environment is said to be strategic.

  **Examples : In chess playing there are two agents and next state of board is strategically decided by the actions of the players.**

- ***Episodic Environment :***

  In episodic environments, the agent's experience is divided into atomic "episodes" (each episode consists of the agent perceiving and then performing a single action), and the ***choice of action in each episode depends only on the episode itself***.

  **Examples : Classification task.**

# Environment of an Agent

- ***Sequential Environment/ Strategic Environment :***

  When the environment is determined by the preceding ***state and actions of multiple agents***, then the environment is said to be strategic. In a sequential environment, the ***agent engages in a series of connected episodes***. Current decision could affect future decisions.

  **Examples : Chess.**

- ***Static Environment :***

  It is static environment means ***the environment does not change from one state to another*** while the agent is considered in course of action.

  **Examples : Crosswords puzzles.**

# Environment of an Agent

- ***Discrete & Continuous Environment :***

  A limited number of distinct, clearly defined states, percepts and actions.

  **Examples : Chess has finite number of discrete states, and has discrete set of percepts and actions. Taxi driving has continuous states, and actions.**

- ***Single agent/Multi-agent :***

  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 don't have multi-agent properties***, whereas most social and economic systems get their complexity from the interactions of (more or less) rational agents.

# Example

| Task Environment | Oberservable | Deterministic | Episodic | Static | Discrete | Agents |
|---|---|---|---|---|---|---|
| *Crossword puzzle* | fully | deterministic | sequential | static | discrete | single |
| *Chess with a clock* | fully | strategic | sequential | semi | discrete | multi |
| *Taxi driver* | partially | stochastic | sequential | dynamic | conti. | multi |
| *mushroom-picking* | partially | stochastic | episodic | dynamic | conti. | single |

- The environment type largely determines the agent design.
- The real world is (of course) partially observable, stochastic, sequential, dynamic, continuous, multi-agent

# Agent Structure

- ***Table based Agent :***

  The information from the ***percept*** of the agent is given in the ***from of a table***. The mapping is done from the table information through **some rule-base system or by ANN**. Thus the mapping from percepts to actions is specified through table.

  **Disadvantage –**

  - the table may become very large to handle
  - the learning from table may take long time
  - such system have little autonomy as well as actions are predicted

# Agent Structure

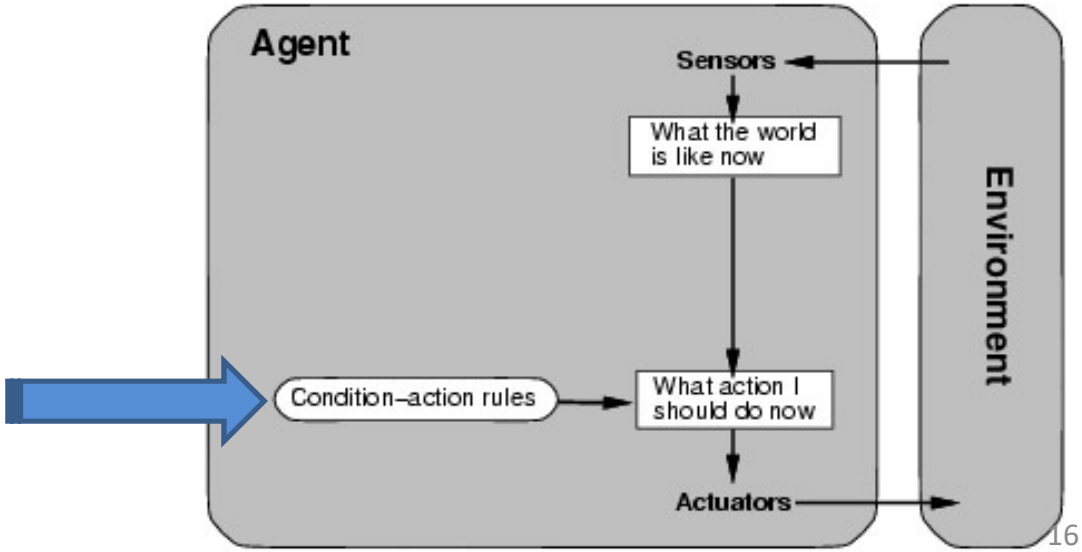
- ***Percepts based or reflex agent :***

  This types of agents ***don't have notion of history*** and action is based upon current percepts.

  This type of agent ***don't have any internal knowledge representation, learning, strategy, planning***.

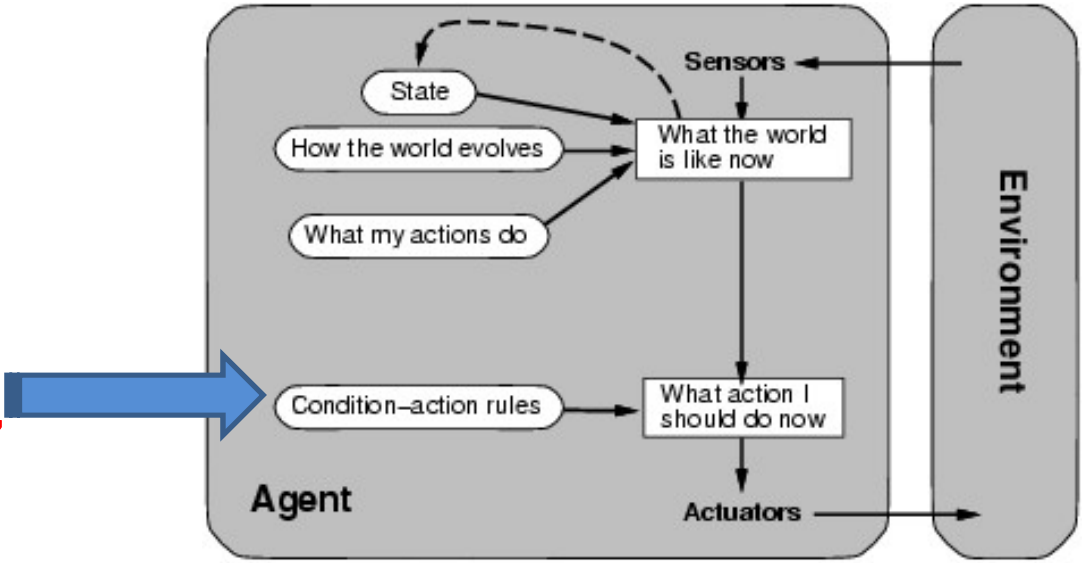
  This type of agents are called as ***reactive or stimulus*** agents.

**If tail-light of car in front is red then brake.**



# Agent Structure

- ***State based or model based reflex agent :***
  - information comes from sensors – percepts
  - based on the information agent changes its current state of world
  - ***based on current state of world and knowledge***, it triggers actions through effectors/actuators.




"Crazy driver in front"
"Safe to change lanes"

*Percepts based or reflex agent*

```
function SIMPLE-REFLEX-AGENT(percept) returns an action
  static: rules, a set of condition–action rules

  state ← INTERPRET-INPUT(percept)
  rule ← RULE-MATCH(state, rules)
  action ← RULE-ACTION[rule]
  return action
```

*State based or model based reflex agent*

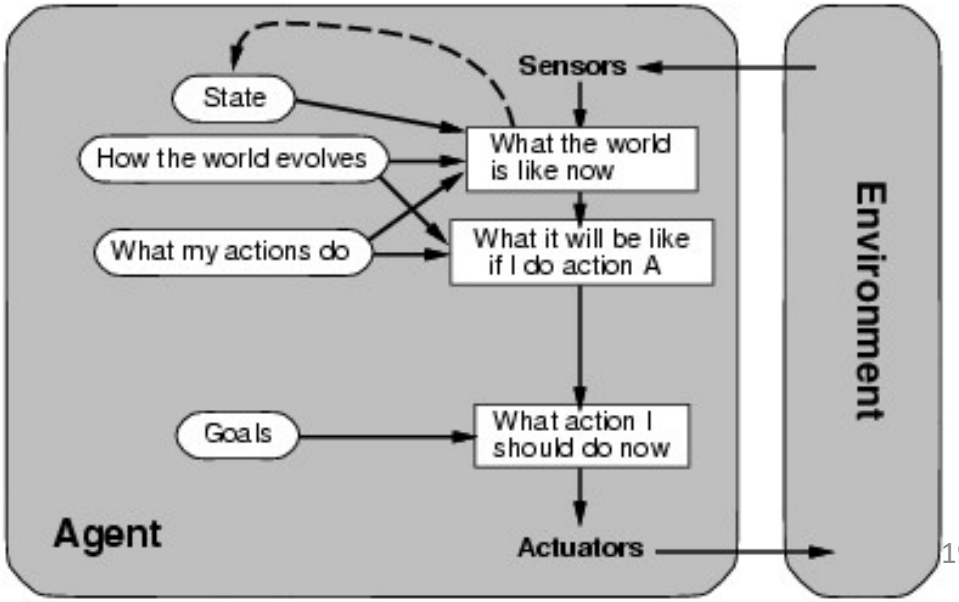
```
function REFLEX-AGENT-WITH-STATE(percept) returns an action
  static: state, a description of the current world state
          rules, a set of condition–action rules
          action, the most recent action, initially none

  state ← UPDATE-STATE(state, action, percept)
  rule ← RULE-MATCH(state, rules)
  action ← RULE-ACTION[rule]
  return action
```

# Agent Structure

- ***Goal based agent :***
  - agents takes decisions based on how far they are currently from their goal.
  - every action of the agent reduces its distance from the goal.
  - the knowledge that supports its decision is represented explicitly and can be modified, which makes these agents more flexible.

**This is a way of solving many AI problems.**



# Agent Structure

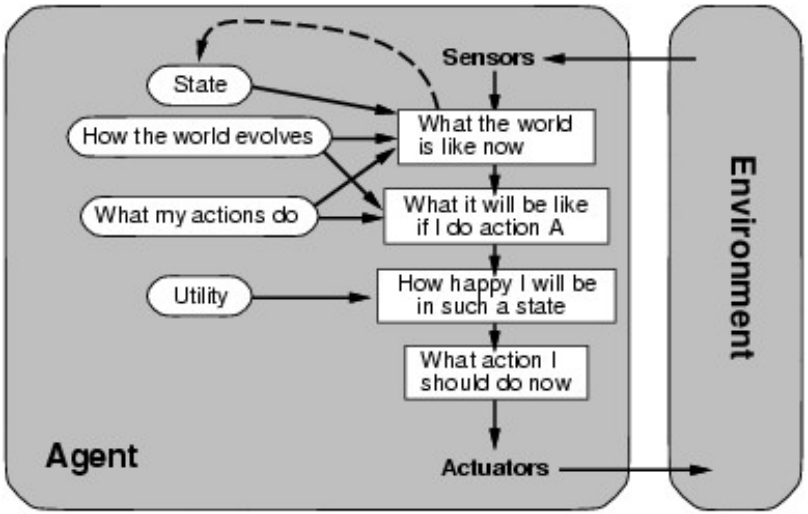
- ***Utility based agent :***

When there are ***multiple possible alternatives***, how to decide which one is best, Utility based agents are used.

***Goals are qualitative*** → A goal specifies a crude distinction between a happy and unhappy state, but often need a more general performance measure that describes **"degree of happiness"**.

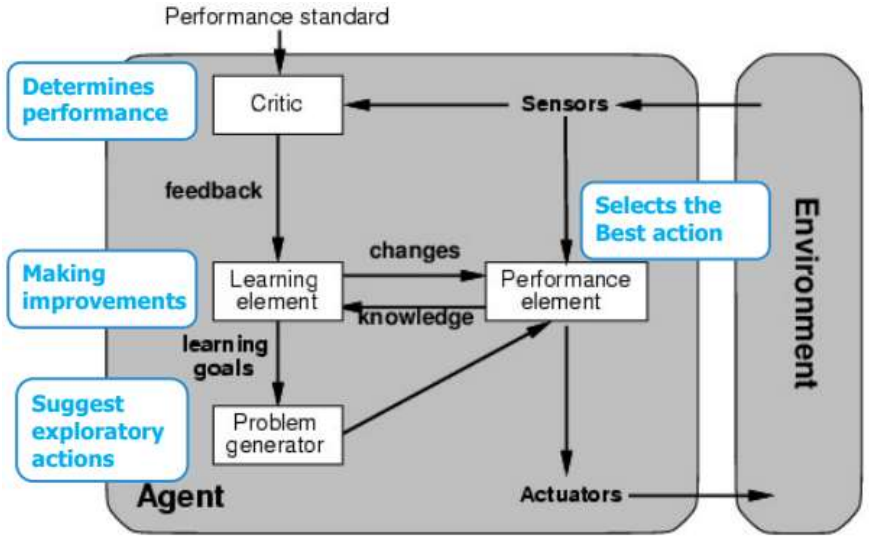
***Utility function*** **U: State → Real** indicating a measure of success or happiness when at a given state.

**The agent acts so as to maximize utility.**



# Agent Structure

- ***Learning agent:***
- It can learn from its past experience.
- Starts with basic knowledge and can act and adapt automatically.
- Learning is through four concepts –
  - **Learning element**
    - ✓ Making improvement by learning
  - **Critic**
    - ✓ Takes feedback from critic to judge performance
  - **Performance element**
    - ✓ Selecting external actions
  - **Problem generator**
    - ✓ Suggesting actions that will lead



# Learning Element

- responsible for making improvements
- uses knowledge about the agent and feedback on its actions to improve performance

# Performance Element

- selects external actions
- collects percepts, decides on actions
- incorporated most aspects of our previous agent design

# Critic

- Informs the learning element about the performance of the action
- Must use a fixed standard of performance
  - should be from the outside
  - an internal standard could be modified to improve performance

# Problem Generator

- Suggests actions that might lead to new experiences
- May lead to some ***sub-optimal decisions*** in the short run
  - in the long run, hopefully better actions may be discovered
- Otherwise no exploration would occur

# Learning Element Design Issues

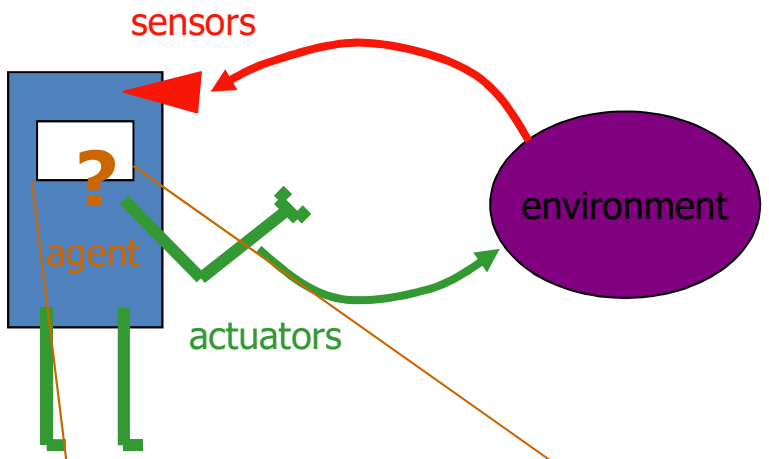
- selections of the components of the performance elements that are to be improved
- representation mechanisms used in those components
- availability of feedback
- availability of prior information

# Feedback

- provides information about the actual outcome of actions
- supervised learning
  - both the input and the output of a component can be perceived by the agent directly
  - the output may be provided by a teacher
- reinforcement learning
  - feedback concerning the desirability of the agent's behavior is available
    - not in the form of the correct output
  - may not be directly attributable to a particular action
    - feedback may occur only after a sequence of actions
  - the agent or component knows that it did something right (or wrong), but not what action caused it

# Problem Solving Agent

- ***Goal formulation***
- ***Problem formulation***
- ***Example problems***
  - ***States***
  - ***Actions***
- ***Search***
  - ***Search strategies***
  - ***Constraint satisfaction***
- ***Solution***




- Formulate Goal
- Formulate Problem
  - States
  - Actions
- Find Solution

# Goal Formation

***Specify the objectives to be achieved***

- ***Goal***

  ***- a set of desirable world states in which the objectives have been achieved***

- ***Current / initial situation***

  ***- starting point for the goal formulation***

- ***Actions***

  ***- cause transitions between world states***

# Problem Formation

## *Actions and states to consider*

- ***States** : possible world states*
- ***Accessibility** : the agent can determine via its sensors in which state it is*
- ***Consequences of actions** : the agent knows the results of its actions levels problems and actions can be specified at various*
- ***Levels** : constraints conditions that influence the problem-solving process*
- ***Performance** : measures to be applied*
- ***Costs** : utilization of resources*

# Search

*Examine possible sequences of actions*

- ***Input***

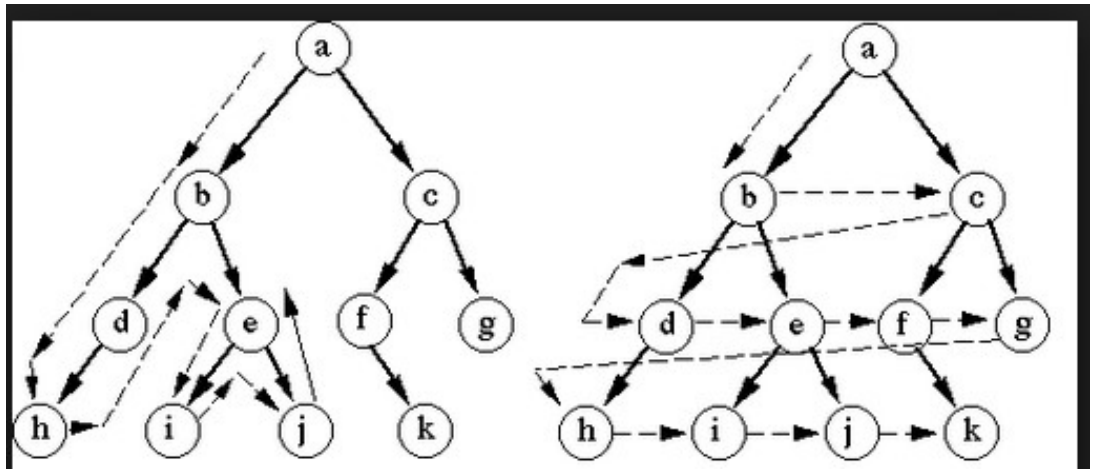
  ***problem description, initial state***

- ***Output***

  ***solution as an action sequence***

- ***Search space***

  ***set of all possible action sequences***



# Solution

***Action sequence that satisfies the goal***

- ***Validation***

  ***Does the solution really achieve the goal?***

- ***Verification***

  ***Is the sequence of actions admissible?***

- ***Feasibility***

  ***With the available resources, can the sequence of actions be carried out?***

- ***Execution***

  ***actions are carried out***

# Subjects of AI

- Learning System
- Knowledge Representation and Reasoning
- Planning
- Intelligent Search
- Knowledge Acquisition
- Logic Programming
- Soft Computing

# Uninformed Search Strategies

- ***Uninformed*** strategies use only the information available in the problem definition
  - **Also known as blind searching**

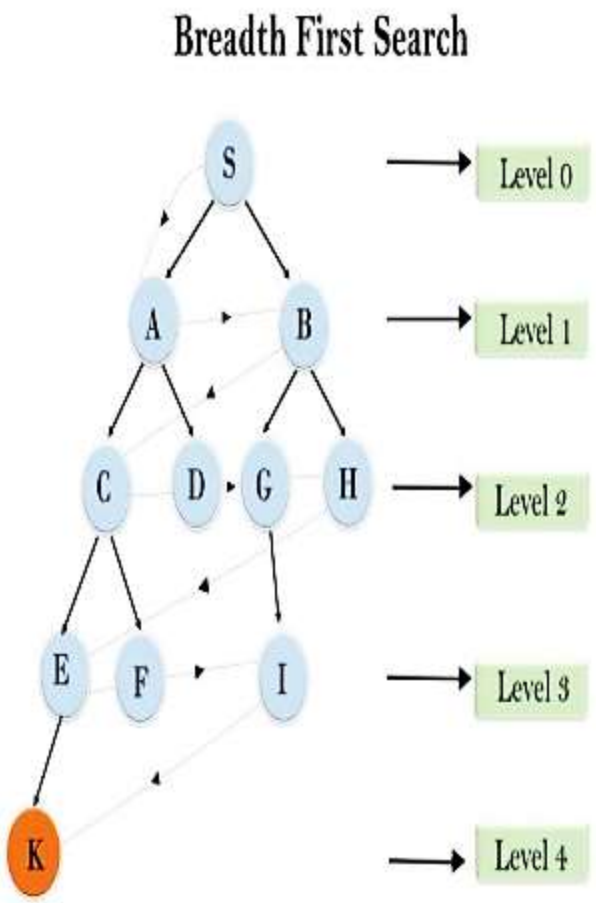
- ***Breadth-first search***
- ***Depth-first search***
- ***Iterative Depending search***
- ***Uniform-cost search***

***Completeness :*** The search guarantees a solution for any random input.

***Optimality :*** If the solution guarantees to be the best solution among all the solutions.

# Breadth-First Search

- BFS is the most common search for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- In BFS search starts from root node and then before moving to next level all successor node from current level is expand.
- It is an example of a general-graph search algorithm.
- It uses queue data structure.



# Breadth First Search

**Begin**

i) Place the starting node in the queue

ii) **Repeat**

Delete the queue to get the front element;

**If** the front element of the queue = goal,

return success and stop;

**Else** do

**Begin**

insert the children of the front element,

if exist, in any order at the rear end of the queue;

**End**

**Until** the queue is empty;

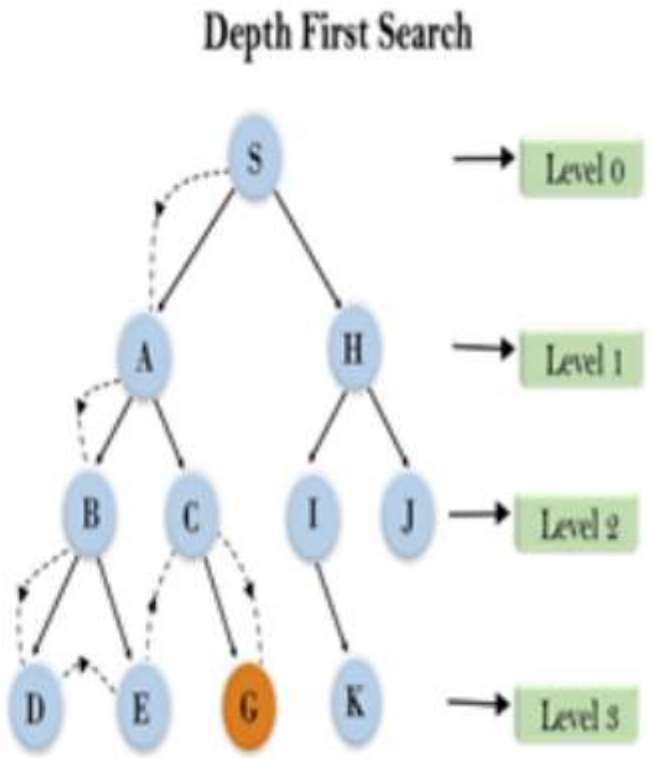
**End**

# Lessons From Breadth First Search

- BFS is a **complete search**.
- BFS finds a solution with the **shortest path** length.
- It has **exponential time** and **space complexity**.
- A complete search tree of depth ‘d’ where each non-leaf node has ‘b’ children, has total of **(b^d)/(b-1)** nodes.
- The **memory requirements are a bigger** problem for breadth-first search than is execution time
- **Exponential-complexity search problems** cannot be solved by uniformed methods for any but the smallest instances

# Depth-First Search

- Depth-first search is a recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- Backtracking is an algorithm technique for finding all possible solutions using recursion.




- DFS takes exponential time.
- If ‘N’ is maximum depth of node in the search space, in the worst case algorithm takes time ‘b^d’.
- The space taken is linear in the depth of the search tree ‘bN’

# Depth First Search

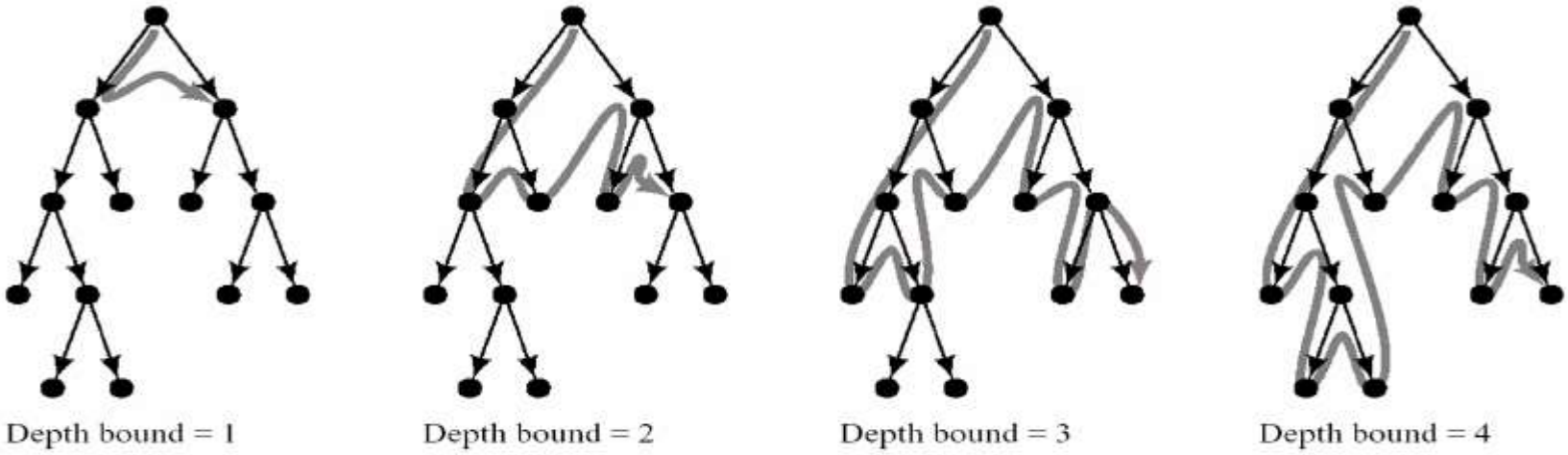
**Begin**

1. Push the starting node at the stack,
   pointed to by the stack-top;
2. **While** stack is not empty do
   **Begin**
   Pop stack to get stack-top element;
   **If** stack-top element = goal, return success and stop
   **Else** push the children of the stack-top element in any order into the stack;
   **End while**;

**End**

# Iterative Deepening Search

- Iterative deepening depth-first search
  - First do depth 0, then, if no solution found, do DFS to depth 1 and gradually increases the depth limit; 0, 1, 2, … until a goal is found
  - For large ‘d’ the ratio of the number of nodes expanded is **b/(b-1)**.
  - ID searching moves along the increasing depth of the nodes, where as BFS moves level by level.
  - In DFS the search propagates at the maximum depth of the tree, whereas in ID searching doesn't go below the current depth.

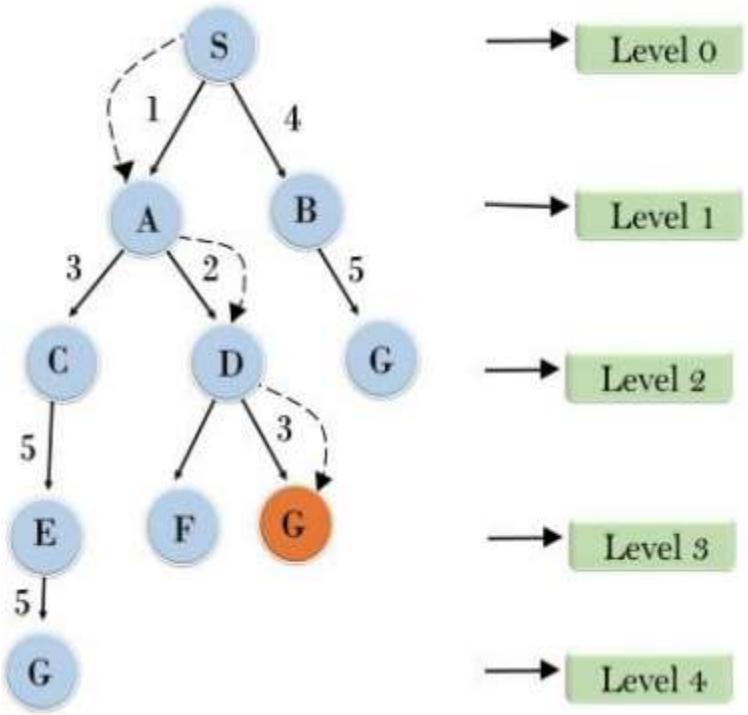



**Depth First Iterative Deepening**

| | Breadth first | Depth first | Iterative deepening |
|---|---|---|---|
| Time | $b^d$ | $b^d$ | $b^d$ |
| Space | $b^d$ | bm | bd |
| Optimum? | Yes | No | Yes |
| Complete? | Yes | No | Yes |

# Uniform-Cost Search

- Same idea as the algorithm for breadth-first search…but…
  - Nodes are expanded in the order of their cost from the source.
  - FIFO queue is ordered by cost.
  - This search is **complete** and **optimal**.
  - Exponential space and time complexity of b^d.
  - the value of the function ‘f’ for a given node ‘n’, f(n), for ***uninformed search algorithms***, takes into consideration g(n), the total action cost **from the root node to the node n**, that is, the path cost.



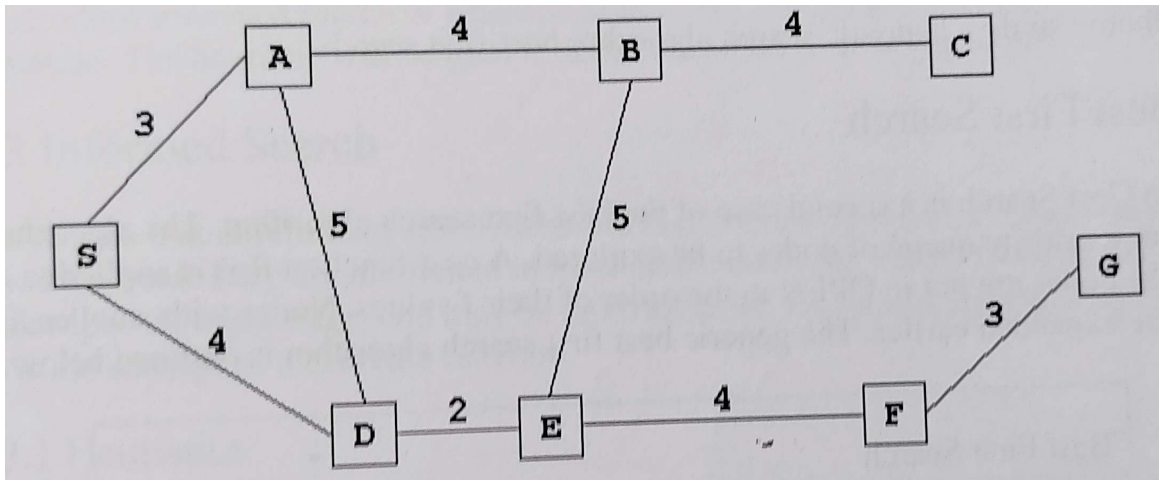
# ***Informed Search***

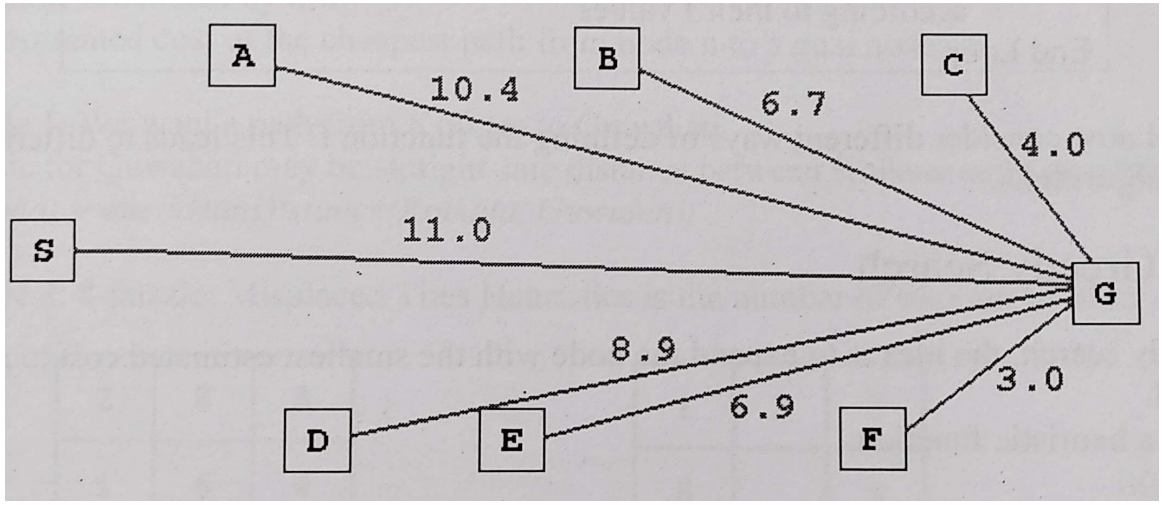
## Best-first Search Algorithm (Greedy Search)

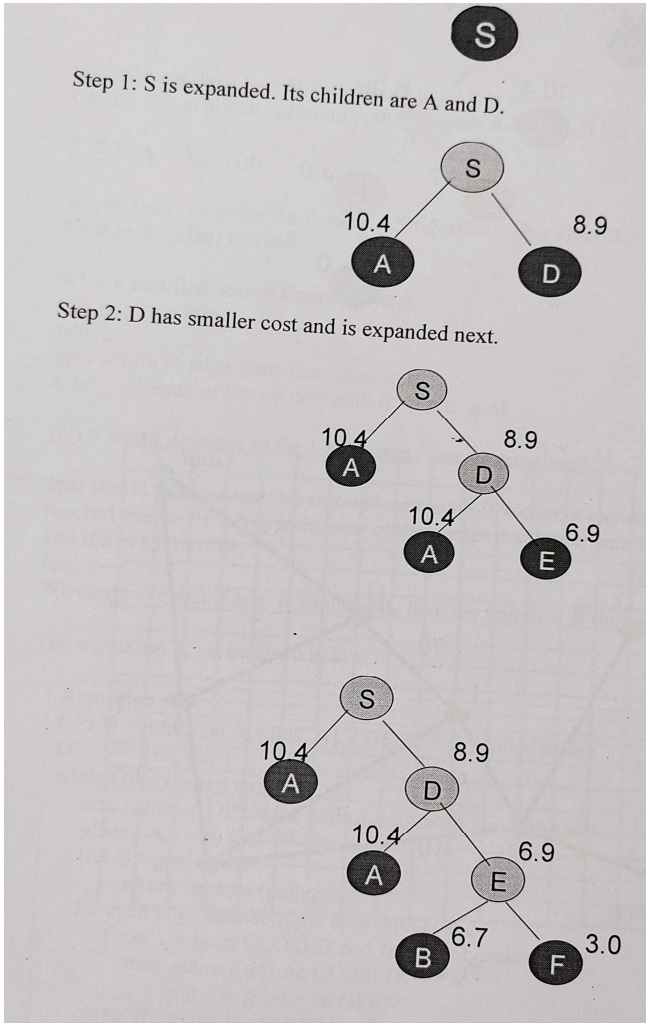
- Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help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

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

- Were, h(n)= estimated cost from node n to the goal.
- The greedy best first algorithm is implemented by the priority queue.



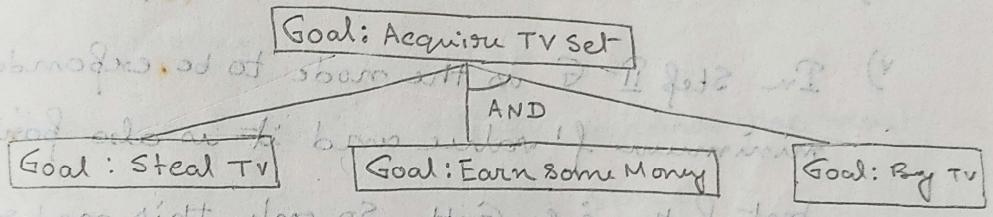



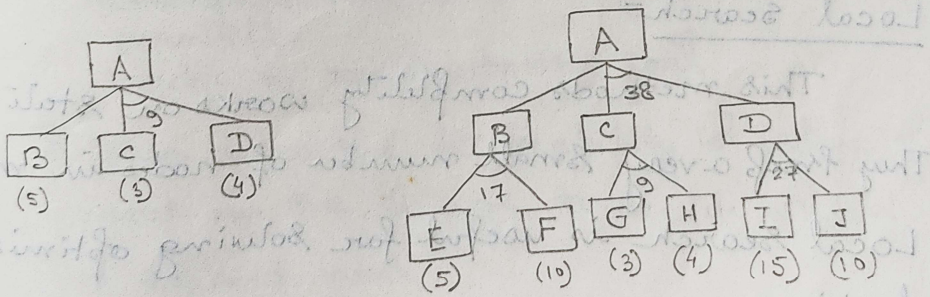

- g(n) is the essential distance remaining to a goal.
- The search is **not an optimal** one **but has completeness**.
- the function that is evaluated to choose which node to expand has the form of f(n) = h(n), where h is the heuristic function for a given node n that returns the estimated value **from this node n to a goal state**.

## Best - First - Search : AND OR Graph / AO* Algo. –

This method of searching is useful when problems can be decomposed into a set of small problems It means, may be two set of nodes leads to the goal point rather than only one goal node. Hence due to this decomposition, arcs are generated which are known as AND arc.




Hence in the above example, the Goal can be achieved by decomposition the problem into two nodes i.e., Earn some money & Buy TV. So it is the AND of these two nodes which can solve the problem.
Lets, consider another example –



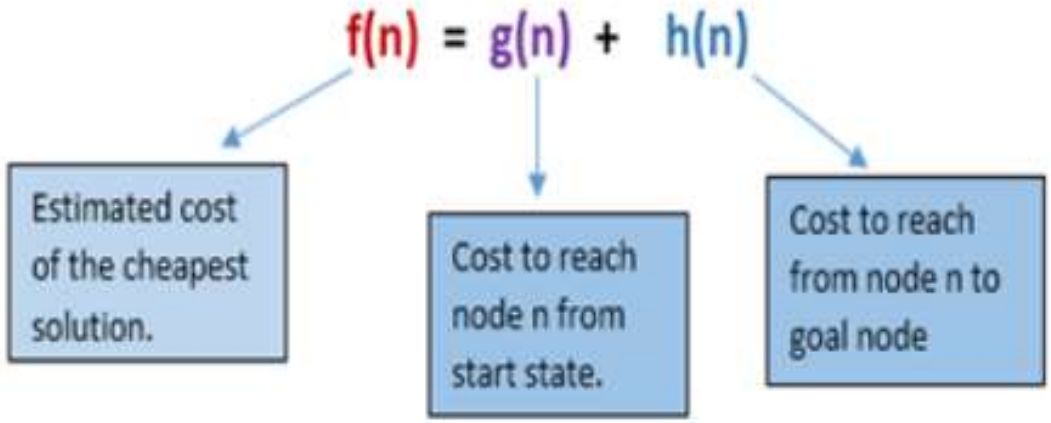

- It is considered to have uniform cost function of every expansion = 1
- Suppose 'B' node is to be selected for expansion but
  (B+C)=5+3+1+1=10
  (C+D)=3+4+1+1=9

So best path for expansion is (C+D)

- It is not only about the value of 'f' but also whether it is part of best node or not.
- This method of algorithm is known as AO* algorithm.

# A* Search Algorithm

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




- At each point in the search space, only those node is expanded which have the lowest value of f(n), and the algorithm terminates when the goal node is found.
- A* search algorithm is optimal and complete.

# Example: Water Pouring

- Given a 4 gallon bucket and a 3 gallon bucket, how can we measure exactly 2 gallons into one bucket?
  - There are no markings on the bucket
  - You must fill each bucket completely
- **Initial state**:
  - The buckets are empty
  - Represented by the tuple ( 0 0 )
- **Goal state**:
  - One of the buckets has two gallons of water in it
  - Represented by either ( x 2 ) or ( 2 x )
- **Path cost**:
  - 1 per unit step

# Example: Water Pouring

**List of PRs for the water-jug problem**

PR 1. $(u, v : u < 4) \rightarrow (4, v)$

PR 2. $(u, v : v < 3) \rightarrow (u, 3)$

PR 3. $(u, v : u > 0) \rightarrow (u - D, v)$, where D is a fraction of the previous content of u.

PR 4. $(u, v : v > 0) \rightarrow (u, v - D)$, where D is a fraction of the previous content of v.

PR 5. $(u, v : u > 0) \rightarrow (0, v)$

PR 6. $(u, v : v > 0) \rightarrow (u, 0)$

PR 7. $(u, v : u + v \geq 4 \wedge v > 0) \rightarrow (4, v - (4 - u))$

PR 8. $(u, v : u + v \geq 3 \wedge u > 0) \rightarrow (u - (3 - v), 3)$

PR 9. $(u, v : u + v \leq 4 \wedge v > 0) \rightarrow (u + v, 0)$

PR 10. $(u, v : u + v \leq 3 \wedge u > 0) \rightarrow (0, u + v)$

# Example: Water Pouring

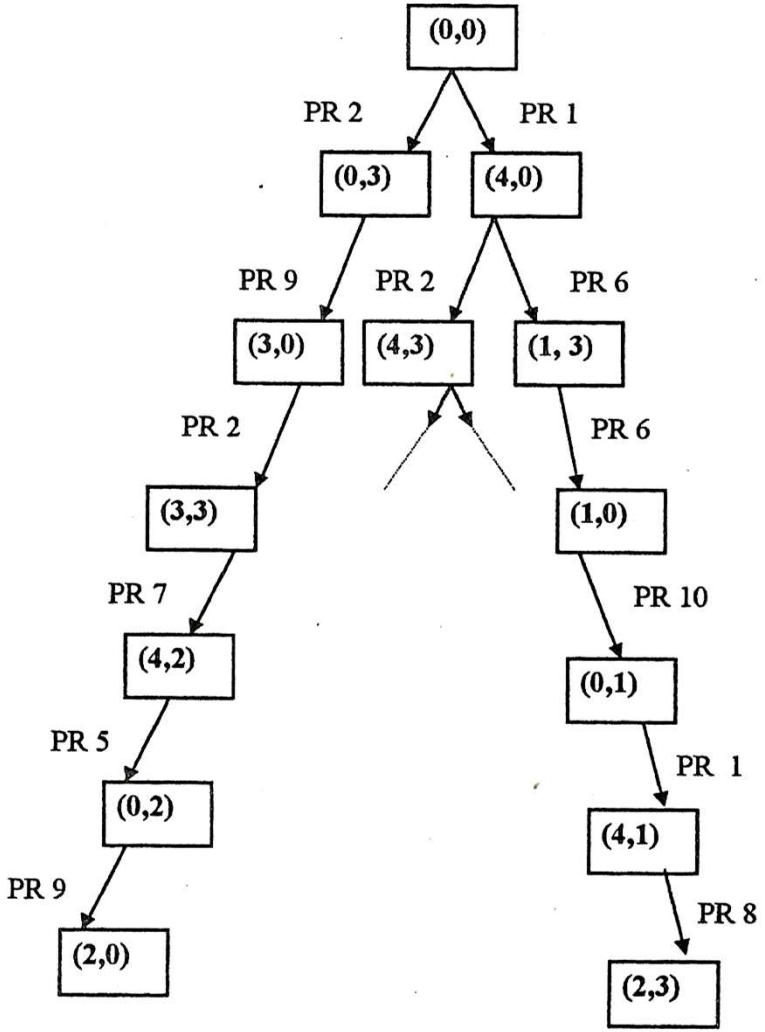
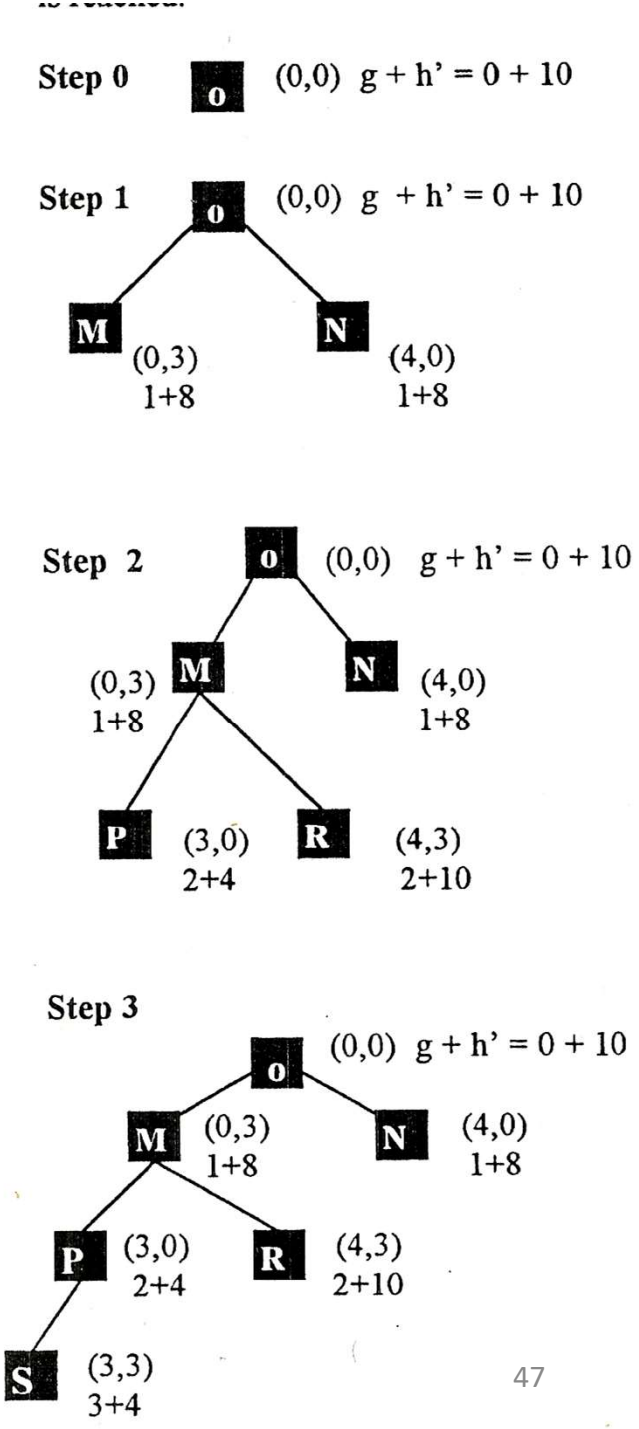



**Fig. 3.2:** The state-space for the water-jug problem.

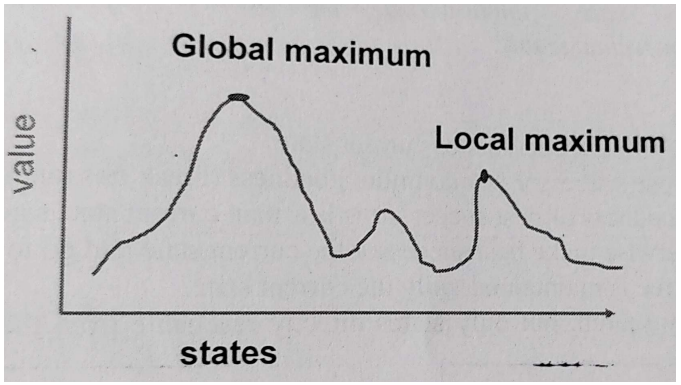
# Heuristic Search

h(x) = 2, when 0<X<4 AND 0<Y<3,
= 4, when 0<X<4 OR 0<Y<3
= 10, when i) X = 0 AND Y = 0
OR ii) X = 4 AND Y = 3
= 8, when i) X = 0 AND Y = 3
OR ii) X = 4 AND Y = 0



# Hill Climbing Algorithm

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




- **Features of Hill Climbing Algorithm:**
  - Generate and Test variant
  - Greedy approach
  - No backtracking
- **Simple Hill Climbing**: It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as a current state.
  - Less time consuming
  - Less optimal solution and the solution is not guaranteed

# Hill Climbing Algorithm

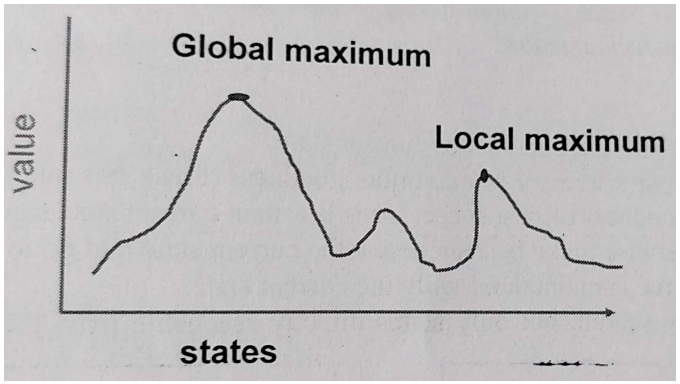
The generate & test type of search algo. expands the search space and examines the existence of the goal in that space. But in this method, the feedback from test procedure helps the generator to decide which direction to move in the search space.

In this method a function $f(x)$ is employed that would give an estimate of the measure of distance of the goal from node 'x'.

After $f(x)$ is evaluted for all possible initial nodes 'x', the nodes are sorted in ~~as~~ ascending order of their function values. So the stack top element has the least function value. It is now popped out and compared with goal state. If the stack top element is not the goal then it is expanded and function is measured for each of its children. Again the previous steps ~~are~~ are repeated until the goal state is ~~ach~~ achieved.

# Problems in Hill Climbing Algorithm

- **Local Maximum-** A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.
- **Plateau-** A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.
- **Ridges-** A ridge is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.



# Simulated Annealing

Annealing is a process of metal casting, where metal is melted at high temperature beyond its melting point and then it is allowed to cool down, until it returns to solid state. Thus in the physical process of annealing, the hot material gradually looses its energy and finally at one point of time reaches at a state of minimum energy.

Here the term objective function is used rather than or in place of heuristic function. Here the attempt is to minimize the object function rather than maximize. Thus actually it describes the process of valley descending rather than hill climbing.

# Simulated Annealing

The concept of rolling ball can be used here. Let, consider a rolling ball that falls from a higher energy state to a valley and moves up to a little higher energy state. The probability of such transition to a higher energy state is very small and is given by

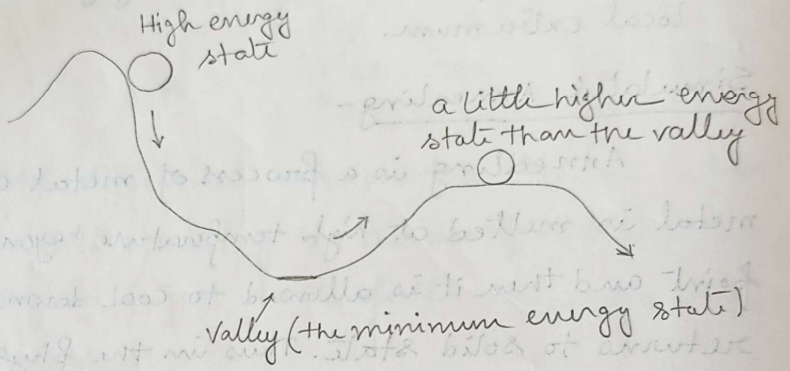
$$p = \exp(-\Delta E/KT)$$

$p$ – probability of transition from lower to higher energy state

$\Delta E$ – positive change in energy

$K$ – Boltzman constant

$T$ – temperature




So, $\Delta E$ small then $p$ is higher

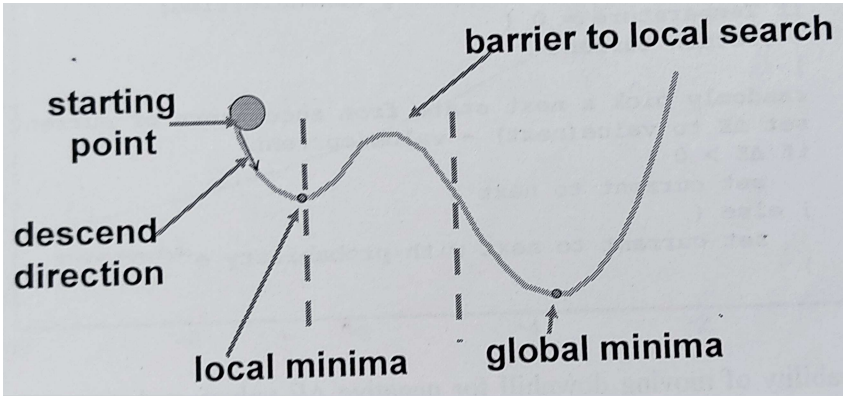
$\Delta E$ large then $p$ is smaller

i.e., the probability of transition to a slightly higher state is more than the probability of transition to a very high state.

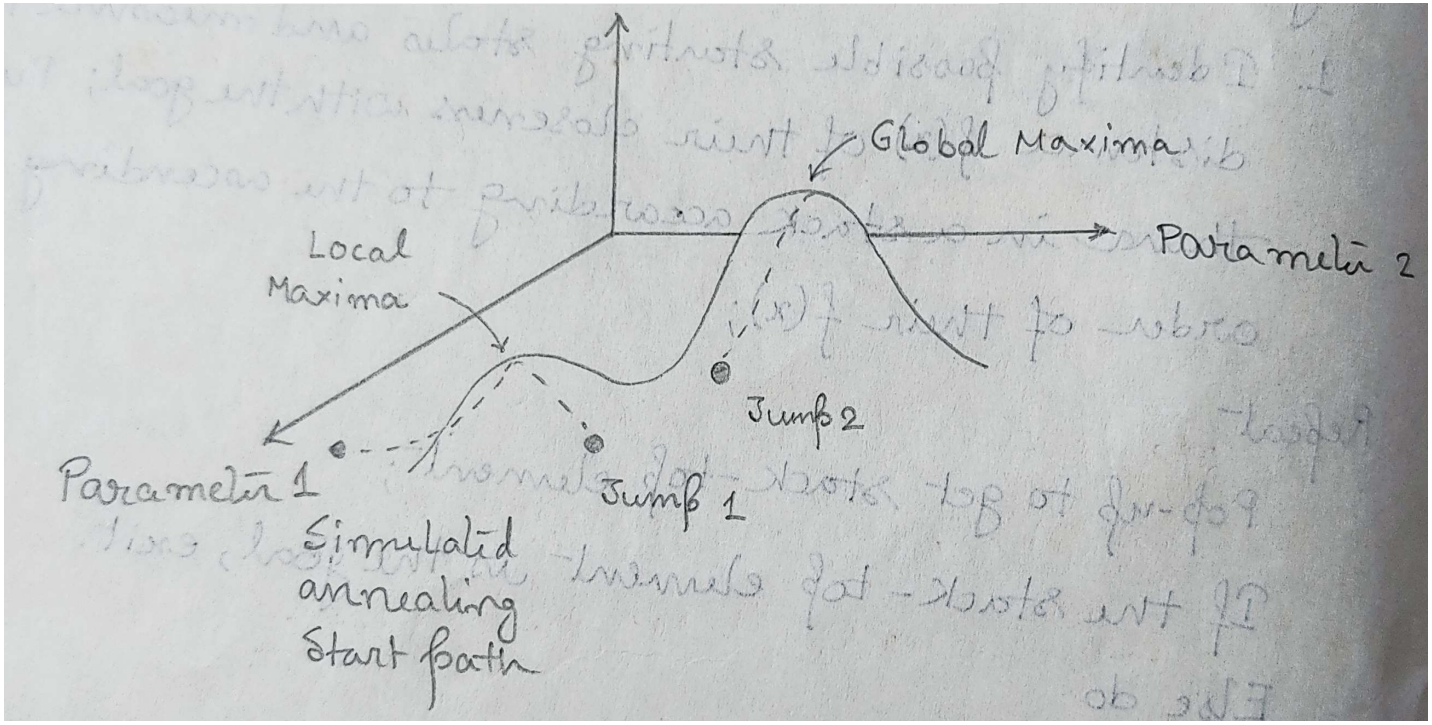
Now under this circumstances $\Delta E$ is calculated for all possible legal next states and $p'$ is also evaluted for each such next state by the formula

$$p' = \exp(-\Delta E/T)$$

A random number in the closed interval of $[0,1]$ is then computed. and $p'$ is compared with the random number. If $p'$ is more than it is selected for next transition.



```
set current to start state
for time = 1 to infinity {
    set Temperature to annealing_schedule[time]
    if Temperature = 0 {
        return current
    }
    randomly pick a next state from successors of current
    set ΔE to value(next) - value(current)
    if ΔE > 0 {
        set current to next
    } else {
        set current to next with probability e^(ΔE/Temperature)
    }
}
```



# Game Playing

- The second major application of **heuristic search** algorithms in AI is game playing.
- One of the original challenges of AI, which in fact predates the term artificial intelligence, was to build a program that could play chess at the level of the best human players.
- **Competitive environments** in which goals of multiple agents are in conflict (often known as games).
- Games problems are like **real world problems**.
- Game theory views as **multi-agent environment** as game.
- Define **optimal move** and algorithm for finding it.

# Minimax Search

- 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.
- Algorithm searches ***forward to a fixed depth*** in the game tree, limited by the amount of time available per move.
- At this search horizon, a ***heuristic evaluation function*** is applied to the frontier nodes.
- the heuristic is a function that takes a board position and returns a number that indicates how ***favorable that position is for one player*** relative to the other.

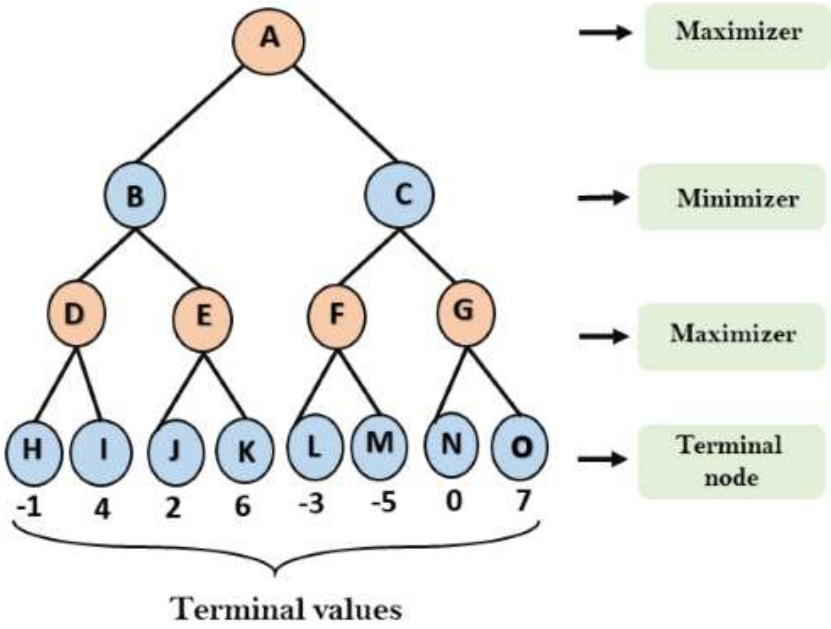
# Minimax Search

- For example, a very simple heuristic evaluator for chess would ***count the total number of pieces on the board for one player***, ***weighted by their relative strength, and subtract the weighted sum of the opponent's pieces***.
- Thus, large positive values would correspond to **strong positions for one player, called MAX**, whereas **negative values of large magnitude would represent advantageous situations for the opponent, called MIN**.
- MAX attempts to maximize its score.
- MIN attempts to minimize MAX's score.

# Minimax Search

- first step, the algorithm generates the entire game-tree and apply the **utility function to get the utility values for the terminal states**.

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

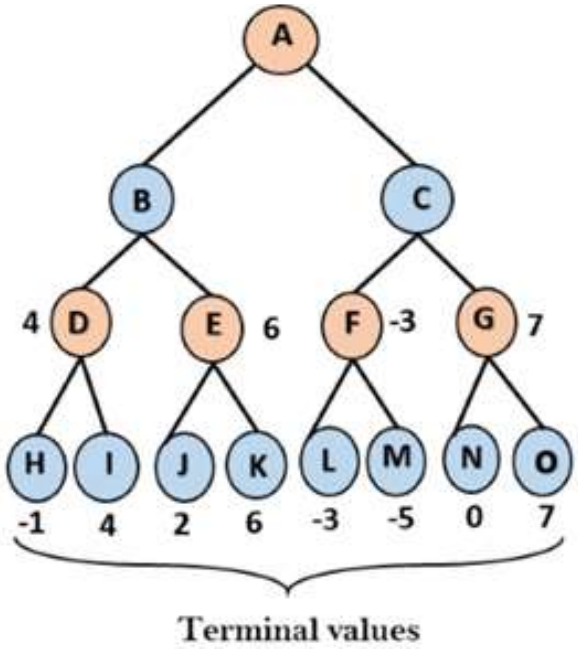



For node D max(-1,- -∞) => max(-1,4)= 4

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

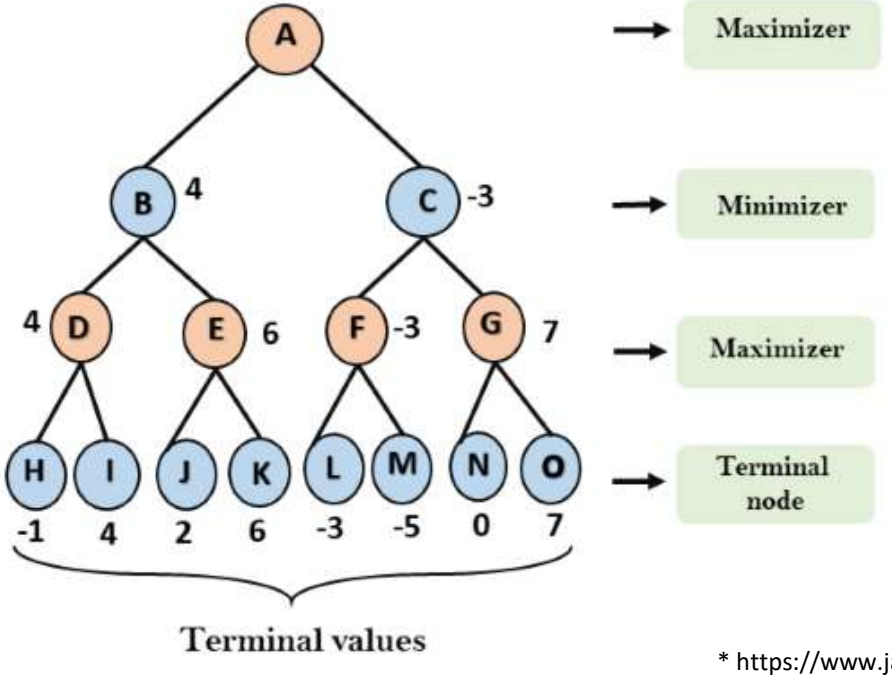




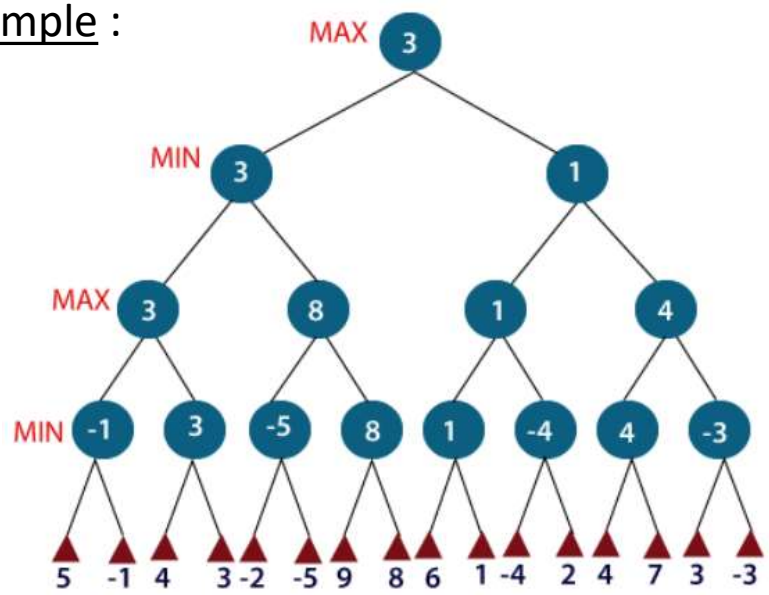

* https://www.javatpoint.com/mini-max-algorithm-in-ai

# Minimax Search

- find the utilities value for the Maximizer, its initial value is -∞, so compare each value in terminal state with initial value of Maximizer and determines the higher nodes values.
- 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.
- Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.




Example :






* https://www.javatpoint.com/mini-max-algorithm-in-ai

# Alpha-Beta Pruning

- Minimax search has to search large number of states.
- But possible to compute ***correct minimax decision without looking at every node*** in search tree.
- ***Eliminating a branch of search tree*** from consideration (without looking at it) is called pruning.
- Alpha-beta pruning – ***Prunes away branches that cannot possibly influence final minimax decision*** – Returns same move as general minimax.
- Alpha-beta name
  - Alpha = value of best (highest-value) choice found so far at any choice point along path for MAX
    - In other words, the worst score (lowest) MAX could possibly get
    - Update alpha only during MAX's turn/ply

# Alpha-Beta Pruning

– Beta = value of best (lowest-value) choice found so far at any choice point along path for MIN

- In other words, the worst score (highest) MIN could possibly get
- Update beta only during MIN's turn/ply

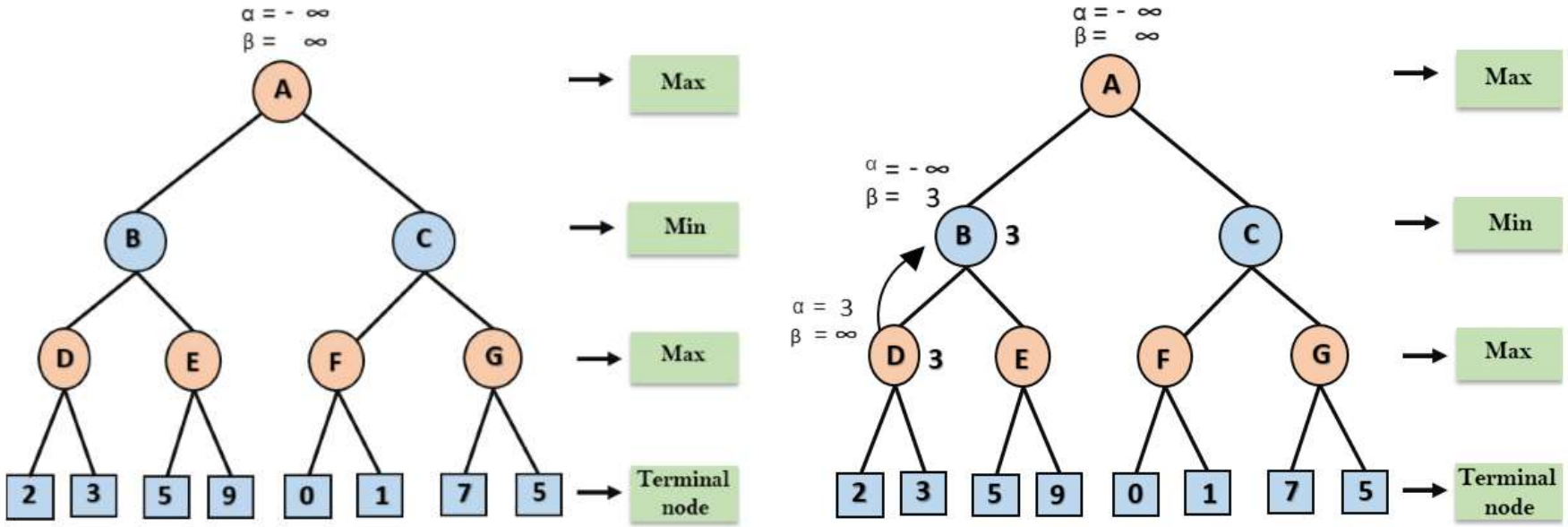
- The two-parameter can be defined as:
  - **Alpha:** The best (highest-value) choice found so far at any point along the path of Maximizer. The initial value of alpha is **-∞**.
  - **Beta:** The best (lowest-value) choice found so far at any point along the path of Minimizer. The initial value of beta is **+∞**.

The main condition which required for alpha-beta pruning is:

$$\alpha >= \beta$$

# Alpha-Beta Pruning

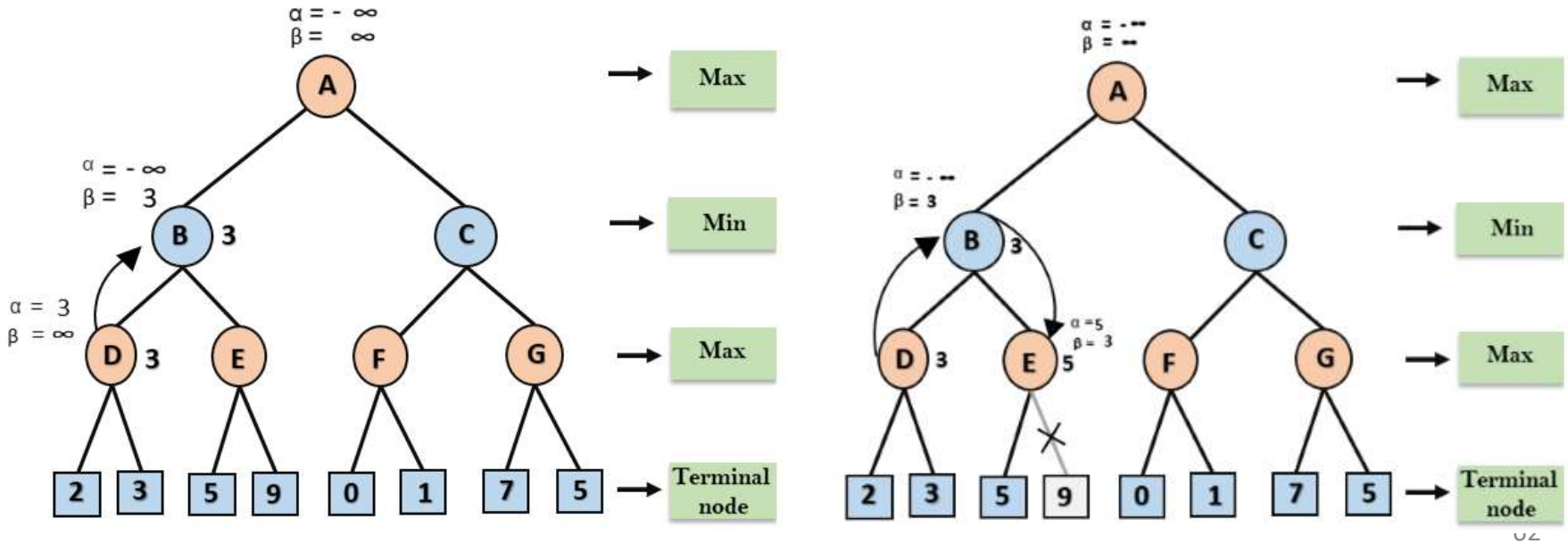
- Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child D.
- At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 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.




* https://www.javatpoint.com/ai-alpha-beta-pruning

# Alpha-Beta Pruning

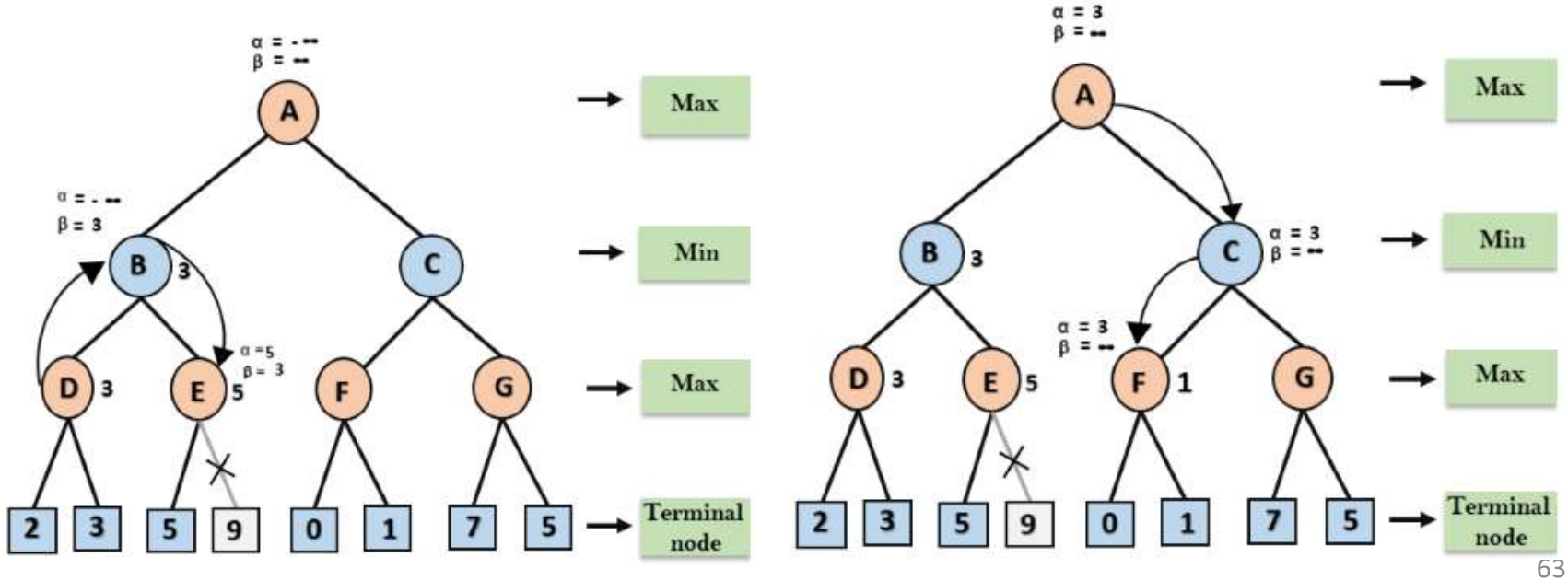
- Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.
- At node E, Max will take its turn, and the value of alpha will change. The current 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.




* https://www.javatpoint.com/ai-alpha-beta-pruning

# Alpha-Beta Pruning

- 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.
- 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.

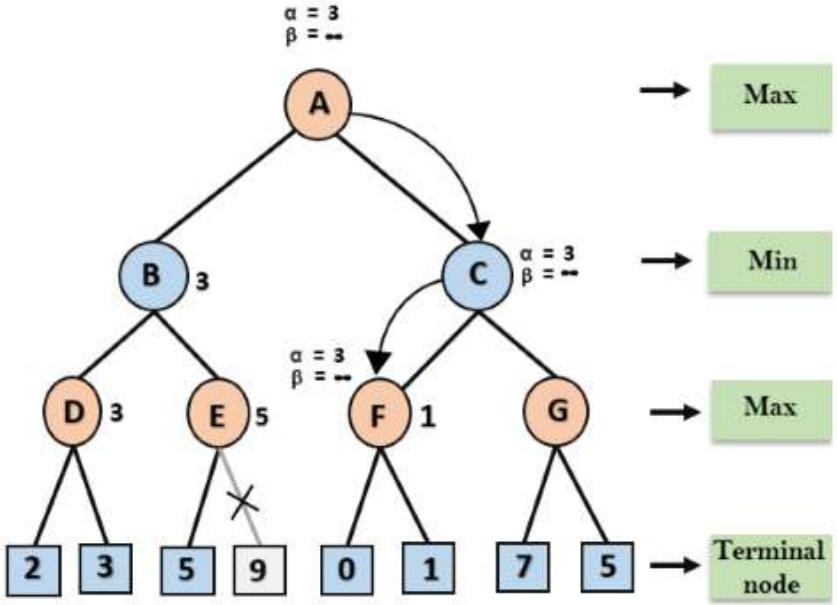


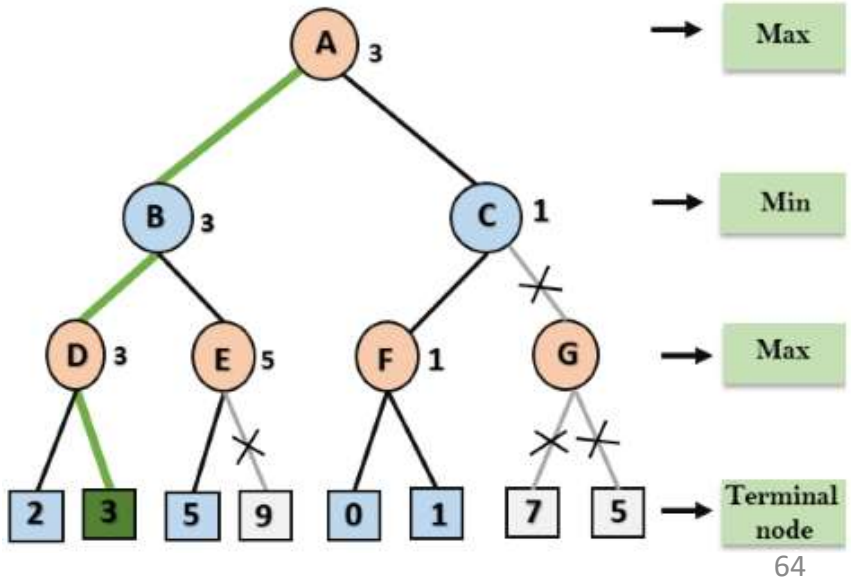



* https://www.javatpoint.com/ai-alpha-beta-pruning

# Alpha-Beta Pruning

- 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.
- 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.




* https://www.javatpoint.com/ai-alpha-beta-pruning