

Unit 1

1.1 Artificial intelligence

- growing technology
- applied in various fields
- universal field
- artificial(man-made) intelligence (thinking power)

It is a branch of computer science by which we can create intelligent machines which can behave like a human, think like humans, and able to make decisions

Goals of Artificial Intelligence

Following are the main goals of Artificial Intelligence:

1. Replicate human intelligence
2. Solve Knowledge-intensive tasks
3. An intelligent connection of perception and action

4. Building a machine which can perform tasks that requires human intelligence such as:
 - Proving a theorem
 - Playing chess
 - Plan some surgical operation
 - Driving a car in traffic
5. Creating some system which can exhibit intelligent behavior, learn new things by itself, demonstrate, explain, and can advise its user.

What Comprises Artificial Intelligence?

combination of Reasoning, learning, problem-solving perception, language understanding, etc.

Advantages of AI:

- high accuracy with less errors
- high speed
- high reliability
- useful for risky areas
- digital assistance
- useful as public utility

Disadvantages of AI:

- High cost
- cant think outside the box
- no feelings and emotions
- increase dependency on machines
- no original creativity

1.2 Foundations and History of Artificial Intelligence

skip this topic, boring

1.3 State of the Art

Researchers now have access to new tools that can help them achieve critical objectives, and these technologies are excellent starting points in and of themselves. The following are some specific domains that have been achieved in recent years:

- ML
- RL
- DL
- NLP

ML:

subtype of ai

uses stat approaches to enable machines to absorb data without being explicitly instructed to do so.

referred to as 'training' the 'model' with 'algorithm'
continuous research on hyperparameters in neural nets

eg. Learning Evolutionary AI Framework(LEAF)

RL:

branch of machine learning

deals with software agents that learn 'goal-oriented' behaviour

by attempting and failing in settings that reward them for their actions (called 'Policy') toward accomplishing the goals.

eg. game agents

DL:

multi level neurons used to discern complicated patterns

eg. medical applications

NLP:

process by which computers learn to interpret, comprehend, and alter textual material.
eg. ChatGPT

1.4 Risks and Benefits of AI

various forms of depictions in movies
eg. the matrix, terminator, space odyssey series

risks

- Unsustainable(building material)
- lesser jobs
- threat to humanity?(self-awareness)

benefits

- reduction in human error
- take risk instead of humans
- available 24×7
- helps in repetitive jobs
- digital assistance
- faster decisions
- daily applications

- new inventions

1.5 Intelligent Agents, Agents and Environments

Agents

ai system = study(rational agents, environment)

agent can be anything that perceives env through sensors and acts upon that env through actuators

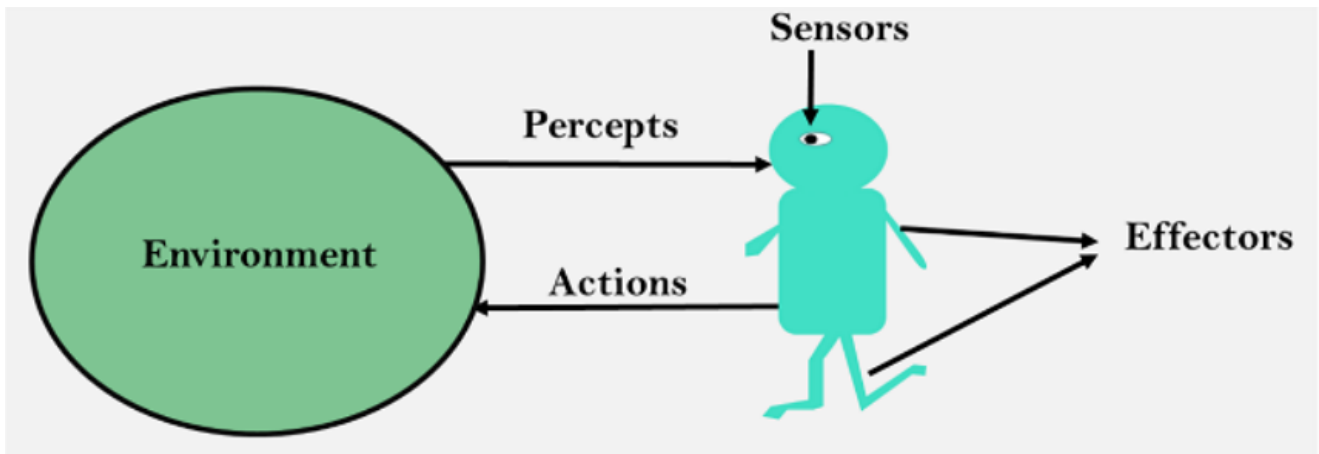
agent runs in cycle: perceive, think, act

agent can be:

- Human agent
- robotic agent
- software agent

components of agent

- sensors: devices to sense
- actuators: energy to motion
- effectors: affect env



Intelligent Agents:

autonomous entity which act upon an environment using sensors and actuators for achieving goals.

An intelligent agent may learn from the environment to achieve their goals.

eg. thermostat

rules for ai agent

1. Rule 1: must have ability to perceive
2. Rule 2: use observation to make decision
3. Rule 3: decision → action
4. Rule 4: action must be rational

Rational Agent:

has clear preference, models uncertainty, acts to maximise performance measure

performs right things

rational agents \approx intelligent agents

Structure of an AI Agent

agent = architecture + agent program

arch : machinery agent runs on

agent function: map percept to actions

agent program: implimentation of agent function

PEAS representation

ai agent model

- P: performance measure
- E: environment
- A: actuators
- S: sensors

1.6 Good Behavior: Concept of Rationality, Nature of Environments

Rationality

The state of being reasonable, sensible, and having a good sense of judgment is

known as rationality.

can be measured by performance measure

- performance Measure
- prior knowledge
- best possible action sequence
- sequence of percepts

rationality \neq omniscience

not everything known about env

1.7 Categories of intelligent agents

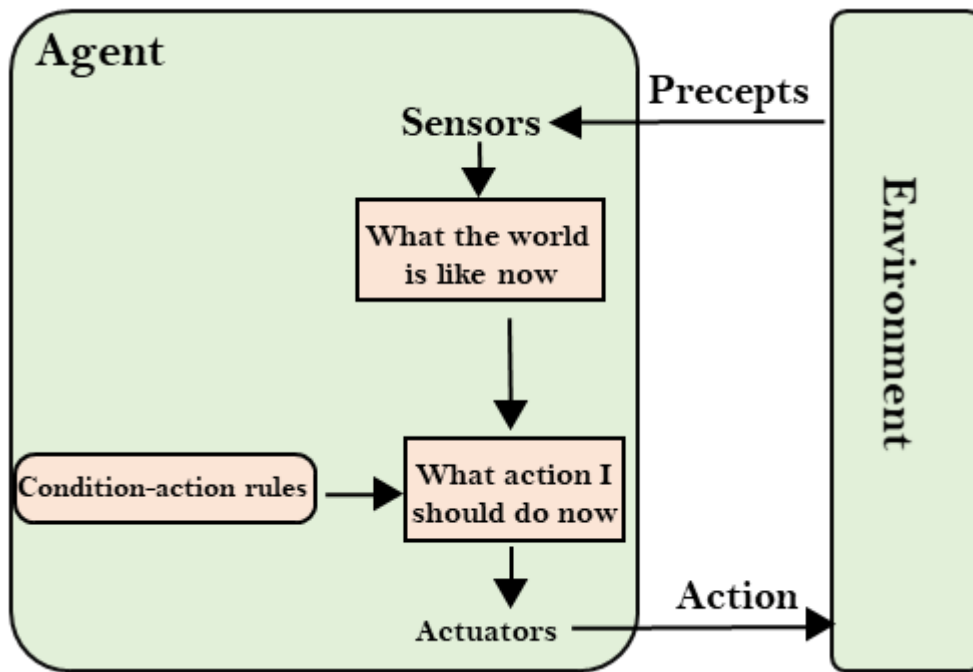
Simple reflex agents

perform actions using the current percept, rather than the percept history.

condition-action rule is used as the basis for the agent function.

fully observable environment is ideal for the

success of the agent function.



Model-based reflex agents

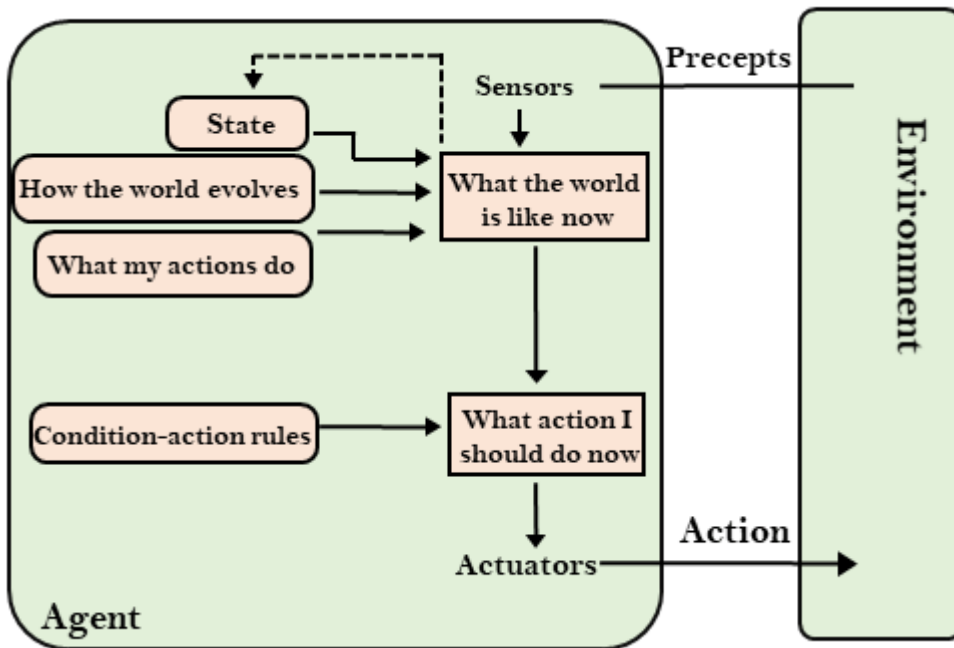
consider the percept history in their actions.

can still work well even in an environment that is not fully observable

use an internal model that determines the percept history and effect of actions

reflect on certain aspects of the present state that

have been unobserved.



Goal-based agents

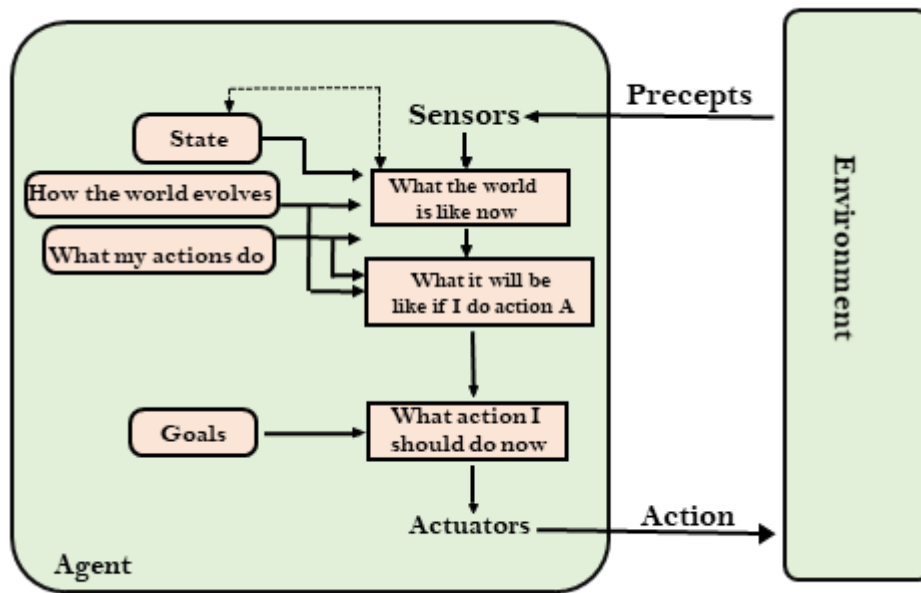
have higher capabilities

use goal information to describe desirable capabilities.

choose among various possibilities.

select the best action that enhances the

attainment of the goal.



Utility-based agents

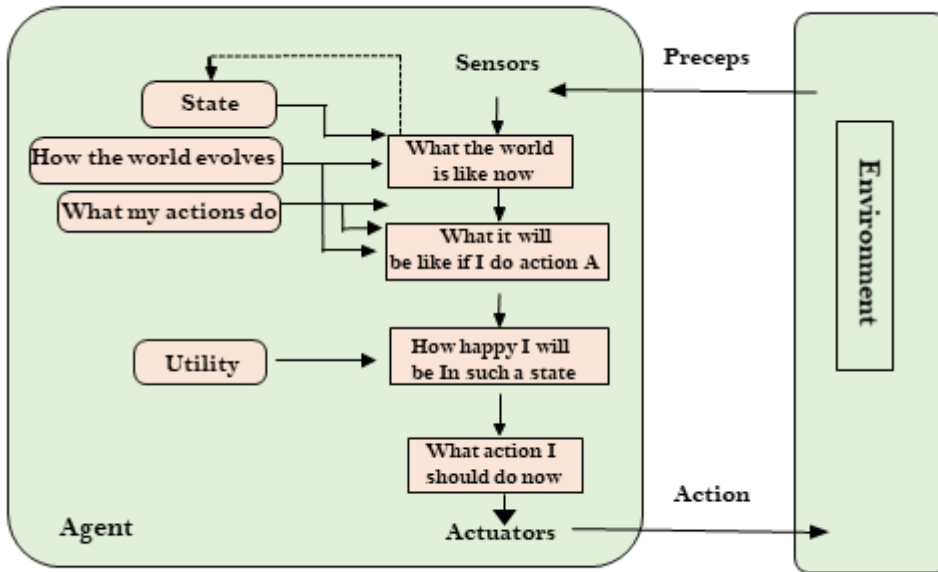
make choices based on utility.

more advanced because of an extra component of utility measurement.

a state is mapped against a certain measure of utility.

A rational agent selects the action that optimizes

the expected utility of the outcome.



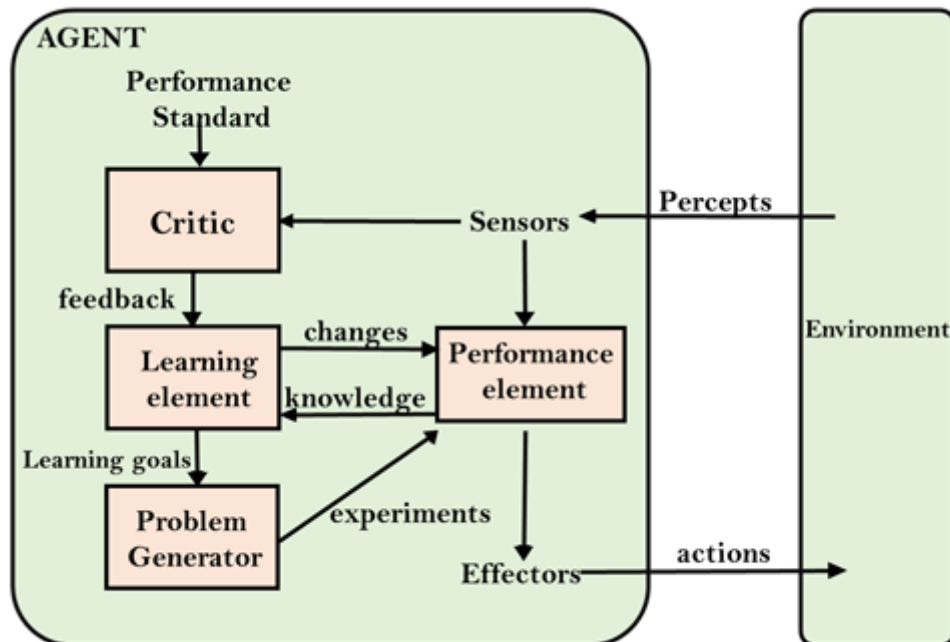
Learning agents

These are agents that have the capability of learning from their previous experience.

Learning agents have the following elements.

- The learning element: This element enables learning agents to learn from previous experiences.
- The critic: It provides feedback on how the agent is doing.
- The performance element: This element decides on the external action that needs to be taken.

- The problem generator: This acts as a feedback agent that performs certain tasks such as making suggestions (new) and keeping history



Unit 2

2.1 Solving Problems by Searching, Problem-Solving Agents

search algo terminologies:

- Search
 - search space
 - start state

- goal test
- search tree
- actions
- transition model
- path cost
- solutions
- optimal solutions

Properties of Search Algorithms:

four essential properties of search algorithms to compare the efficiency of these algorithms:

- Completeness
- Optimality
- Time Complexity
- Space Complexity

search algorithms

types:

- uninformed
 - BFS

- DFS
- uniform cost search
- depth limited search
- iterative deepening DFS
- bidirectional search
- informed search
 - best first search(greedy approach)
 - A* search

Uninformed search

1. Breadth-first Search:

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

Advantages:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

Disadvantages:

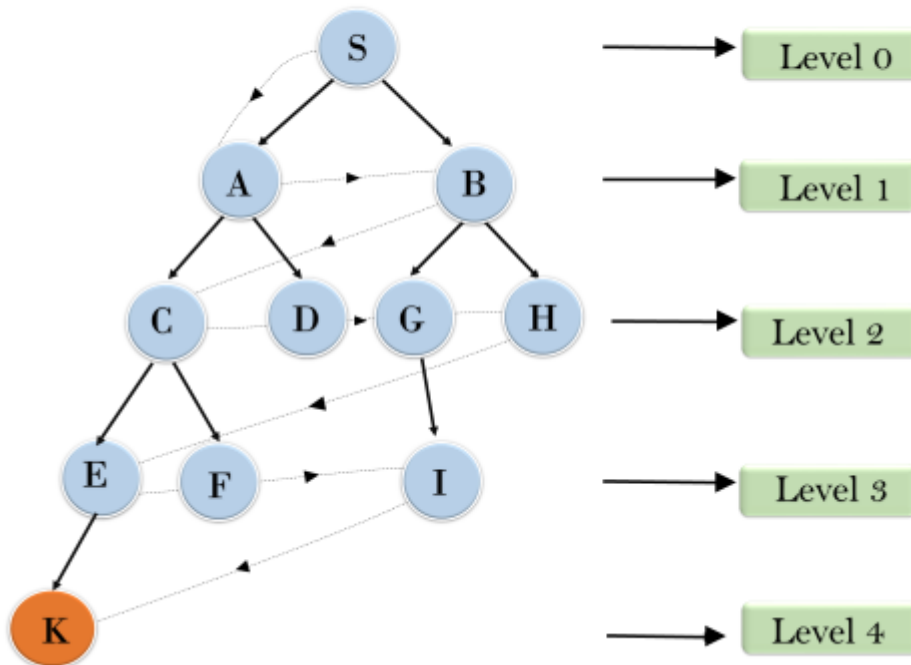
- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

Example:

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

S----> A---->B----->C--->D----->G--->H--->E----->F---
->I----->K

Breadth First Search



Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d = depth of shallowest solution and b is a node at every state.

$$T(b) = 1 + b + b^2 + b^3 + \dots + b^d = O(b^{d+1})$$

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.

Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

2. Depth-first Search

- Depth-first search 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.
- The process of the DFS algorithm is similar to the BFS algorithm.

Note: Backtracking is an algorithm technique for finding all possible solutions using recursion.

Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage:

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

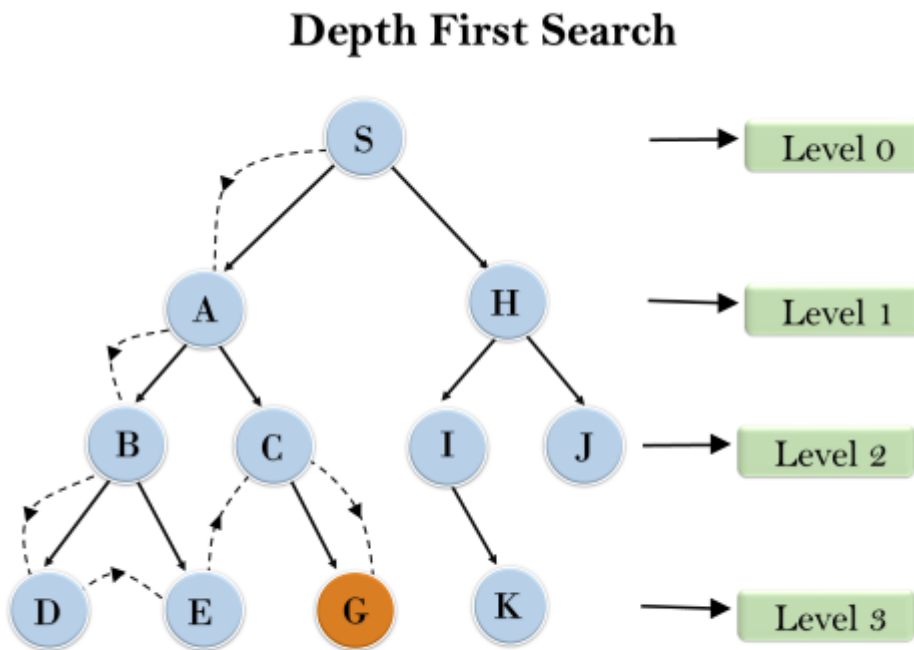
Example:

In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:

Root node —→ Left node - —→ right node.

It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After

backtracking it will traverse node C and then G, and here it will terminate as it found goal node.



Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

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

Where, m = maximum depth of any node and this can be much larger than d (Shallowest solution depth)

Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **$O(bm)$** .

Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

3. Depth-Limited Search Algorithm:

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

Depth-limited search can be terminated with two Conditions of failure:

- Standard failure value: It indicates that problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit.

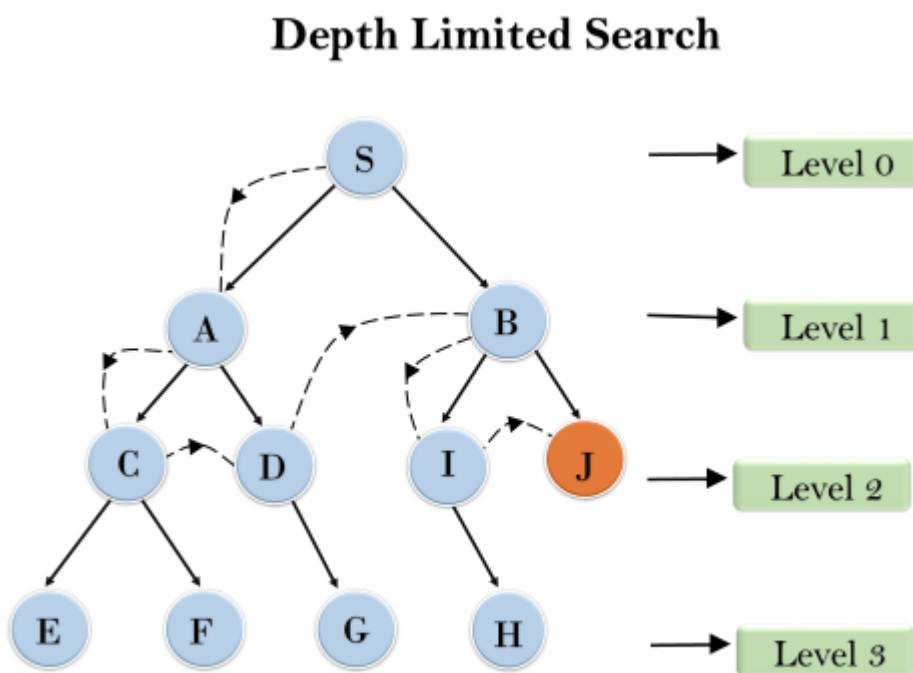
Advantages:

Depth-limited search is Memory efficient.

Disadvantages:

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

Example:



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

Time Complexity: Time complexity of DLS algorithm is $O(b^l)$.

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

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

4. Uniform-cost Search Algorithm:

Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost.

Uniform-cost search expands nodes according to their path costs from the root node. It can be used to solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost.

Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

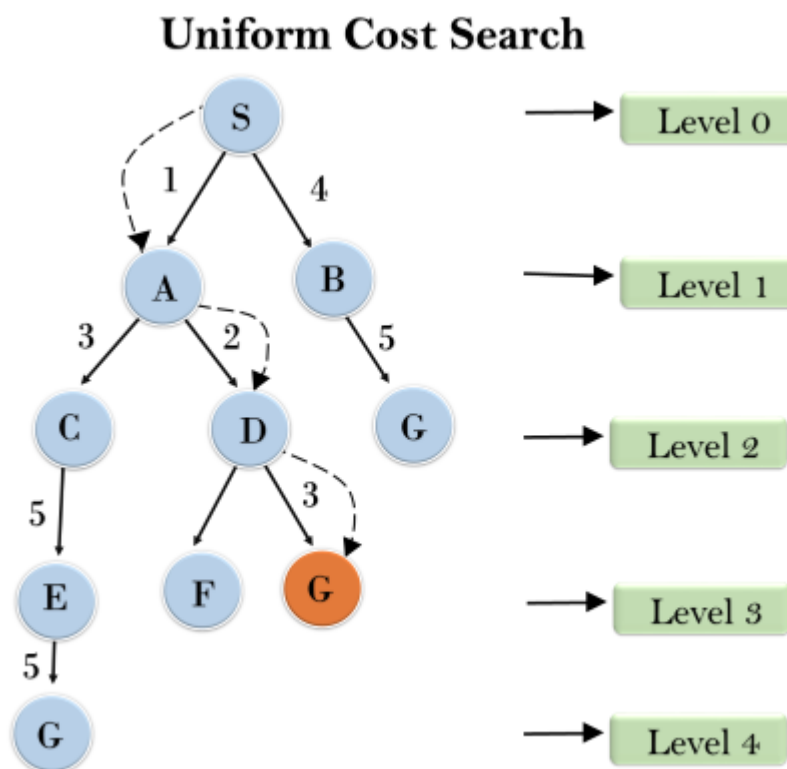
Advantages:

- Uniform cost search is optimal because at every state the path with the least cost is chosen.

Disadvantages:

- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Example:



Completeness:

Uniform-cost search is complete, such as if there is a solution, UCS will find it.

Time Complexity:

Let C **is Cost of the optimal solution**, and ϵ is each step to get closer to the goal node. Then the number of steps is $= C/\epsilon + 1$. Here we have taken $+1$, as we start from state 0 and end to C^*/ϵ .

Hence, the worst-case time complexity of Uniform-cost search is $O(b^1 + \lceil C^*/\epsilon \rceil)$.

Space Complexity:

The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $O(b^1 + \lceil C^*/\epsilon \rceil)$.

Optimal:

Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

5. Iterative deepening depth-first Search:

The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.

This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.

This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.

The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Advantages:

- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

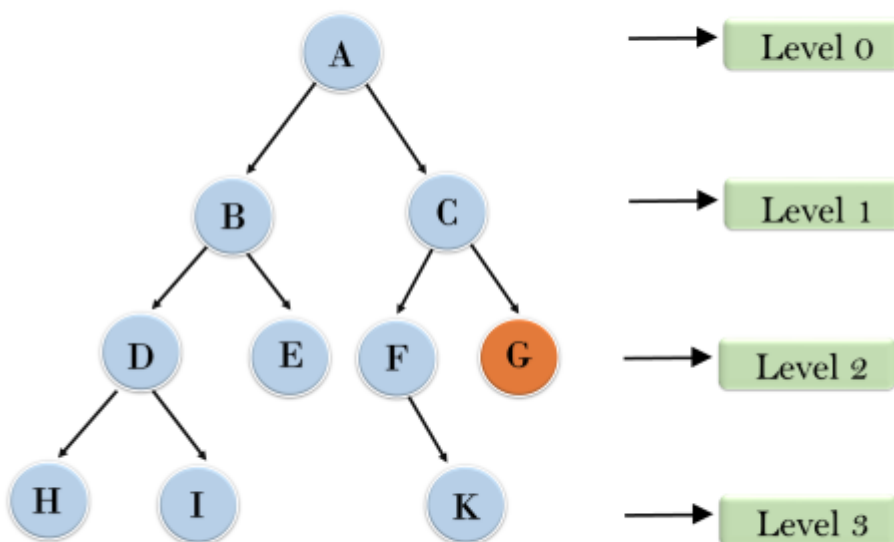
Disadvantages:

- The main drawback of IDDFS is that it repeats all the work of the previous phase.

Example:

Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:

Iterative deepening depth first search



1'st Iteration--→ A

2'nd Iteration--→ A, B, C

3'rd Iteration---→ A, B, D, E, C, F, G

4'th Iteration---→ A, B, D, H, I, E, C, F, K, G

In the fourth iteration, the algorithm will find the goal node.

Completeness:

This algorithm is complete if the branching factor is finite.

Time Complexity:

Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **$O(bd)$** .

Space Complexity:

The space complexity of IDDFS will be **$O(bd)$** .

Optimal:

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

6. Bidirectional Search Algorithm:

Bidirectional search algorithm runs two simultaneous searches, one from initial state called as forward-search and other from goal node called as backward-search, to find the goal node. Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and

other starts from goal vertex. The search stops when these two graphs intersect each other.

Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Advantages:

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

Disadvantages:

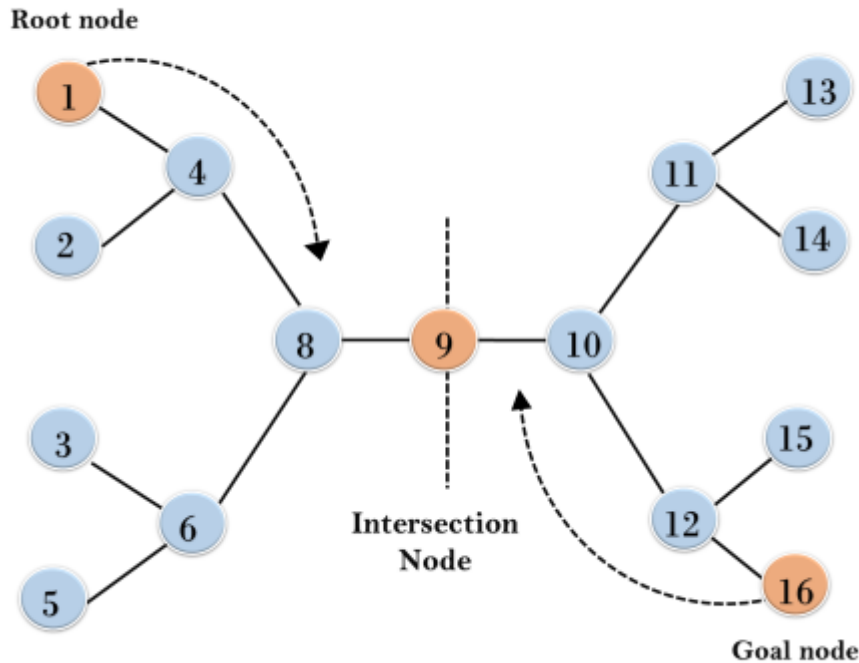
- Implementation of the bidirectional search tree is difficult.
- **In bidirectional search, one should know the goal state in advance.**

Example:

In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.

The algorithm terminates at node 9 where two searches meet.

Bidirectional Search



Completeness: Bidirectional Search is complete if we use BFS in both searches.

Time Complexity: Time complexity of bidirectional search using BFS is $O(bd)$.

Space Complexity: Space complexity of bidirectional search is $O(bd)$.

Heuristics function: Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution

in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by $h(n)$, and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Admissibility of the heuristic function is given as:

$$1. h(n) \leq h^*(n)$$

Here $h(n)$ is heuristic cost, and $h^*(n)$ is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.

Pure Heuristic Search(informed search):

Pure heuristic search is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value $h(n)$. It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.

On each iteration, each node n with the lowest heuristic value is expanded and generates all its

successors and n is placed to the closed list. The algorithm continues until a goal state is found.

In the informed search we will discuss two main algorithms which are given below:

- **Best First Search Algorithm(Greedy search)**
- **A* Search Algorithm**

1. Best-first Search Algorithm (Greedy Search):

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

$$1. f(n) = g(n).$$

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

The greedy best first algorithm is implemented by the priority queue.

Best first search algorithm:

- **Step 1:** Place the starting node into the OPEN list.
- **Step 2:** If the OPEN list is empty, Stop and return failure.
- **Step 3:** Remove the node n , from the OPEN list which has the lowest value of $h(n)$, and places it in the CLOSED list.
- **Step 4:** Expand the node n , and generate the successors of node n .
- **Step 5:** Check each successor of node n , and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- **Step 6:** For each successor node, algorithm checks for evaluation function $f(n)$, and then check if the node has been in either OPEN or

CLOSED list. If the node has not been in both list, then add it to the OPEN list.

- **Step 7:** Return to Step 2.

Advantages:

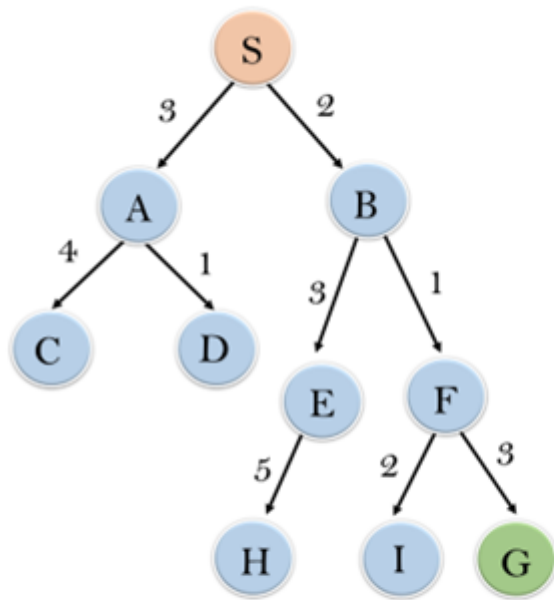
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

Disadvantages:

- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.

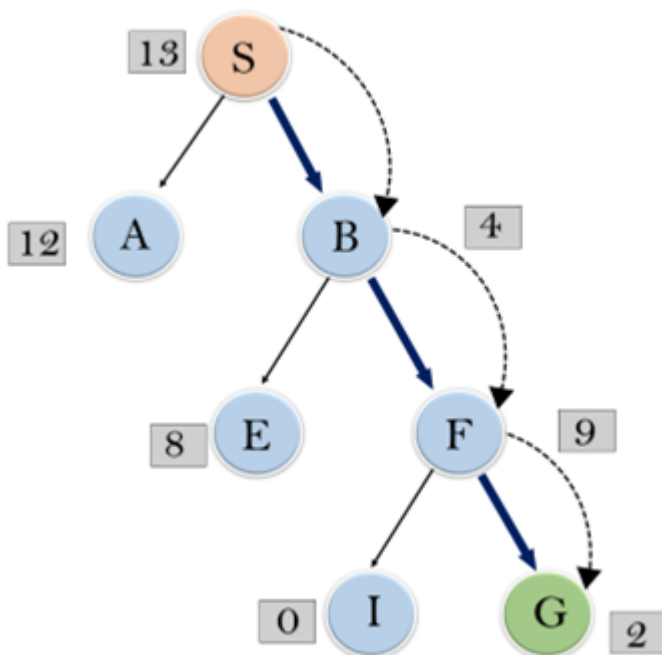
Example:

Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function $f(n)=h(n)$, which is given in the below table.



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

In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists. Following are the iteration for traversing the above example.



Expand the nodes of S and put in the **CLOSED** list

Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration 2: Open [E, F, A], Closed [S, B]

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

Iteration 3: Open [I, G, E, A], Closed [S, B, F]

: Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: **S** \longrightarrow **B** -----
>F \longrightarrow **G**

Time Complexity: The worst case time complexity of Greedy best first search is $O(bm)$.

Space Complexity: The worst case space complexity of Greedy best first search is $O(bm)$.
Where, m is the maximum depth of the search space.

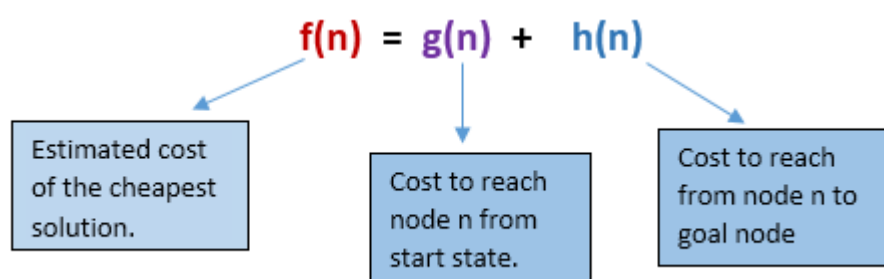
Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

Optimal: Greedy best first search algorithm is not optimal.

2. A* Search Algorithm:

A search is the most commonly known form of best-first search. It uses heuristic function $h(n)$, and cost to reach the node n from the start state $g(n)$. It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. A* algorithm is similar to UCS except that it uses $g(n)+h(n)$ instead of $g(n)$.

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



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.

Algorithm of A* search:

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

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

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n' , check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest $g(n')$ value.

Step 6: Return to **Step 2**.

Advantages:

- A* search algorithm is the best algorithm than other search algorithms.

- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

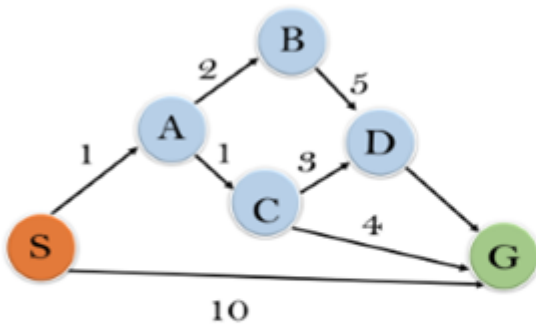
Disadvantages:

- It does not always produce the shortest path as it is mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

Example:

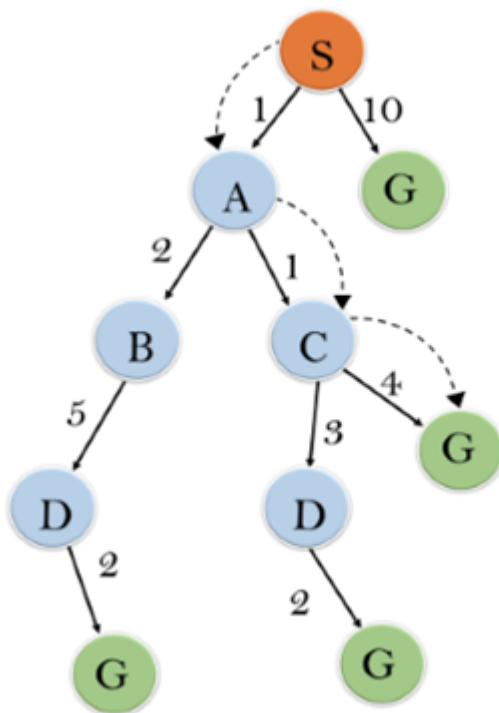
In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the $f(n)$ of each state using the formula $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach any node from start state.

Here we will use OPEN and CLOSED list.



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

Solution:



Initialization: $\{(S, 5)\}$

Iteration1: $\{(S \rightarrow A, 4), (S \rightarrow G, 10)\}$

Iteration2: $\{(S \rightarrow A \rightarrow C, 4), (S \rightarrow A \rightarrow B, 7), (S \rightarrow G, 10)\}$

Iteration3: $\{(S \rightarrow A \rightarrow C \rightarrow G, 6), (S \rightarrow A \rightarrow C \rightarrow D, 11), (S \rightarrow A \rightarrow B, 7), (S \rightarrow G, 10)\}$

Iteration 4 will give the final result, as $S \rightarrow A \rightarrow C \rightarrow G$ it provides the optimal path with cost 6.

Points to remember:

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition $f(n)$

Complete: A* algorithm is complete as long as:

- Branching factor is finite.
- Cost at every action is fixed.

Optimal: A* search algorithm is optimal if it follows below two conditions:

- **Admissible:** the first condition requires for optimality is that $h(n)$ should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
- **Consistency:** Second required condition is consistency for only A* graph-search.

If the heuristic function is admissible, then A* tree search will always find the least cost path.

Time Complexity: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d . So the time complexity is $O(b^d)$, where b is the branching factor.

Space Complexity: The space complexity of A* search algorithm is $O(b^d)$

2.8 Local Search and Optimization Problems

Hill Climbing Algorithm in Artificial Intelligence

- Hill climbing algorithm is a local search algorithm which continuously moves in the

direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.

- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.
- It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.
- A node of hill climbing algorithm has two components which are state and value.
- Hill Climbing is mostly used when a good heuristic is available.
- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

Features of Hill Climbing:

Following are some main features of Hill Climbing Algorithm:

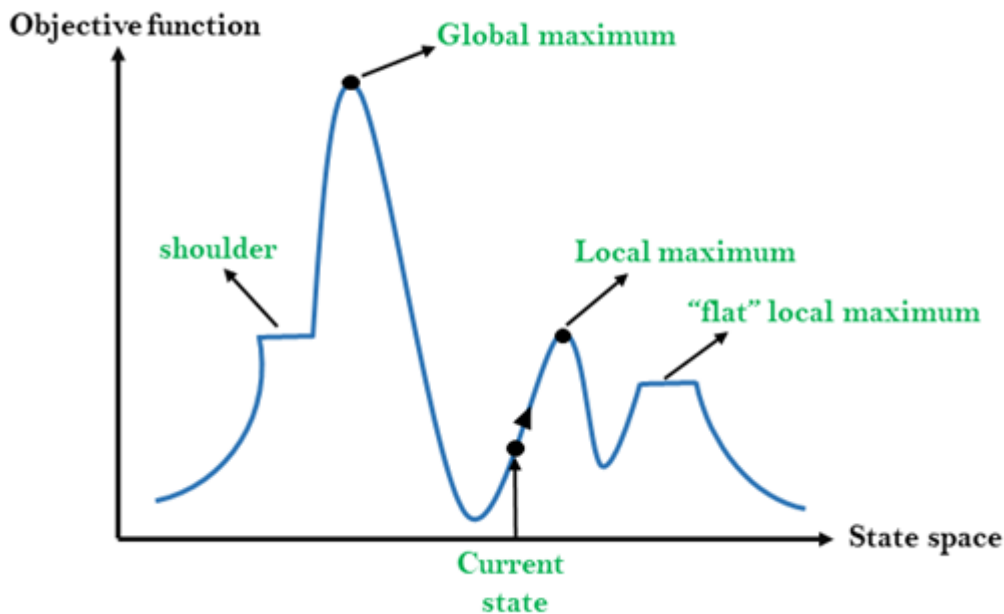
- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

State-space Diagram for Hill Climbing:

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

On Y-axis we have taken the function which can be an objective function or cost function, and state-space on the x-axis. If the function on Y-axis is

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



Different regions in the state space landscape:

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

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

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

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

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

Types of Hill Climbing Algorithm:

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

1. Simple Hill Climbing:

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

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

Algorithm for Simple Hill Climbing:

- **Step 1:** Evaluate the initial state, if it is goal state then return success and Stop.
- **Step 2:** Loop Until a solution is found or there is no new operator left to apply.
- **Step 3:** Select and apply an operator to the current state.
- **Step 4:** Check new state:
 1. If it is goal state, then return success and quit.
 2. Else if it is better than the current state then assign new state as a current state.
 3. Else if not better than the current state, then return to step2.
- **Step 5:** Exit.

2. Steepest-Ascent hill climbing:

The steepest-Ascent algorithm is a variation of simple hill climbing algorithm. This algorithm

examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state. This algorithm consumes more time as it searches for multiple neighbors

Algorithm for Steepest-Ascent hill climbing:

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

5. If the SUCC is better than the current state, then set current state to SUCC.

- **Step 3:** Exit.

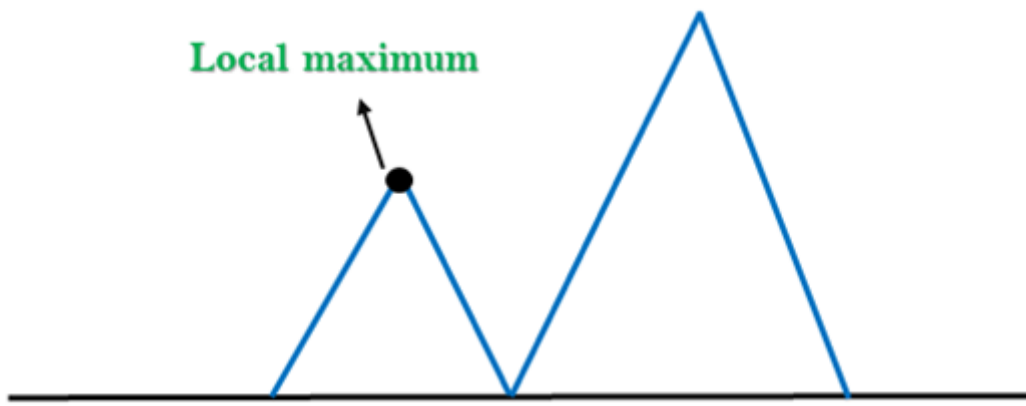
3. Stochastic hill climbing:

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

Problems in Hill Climbing Algorithm:

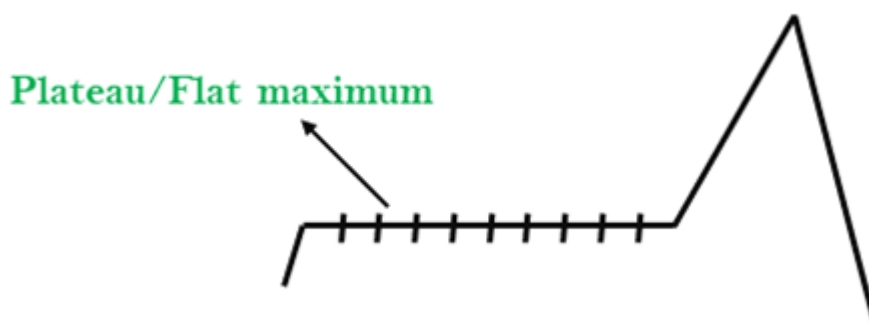
1. Local Maximum: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

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



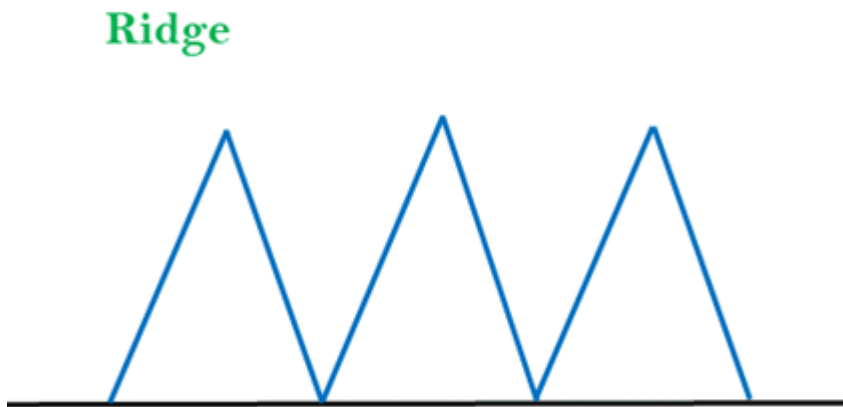
2. Plateau: A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

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



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

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



Simulated Annealing:

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

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