UNIT II PROBLEM SOLVING METHODS

Problem solving Methods – Search Strategies- Uninformed – Informed – Heuristics – Local Search Algorithms and Optimization Problems -Searching with Partial Observations – Constraint Satisfaction Problems – Constraint Propagation – Backtracking Search – Game Playing – Optimal Decisions in Games – Alpha – Beta Pruning – Stochastic Games.

2.1 PROBLEM-SOLVING AGENTS

In Artificial Intelligence, Search techniques are universal problem-solving methods. Rational agents or Problem-solving agents in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result. Problem-solving agents are the goal-based agents and use atomic representation.

2.1.1 Search Algorithm Terminologies

Search: Searching is a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:

- Search Space: Search space represents a set of possible solutions, which a system may have.
- Start State: It is a state from where agent begins the search.
- Goal test: It is a function which observe the current state and returns whether the goal state is achieved or not.

Search tree: A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.

Actions: It gives the description of all the available actions to the agent.

Transition model: A description of what each action do, can be represented as a transition model.

Path Cost: It is a function which assigns a numeric cost to each path.

Solution: It is an action sequence which leads from the start node to the goal node.

Optimal Solution: If a solution has the lowest cost among all solutions.

2.1.2 Properties of Search Algorithms:

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

- Completeness: A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.
- Optimality: If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.
- Time Complexity: Time complexity is a measure of time for an algorithm to complete its task.
- Space Complexity: It is the maximum storage space required at any point during the search, as the complexity of the problem.

2.1.3 Types of search algorithms

Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.

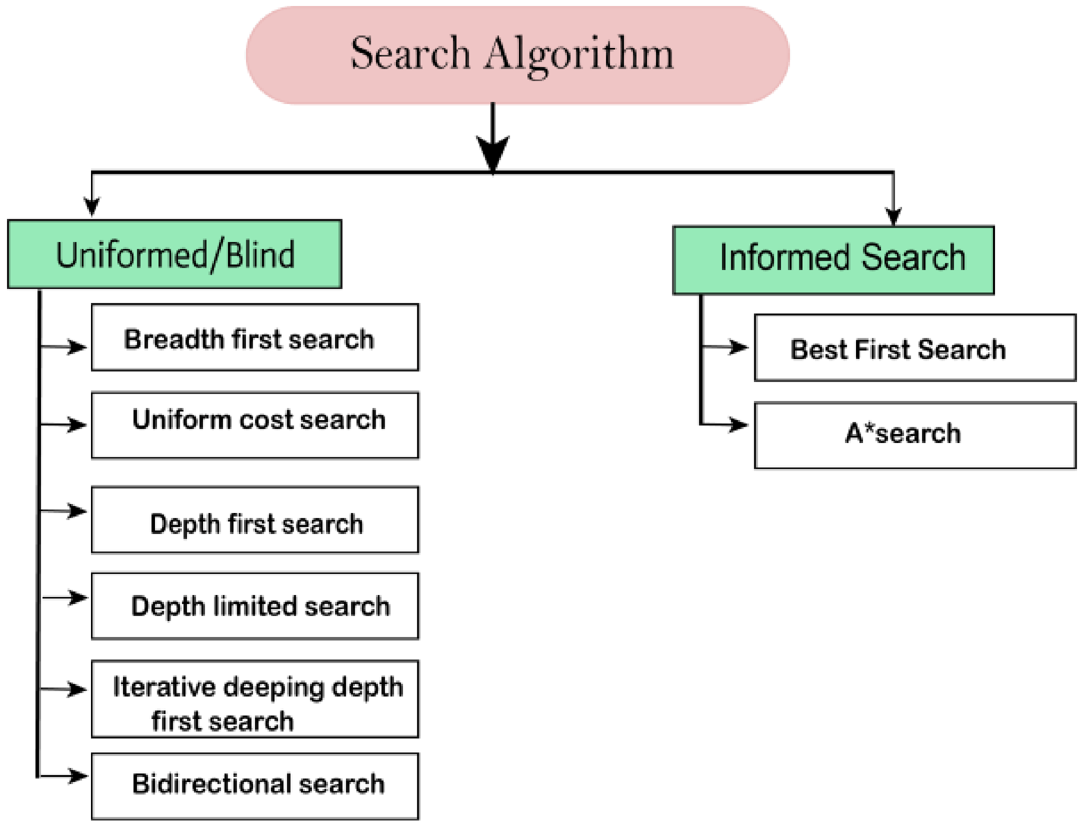


Fig: Types of Search Algorithms

2.2 UNINFORMED/BLIND SEARCH

The uninformed search does not contain any domain knowledge such as closeness, the location of the goal. It operates in a brute-force way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes. Uninformed search applies a way in which search tree is searched without any information about the search space like initial state operators and test for the goal, so it is also called blind search. It examines each node of the tree until it achieves the goal node.

It can be divided into five main types:

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

2.2.1. 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 nodes 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 is more than one solution for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

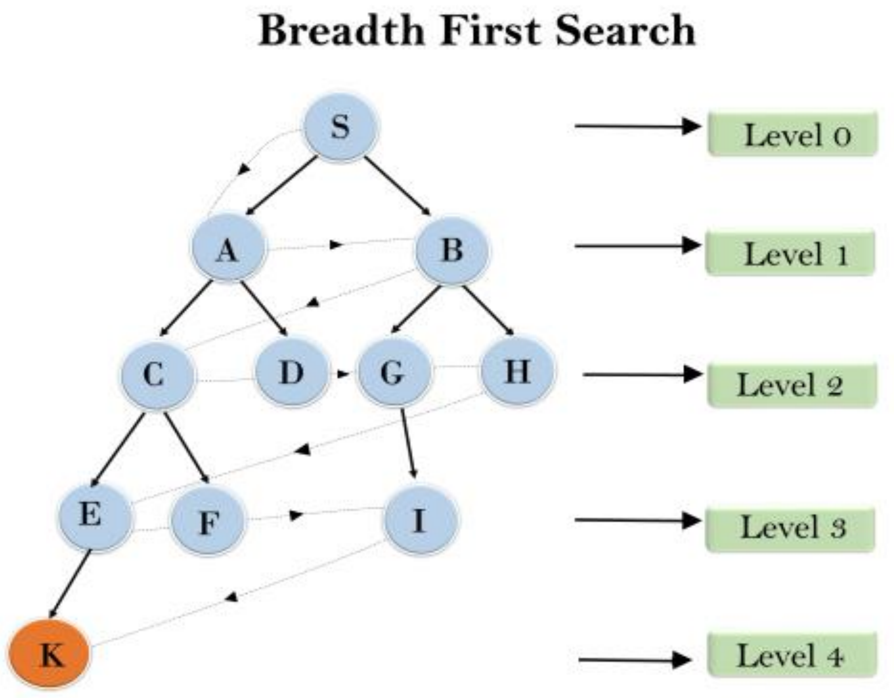
Disadvantages:

- 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:

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



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) = $1+b^2+b^3+.......+ b^d= O (b^d)$

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.

2.2.2 Depth-first Search

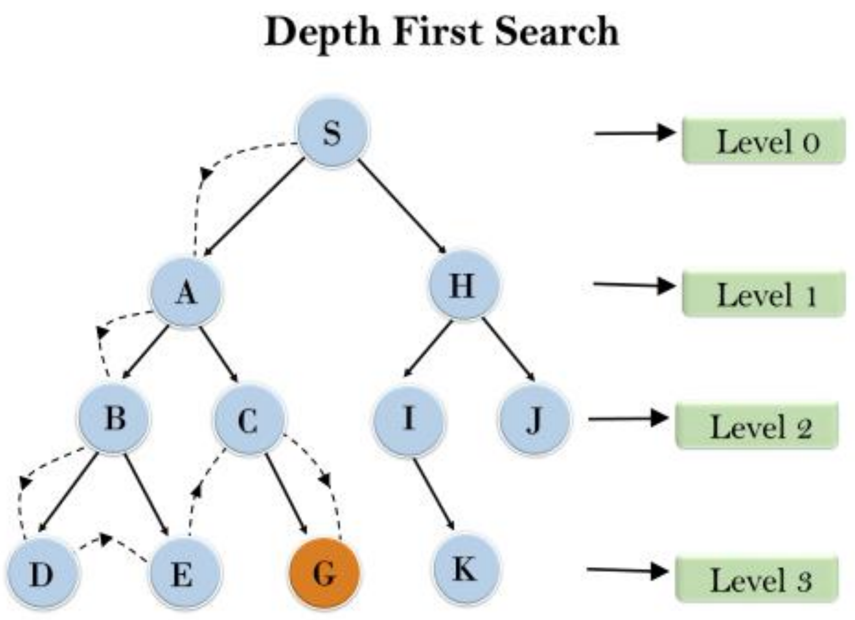
- Depth-first search is a recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.

- 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 above 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:

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

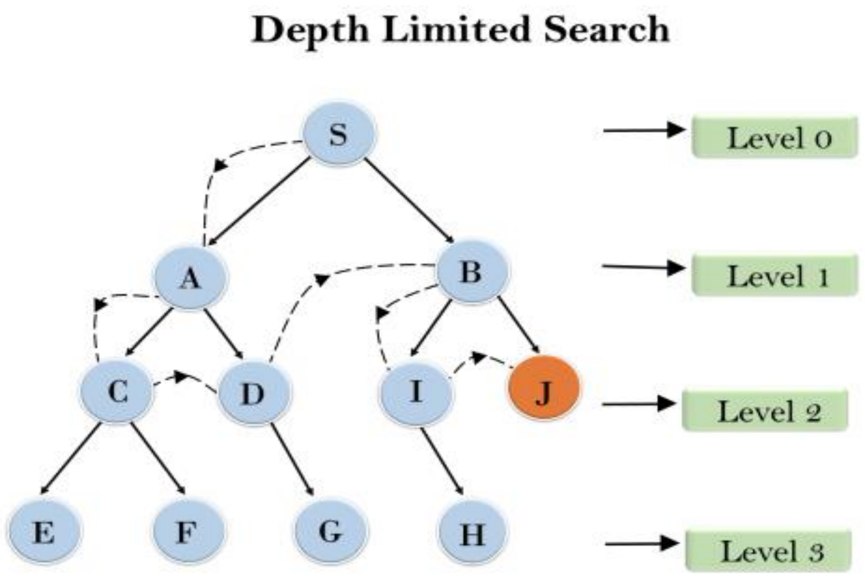
Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)

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.

2.2.3 Depth-Limited Search Algorithm

Example:



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.

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

Time Complexity: Time complexity of DLS algorithm is $\mathbf{O(b^{\ell})}$.

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

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

2.2.4. 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 form 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.

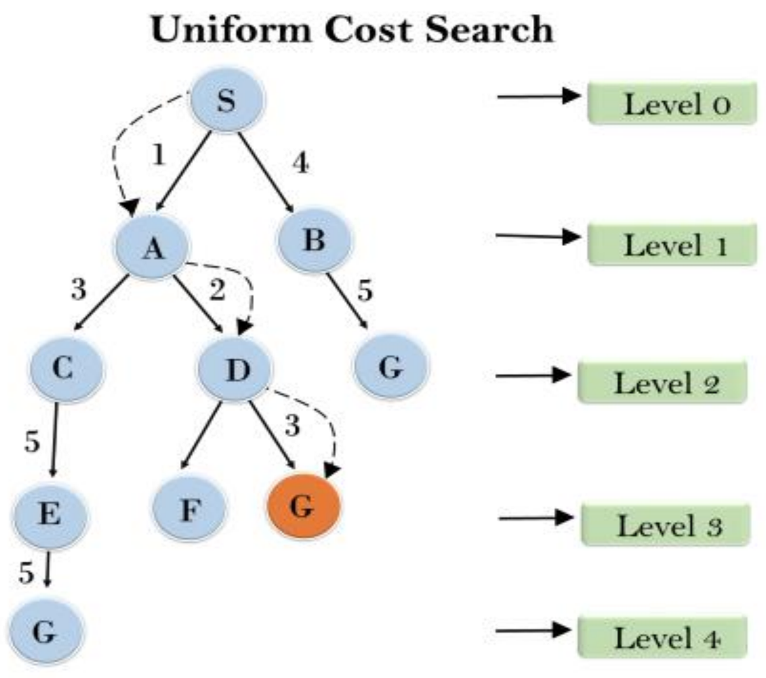
Advantages:

- 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*/ε+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**$O(b^{1 + [C^*/\varepsilon]})$/**.

Space Complexity:

The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is **$O(b^{1 + [C^*/\varepsilon]})$**.

Optimal:

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

2.2.5. 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:

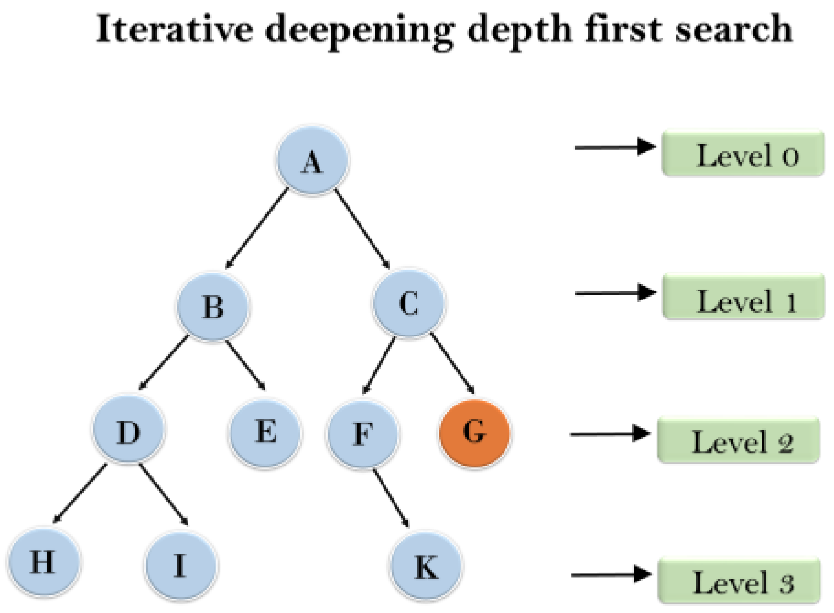
- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

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:



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 is ifthe 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(b^d)**.

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.

2.2.6. Bidirectional Search Algorithm

Bidirectional search algorithm 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. 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:

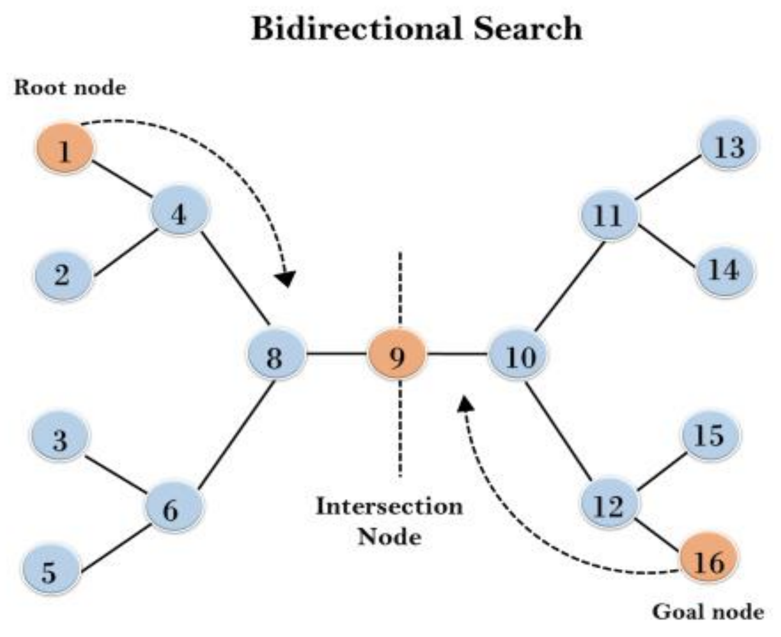
- Bidirectional search is fast.
- Bidirectional search requires less memory

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.



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

Time Complexity: Time complexity of bidirectional search using BFS is **O(b^d)**.

Space Complexity: Space complexity of bidirectional search is **O(b^d)**.

Optimal: Bidirectional search is Optimal.

2.3 INFORMED (HEURISTIC) SEARCH STRATEGIES

To solve large problems with large number of possible states, problem-specific knowledge needs to be added to increase the efficiency of search algorithms.

Heuristic Evaluation Functions

They calculate the cost of optimal path between two states. A heuristic function for sliding-tiles games is computed by counting number of moves that each tile makes from its goal state and adding these numbers of moves for all tiles.

Pure Heuristic Search

It expands nodes in the order of their heuristic values. It creates two lists, a closed list for the already expanded nodes and an open list for the created but unexpanded nodes.

In each iteration, a node with a minimum heuristic value is expanded; all its child nodes are created and placed in the closed list. Then, the heuristic function is applied to the child nodes and they are placed in the open list according to their heuristic value. The shorter paths are saved and the longer ones are disposed.

Admissibility of the heuristic function is given as:

$$h(n) <= h^*(n)$$

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.

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

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

2.3.1 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.

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

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

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

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:

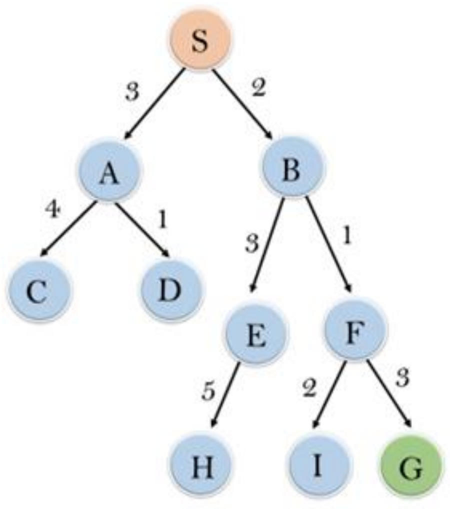
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

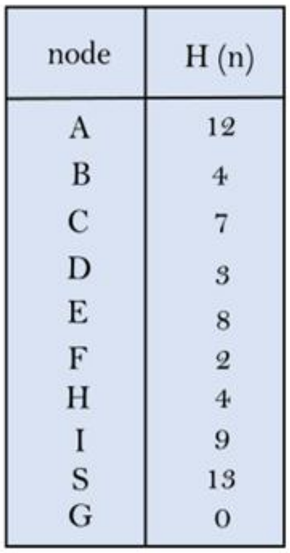
Disadvantages:

- 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:

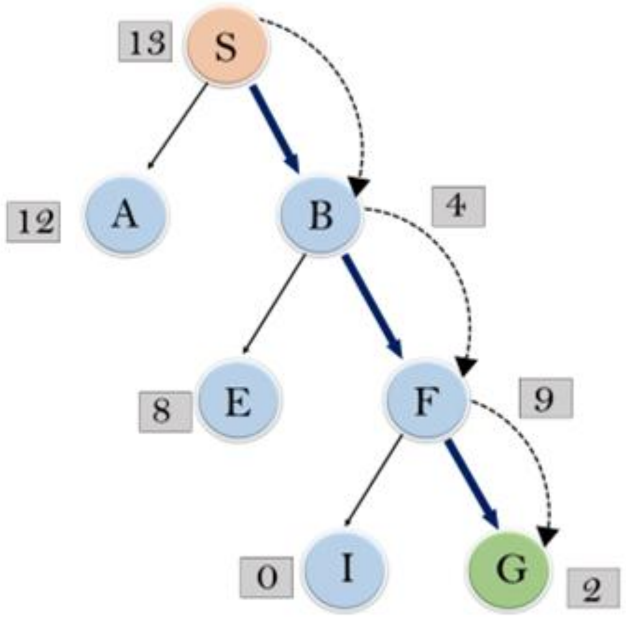
Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.





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----> B----->F----> G**

Time Complexity: The worst case time complexity of Greedy best first search is O(b^m).

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

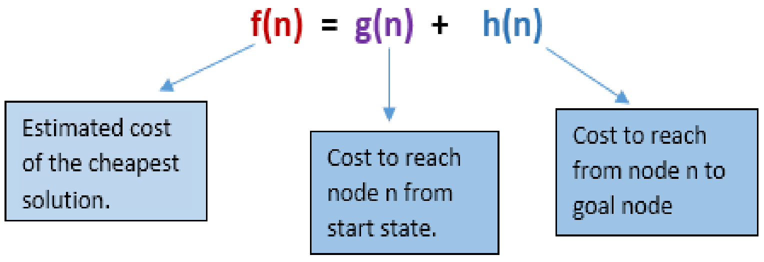
Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

Optimal: Greedy best first search algorithm is not optimal.

2.3.2 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**.



f(n) = g(n) + h(n)
Estimated cost of the cheapest solution.
Cost to reach node n from start state.
Cost to reach from node n to goal node

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:

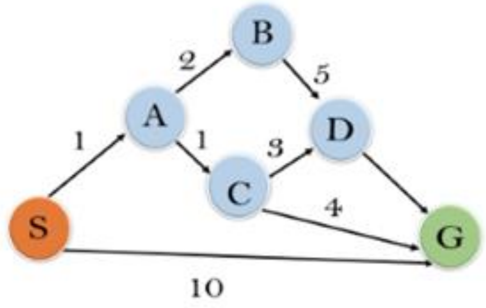
- It does not always produce the shortest path as it mostly based on heuristics and approximation.

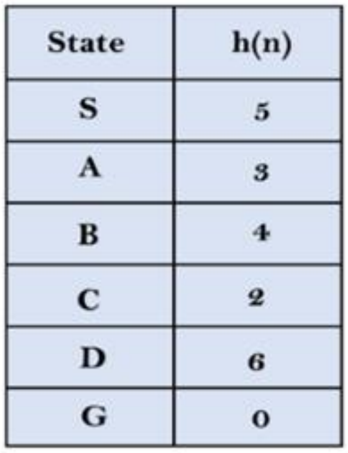
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

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.

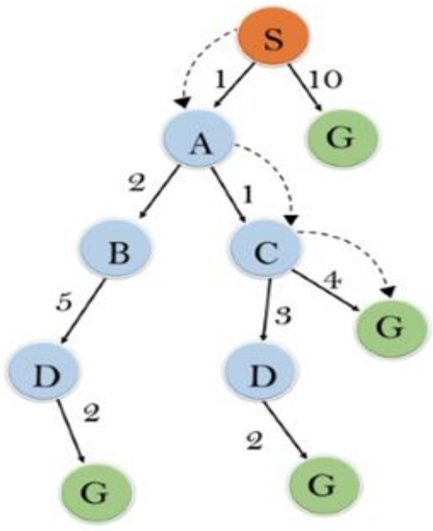
Here we will use OPEN and CLOSED list.





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

Solution:



Initialization: {(S, 5)}

Iteration1: {(S--> A, 4), (S-->G, 10)}

Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

Iteration 4 will give the final result, as **S--->A--->C--->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)<="" li="">

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)**

2.4 LOCAL SEARCH AND OPTIMIZATION

Local search is a heuristic method for solving computationally hard optimization problems. Local search can be used on problems that can be formulated as finding a solution maximizing a criterion among a number of candidate solutions. Local search algorithms move from solution to solution in the space of candidate solutions (the search space) by applying local changes, until a solution deemed optimal is found or a time bound is elapsed.

Local search algorithms are widely applied to numerous hard computational problems, including problems from computer science (particularly artificial intelligence), mathematics, operations research, engineering, and bioinformatics. Examples of local search algorithms are WalkSAT, the 2-opt algorithm for the Traveling Salesman Problem and the Metropolis–Hastings algorithm.
A local search algorithm starts from a candidate solution and then iteratively moves to a neighbor solution. This is only possible if a neighborhood relation is defined on the search space. As an example, the neighborhood of a vertex cover is another vertex cover only differing by one node. For boolean satisfiability, the neighbors of a truth assignment are usually the truth assignments only differing from it by the evaluation of a variable. The same problem may have multiple different neighborhoods defined on it; local optimization with neighborhoods that involve changing up to k components of the solution is often referred to as k-opt.

Typically, every candidate solution has more than one neighbor solution; the choice of which one to move to is taken using only information about the solutions in the neighborhood of the current one, hence the name local search. When the choice of the neighbor solution is done by taking the one locally maximizing the criterion, the metaheuristic takes the name hill climbing. When no improving configurations are present in the neighborhood, local search is stuck at a locally optimal point. This local-optima problem can be cured by using restarts (repeated local search with different initial conditions), or more complex schemes based on iterations, like iterated local search, on memory, like reactive search optimization, on memory-less stochastic modifications, like simulated annealing.

Termination of local search can be based on a time bound. Another common choice is to terminate when the best solution found by the algorithm has not been improved in a given number of steps. Local search is an anytime algorithm: it can return a valid solution even if it's interrupted at any time before it ends. Local search algorithms are typically approximation or incomplete algorithms, as the search may stop even if the best solution found by the algorithm is not optimal. This can happen even if termination is due to the impossibility of improving the solution, as the optimal solution can lie far from the neighborhood of the solutions crossed by the algorithms.

For specific problems it is possible to devise neighborhoods which are very large, possibly exponentially sized. If the best solution within the neighborhood can be found efficiently, such algorithms are referred to as very large-scale neighborhood search algorithms.

Features of Local search

- Keep track of single current state
- Move only to neighboring states
- Ignore paths

Advantages:

- Use very little memory
- Can often find reasonable solutions in large or infinite (continuous) state spaces.
- Features of “Pure optimization” problems
- All states have an objective function
- Goal is to find state with max (or min) objective value
- Does not quite fit into path-cost/goal-state formulation
- Local search can do quite well on these problems.

2.4.1 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.

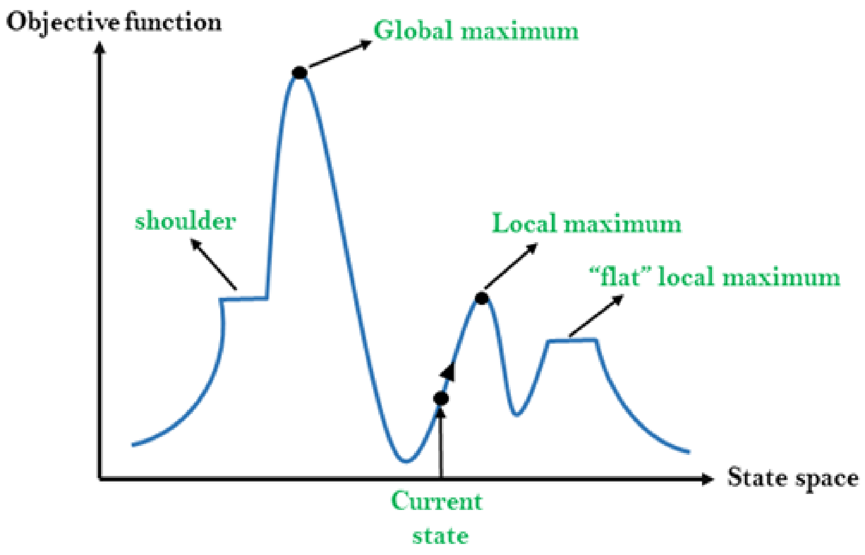
Features of Hill Climbing:

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.

2.4.2 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 it's 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:
    - If it is goal state, then return success and quit.
    - Else if it is better than the current state then assign new state as a current state.
    - 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.
  - ✓ Let SUCC be a state such that any successor of the current state will be better than it.
  - ✓ For each operator that applies to the current state:
    - ▪ Apply the new operator and generate a new state.
    - ▪ Evaluate the new state.
    - ▪ If it is goal state, then return it and quit, else compare it to the SUCC.
    - ▪ If it is better than SUCC, then set new state as SUCC.
    - ▪ If the SUCC is better than the current state, then set current state to SUCC.
- **Step 5:** Exit.

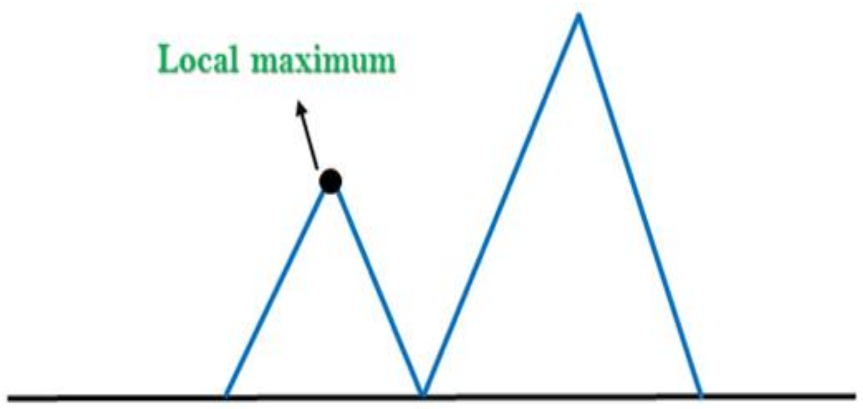
3. Stochastic hill climbing:

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:

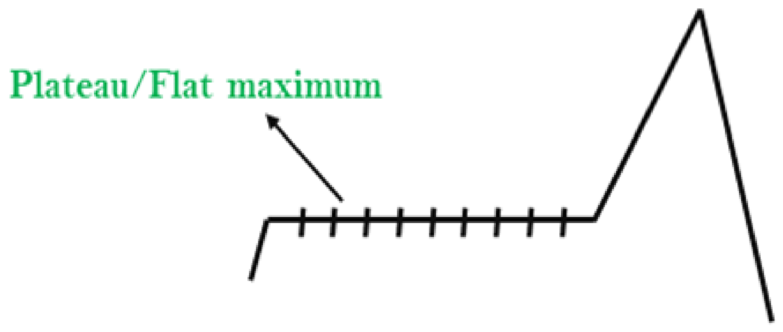
a. Local Maximum: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

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.



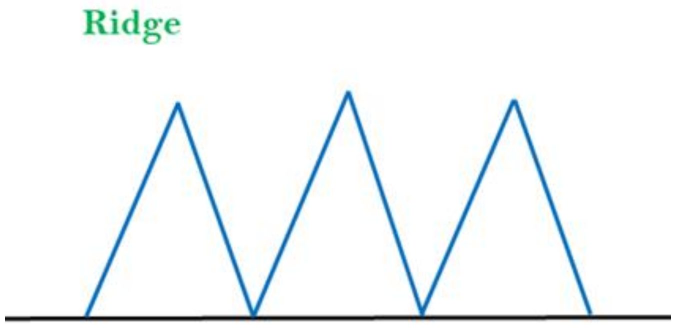
b. Plateau: A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

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.



c. 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.



2.4.3 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.

Annealing is the process of heating and cooling a metal to change its internal structure for modifying its physical properties. When the metal cools, its new structure is seized, and the metal retains its newly obtained properties. In simulated annealing process, the temperature is kept variable. We initially set the temperature high and then allow it to 'cool' slowly as the algorithm proceeds. When the temperature is high, the algorithm is allowed to accept worse solutions with high frequency.

- Start
- Initialize k = 0; L = integer number of variables;
- From i → j, search the performance difference Δ.
- If Δ <= 0 then accept else if exp(-Δ/T(k)) > random(0,1) then accept;
- Repeat steps 1 and 2 for L(k) steps.
- k = k + 1;
- Repeat steps 1 through 4 till the criteria is met.

- End

Hill-climbing (Greedy Local Search) max version

```
 function HILL-CLIMBING( problem) return a state that is a local maximum
 input: problem, a problem local
variables: current, a node. neighbor, a node.
 current ← MAKE-NODE(INITIAL-STATE[problem])
 loop do
neighbor ← a highest valued successor of current
 if VALUE [neighbor] ≤ VALUE[current] then return STATE[current]
current ← neighbor
```

2.4.4 Travelling Salesman Problem

In this algorithm, the objective is to find a low-cost tour that starts from a city, visits all cities en-route exactly once and ends at the same starting city.

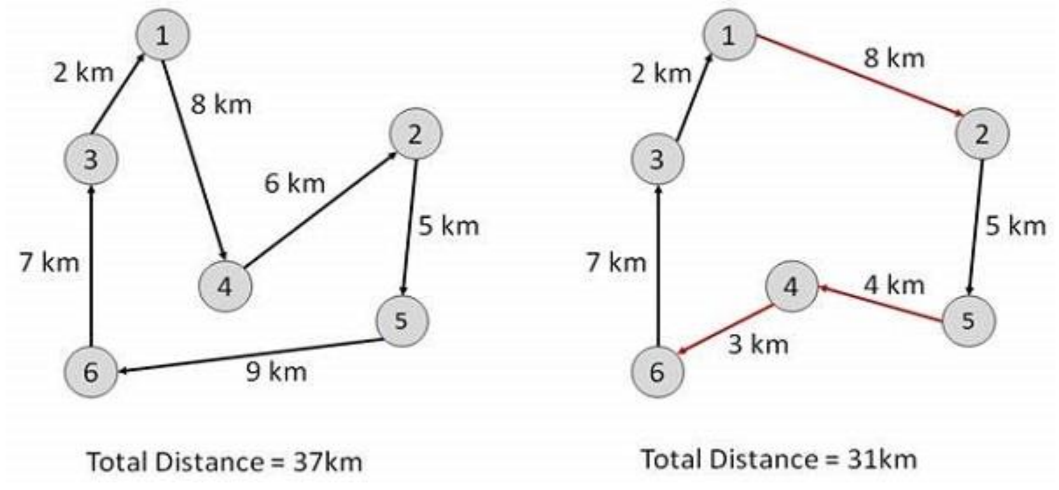
Start

Find out all (n -1)! Possible solutions, where n is the total number of cities.

Determine the minimum cost by finding out the cost of each of these (n -1)! solutions.

Finally, keep the one with the minimum cost.

end



2.5 CONSTRAINT SATISFACTION PROBLEMS

Constraint Satisfactory problems, as the name suggests are the problems which have some constraints which need to be satisfied while solving any problem. In simpler words, we can say that while solving any problem or changing any state to reach to the goal state, our Agent cannot violate the constraints which are defined for any problem in prior.

In solving the **constraint satisfactory problems**, an Agent has the following parameters to consider: **a set of variables**, **Domain**, **Set of constraints**.

Set of variables:

The variables are like empty containers in an agent (system) where the data is to be placed. Any information or data that the agent deals with has to be stored somewhere and this work is done by the variables. The collection of all the variables that the agent uses to solve that particular problem is known as the set of variables for that problem.

Domain:

Domain is the field in which knowledge is to be attained by the agent. Any problem deals with the information or data in any particular field and the domain defines this field. A domain can simply be defined as what data or information is to be collected or placed.

Set of Constraints:

It is the collection of all the restrictions and regulations that are imposed on the agent while solving the problem. The Agent cannot violate or avoid these restrictions while performing any action.

A constraint satisfaction problem (or CSP) is defined by a set of variables, X1, X2, . . . , Xn, and a set of constraints, C1, C2, . . . , Cm. Each variable Xi has a nonempty domain Di of possible values. Each constraint Ci involves some subset of the variables and specifies the allowable combinations of values for that subset. A state of the problem is defined by an assignment of values to some or all of the variables, {Xi = vi , Xj = vj , . . .}. An assignment that does not violate any constraints is called a consistent or legal assignment. A complete assignment is one in which every variable is mentioned, and a solution to a CSP is a complete assignment that satisfies all the constraints. Some CSPs also require a solution that maximizes an objective function

Any agent (Artificial Intelligence-based) who solves the **constraint Satisfactory problems** has the above-mentioned parameters embedded in its system. There are many games and puzzles which lie under the **constraint satisfactory problem**. Some commonly known among them are Sudoku, N-queens problem, Map Coloring Problem, Crossword, Crypt-Arithmetic Problem, Latin Square Problem, etc. All these problems have their own set of rules and regulations that should not be violated and these rules and regulations act as constraints for each of them.

Many problems in AI can be considered as problems of constraint satisfaction, in which the goal state satisfies a given set of constraint. Constraint satisfaction problems can be solved by using any of the search strategies. The general form of the constraint satisfaction procedure is as follows:

Until a complete solution is found or until all paths have led to lead ends, do

a. Select an unexpanded node of the search graph.
b. Apply the constraint inference rules to the selected node to generate all possible new constraints.
c. If the set of constraints contains a contradiction, then report that this path is a dead end.
d. If the set of constraints describes a complete solution then report success.
e. If neither a constraint nor a complete solution has been found then apply the rules to generate new partial solutions. Insert these partial solutions into the search graph.

2.5.1 Solution

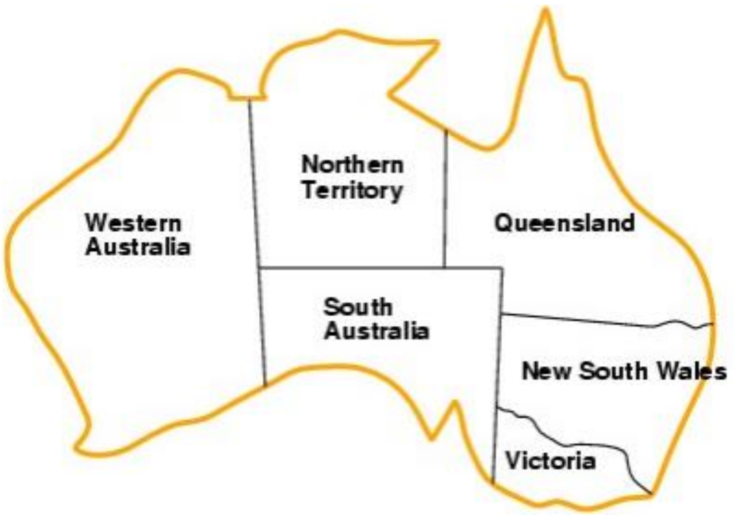
- Each state in a CSP is defined by an assignment of values to some or all of the variables
- An assignment that does not violate any constraints is called a consistent or legal assignment
- A complete assignment is one in which every variable is assigned.
- A solution to a CSP is consistent and complete assignment.
- Allows useful general-purpose algorithms with more power than standard search algorithms

Example: Map Coloring

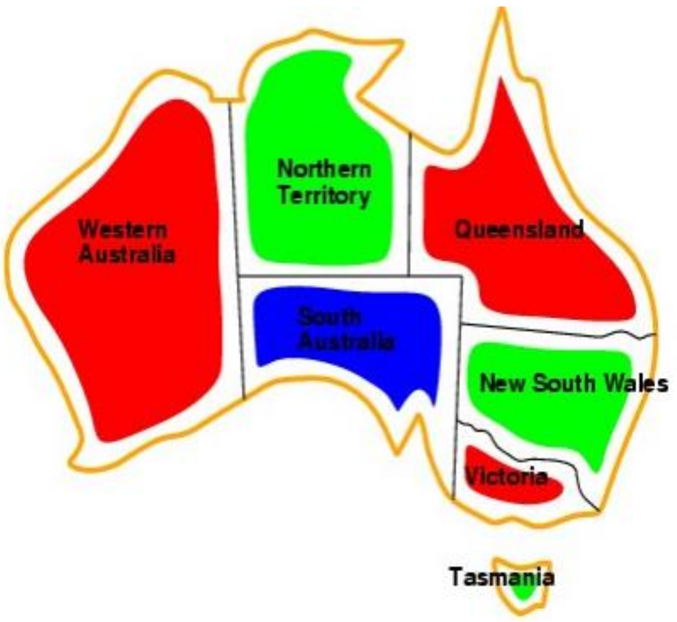
Variables: X = {WA, NT, Q, NSW, V, SA, T}

Domains: Di = {red, green, blue}

Constraints: adjacent regions must have different colors



Solution: Complete and Consistent Assignment



Variables: X = {WA, NT, Q, NSW, V, SA, T}.

Domains: Di = {red, green, blue}

Constraints: adjacent regions must have different colors.

Solution: {WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red}.

2.5.2 Constraint Graph

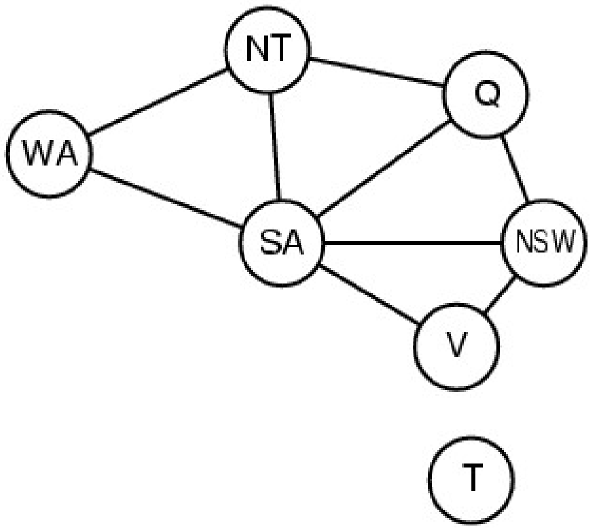
Constraint graph: nodes are variables, arcs are constraints.

Binary CSP: each constraint relates two variables.

CSP conforms to a standard pattern

- ✓ a set of variables with assigned values

- ✓ generic successor function and goal test
- ✓ generic heuristics
- ✓ reduce complexity



2.5.3 CSP as a Search Problem

Initial state:

{} – all variables are unassigned

Successor function: a value is assigned to one of the unassigned variables with no conflict

Goal test: a complete assignment

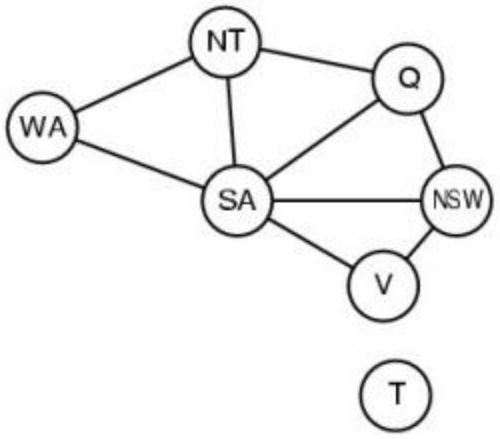
Path cost: a constant cost for each step; Solution appears at depth n if there are n variables; Depth-first or local search methods work well.

CSP Solvers Can be Faster

CSP solver can quickly eliminate large part of search space

If {SA = blue}

Then 3^5 assignments can be reduced to 2^5 assignments, a reduction of 87%



In a CSP, if a partial assignment is not a solution, we can immediately discard further refinements of it.

2.5.4 Types of Constraints

- Unary constraints involve a single variable, e.g., SA ≠ green
- Binary constraints involve pairs of variables, e.g., SA ≠ WA
- Higher-order constraints involve 3 or more variable, e.g., crypt arithmetic column constraints

2.5.5 Real-World CSPs

- Assignment problems

  e.g., who teaches what class
- Timetabling problems

  e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling

2.5.6 Commutativity

A crucial property to all CSPs: commutativity

The order of application of any given set of actions has no effect on the outcome .Variable assignments are commutative, i.e., [WA = red then NT = green] same as [NT = green then WA = red]

2.5.7 Varieties of CSPs

a) Discrete variables

Finite domains:

- n variables, domain size d Æ O(dn) complete assignments
- e.g., Boolean CSPs, incl.~Boolean satisfiability (NP-complete)

Infinite domains:

- integers, strings, etc.
- e.g., job scheduling, variables are start/end days for each job
- need a constraint language, e.g., StartJob1 + 5 ≤ StartJob3

b) Continuous variables

- e.g., start/end times for Hubble Space Telescope observations
- linear constraints solvable in polynomial time by linear programming

2.5.8 Varieties of constraints

Unary constraints involve a single variable,

• e.g., SA ≠ green

Binary constraints involve pairs of variables,

• e.g., SA ≠ WA

Higher-order constraints involve 3 or more variables,

• e.g., crypt arithmetic column constraints

2.5.9 Example: consider the crypt arithmetic problems

The **Crypt-Arithmetic problem in** Artificial Intelligence is a type of encryption problem in which the written message in an alphabetical form which is easily readable and understandable is converted into a numeric form which is neither easily readable nor understandable. In simpler words, the crypt-arithmetic problem deals with the converting of the message from the readable plain text to the non-readable ciphertext. The constraints which this problem follows during the conversion is as follows:

- A number 0-9 is assigned to a particular alphabet.
- Each different alphabet has a unique number.
- All the same, alphabets have the same numbers.
- The numbers should satisfy all the operations that any normal number does.

Let us take an example of the message: **SEND MORE MONEY.**

Here, to convert it into numeric form, we first split each word separately and represent it as follows:

```
S E N D
M O R E
-------------
M O N E Y
```

These alphabets then are replaced by numbers such that all the constraints are satisfied. So initially we have all blank spaces. We first look for the MSB in the last word which is **'M'** in the word **'MONEY'** here. It is the letter which is generated by carrying. So, carry generated can be only one. SO, we have **M=1**.

Now, we have **S+M=O** in the second column from the left side. Here **M=1**. Therefore, we have, **S+1=O**. So, we need a number for **S** such that it generates a carry when added with **1**. And such a number is **9**. Therefore, we have **S=9** and **O=0**.

Now, in the next column from the same side we have **E+O=N**. Here we have **O=0**. Which means **E+0=N** which is not possible. This means a carry was generated by the lower place digits. So we have:

1+E=N ----------(i)

Next alphabets that we have are **N+R=E -------(ii)**

So, for satisfying both equations **(i) and (ii)**, we get **E=5 and N=6**.

Now, **R** should be **9**, but **9** is already assigned to **S**, So, **R=8** and we have **1** as a carry which is generated from the lower place digits.

Now, we have **D+5=Y** and this should generate a carry. Therefore, **D** should be greater than **4**. As **5, 6, 8** and **9** are already assigned, we have **D=7** and therefore **Y=2**.

Therefore, the **solution to the given Crypt-Arithmetic problem is**:

S=9; E=5; N=6; D=7; M=1; O=0; R=8; Y=2

Which can be shown in layout form as:

```
  9 5 6 7
  1 0 8 5
-------------
1 0 6 5 2
--------------
```

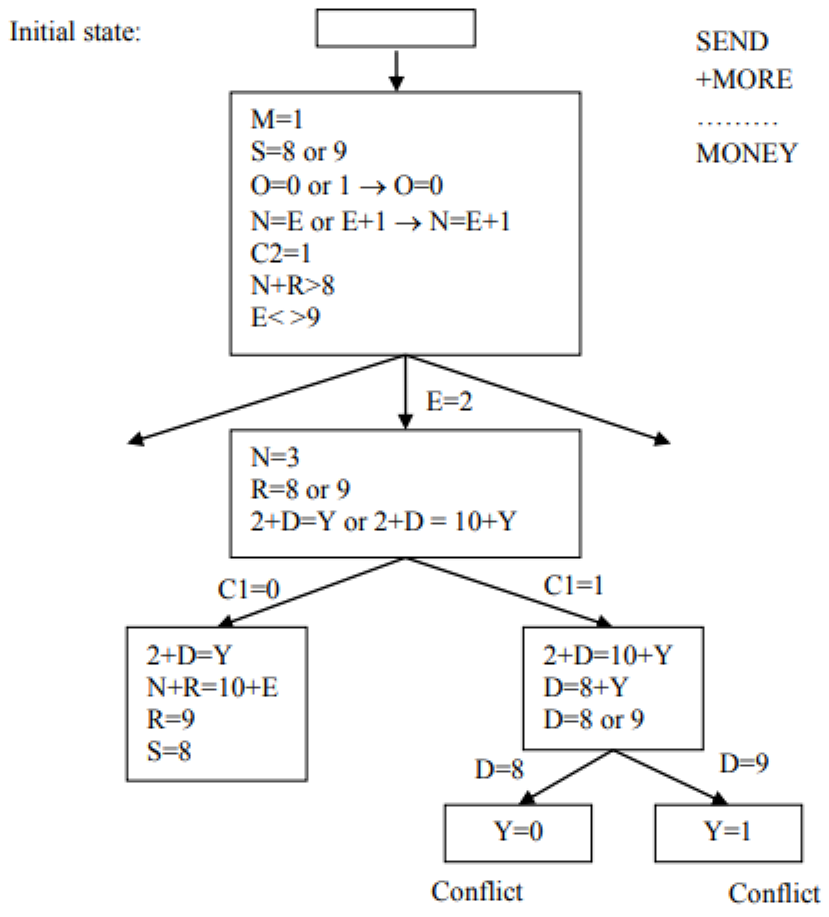


Fig – showsSolving a cryptarithmetic problem

<u>**Other two kinds of constraints:**</u>

(i) Simple, value – listing constraints are always dynamic and must always be represented explicitly in each problem state.

(ii) Complicated, relationship – expressing constraints are dynamic in the cryptarithmatic domain since they are different for each cryptarithmetic problem. But in many other domains they are static.

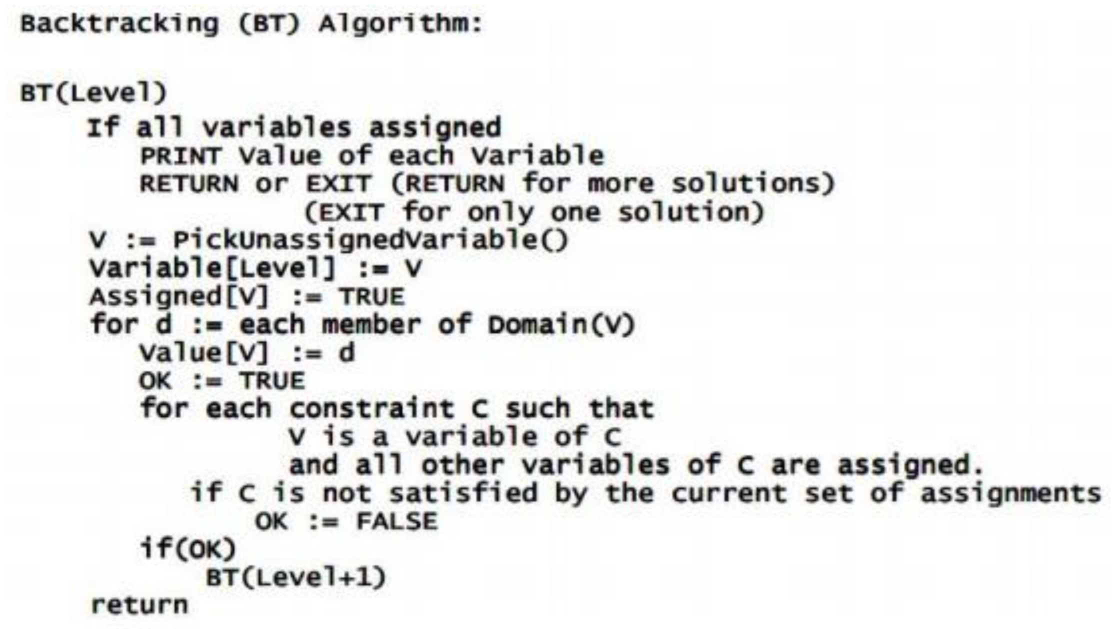
Algorithm for constraint satisfaction in which chronological backtracking is used when guessing leads to an inconsistent set of constraints. Constraints are generated are left alone if they are independent of the problem and its cause. This approach is called dependency directed Backtracking (DDB).

2.6 SOLVING CSPS USING BACKTRACKING SEARCH

- CSPs can be solved by a specialized version of depth first search.
- Key intuitions:

- ✓ a solution is built by searching through the space of partial assignments.
- ✓ Order in which variables are assigned does not matter; eventually they all have to be assigned.
- ✓ If during the process of building up a solution a constraint fails reject all possible ways of extending the current partial assignment.

2.6.1 Backtracking Search Algorithm



```
Backtracking (BT) Algorithm:

BT(Level)
    If all variables assigned
       PRINT Value of each Variable
       RETURN or EXIT (RETURN for more solutions)
                (EXIT for only one solution)
    V := PickUnassignedVariable()
    Variable[Level] := V
    Assigned[V] := TRUE
    for d := each member of Domain(V)
       Value[V] := d
       OK := TRUE
       for each constraint C such that
                V is a variable of C
                and all other variables of C are assigned.
          if C is not satisfied by the current set of assignments
              OK := FALSE
       if(OK)
           BT(Level+1)
    return
```

<u>Measure of Performance and Analysis of Search Algorithm:</u>

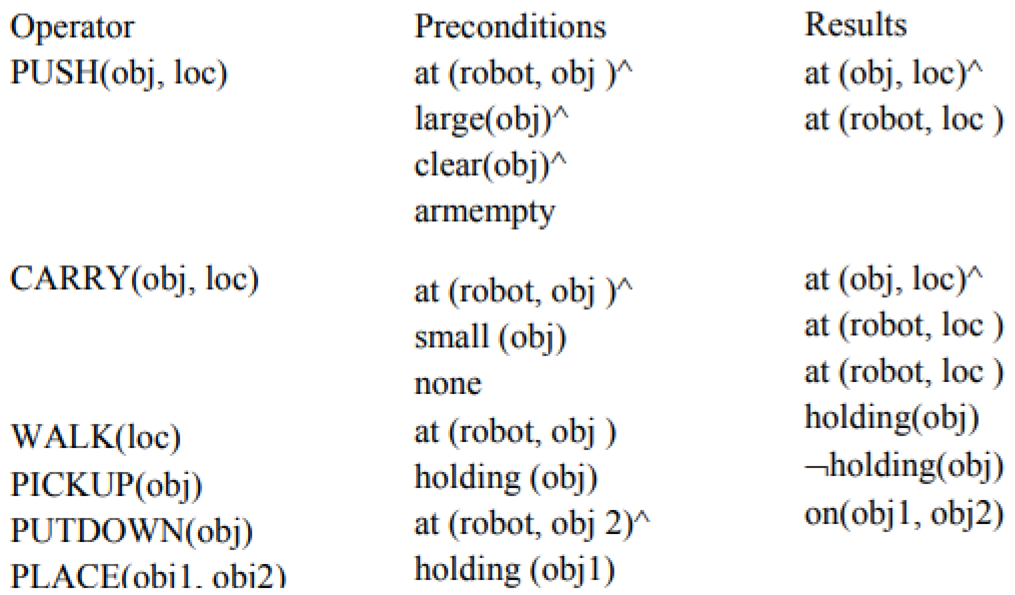
Search strategies that can reason either forward or backward but for a given problem, one direction or the other must be chosen. A mixture of the two directions is appropriate mixed strategy solve the major parts of the problem first and then go back and solve the small problems. This technique is known as Means – ends Analysis.

- Difference between the current state and the goal state.
- Operator reduce the difference must be found.
- Operator can‟t be applied to the current state.
- Setting up a sub problem of getting to a state in which it can be applied.
- Backward chaining in which operators are selected and then sub goals are setup to establish the pre-conditions of the operators is called operator sub goaling.
- If the operator doesn‟t produce the exact goal state then the second sub problem of getting from the state it does produce a goal.

- If the difference chosen correctly and if operator is effective at reducing the difference, then the two sub problems should be easier to solve than the original problem.
- To solve big problems first, the differences assigned priority levels.
- Higher priority is considered before lower priority.
- General problem solver (GPS), people use this GPS to solve problem.
- GPS provide fuzziness of the boundary between building programs that simulate what people do and building programs that solve a problem.
- Mean – ends Analysis relies on a set of rules that transform one problem state into another.
- Rules are represented as a left side that describes the conditions that must be met for the rule to be applicable and a right side describes aspects of the problem state that will be changed by the application of the rule.
- A separate data structure called a difference table indexes the rules by the differences that they can be used to reduce.

Example: House hold robot domain

The available operators shown in below fig. along with preconditions and results.



Operator	Preconditions	Results
PUSH(obj, loc)	at (robot, obj)^	at (obj, loc)^
	large(obj)^	at (robot, loc)
	clear(obj)^	
	armempty	
CARRY(obj, loc)	at (robot, obj)^	at (obj, loc)^
	small (obj)	at (robot, loc)
	none	at (robot, loc)
WALK(loc)	at (robot, obj)	holding(obj)
PICKUP(obj)	holding (obj)	¬holding(obj)
PUTDOWN(obj)	at (robot, obj 2)^	on(obj1, obj2)
PLACE(obj1, obj2)	holding (obj1)	

Fig Shows The Robot's operation

The below fig shows the different table that describes when each of the operator is appropriate. More operator reduce a different and a given operator able to reduce more than one difference.

- **Working of Robot:**
  Robot job is to moving a desk with two things on it from one room to another.
- The objects on top must be moved.
- Difference between the start state and the goal state is location of the desk.
- To reduce the difference, either PUSH or CARRY could be chosen.
- If CARRY chosen, its preconditions met.
- It Result in two more differences that must be reduced, the location of robot and size of desk.
- Location of robot by applying WALK. But no operator change the size of on object.
- This path leads to dead – end.
- We attempt to apply PUSH.
  Below fig shows the problem solver's progress.
- The final difference between C and E reduced by using WALK to get the robot back to the objects, followed by PICKUP and CARRY.

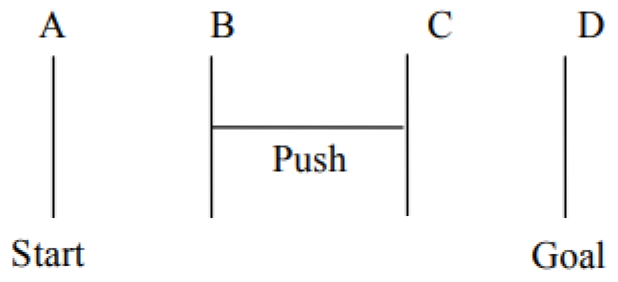


Fig –ShowsThe progress of the Means – ends Analysis method

Algorithm: Means – Ends Analysis (CURRENT, GOAL)

1) Compare CURRENT to GOAL. If there are no difference between them then return.

2) Otherwise, select the most important difference and reduce it by doing the following until success or failure is signaled:

a) Select an as yet untried operator O that is applicable to the current difference. If there are no such operator, then signal failure.

b) Attempt to apply O to CURRENT. Generate description of two states: O-START, a state in which O‟s preconditions are satisfied and O-RESULT, the state that would result if O were applied in O-START.

c) If

(FIRST-PART MEA(CURRENT, O-START)

And

(LAST-PART MEMO-RESULT, GOAL)

Are successful, then signal success and return the result of concatenating

FIRST-PART, O, and LAST-PAR

2.7 ADVERSARIAL SEARCH (GAME PLAYING)

Adversarial search is a search, where we examine the problem which arises when we try to plan ahead of the world and other agents are planning against us. The environment with more than one agent is termed as **multi-agent environment**, in which each agent is an opponent of other agent and playing against each other. Each agent needs to consider the action of other agent and effect of that action on their performance. So, **Searches in which two or more players with conflicting goals are trying to explore the same search space for the solution, are called adversarial searches, often known as Games**. Games are modeled as a Search problem and heuristic evaluation function, and these are the two main factors which help to model and solve games in AI.

Types of Games in AI:

	Deterministic	**Chance Moves**
Perfect information	Chess, Checkers, go, Othello	Backgammon, monopoly
Imperfect information	Battleships, blind, tic-tac-toe	Bridge, poker, scrabble, nuclear war

Perfect information: A game with the perfect information is that in which agents can look into the complete board. Agents have all the information about the game, and they can see each other moves also. Examples are Chess, Checkers, Go, etc.

Imperfect information: If in a game agents do not have all information about the game and not aware with what's going on, such type of games are called the game with imperfect information, such as tic-tac-toe, Battleship, blind, Bridge, etc.

Deterministic games: Deterministic games are those games which follow a strict pattern and set of rules for the games, and there is no randomness associated with them. Examples are chess, Checkers, Go, tic-tac-toe, etc.

Non-deterministic games: Non-deterministic are those games which have various unpredictable events and have a factor of chance or luck. This factor of chance or luck is introduced by either dice or cards. These are random, and each action response is not fixed. Such games are also called as stochastic games.
Example: Backgammon, Monopoly, Poker, etc.

Note: In this topic, we will discuss deterministic games, fully observable environment, zero-sum, and where each agent acts alternatively.

2.7.1 Zero-Sum Game

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

Zero-sum game: Embedded thinking

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

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

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

Formalization of the Problem:

Competitive environments: in which the agents' goal are in conflict, giving rise to adversarial search problems.

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

A **zero-sum game** is one where the total payoff to all players is the same for every instance of the game. e.g. Chess, 0+1=1+0=1/2+1/2.

Game tree: Defined by the initial state, ACTIONS function and RESULT function, a tree where the nodes are game states and the edges are move

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

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

2.7.2 Game tree

A game tree is a tree where nodes of the tree are the game states and Edges of the tree are the moves by players. Game tree involves initial state, actions function, and result Function. A Game Tree is a structure for organizing all possible (legal) game states by the moves which allow transition from one game state to the next.

This structure helps the computer to evaluate which moves to make because, by traversing the game tree, a computer (program) can easily see the outcome of a move and can decide whether to take it or not.

The following states are used to represent a game tree.

- The board state: This is an initial stage.
- The current player: It refers to the player who will be making the next move.
- The next available moves: For humans, a move involves placing a game token while the computer selects the next game state.
- The game state: It includes the grouping of the three previous concepts.
- Final Game States

In final game states, AI should select the winning move in such a way that each move assigns a numerical value based on its board state. The ranking should be given as:

a) Win: 1

b) Draw: O

c) Lose: -1

It is important to consider the aspects related to winning with the highest ranking, losing to the lowest, and a draw between the two players. The Max part of Minimax algorithm states that the user has to select the move with the highest value. Final Game States are ranked on the basis of their status of winning, losing or a draw. Ranking of Intermediate Game States is based on the turn of player to make available moves. If it's X's turn, set the rank to that of the maximum available move. If a move results into a win, X can take it. If it's O's turn, set the rank to that of the minimum available move. If a move results into a loss, X can avoid it.

Search tree: A tree that is superimposed on the full game tree, and examines enough nodes to allow a player to determine what move to make.

2.7.3 Optimal decisions in games

Optimal solution: In adversarial search, the optimal solution is a contingent 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 opponent), then MAX's moves in the states resulting from every possible response by MIN to those moves, and so on.

Minimax Algorithm: The Min-Max algorithm is generally used for a game consisting of two players such as tic-tac-toe, checkers, chess etc. All these games are logical games, so they can be described by set of rules. It is possible to determine the next available moves from a given point in the game. Consider that the game has two players MAX and MIN. MAX will move first and then MIN. A game can be defined as a kind of search problem.

***One move deep*:** If a particular game ends after one move each by MAX and MIN, we say that this tree is one move deep, consisting of two half-moves, each of which is called a **ply**.

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 of a terminal state is just its utility.

Given a game tree, the optimal strategy can be determined from the minimax value of each node, i.e. MINIMAX(n). MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value.

Example: Tic-Tac-Toe game tree:

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

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

<u>Example Explanation</u>

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

- In this either MIN wins, MAX wins, or it's a draw. This game-tree is the whole search space of possibilities that MIN and MAX are playing tic-tac-toe and taking turns alternately.

Hence adversarial Search for the minimax procedure works as follows:

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

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

For a state S MINIMAX(s) =

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

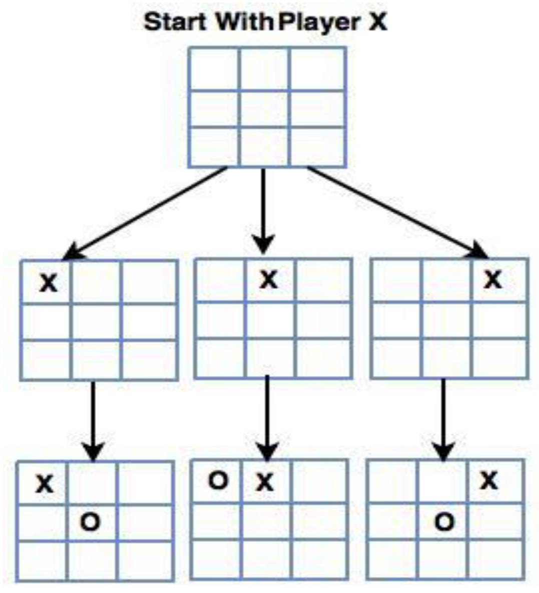


Fig (a): Initial Moves

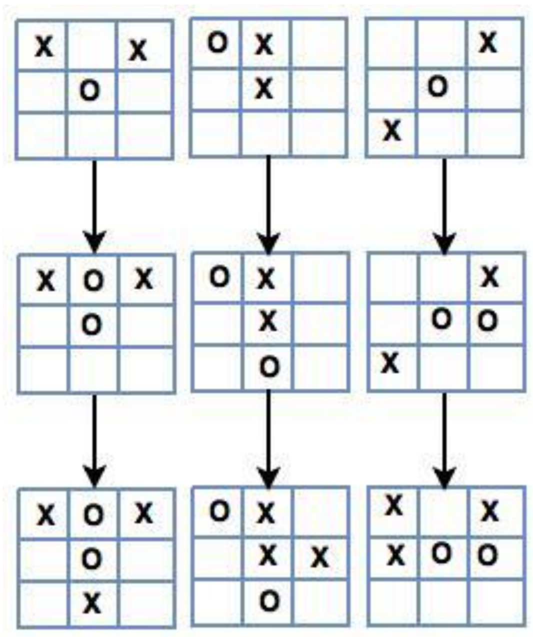


Fig (b): Intermediate Moves (Continue from Fig(a))

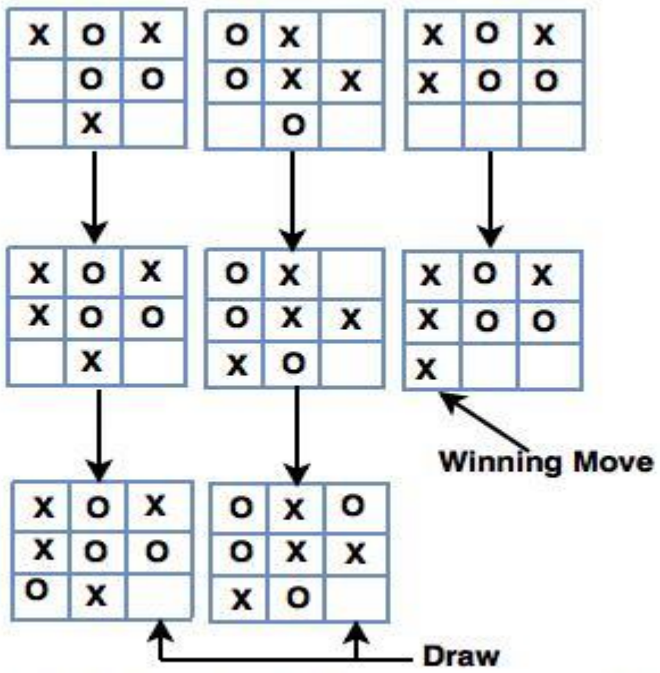


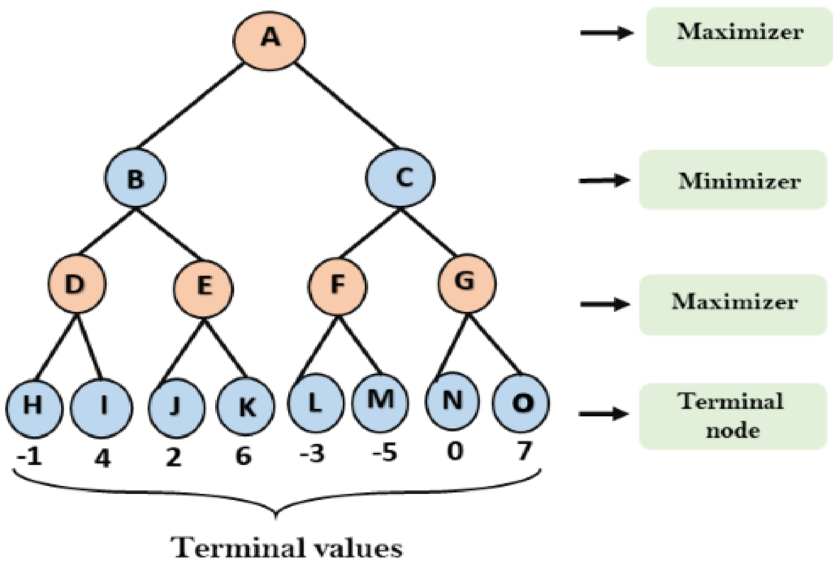
Fig (c): Final Stage (continue from Fig(b))

Working of Min-Max Algorithm:

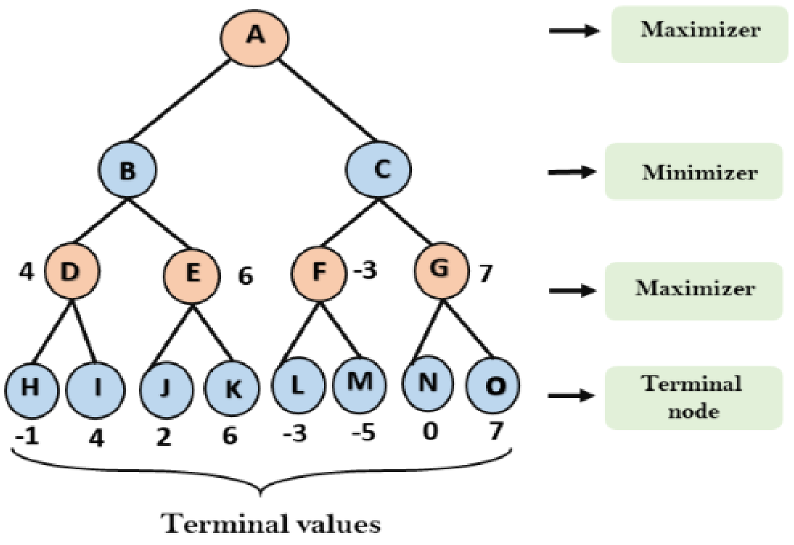
The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.

In this example, there are two players one is called Maximizer and other is called Minimizer. Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score. This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes. At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

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



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



It will find the maximum among the all.

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

For Node E max(2, -∞) => max(2, 6)= 6

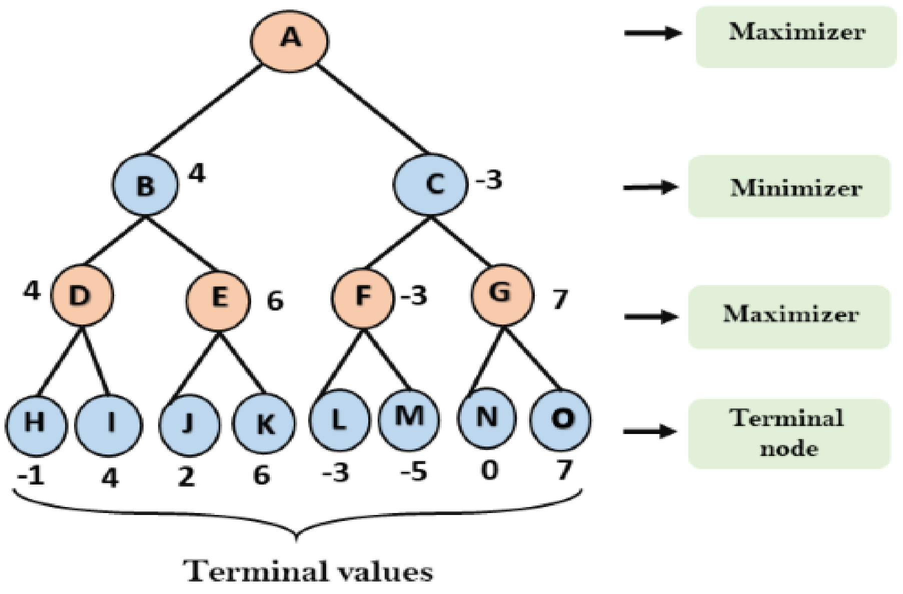
For Node F max(-3, -∞) => max(-3,-5) = -3

For node G max(0, -∞) = max(0, 7) = 7

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

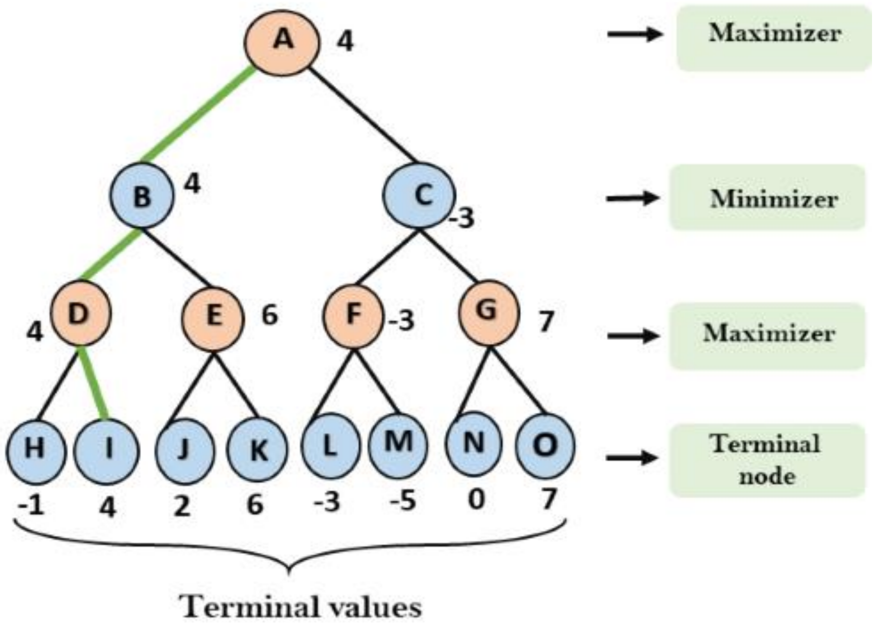
For node B= min(4,6) = 4

For node C= min (-3, 7) = -3



Step 3: Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

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



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

Properties of Mini-Max algorithm:

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

Limitation of the minimax Algorithm:

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

2.8 ALPHA-BETA PRUNING

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

Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.

The two-parameter can be defined as:

- **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is $-\infty$.
- **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is $+\infty$.

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

Note: To better understand this topic, kindly study the minimax algorithm.

<u>Condition for Alpha-beta pruning</u>

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

$$\alpha >= \beta$$

2.8.1 Key points about alpha-beta pruning:

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

2.8.2 Pseudo-code for Alpha-beta Pruning

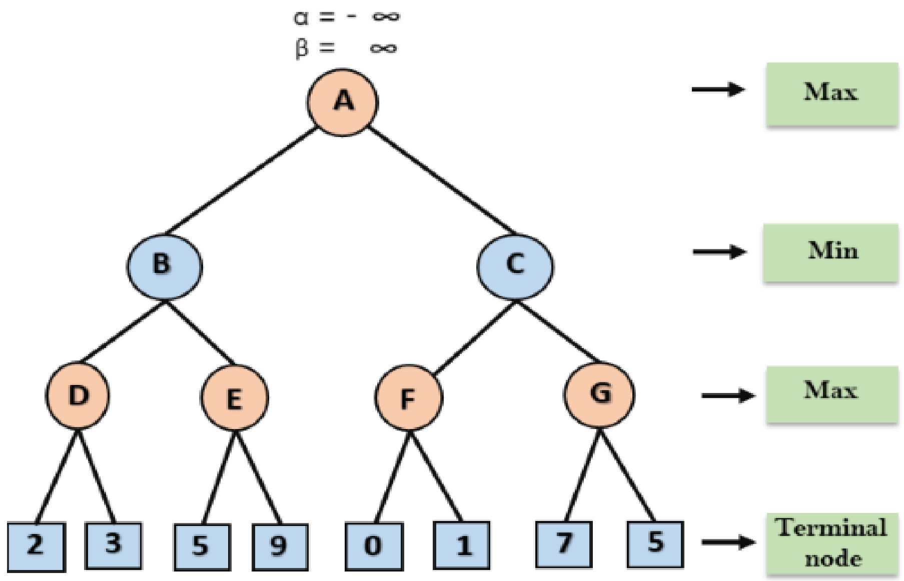
```
function minimax(node, depth, alpha, beta, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node
if MaximizingPlayer then      // for Maximizer Player
   maxEva= -infinity
   for each child of node do
   eva= minimax(child, depth-1, alpha, beta, False)
  maxEva= max(maxEva, eva)
  alpha= max(alpha, maxEva)
   if beta<=alpha
 break
 return maxEva
else                          // for Minimizer player
   minEva= +infinity
   for each child of node do
```

```
        eva= minimax(child, depth-1, alpha, beta, true)
        minEva= min(minEva, eva)
        beta= min(beta, eva)
         if beta<=alpha
       break
      return minEva
```

2.8.3 Working of Alpha-Beta Pruning

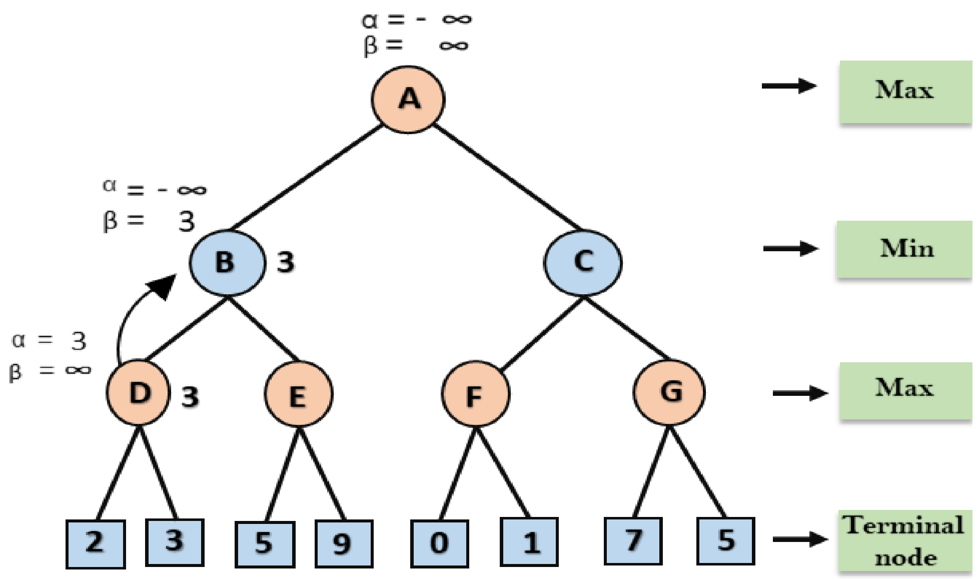
Let's take an example of two-player search tree to understand the working of Alpha-beta pruning.

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



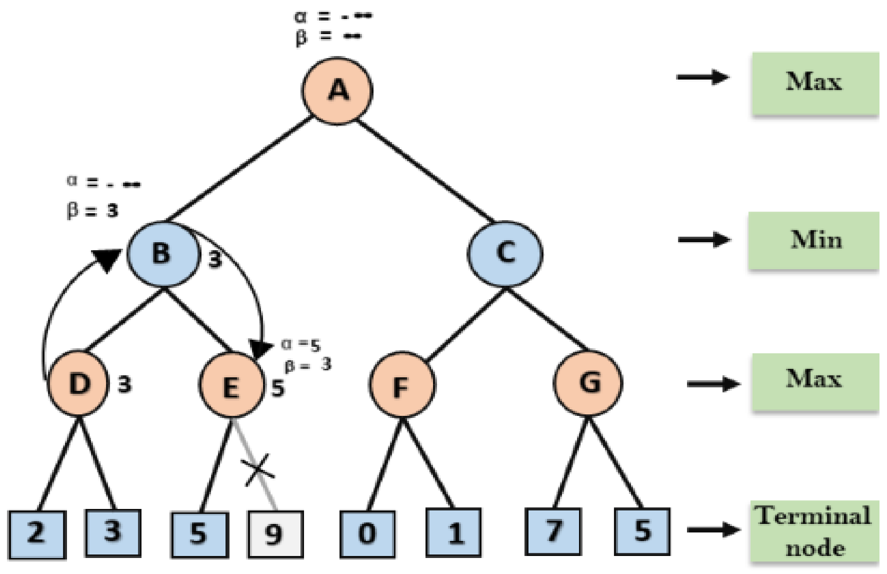
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

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



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

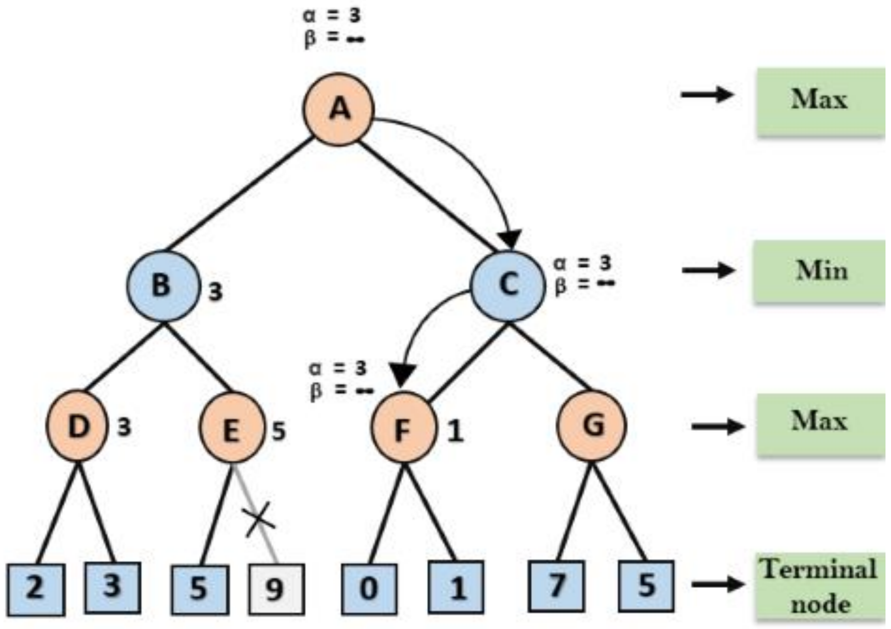
Step 4:



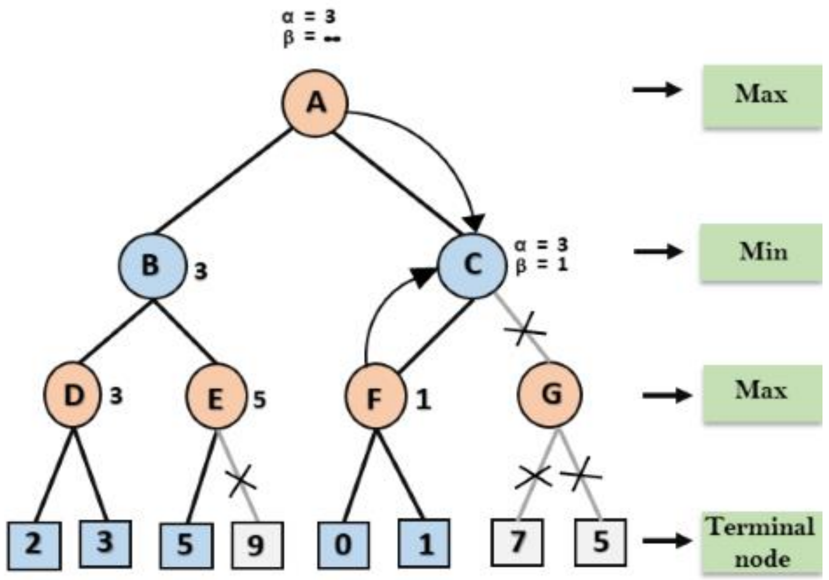
At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.

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

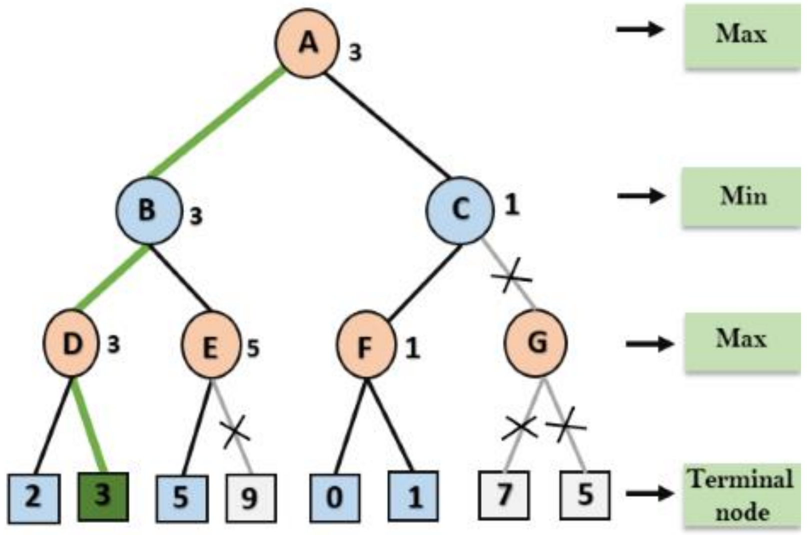
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



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



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



2.8.4 Move Ordering in Alpha-Beta pruning

The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning. It can be of two types:

- **Worst ordering:** In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is $O(b^m)$.
- **Ideal ordering:** The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is $O(b^{m/2})$.

2.8.5 Game Strategies

Strategy is a complete approach for playing a game, while move is an action taken by a player at some point during the course of the game.

- ***Dominant Strategies:*** One strategy is better than another for the same player, irrespective of other player‟s game.
- ***Pure Strategies:*** It‟s the complete approach of a player‟s game plan.
- ***Mixed Strategy:*** Randomly the player can choose a strategy based on probability assigned for any strategy.
- ***Backward Induction:*** Optimal actions are executed by backward reasoning.

2.9 GAME EQUILIBRIUM

Equilibrium is the process of selection of a stable state. Game theory tries to find state of equilibrium. It depends on field of application, which might coincide. Some important equilibrium are Nash Equilibrium, Sub-Game Perfection, Bayesian Nash, Perfect Bayesian, Proper Equilibrium, Correlated Equilibrium, Sequential Equilibrium, Parto Efficiency, Self-Confirming Equilibrium, Trembling hand.

2.10 STOCHASTIC GAMES

A stochastic game is a collection of normal-form games that the agents play repeatedly . The particular game played at any time depends probabilistically on,

- the previous game played
- the actions of the agents in that game

A stochastic (or Markov) game includes the following:

- a finite set Q of states (games),
- a set N = {1, …, n} of agents,
- For each agent i, a finite set Ai of possible actions
- A transition probability function $P : Q \times A1 \times \cdot\cdot\cdot \times An \times Q \rightarrow [0, 1]$ P(q, a1, …, an , q'') = probability of transitioning to state q'' if the action profile (a1, …, an) is used in state q
- For each agent i, a real-valued payoff function $ri : Q \times A1 \times \cdot\cdot\cdot \times An \rightarrow \Re$

This definition makes the inessential but simplifying assumption that each agent's strategy space is the same in all games. So the games differ only in their payoff functions. We will only consider strategies called **Markov strategies**. A strategy is call a **Markov strategy** if the behaviour dictated is not time dependent.

2.11 SEARCHING WITH PARTIAL OBSERVATIONS

Above we (unrealistically) assumed that the environment is fully observable and deterministic. Moreover, we assumed that the agent knows what the effects of each action are. Therefore, the agent can calculate exactly which state results from any sequence of actions and always knows which state it is in. Its percepts provide no new information after each action. In a more realistic situation the agent's knowledge of states and actions is incomplete.

If the agent has no sensors at all, then as far as it knows it could be in one of several possible initial states, and each action might therefore lead to one of several possible successor states. An agent without sensors, thus, must reason about sets of states that it might get to, rather than single states. At each instant the agent has a belief of in which state it might be

If the environment is partially observable or if the effects of actions are uncertain, then each new action yields new information. Every possible contingency that might arise during execution need considering. The cause of uncertainty may be another agent, an adversary. When the states of the environment and actions are uncertain, the agent has to explore its environment to gather information. In a partially observable world one cannot determine a fixed action sequence in advance, but needs to condition actions on future percepts. As the agent can gather new knowledge through its actions, it is often not useful to plan for each possible situation. Rather, it is better to interleave search and execution.

Now we come back to a world where the actions of the robot are deterministic again (no erratic behavior like before) but, the robot no longer has complete sense of its current state or its environment.

Vacuum World with no observation

In this world, the vacuum cleaner has no idea initially about its own location and the location of dirt in the world. Since the robot has no percept, it should be able to figure out a sequence of actions that will work despite its current state.

Given below are 8 random initial states. You can record a sequence of actions and see it in action just like before. Assume that illegal moves (like moving right in the right-most tile) have no effect on the world. Try to find a sequence of actions that will lead to a final state (Clean all the dirt), no matter what the initial state of the world. Different types of incompleteness lead to three distinct problem types:

Sensorless problems (conformant): If the agent has no sensors at all

Contingency problem: if the environment if partially observable or if action are uncertain (adversarial)

Exploration problems: When the states and actions of the environment are unknown.

- ✓ No sensor
- ✓ Initial State(1,2,3,4,5,6,7,8)
- ✓ After action [Right] the state (2,4,6,8)
- ✓ After action [Suck] the state (4, 8)
- ✓ After action [Left] the state (3,7)
- ✓ After action [Suck] the state (8)
- ✓ Answer : [Right, Suck, Left, Suck] coerce the world into state 7 without any sensor
- ✓ Belief State: Such state that agent belief to be there

Partial knowledge of states and actions:

Sensorless or conformant problem

Agent may have no idea where it is; solution (if any) is a sequence.

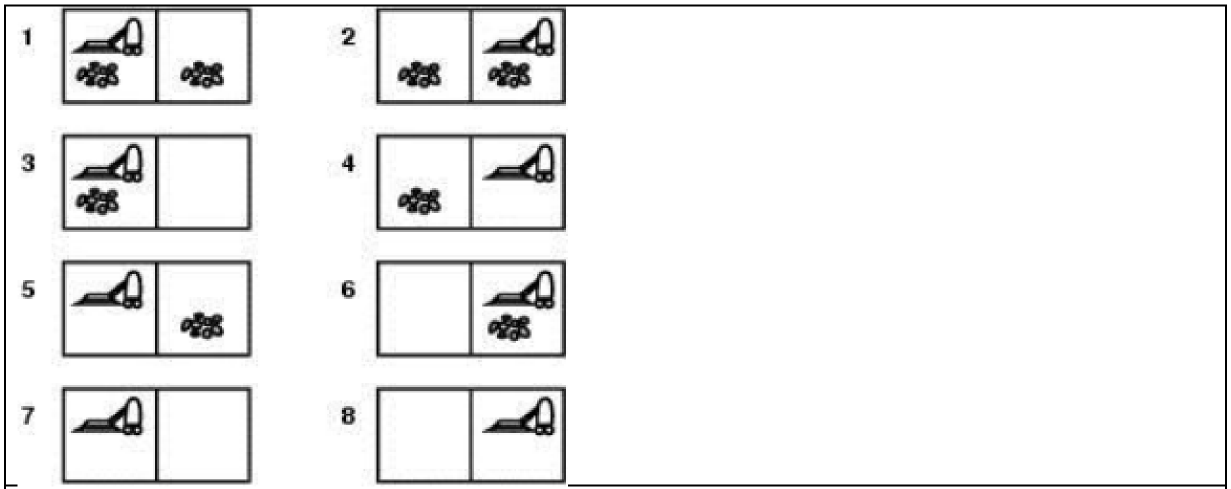
Contingency problem

Percepts provide new information about current state; solution is a tree or policy; often interleave search and execution.

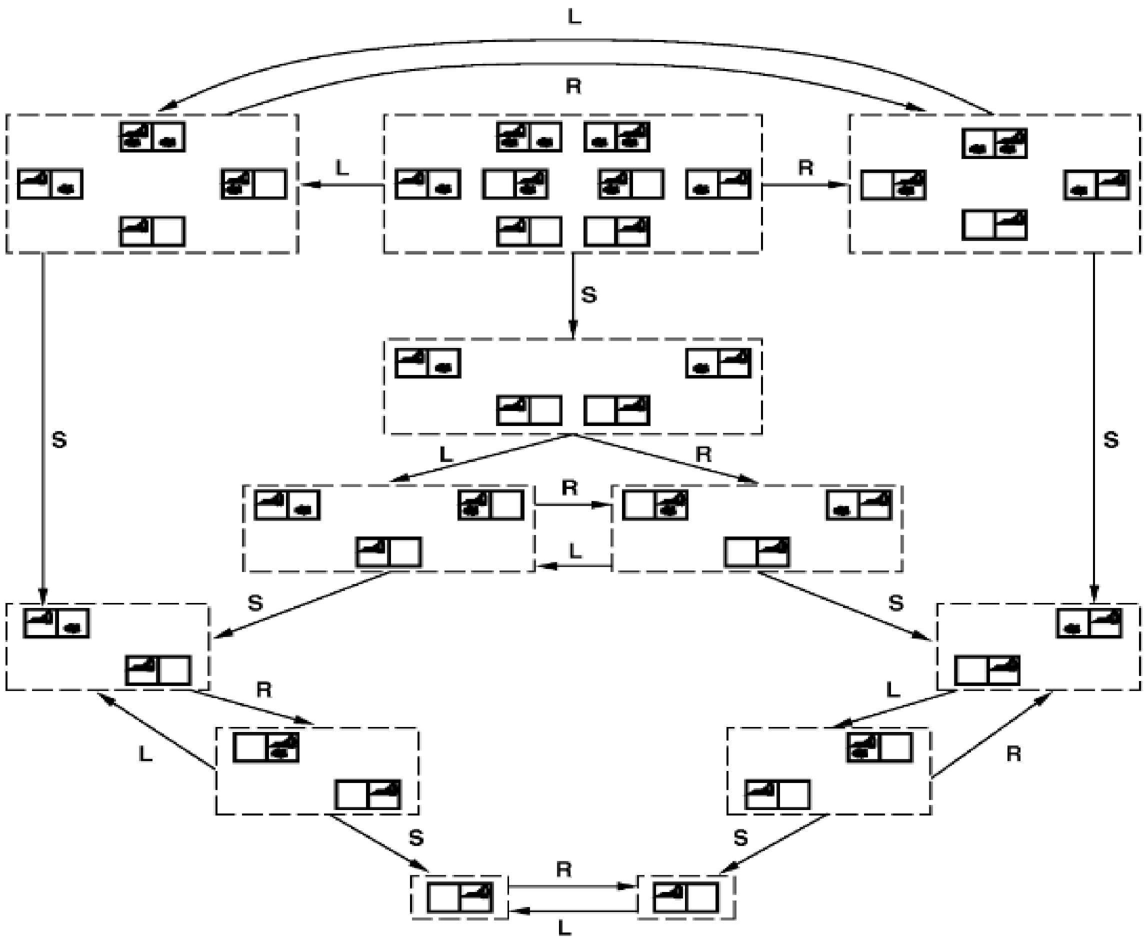
If uncertainty is caused by actions of another agent: adversarial problem

Exploration problem

When states and actions of the environment are unknown.



Figure



Contingency, start in {1,3}.

By Murphy's law, Suck can dirty a clean carpet.

Local sensing: dirt, location only.

Percept = [L,Dirty] ={1,3}

[Suck] = {5,7} – [Right] ={6,8}

[Suck] in {6}={8} (Success) – BUT [Suck] in {8} = failure

Solution:

– Belief-state: no fixed action sequence guarantees solution

Relax requirement:

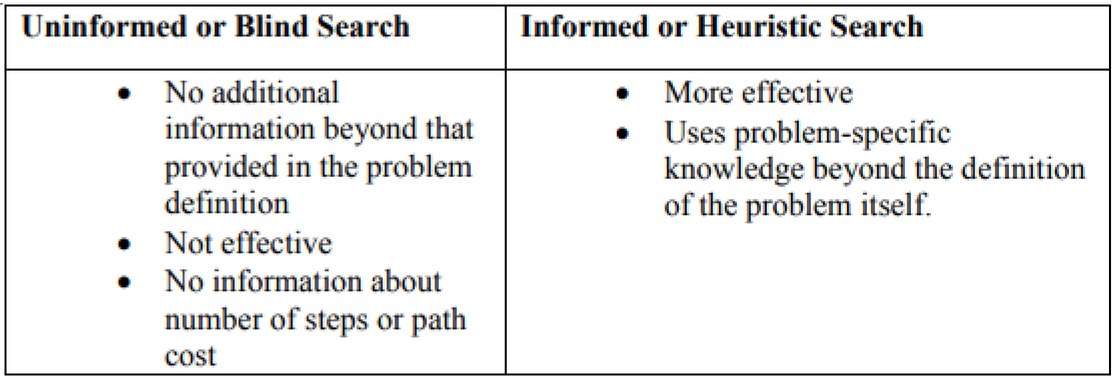
– [Suck, Right, if [R,dirty] then Suck]

– Select actions based on contingencies arising during execution.

Time and space complexity are always considered with respect to some measure of the problem difficulty. In theoretical computer science, the typical measure is the size of the state space.

Two Marks Questions and Answers

1. Compare Uninformed Search (Blind search) and informed Search (Heuristic Search) strategies.



Uninformed or Blind Search	**Informed or Heuristic Search**
• No additional information beyond that provided in the problem definition • Not effective • No information about number of steps or path cost	• More effective • Uses problem-specific knowledge beyond the definition of the problem itself.

2. Define Best-first-search.

Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on the evaluation function f(n). Traditionally, the node with the lowest evaluation function is selected for expansion.

3. What is a heuristic function?

A heuristic function or simply a heuristic is a function that ranks alternatives in various search algorithms at each branching step basing on available information in order to make a decision which branch is to be followed during a search. For example, for shortest path problems, a heuristic is a function, h (n) defined on the nodes of a search tree, which serves as an estimate of the cost of the cheapest path from that node to the goal node. Heuristics are used by informed search algorithms such as Greedy best-first search and A* to choose the best node to explore.

4. What is admissible heuristic?

Admissible Heuristic is a heuristic h(n) that never overestimates the cost from node n the goal node.

5. What are relaxed problems?

- A problem with fewer restrictions on the actions is called a relaxed problem.
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.
- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then hoop(n) gives the shortest solution.
- If the rules are relaxed so that a tile can move to any adjacent square, then hmd(n) gives the shortest solution.

6. What is greedy best-first-search?

Greedy best-first-search tries to expand the node that is closest to the goal, on the grounds that that is likely to lead to a solution quickly. For example, it evaluates nodes by using just the heuristic function: f (n) = h (n).

7. What is A* search?

A* search is the most widely-known form of best-first search. It evaluates the nodes by combining g(n),the cost to reach the node, and h(n),the cost to get from the node to the goal:

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

Where f (n) = estimated cost of the cheapest solution through n.

g (n) is the path cost from the start node to node n.

h (n) = heuristic function

A* search is both complete and optimal.

8. What is Recursive best-first search?

Recursive best-first search is a simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space.

9. What are local search algorithms?

Local search algorithms operate using a single current state (rather than multiple paths) and generally move only to neighbors of that state. The local search algorithms are not systematic. The key two advantages are (i) they use very little memory – usually a constant amount, and (ii) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

10. What are the advantages of local search?

*Use very little memory – usually a constant amount

*Can often find reasonable solutions in large or infinite state spaces (e.g., continuous)

- ✓ Unsuitable for systematic search

*Useful for pure optimization problems

- ✓ Find the best state according to an objective function
- ✓ Traveling salesman

11. What are optimization problems?

In optimization problems, the aim is to find the best state according to an objective function the optimization problem is then: Find values of the variables that minimize or maximize the objective function while satisfying the constraints.

12. What is Hill-climbing search?

The Hill-climbing algorithm is simply a loop that continually moves in the direction of increasing value –that is uphill. It terminates when it reaches a "peak" where no neighbor has a higher value. The algorithm does not maintain a search tree so the current node data structure need only record the state and its objective function value. Hill-climbing does not look ahead beyond the immediate neighbors of the current state.

13. What is the problem faced by hill-climbing search?

Hill-climbing often get stuck for the following reasons:

i. Local maxima – A local maxima is a peak that is higher than each of its neighboring states, but lower than the local maximum. Hill climbing algorithm that reach the vicinity of a local maximum will be drawn upwards towards the peak, but will then be stuck with nowhere else to go.

ii. Ridges – Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

iii. Plateaux- a plateau is an area of state space landscape where the evaluation function is flat. A hill-climbing search might be unable to find its way off the plateau.

14. What is local beam search?

The local beam search algorithm keeps track of k states rather than just one. It begins with k randomly generated states. At each step, all the successors of all k states are generated. If anyone is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats.

15. Explain briefly simulated annealing search.

Simulated annealing is an algorithm that combines hill climbing with random walk in some way that yields both efficiency and completeness.

16. What is genetic algorithm?

A genetic algorithm (or GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states, rather than by specifying a single state.

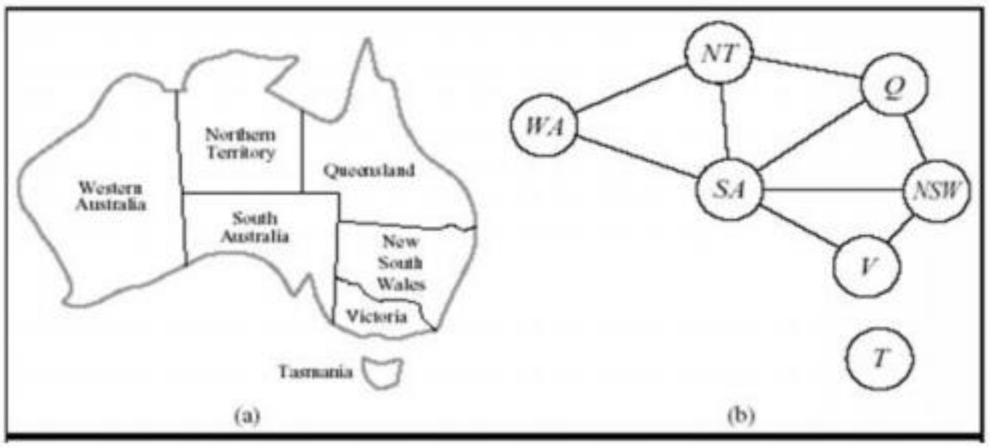
17. Define constraint satisfaction problem.

A Constraint Satisfaction problem (or CSP) is defined by a set of variables X1, X2,…..,Xn, and a set of constraints, C1,C2,…..,Cm. Each variable Xi has a nonempty domain Di of possible values. Each constraint Ci involves some subset of the variables and specifies the allowable combinations of values for that subset. A state of the problem is defined by an assignment of values to some or all of the variables,{Xi = vi,Xj=vj,…} A solution to a CSP is a complete assignment that satisfies all the constraints.

18. What is a constraint graph?

It is helpful to visualize the Constraint Satisfaction Problem as a Constraint Graph. A Constraint Graph is a graph where the nodes of the graph correspond to variables of the problem and the arcs corresponds to constraints.

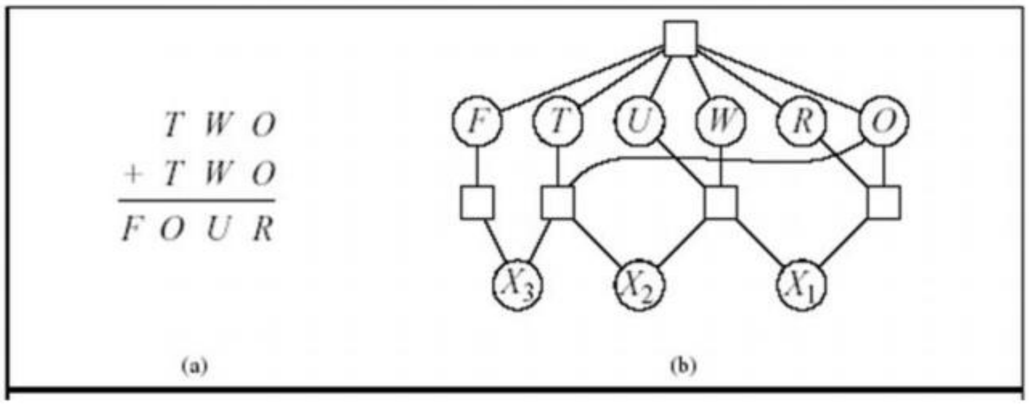
Constraint graph – Example:



The map coloring problem represented as a constraint graph

19. What are crypt arithmetic problem? Give an example.

Verbal arithmetic, also known as alphabetic, crypt arithmetic, cryptarithm or word addition, is a type of mathematical game consisting of a mathematical equation among unknown numbers, whose digits are represented by letters. The goal is to identify the value of each letter.



(a) **A crypt arithmetic problem.** Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed.

(b) The constraint hyper graph for the crypt arithmetic problem, showing the All diff constraint as well as the column addition constraints. Each constraint is a square box connected to the variables it constrains.

20. What is backtracking search?

Backtracking search is 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.

21. What is adversarial search?

Competitive environments, in which the agents' goals are in conflict, give rise to adversarial search problems – often known as games.

22. How can we avoid ridge and plateau in hill climbing?

A ridge is a special kind of local maximum. It is an area of the search space that is higher that the surrounding areas and that it have a slope. But the orientation of the high region, compared to the set of available moves and the directions in which they move, makes it impossible to traverse a ridge by single moves. Any point on a ridge can look like peak because movement in all probe directions is downward. A plateau is a flat area of the search space in which a whole set of neighboring states have the same value. On a plateau, it i not possible to determine the best direction in which to move by making local comparisons.

PART B (13 MARK QUESTIONS)

1. Explain in detail about the Uninformed Search Strategies.
2. Explain in detail about the Informed Search Strategies.
3. What is Constraint Satisfaction Problem? Explain it in detail.
4. Describe the Concept of Game playing.
5. Explain Min-Max Algorithm with a suitable Example
6. What is Hill Climbing? Write short notes on it.
7. Discuss in detail about Alpha-Beta Pruning.
8. What is Heuristic Function? Explain it in detail.