Module: 3.3
Problem Solving and Search

Motivation:
This module 3.3 will brief about search optimization methods and advance search techniques.

Syllabus:

Lecture no	Content	Duration (Hr)	Self-Study (Hrs)
1	Local Search Methods(Optimization Methods)	1	1
2	Adversarial Search	1	
3	Constraint Satisfaction Problems	1	

Learning Objective:

- Learner should know about various search optimization methods and advance search techniques and apply same on given problem.
- **Theoretical Background:**
  The optimal solution is a set of decision variables that maximizes or minimizes the objective function while satisfying the constraints. In general, optimal solution is obtained when the corresponding values of the decision variables yield the best value of the objective function, while satisfying all the model constraints.

  Apart from the gradient-based optimization methods, some new optimization methods have also been proposed that help solve complex problems. In the available classifications, these methods are recognized as “intelligent optimization,” “optimization and evolutionary computing,” or “intelligent search.” One of the advantages of these algorithms is that they can find the optimal point without any need to use objective function derivatives. Moreover, compared to the gradient-based methods, they are less likely to be trapped in local optima.

  Optimization algorithms are classified into two types: exact algorithms and approximate algorithms. Exact algorithms are capable of precisely finding optimal solutions, but they are not applicable for complicated optimization problems, and their solution time increases exponentially in such problems. Approximate algorithms can find close-to-optimal solutions for difficult optimization problems within a short period of time.

  Constraint satisfaction problem represent the entities in a problem as a homogeneous collection of finite constraints over variables, which is solved by constraint satisfaction methods. CSPs are the subject of intense research in both artificial intelligence and operations research, since the regularity in their formulation provides a common basis to analyze and solve problems of many seemingly unrelated families. CSPs often exhibit high complexity, requiring a combination of heuristics and combinatorial search methods to be solved in a reasonable time. Constraint Programming (CP) is the field of research that specifically focuses on tackling with this kind of problems.

Key Definitions:

- **Optimization Methods:** Optimization is the process of setting decision variable values in such a way that the objective in question is optimized.
- **Adversarial Search** :It is search when there is an "enemy" or "opponent" changing the state of the problem every step in a direction you do not want. E.g Chess, business, trading, war.
- **Constraint satisfaction problems** : They are mathematical questions defined as a set of objects whose state must satisfy a number of constraints or limitations.

Course Content:

Lecture : 1

LOCAL SEARCH METHODS

- 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.

Hill Climbing Algorithm in Artificial Intelligence

1. Pick a random point in the search space
2. Consider all the neighbours of the current state
3. Choose the neighbour with the best quality and move to that state
4. Repeat 2 thru 4 until all the neighbouring states are of lower quality
5. Return the current state as the solution state

In computer science, **hill climbing** is a mathematical optimization technique which belongs to the family of local search. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution. If

the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found.

For example, hill climbing can be applied to the travelling salesman problem. It is easy to find an initial solution that visits all the cities but will be very poor compared to the optimal solution. The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited. Eventually, a much shorter route is likely to be obtained.

Hill climbing is good for finding a local optimum (a solution that cannot be improved by considering a neighbouring configuration) but it is not necessarily guaranteed to find the best possible solution (the global optimum) out of all possible solutions (the search space). In convex problems, hill-climbing *is* optimal. Examples of algorithms that solve convex problems by hill-climbing include the simplex algorithm for linear programming and binary search.

The characteristic that only local optima are guaranteed can be cured by using restarts (repeated local search), or more complex schemes based on iterations, like iterated local search, on memory, like reactive search optimization and tabu search, or memory-less stochastic modifications, like simulated annealing.

The relative simplicity of the algorithm makes it a popular first choice amongst optimizing algorithms. It is used widely in artificial intelligence, for reaching a goal state from a starting node. Choice of next node and starting node can be varied to give a list of related algorithms. Although more advanced algorithms such as simulated annealing or tabu search may give better results, in some situations hill climbing works just as well. Hill climbing can often produce a better result than other algorithms when the amount of time available to perform a search is limited, such as with real-time systems. It is an anytime algorithm.

Features of Hill Climbing:

Following are some main features of Hill Climbing Algorithm:

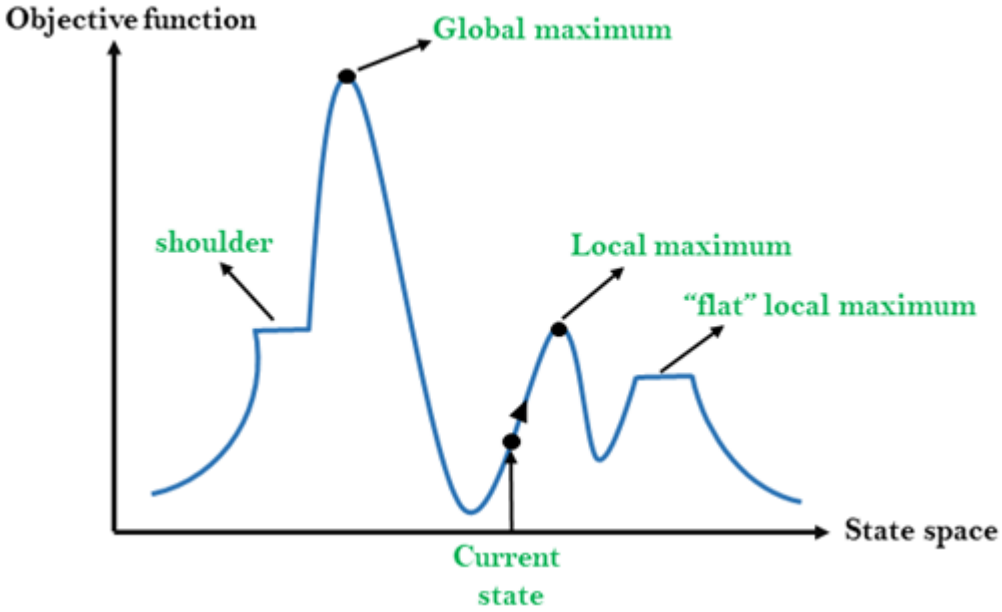
- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.

- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

State-space Diagram for Hill Climbing:

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

On Y-axis we have taken the function which can be an objective function or cost function, and state-space on the x-axis. If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum. If the function of Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.



Different regions in the state space landscape:

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

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

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

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

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

Types of Hill Climbing Algorithm:

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

1. Simple Hill Climbing:

Simple hill climbing is the simplest way to implement a hill climbing algorithm. **It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as a current state**. It only checks 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:

    1. If it is goal state, then return success and quit.
    2. Else if it is better than the current state then assign new state as a current state.
    3. Else if not better than the current state, then return to step2.

- **Step 5:** Exit.

2. Steepest-Ascent hill climbing:

The steepest-Ascent algorithm is a variation of simple hill climbing algorithm. This algorithm examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state. This algorithm consumes more time as it searches for multiple neighbors

Algorithm for Steepest-Ascent hill climbing:

- **Step 1:** Evaluate the initial state, if it is goal state then return success and stop, else make current state as initial state.
- **Step 2:** Loop until a solution is found or the current state does not change.

    1. Let SUCC be a state such that any successor of the current state will be better than it.
    2. For each operator that applies to the current state:

        I. Apply the new operator and generate a new state.
        II. Evaluate the new state.
        III. If it is goal state, then return it and quit, else compare it to the SUCC.
        IV. If it is better than SUCC, then set new state as SUCC.
        V. 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.

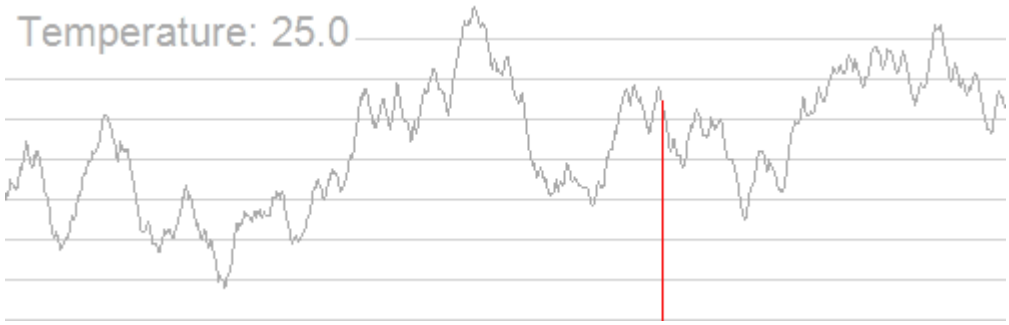
Problems in Hill Climbing Algorithm:

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

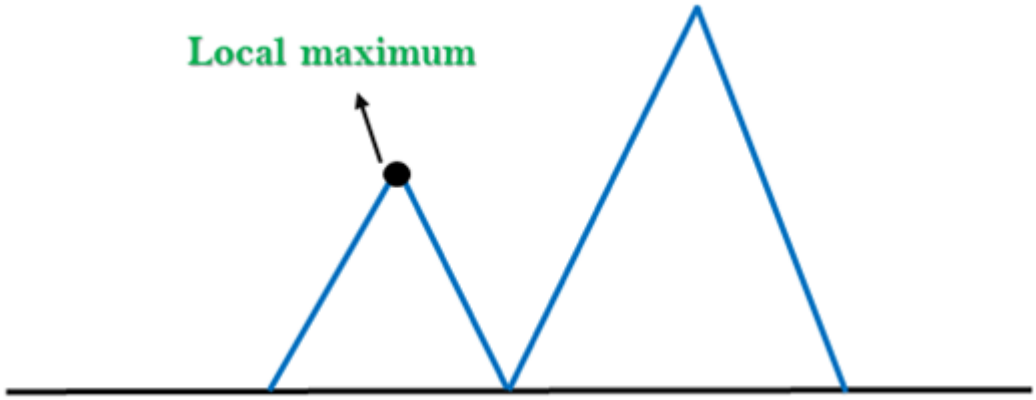
Solution: Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.

A surface with two local maxima. (Only one of them is the global maximum.) If a hill-climber begins in a poor location, it may converge to the lower maximum.

Hill climbing will not necessarily find the global maximum, but may instead converge on a local maximum. This problem does not occur if the heuristic is convex. However, as many functions are not convex hill climbing may often fail to reach a global maximum. Other local search algorithms try to overcome this problem such as stochastic hill climbing, random walks and simulated annealing.



Despite the many local maxima in this graph, the global maximum can still be found using simulated annealing. Unfortunately, the applicability of simulated annealing is problem-specific because it relies on finding *lucky jumps* that improve the position. In problems that involve more dimensions, the cost of finding such a jump may increase exponentially with dimensionality. Consequently, there remain many problems for which hill climbers will efficiently find good results while simulated annealing will seemingly run forever without making progress. This depiction shows an extreme case involving only one dimension.



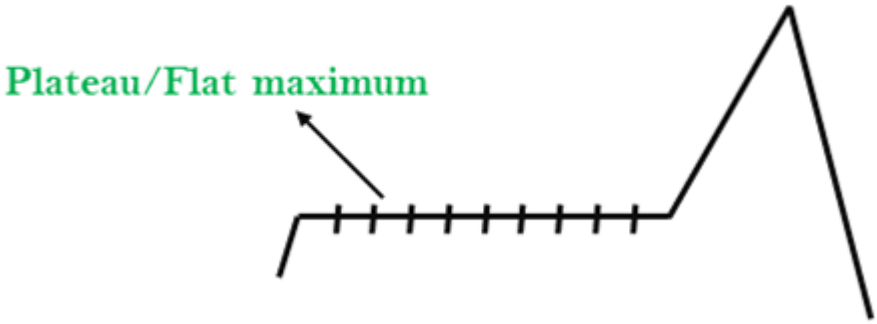
2. Plateau: A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

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

Another problem that sometimes occurs with hill climbing is that of a plateau. A plateau is encountered when the search space is flat, or sufficiently flat that the value returned by the target function is indistinguishable from the value returned for nearby regions due to the precision used by the machine to represent its value. In such cases, the hill climber may not be able to determine in which direction it should step, and may wander in a direction that never leads to improvement.

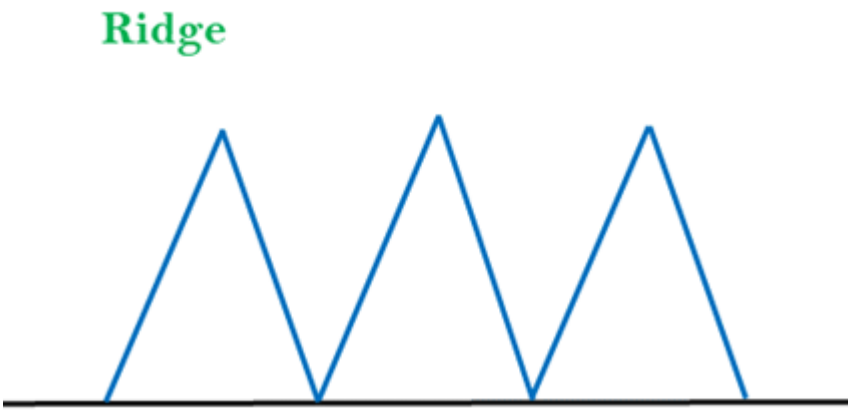
Ridges are a challenging problem for hill climbers that optimize in continuous spaces. Because hill climbers only adjust one element in the vector at a time, each step will move in an axis-aligned direction. If the target function creates a narrow ridge that ascends in a non-axis-aligned direction (or if the goal is to minimize, a narrow alley that descends in a non-axis-aligned direction), then the hill climber can only ascend the ridge (or descend the alley) by zig-zagging. If the sides of the ridge (or alley) are very steep, then the hill climber may be forced to take very tiny steps as it zig-zags toward a better position. Thus, it may take an unreasonable length of time for it to ascend the ridge (or descend the alley).

By contrast, gradient descent methods can move in any direction that the ridge or alley may ascend or descend. Hence, gradient descent or conjugate gradient method is generally preferred over hill climbing when the target function is differentiable. Hill climbers, however, have the advantage of not requiring the target function to be differentiable, so hill climbers may be preferred when the target function is complex.



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

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



Simulated Annealing:

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

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

Artificial neural networks are among the most effective learning methods currently known for certain types of problems. But BP training algorithm is based on the error gradient descent mechanism that the weight inevitably fall into the local minimum points. It is well known that simulated annealing (SA) and genetic algorithm (GA) are two global methods and can then be used to determine the optimal solution of NP-hard problem. In this paper, due to difficulty of obtaining the optimal solution in medium and large-scaled problems, a hybrid genetic algorithm (HGA) was also developed. The proposed HGA incorporates simulated annealing into a basic genetic algorithm that enables the algorithm to perform genetic search over the subspace of local optima. The two proposed solution methods were compared on Rosenbrock and Shaffer function global optimal problems, and computational results suggest that the HGA algorithm have good ability of solving the problem and the performance of HGA is very promising because it is able to find an optimal or near-optimal solution for the test problems. To evaluate the performance of the hybrid genetic algorithm-based neural network, BP neural network is also involved for a comparison purpose. The results compared with genetic algorithm-based indicated that this method was successful in evolving ANNs

There are certain optimization problems that become unmanageable using combinatorial methods as the number of objects becomes large. A typical example is the traveling salesman problem, which belongs to the NP-complete class of problems. For these problems, there is a very effective practical algorithm called simulated annealing (thus named because it mimics the process undergone by misplaced atoms in a metal when its heated and then slowly cooled). While this technique is unlikely to find the *optimum* solution, it can often find a very good solution, even in the presence of noisy data.

The traveling salesman problem can be used as an example application of simulated annealing. In this problem, a salesman must visit some large number of cities while minimizing the total mileage traveled. If the salesman starts with a random itinerary, he can then pairwise trade the order of visits to cities, hoping to reduce the mileage with each exchange. The difficulty with this approach is that while it rapidly finds a local minimum, it cannot get from there to the global minimum.

Simulated annealing improves this strategy through the introduction of two tricks. The first is the so-called "Metropolis algorithm" (Metropolis *et al.*1953), in which some trades that do not lower the mileage are accepted when they serve to allow the solver to "explore" more of the possible space of solutions. Such "bad" trades are allowed using the criterion that

$$e^{-\Delta D/T} > R\,(0, 1),$$

where ΔD is the change of distance implied by the trade (negative for a "good" trade; positive for a "bad" trade), T is a "synthetic temperature," and $R(0, 1)$ is a random number in the interval $[0, 1]$. D is called a "cost function," and corresponds to the free energy in the case of annealing a metal (in which case the temperature parameter would actually be the kT, where k is Boltzmann's Constant and T is the physical temperature, in the Kelvin absolute temperature scale). If T is large, many "bad" trades are accepted, and a large part of solution space is accessed. Objects to be traded are generally chosen randomly, though more sophisticated techniques can be used

Genetic Algorithms

Terminology:

- Fitness function
- Population
- Encoding schemes
- Selection
- Crossover
- Mutation
- Elitism

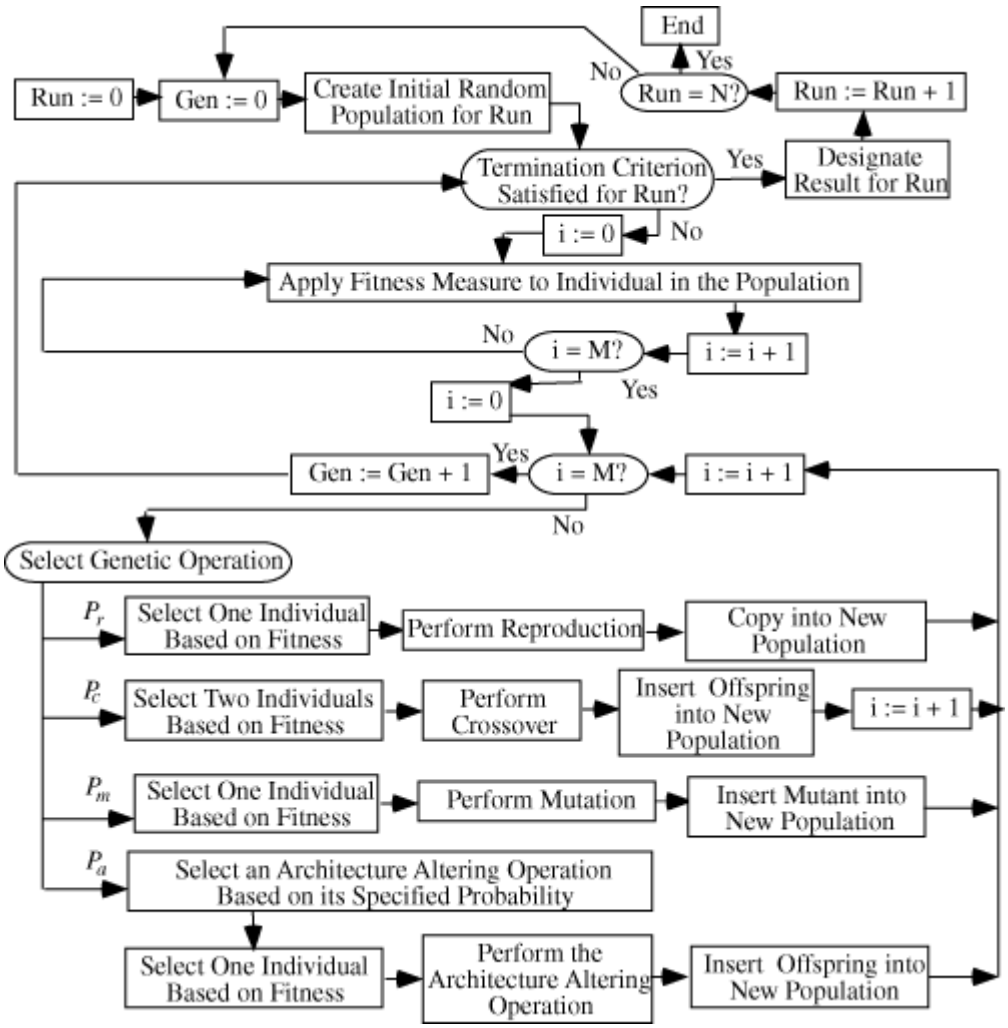
Flowchart of Genetic Algorithm:

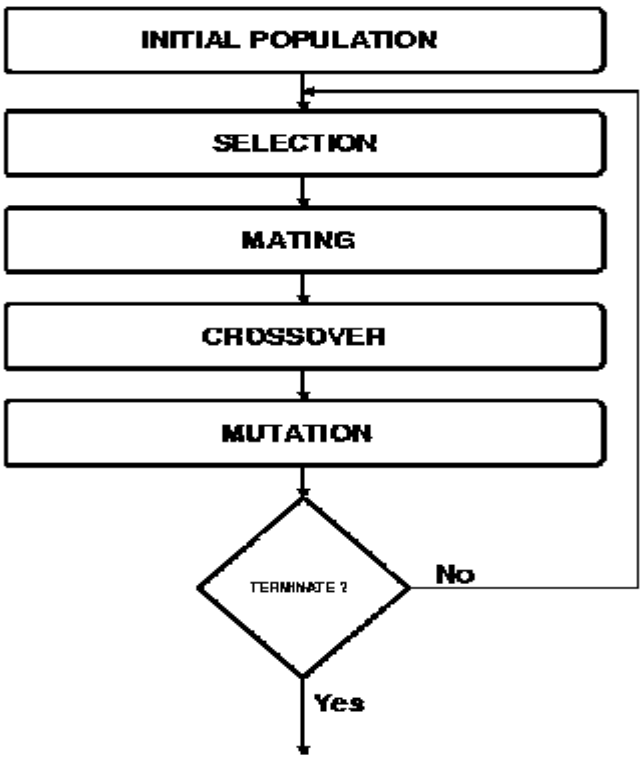
Genetic Algorithms are a form of Artificial Intelligence. They learn by evolving a fit set of solution to previously specified problem. The fundamental idea behind genetic algorithms is the same idea behind Darwinian evolution, survival of the fittest. Each potential solution to a problem in a genetic algorithm is represented by one individual. The fitter individuals are allowed to breed more often. By allowing fitter individuals to breed more often, the population tends toward the desired solution with time. This ability to tend towards a solution is comparable to reasoning capability.

Genetic programming is problem-independent in the sense that the flowchart specifying the basic sequence of executional steps is not modified for each new run or each new problem.

There is usually no discretionary human intervention or interaction during a run of genetic programming (although a human user may exercise judgment as to whether to terminate a run).

The figure below is a flowchart showing the executional steps of a run of genetic programming. The flowchart shows the genetic operations of crossover, reproduction, and mutation as well as the architecture-altering operations. This flowchart shows a two-offspring version of the crossover operation.


End
Yes
No
Run := 0
Gen := 0
Create Initial Random Population for Run
Run = N?
Run := Run + 1
Termination Criterion Satisfied for Run?
Yes
Designate Result for Run
i := 0
No
Apply Fitness Measure to Individual in the Population
No
i = M?
i := i + 1
i := 0
Yes
Yes
Gen := Gen + 1
i = M?
i := i + 1
No
Select Genetic Operation
P_r
Select One Individual Based on Fitness
Perform Reproduction
Copy into New Population
P_c
Select Two Individuals Based on Fitness
Perform Crossover
Insert Offspring into New Population
i := i + 1
P_m
Select One Individual Based on Fitness
Perform Mutation
Insert Mutant into New Population
P_a
Select an Architecture Altering Operation Based on its Specified Probability
Select One Individual Based on Fitness
Perform the Architecture Altering Operation
Insert Offspring into New Population


INITIAL POPULATION
SELECTION
MATING
CROSSOVER
MUTATION
TERMINATE ?
No
Yes

Let's check the take away from this lecture

Exercise

Q.1……..not local search algorithm

(a) A*

(b) Hill Climbing

(c) Simulated Hill Climbing

(d) Genetic Algorithm

Q.2 **Hill climbing cannot get stuck due to which of the following phenomenon?**

(a) Local Maxima

(b) Inadmissible heuristic

(c)Ridges

(d)Plateau

Q.3……..is not a term of Genetic Algorithm.

(a) Selection

(b)Crossover

(c)Mutation

(d) Successor function

Learning from this lecture: Learners will be able to understand various optimization algorithm techniques.

Lecture : 2

Adversarial Search

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

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

Types of Games in AI:

	Deterministic	Chance Moves
Perfect information	Chess, Checkers, go, Othello	Backgammon, mono
Imperfect information	Battleships, blind, tic-tac-toe	Bridge, poker, scrabb

- **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 has 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.

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:

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

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

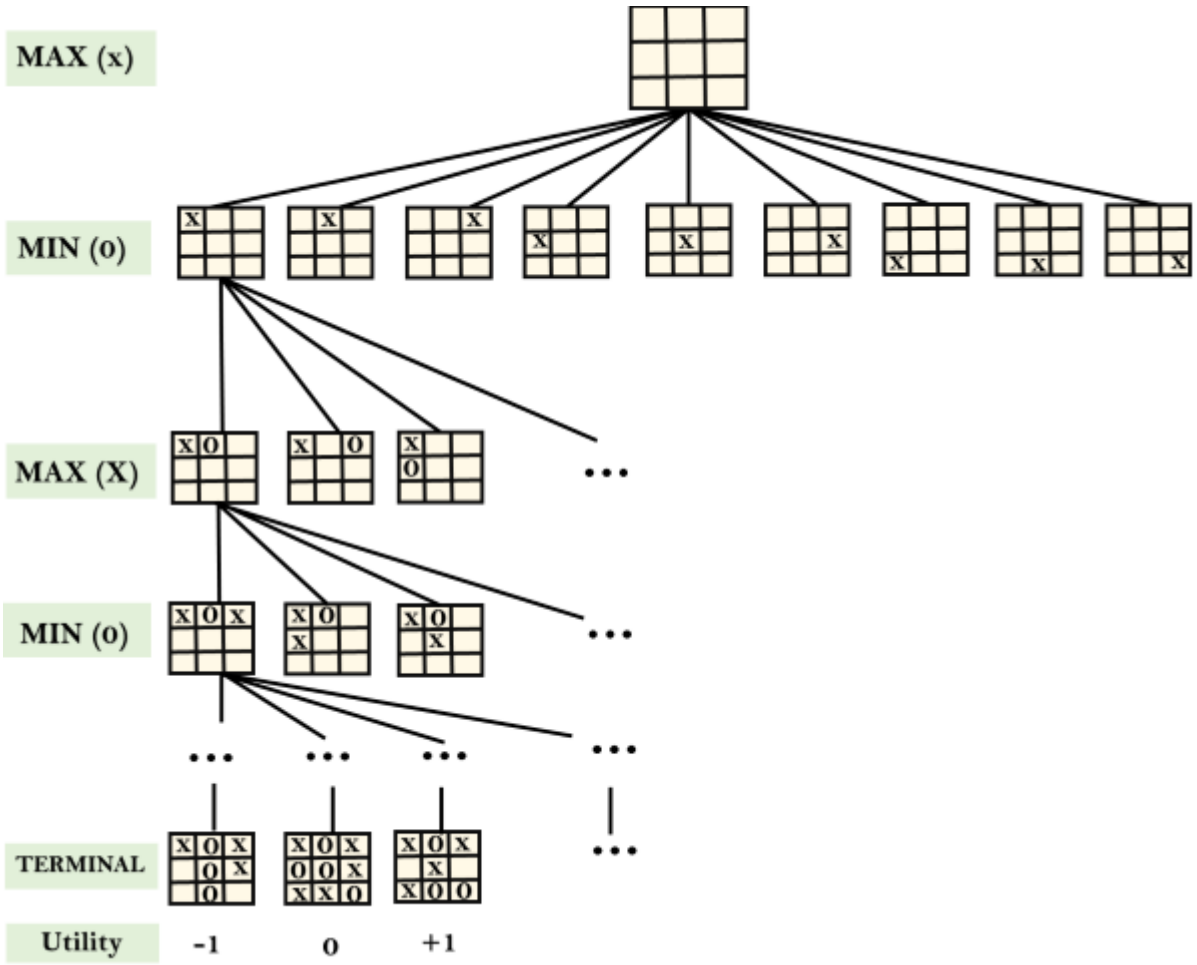
Game tree:

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

Example: Tic-Tac-Toe game tree:

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

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



Example Explanation:

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

Hence adversarial Search for the minimax procedure works as follows:

- It aims to find the optimal strategy for MAX to win the game.

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

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

For a state S MINIMAX(s) =

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

Mini-Max Algorithm in Artificial Intelligence

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

Pseudo-code for MinMax Algorithm:

```
1. function minimax(node, depth, maximizingPlayer) is
2. if depth ==0 or node is a terminal node then
3. return static evaluation of node
4.
```

```
5. if MaximizingPlayer then      // for Maximizer Player
6. maxEva= -infinity
7.  for each child of node do
8.  eva= minimax(child, depth-1, false)
9. maxEva= max(maxEva,eva)       //gives Maximum of the values
10. return maxEva
11.
12. else                         // for Minimizer player
13.  minEva= +infinity
14.  for each child of node do
15.  eva= minimax(child, depth-1, true)
16.  minEva= min(minEva, eva)       //gives minimum of the values
17.  return minEva
```

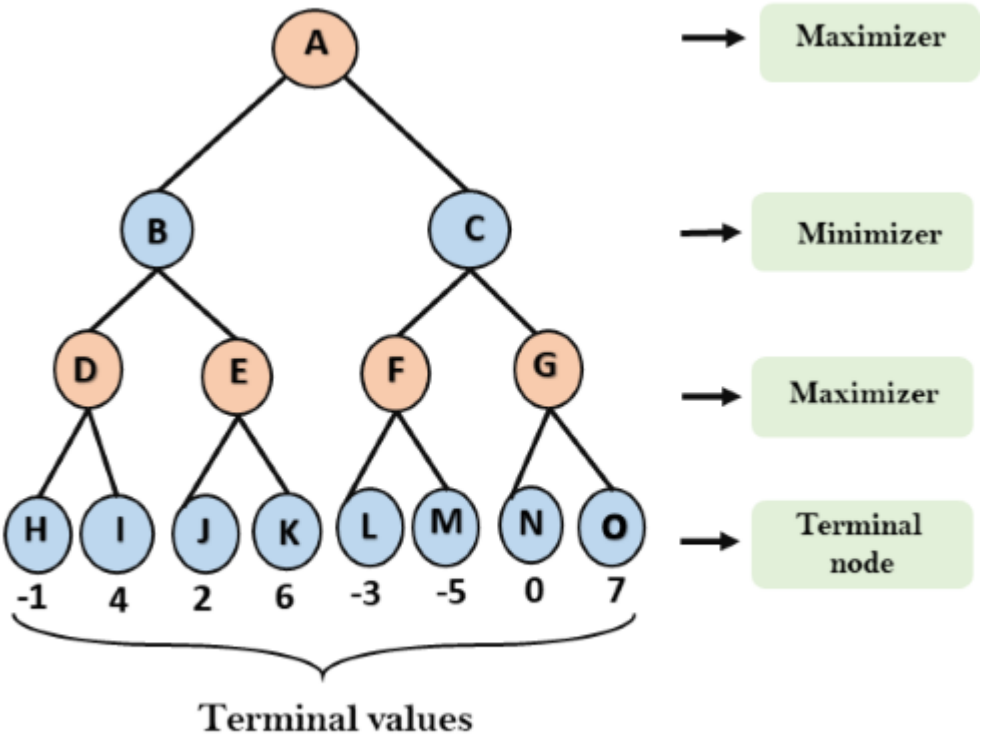
Initial call:

Minimax(node, 3, true)

Working of Min-Max Algorithm:

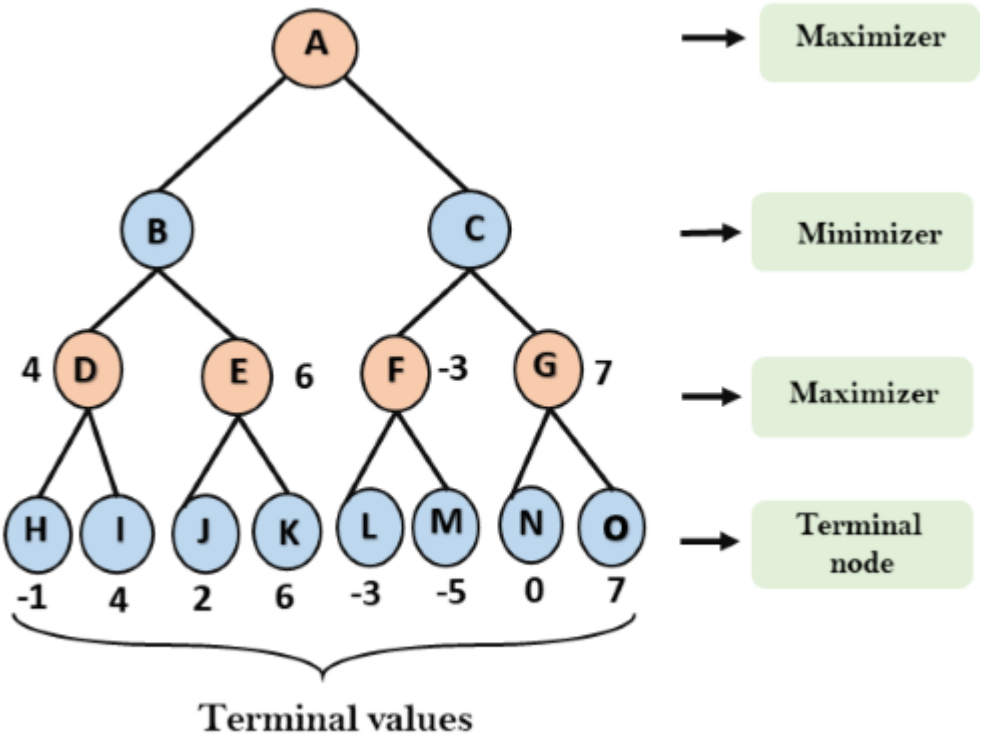
- The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.
- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.
- At the terminal node, the terminal values are given so we will compare those 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.



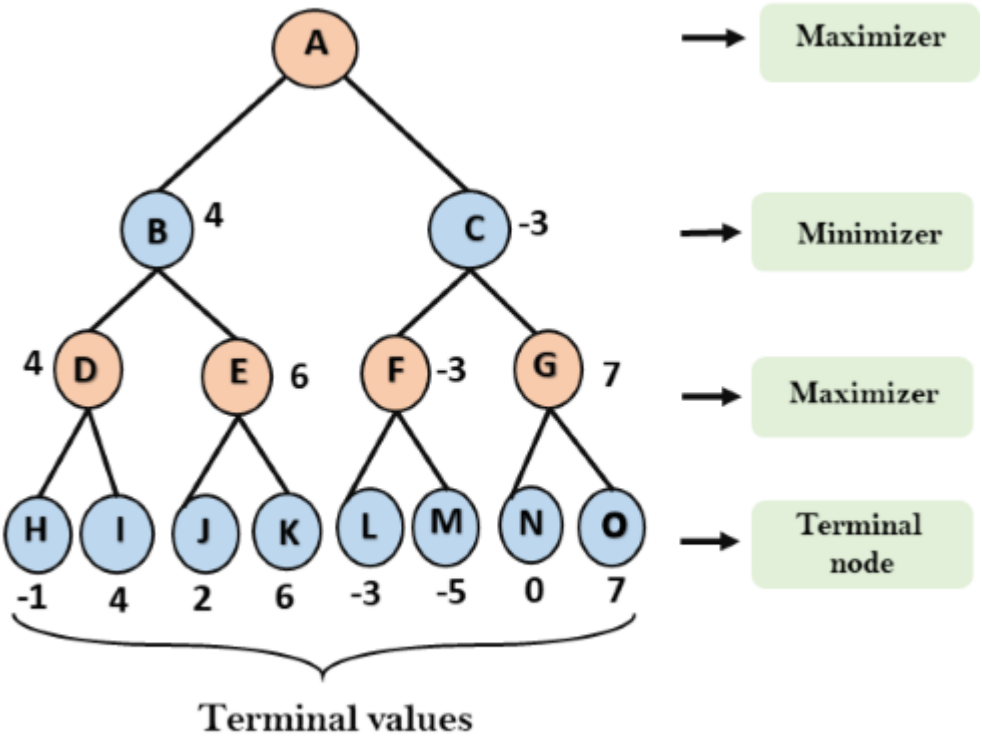
Step 2: Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

- 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



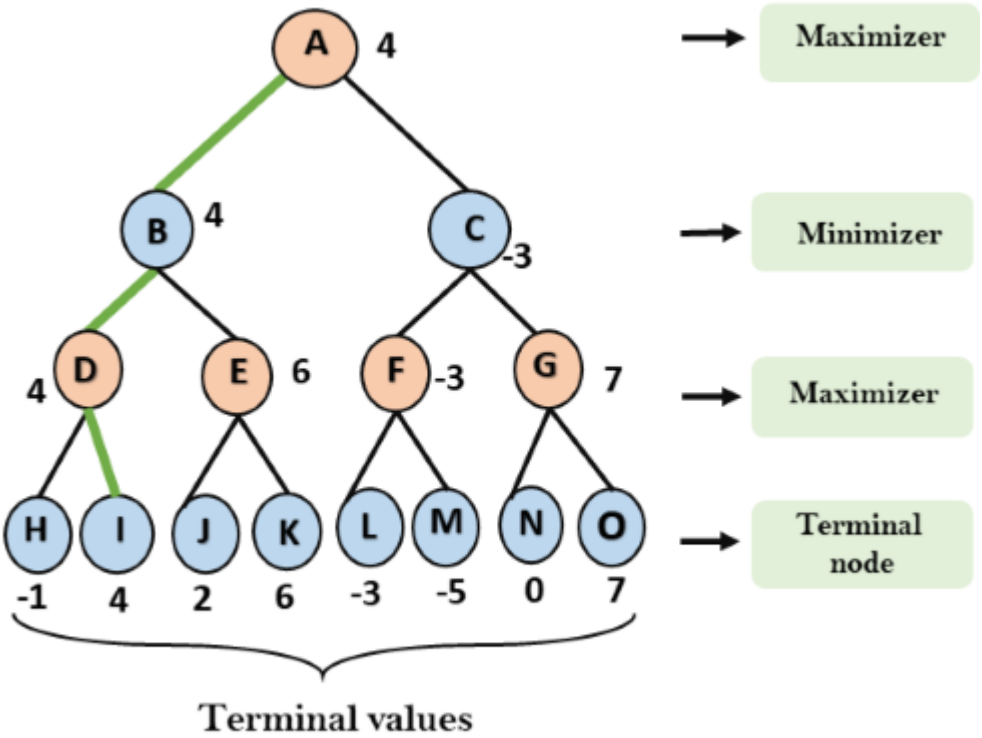
Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

- 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. This limitation of the minimax algorithm can be improved from **alpha-beta pruning** which we have discussed in the next topic.

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:

    1. **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 -∞.
    2. **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.

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

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

Condition for Alpha-beta pruning:

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

1. α>=β

Key points about alpha-beta pruning:

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

Pseudo-code for Alpha-beta Pruning:

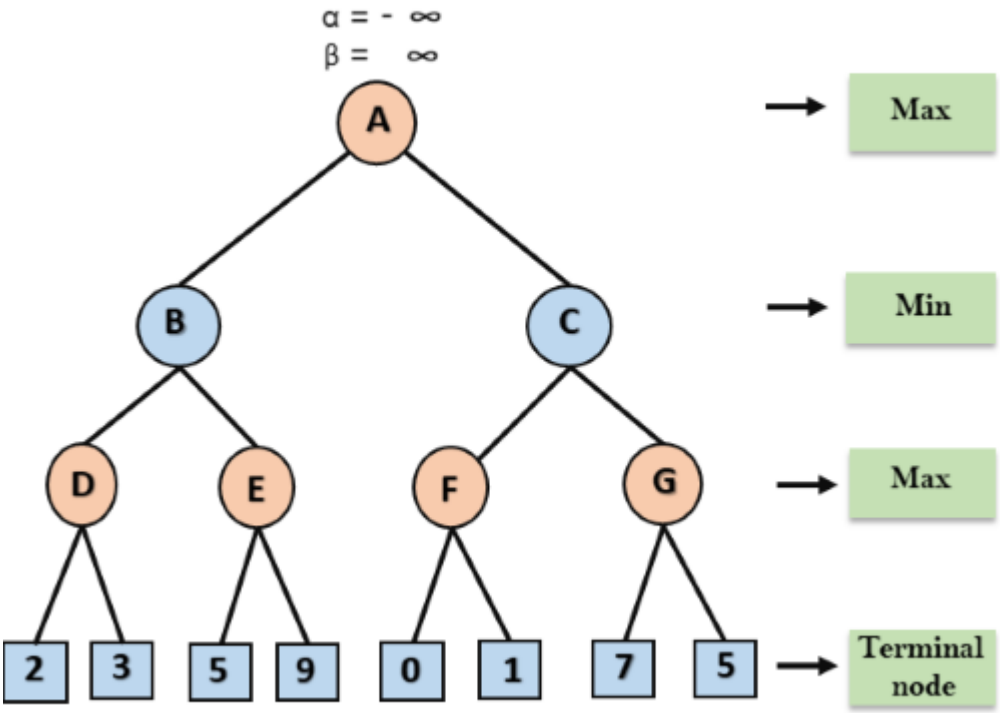
```
1. function minimax(node, depth, alpha, beta, maximizingPlayer) is
2. if depth ==0 or node is a terminal node then
3. return static evaluation of node
4.
```

```
5.  if MaximizingPlayer then      // for Maximizer Player
6.     maxEva= -infinity
7.     for each child of node do
8.     eva= minimax(child, depth-1, alpha, beta, False)
9.    maxEva= max(maxEva, eva)
10.   alpha= max(alpha, maxEva)
11.    if beta<=alpha
12.  break
13.  return maxEva
14.
15. else                    // for Minimizer player
16.    minEva= +infinity
17.    for each child of node do
18.    eva= minimax(child, depth-1, alpha, beta, true)
19.    minEva= min(minEva, eva)
20.    beta= min(beta, eva)
21.     if beta<=alpha
22.   break
23.  return minEva
```

Working of Alpha-Beta Pruning:

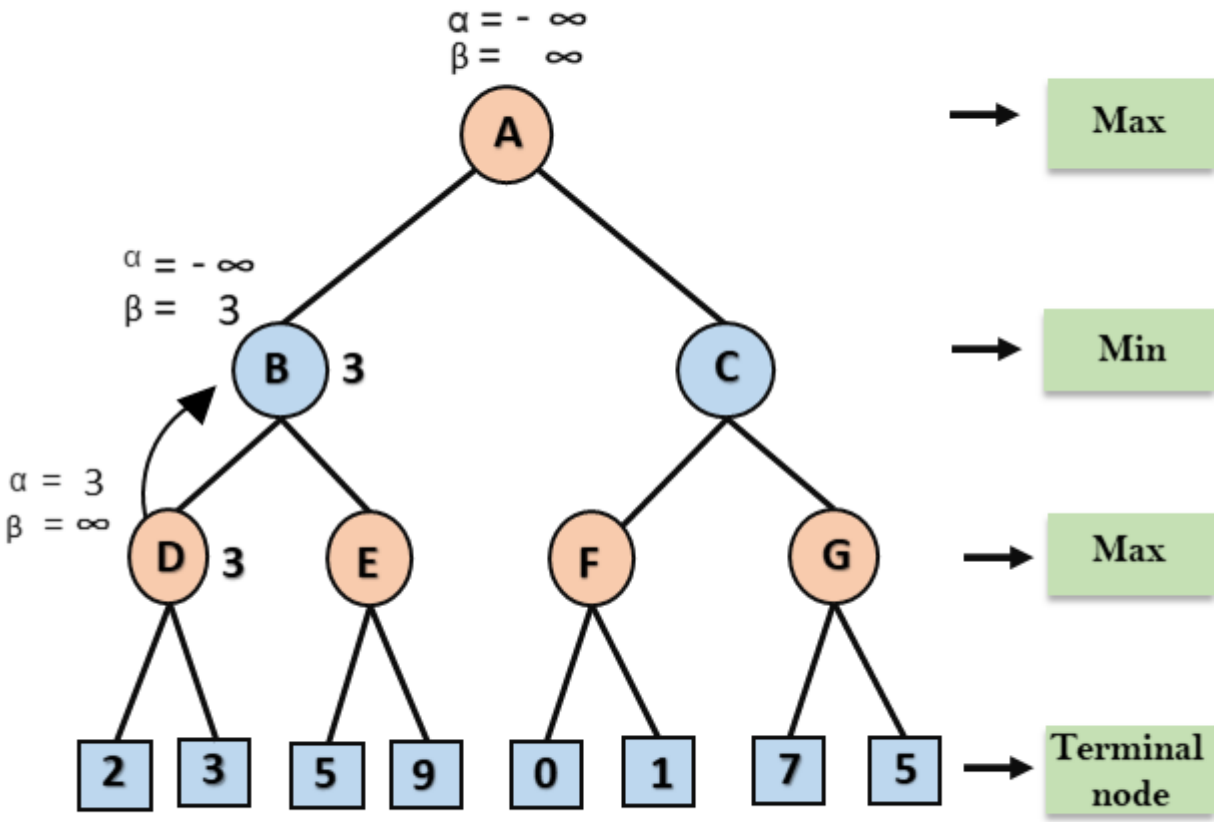
Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

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



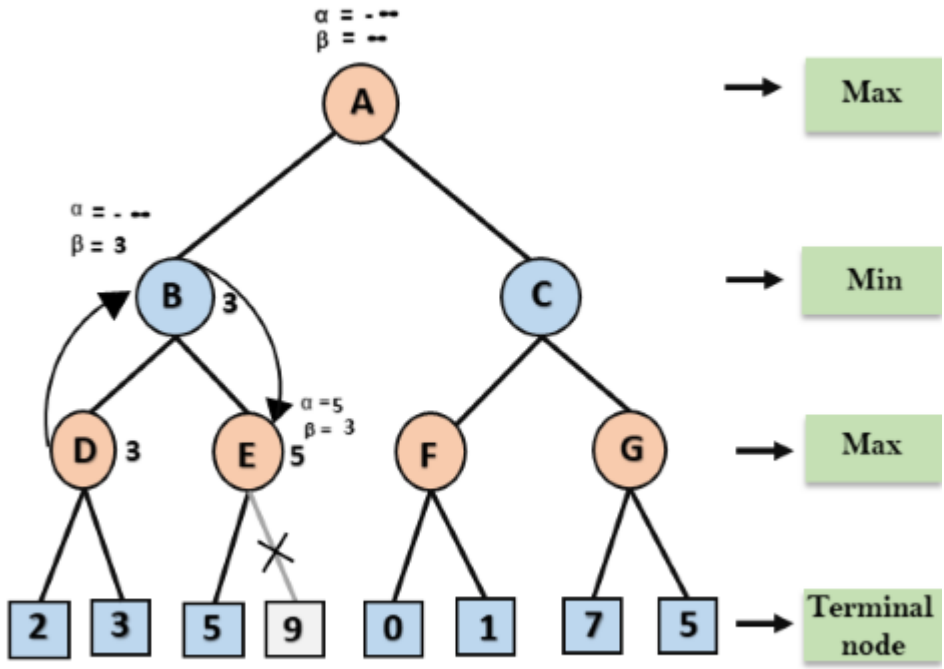
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

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



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

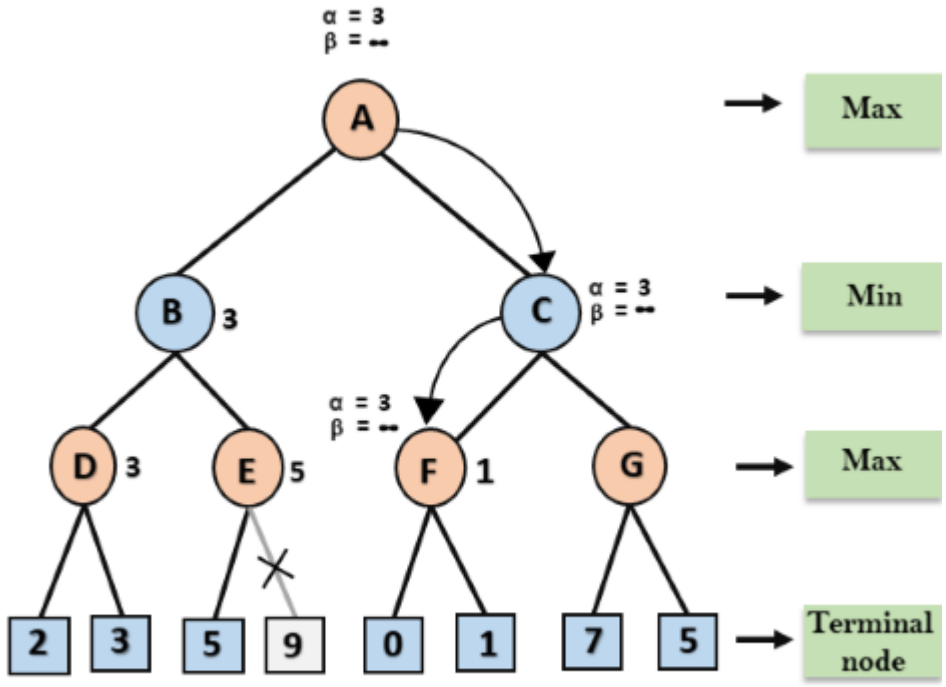
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will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



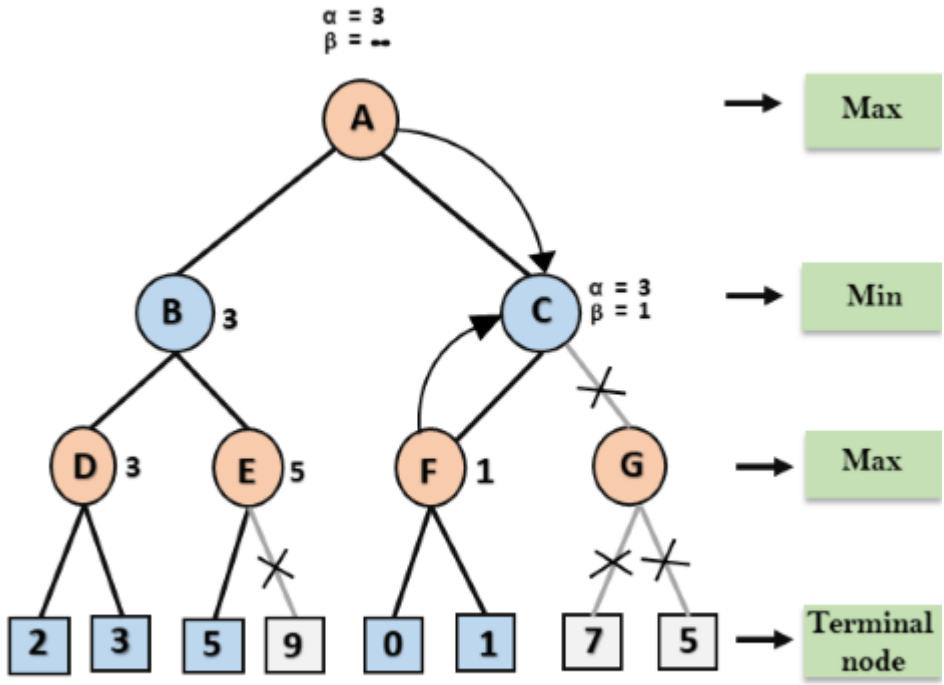
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

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

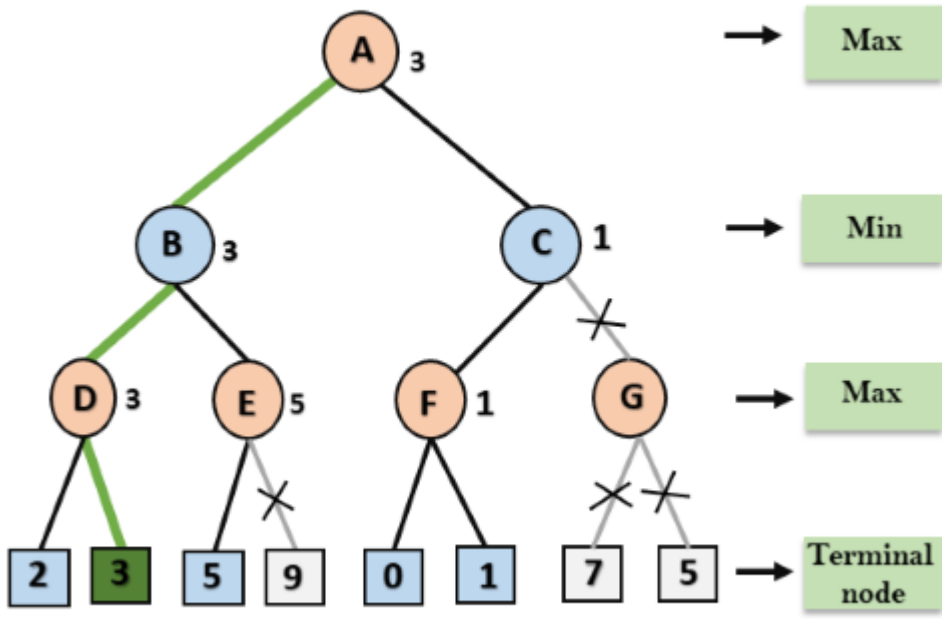
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



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



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



Move Ordering in Alpha-Beta pruning:

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

It can be of two types:

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

Rules to find good ordering:

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

- Occur the best move from the shallowest node.
- Order the nodes in the tree such that the best nodes are checked first.

- Use domain knowledge while finding the best move. Ex: for Chess, try order: captures first, then threats, then forward moves, backward moves.
- We can bookkeep the states, as there is a possibility that states may repeat.

Let's check the take away from this lecture

Exercise

Q. 1. Which search is equal to minimax search but eliminates the branches that can't influence the final decision?

a) Depth-first search
b) Breadth-first search
c) Alpha-beta pruning
d) None of the mentioned

Q.2 To which depth does the alpha-beta pruning can be applied?

a) 10 states
b) 8 States
c) 6 States
d) Any depth

Q.3 Which value is assigned to alpha and beta in the alpha-beta pruning?

a) Alpha = max
b) Beta = min
c) Beta = max
d) Both Alpha = max & Beta = min

Learning from this lecture: Learners will be able to understand theory behind two player game theory

Lecture : 3

Constraint Satisfaction Problem

Constraint satisfaction problems, or **CSPs** for short, are a flexible approach to searching that have proven useful in many AI-style problems

CSPs can be used to solve problems such as

- **graph-coloring**: given a graph, the a coloring of the graph means assigning each of its vertices a color such that no pair of vertices connected by an edge have the same color
  - in general, this is a very hard problem, e.g. determining if a graph can be colored with 3 colors is NP-hard
  - many problems boil down to graph coloring, or related problems
- **job shop scheduling**: e.g. suppose you need to complete a set of tasks, each of which have a duration, and constraints upon when they start and stop (e.g. task c can't start until both task a and task b are finished)
  - CSPs are a natural way to express such problems

- **cryptarithmetic puzzles**: e.g. suppose you are told that TWO + TWO = FOUR, and each of the letters corresponds to a *different* digit from 0 to 9, and that a number can't start with 0 (so T and F are not 0); what, if any, are the possible values for the letters?
    - while these are not directly useful problems, they are a simple test case for CSP solvers

the basic idea is to have a set of variables that can be assigned values in a constrained way

a CSP consists of three main components:

- XX: a set of **variables** {X1,…,Xn}{X1,…,Xn}
- DD: a set of **domains** {D1,…,Dn}{D1,…,Dn}, one domain per variable
    - i.e. the domain of XiXi is DiDi, which means that XiXi can only be assigned values from DiDi
    - we're only going to consider **finite domains**; you can certainly have CSPs with infinite domains (e.g. real numbers), but we won't consider such problems here
- CC is a set of **constraints** that specify allowable assignments of values to variables
    - for example, a **binary constraint** consists of a pair of different variables, (Xi,Xj)(Xi,Xj), and a set of pairs of values that XiXi and XjXj can take on at the same time
    - we will usually only deal with binary constraints; constraints between three or more variables are possible (e.g. Xi,Xj,XkXi,Xj,Xk are all different), but they don't occur too frequently, and can be decomposed into binary constraints

example 1: suppose we have a CSP as follows:

- three variables X1X1, X2X2, and X3X3
- domains: D1={1,2,3,4}D1={1,2,3,4}, D2={2,3,4}D2={2,3,4}, D3={3,7}D3={3,7}
    - so X1X1 can only be assigned one of the values 1, 2, 3, or 4
- constraints: no pair of variables have the same value, i.e. X1≠X2X1≠X2, X1≠X3X1≠X3, and X2≠X3X2≠X3; we can explicitly describe each of these constraints as a relation between the two variables where the pairs show the allowed values that the variables can be simultaneously assigned, i.e.
    - X1≠X2={(1,2),(1,3),(1,4),(2,3),(2,4),(3,2),(3,4),(4,2),(4,3)}X1≠X2={(1,2),(1,3),(1,4),(2,3),(2,4),(3,2),(3,4),(4,2),(4,3)}
    - X1≠X3={(1,3),(1,7),(2,3),(2,7),(3,7),(4,3),(4,7)}X1≠X3={(1,3),(1,7),(2,3),(2,7),(3,7),(4,3),(4,7)}
    - X2≠X3={(2,3),(2,7),(3,7),(4,3),(4,7)}X2≠X3={(2,3),(2,7),(3,7),(4,3),(4,7)}

these three constraints are each binary constraints because they each constrain 2 variables

example 2: suppose we have the same variables and domains as in the previous example, but now the constraints are X1<X2X1<X2 and X2+X3≤5X2+X3≤5

- domains: D1={1,2,3,4}D1={1,2,3,4}, D2={2,3,4}D2={2,3,4}, D3={3,7}D3={3,7}
- both of the constraints are binary constraints, because they each involve two variables
- X1<X2={(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)}X1<X2={(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)}
- X2+X3≤9={(2,3),(3,3)}X2+X3≤9={(2,3),(3,3)}

by looking at the constraints for this problem, we note the following:

- the only possible value for X3X3 is 3, because 3 is the only value for X3X3 in the constraint X2+X3≤9X2+X3≤9
- we can also see the constraint X2+X3≤9X2+X3≤9 that X2X2 *cannot* be 4; so in the constraint X1<X2X1<X2 we can discard all pairs whose second value is 4, giving: X1<X2={(1,2),(1,3),(2,3)}X1<X2={(1,2),(1,3),(2,3)}
- so the final solutions to this problem are: (X1=1,X2=1,X3=3)(X1=1,X2=1,X3=3), (X1=1,X2=3,X3=3)(X1=1,X2=3,X3=3), (X1=2,X2=3,X3=3)(X1=2,X2=3,X3=3)

example 3: suppose we have the same variables and domains as in the previous example, but now with the two constraints X1+X2=X3X1+X2=X3, and X1X1 is even

- domains: D1={1,2,3,4}D1={1,2,3,4}, D2={2,3,4}D2={2,3,4}, D3={3,7}D3={3,7}
- X1+X2=X3X1+X2=X3 is a **ternary constraint**, because it involves 3 variables
- X1+X2=X3={(1,2,3),(3,4,7),(4,3,7)}X1+X2=X3={(1,2,3),(3,4,7),(4,3,7)}
- X1X1 is even is a **unary** constraint, because it involves one variable
- X1X1 is even ={2,4}={2,4}
- looking at the triples in X1+X2=X3X1+X2=X3, the only one where *X_1* is even is (4,3,7)(4,3,7), and so the only solution to his problem is (X1=4,X2=3,X3=7)(X1=4,X2=3,X3=7)

- variables can be assigned, or unassigned
- if a CSP has some variables assigned, and those assignments don't violate any constraints, we say the CSP is **consistent**, or that it is **satisfied**
    - that partial sets of CSP variable can be satisfied is an important part of many CSP algorithms that work by satisfying one variable at a time
- if all the variables of a CSP are assigned a value, we call that a **complete assignment**
- a **solution** to a CSP complete assignment that is consistent, i.e. a complete assignment that does not violate any constraints
- depending on the constraints, a CSP may have 0, 1, or many solutions
    - a CSP with no solutions is sometimes referred to as **over-constrained**
    - a CSP with many solutions is sometimes referred to as **under-constrained**
- depending upon the problem being solved and its application, we may want just 1 solution, or we might want multiple solutions
    - for over-constrained CSPs, we may even be satisfied with a solution that violates the fewest constraints
- if a domain is empty, then the CSP is no solution
- if a domain has only one value, then that is the only possible value for the corresponding variable
- the size of the **search space** for a CSP is |D1|·|D2|·…·|Dn||D1|·|D2|·…·|Dn|
    - this is the number of n-tuples that the CSP is searching through
    - this is often a quick and easy way to estimate the potential difficulty of a CSP: the bigger the search space, the harder the problem might be
        - this is just a rule of thumb: it's quite possible that a problem with a large search space is easier than a problem with s small search space, perhaps because there are many more solutions to be found in the large search space

Constraint Propagation: Arc Consistency

as mentioned above, we are only considering CSPs with finite domains, and with binary constraints between variables

it turns out that many such CSPs can be translated into a simpler CSP that is guaranteed to have the same solutions

the idea we will consider here is **arc consistency**

the CSP variable XiXi is said to be **arc consistent** with respect to variable XjXj if for every value in DiDi there is at least one corresponding value in DjDj that satisfies the binary constraint on XiXi and XjXj

an entire CSP is said to be **arc consistent** if every variable is arc consistent with every other variable

if a CSP is not arc consistent, then it turns out that there are relatively efficient algorithms that will make it arc consistent

- and the resulting arc consistent CSP will have the same solutions as the original CSP, but often a smaller search space
- for some problems, making a CSP arc consistent may be all that is necessary to solve it
- *Let's check the take away from this lecture*

Exercise

Q.1 Which of the Following problems can be modeled as CSP?

a) 8-Puzzle problem

b) 8-Queen problem

c) Map coloring problem

d) All of the mentioned

Learning from this lecture: Learners will be able to understand real world constraint satisfaction problem

Conclusion

This chapter was introduction to optimization and advance search methods with it's application for various problems

Short Answer Questions:

Q.1 Write a short note on Simulated Annealing.

Q.2 Discuss any CSP problem.

Long Answer Questions:

Q.1 Consider an 8 puzzle problem with following initial and goal states.

7	2	4
5		6
8	3	1

	1	2
3	4	5
6	7	8

Successors at next two levels. Apply no. of misplaced tiles as the heuristic function. Which successor nodes will be selected at each level if we apply Hill climbing algorithm.

Q.2 Explain how GA work. Define terms chromosome, fitness function, crossover, mutation.

Q. 3 Define minimax, alpha beta pruning with suitable example. Give alpha beta search algorithm. Why is it suitable for two player games.

MCQs Missed Module 3

1. Which of the following is not an uninformed search technique?
   i) Breadth-first search ii) Uniform-cost search
   iii) Depth-first search iv) **A* search**
2. Which of the following is not an informed search technique?
   i) Greedy best-first search ii) A* search iii)**Depth-first search**
3. Which of the following is not an application of CSP?
   i) Floor Planning ii) Transportation Scheduling **iii) both i and ii** iv) none
4. Which of the following gives an optimal solution?
   i) Breadth First ii) Uniform cost **iii) both i and ii** iv) none
5. Which of the following gives a complete solution?
   i) Iterative Deepening ii) Bidirectional **iii) both i and ii** iv) none

References:

Books:

	Title	Authors	Publisher	Edition	Year	Chapter No
1	Artificial Intelligence a Modern Approach	Stuart J. Russell and Peter Norvig	McGraw Hill	3rd Edition	2009	
2	A First Course in Artificial Intelligence	Deepak Khemani	McGraw Hill Education (India)	1st Edition	2013	

Online Resources:

- https://onlinecourses.nptel.ac.in/noc20_cs81/preview
- https://nptel.ac.in/courses/106/102/106102220/
- https://www.coursera.org/learn/introduction-to-ai/
- https://www.coursera.org/learn/mind-machine-problem-solving-methods
- https://www.javatpoint.com/history-of-artificial-intelligence
- https://en.wikipedia.org/
- http://people.eecs.berkeley.edu/~russell/slides/