Unit-4
Adversarial Search and Constraint Satisfaction Problem

Adversarial Search

Adversarial search is a game-playing technique where the agents are surrounded by a competitive environment. A conflicting goal is given to the agents (multiagent). These agents compete with one another and try to defeat one another in order to win the game. Such conflicting goals give rise to the ***adversarial search*** often known as ***games***.

Mathematically, this search is based on the concept of '***Game Theory***'. According to game theory, a game is played between two players. To complete the game, one has to win the game and the other loosed automatically.

We are opponents- I win, you loose.

Popular ***examples of adversarial search*** include games like chess, checkers, tic-tac-toe, and Go, where players strategically compete to achieve their goals.

Techniques required to get the best optimal solution

There is always need to choose those algorithms which provide the best optimal solution in a limited time. So, we use the following techniques which could fulfill our requirements:

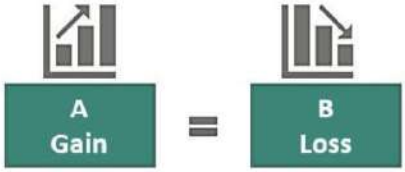
- ***Pruning:*** A technique which allows ignoring the unwanted portions of a search tree which make no difference in its final result.
- ***Heuristic Evaluation Function:*** It allows to approximate the cost value at each level of the search tree, before reaching the goal node.

Types of Games in AI

- ***Perfect Information:*** A game with 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 agent 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, 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. Such games are also called as stochastic games. Examples: Backgammon, Monopoly, Poker etc.

Zero-Sum Game

A ***zero-sum game*** is a type of adversarial game in which the total gain or loss among all players is always zero. In these games, one player's gain is exactly equal to the other player's loss. In other words, the interests of players are strictly opposing, and the game is purely competitive.



Chess and tic-tac-toe are examples of a Zero-sum game.

Tic-Tac-Toe: In a zero-sum game setup: If Player A wins, their payoff is +1, and Player B's is -1. The total payoff is (+1) + (−1) = 0. If Player B wins, the payoffs are reversed: (−1) + (+1) = 0.. If it's a draw, both players receive a payoff of 0, making the total 0 + 0 = 0.

Formalization of the problem

A game can be formally defined with the following elements:

- ***Initial State(S_0):*** 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*. For chess, the outcomes are a win, loss or draw and its payoff values are +1, 0, 1/2. 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, action's function, and the 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.
- From the initial state, MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing an O until we reach leaf nodes corresponding to terminal states such that one player has three squares in a row or all the squares are filled.
- The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are good for MAX and bad for MIN (which is how the players get their names).

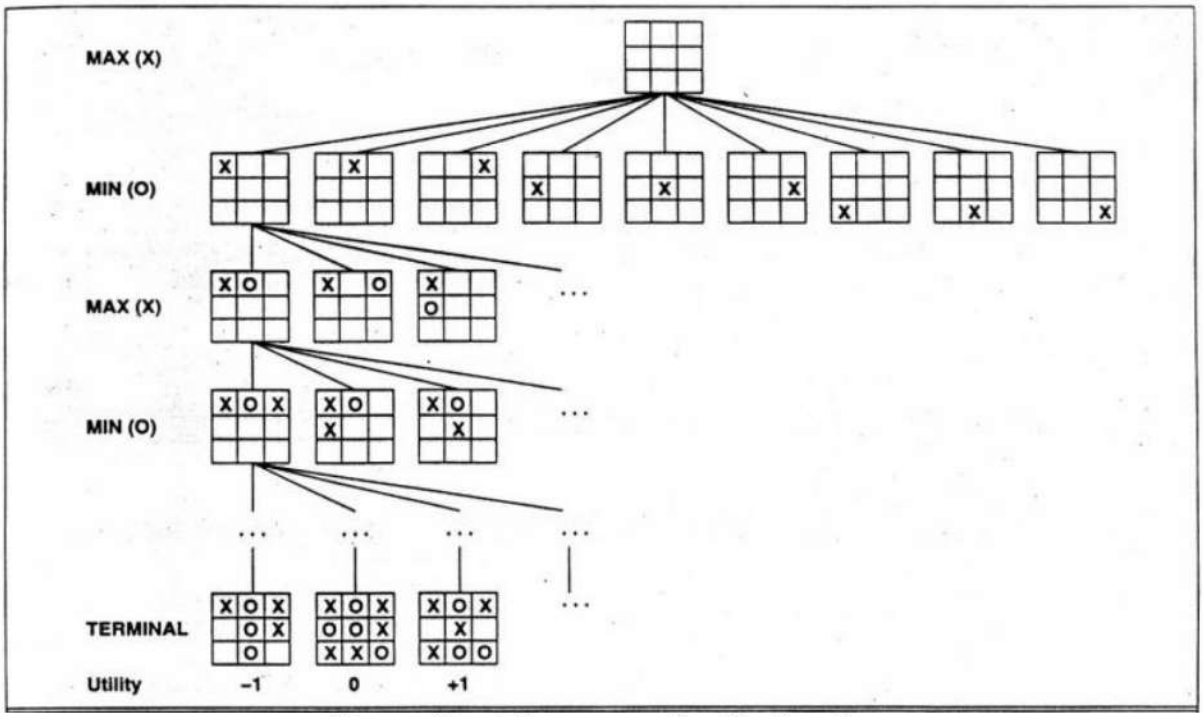


Figure: Partial game tree for Tic-Tac-Toe

- ***Initial State(S_0):*** The top node in the game-tree represents the initial state in the tree and shows all the possible choices to pick out one.
- ***Player(s):*** There are two players, MAX and MIN. MAX begins the game by picking one best move and place X in the empty square box.
- ***Action(s):*** Both the players can make moves in the empty boxes chance by chance.
- ***Result(s, a):*** The move made by MIN and MAX will decide the outcome of the game.
- ***Terminal-Test(s):*** When all the empty boxes will be filled, it will be the terminating state of the game.
- ***Utility:*** At the end, we will get to know who wins: MAX or MIN, and accordingly, the price will be given to them.

Types of Algorithms in Adversarial Search

In a normal search, we follow a sequence of actions to reach the goal or to finish the game optimally. But in adversarial search, the result depends on the players which will decide the result of the game. It is also obvious that the solution for the goal state will be an optimal solution because the player will try to win the game with the shortest path and under limited time.

There are following types of adversarial search:

- Min-max Algorithm
- Alpha-beta Pruning

Min-max Algorithm

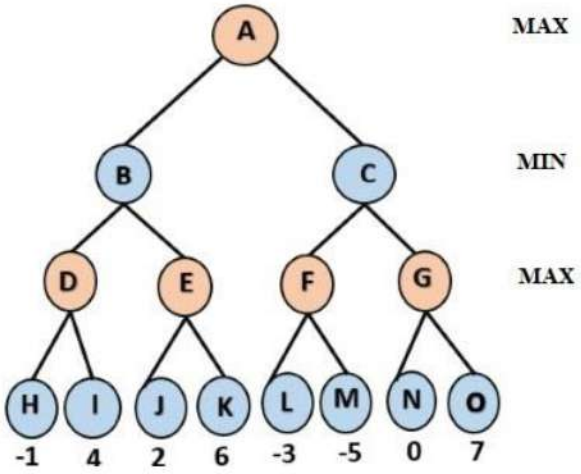
Min-max algorithm is a backtracking algorithm which is used in decision-making and game theory to find the optimal move for a player assuming that opponent is also playing optimally. In this algorithm ***two players*** play the game, one is called ***MAX*** and other is called ***MIN***. Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit. The MAX tries to get the highest score possible while the MIN tries to do the opposite and get the lowest score possible.

Algorithm Steps:

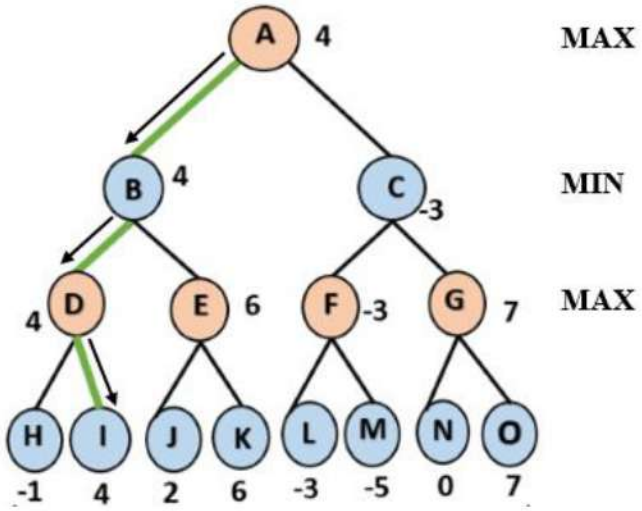
1. Generate all possible game states from the current state by forming a game tree.
2. Apply the utility function to leaf nodes to get their values.
3. Propagate the values from the leaf nodes till the root, applying the minmax strategy at each level.
   - If the parent state is MAX, give it the MAX of its children.
   - If the parent state is MIN, give it the MIN of its children.
4. Choose the optimal move from the root based on the propagated values.

Example:

Consider a following state space representing a game:



The min-max search produce the following tree:



Max moves to B, then Min moves to D, and finally Max moves to I.

Properties of Min-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 $O(b^m)$.
- ***Space Complexity:*** Space complexity of Min-max algorithm is also similar to DFS which is $O(bm)$.

Limitations of the min-max Algorithm

- The min-max algorithm slows down for complex games like Chess and Go due to their huge branching factor and the exponential growth of computations with tree depth.
- Evaluating all possible moves to terminal nodes is computationally expensive, especially for games with large search trees, as the evaluations grow exponentially with depth.

Alpha-Beta Pruning

The problem with minmax algorithm search is that the number of game states has to be examined exponentially with the greater number of moves. ***Alpha–beta pruning*** is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minmax algorithm in its search tree. It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minmax tree, it returns the same move as minmax would, but prunes away branches that cannot possibly influence the final decision. Hence by pruning these nodes, *it makes the algorithm fast and reduces computational cost.*

Alpha-beta pruning uses two parameters, *alpha* and *beta*.

- ***Alpha:*** The best choice (i.e. highest value) found till the current state on the path traversed by the MAX.
- ***Beta:*** The best choice (i.e. lowest-value) found till the current state on the path traversed by the MIN.

Properties and Conditions of the Alpha Beta Pruning Algorithm

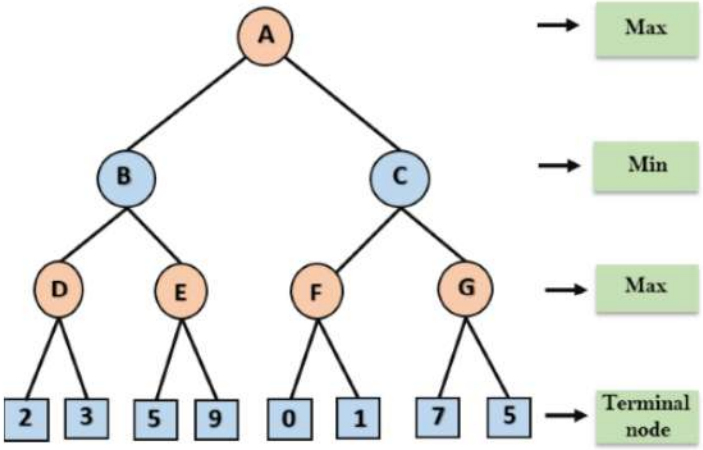
- The initialization of the parameters
  - Alpha (α) is initialized with $-\infty$.
  - Beta (β) is initialized with $+\infty$.
- Updating the parameters
  - Alpha (α) is updated only by the MAX at its turn.
  - Beta (β) is updated only by the MIN at its turn.
- Passing the parameters
  - Alpha (α) and Beta (β) are passed on to only child nodes.
  - While backtracking the game tree, the node values are passed to parent nodes.
- Pruning Condition
  - The child sub-trees, which are not yet traversed, are pruned if the condition $\alpha \geq \beta$ holds.

Pseudocode for Alpha Beta Pruning

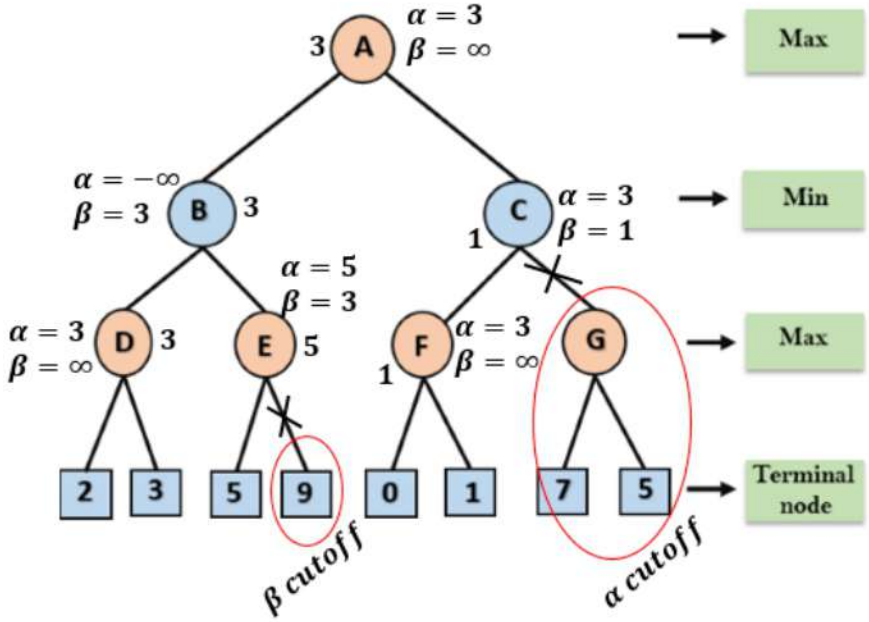
```
ALPHA_BETA(N, α, β)                          Initially:
    if N is a terminal node                      α = −∞
        return eval(N)                           β = +∞
    if N is a MAX node
        for each child C of N
            α ← max(α, ALPHA_BETA(C, α, β))
            if α ≥ β then return β     /* β cutoff */
        return α
    else (N is a MIN node)
        for each child C of N
            β ← min(β, ALPHA_BETA(C, α, β))
            if α ≥ β then return α     /* α cutoff */
        return β
```

Example

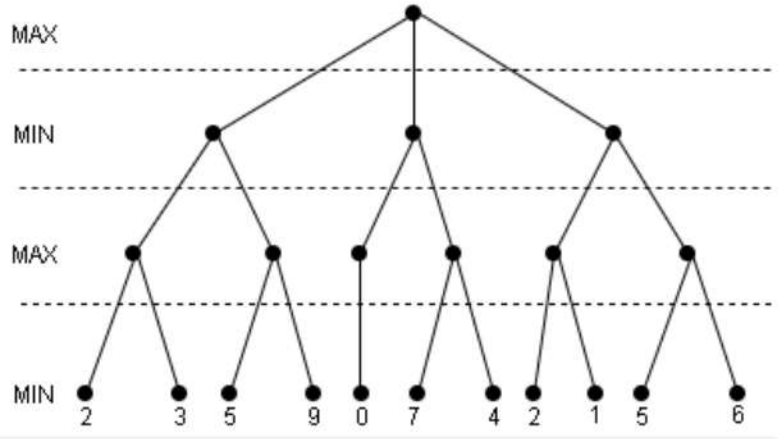
Consider a following state space representing a game:



The pruned branches are shown on the figure below:

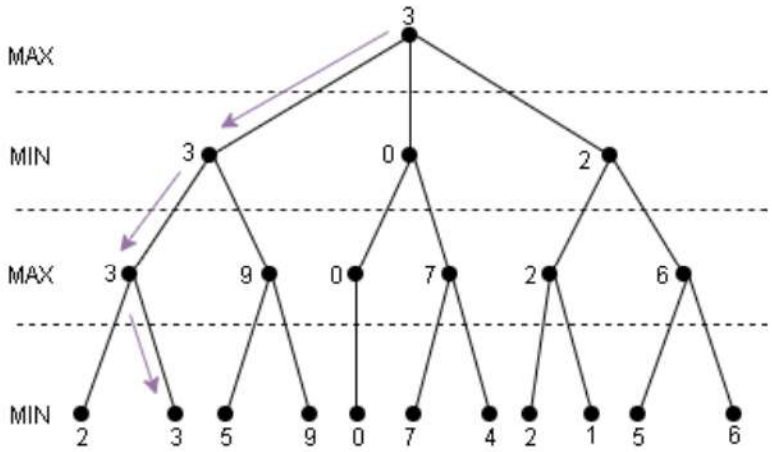


Q. Consider a following state space representing a game. Use minimax search to find solution and perform alpha-beta pruning, if exists.

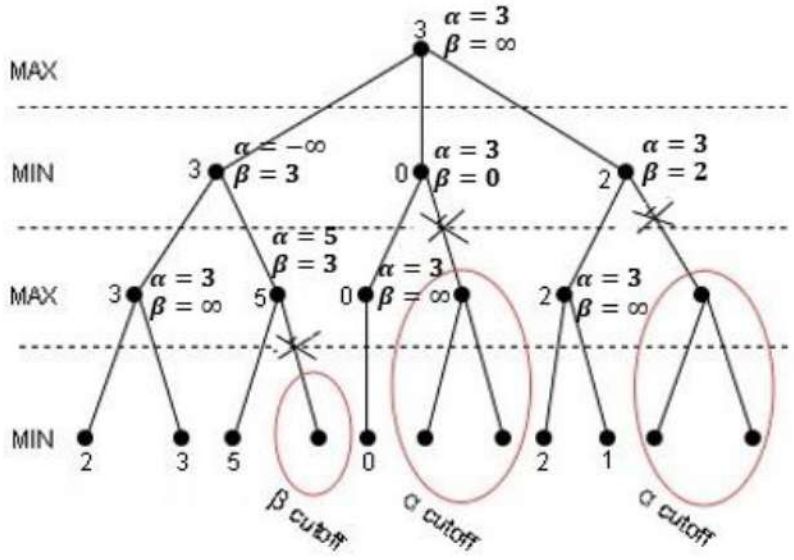


Solution:

The minimax search produce the following tree



The pruned branches are shown on the figure below:



Constraint Satisfaction Problems

A *Constraint Satisfaction Problem (CSP)* is a specific category of problems in AI involves set of variables each of which can take a domain of possible values and set of constraints to be satisfied by the variables. The goal of a CSP problem is to find a suitable value for each variable which satisfies all the constraints.

CSP consists of 3 main components:

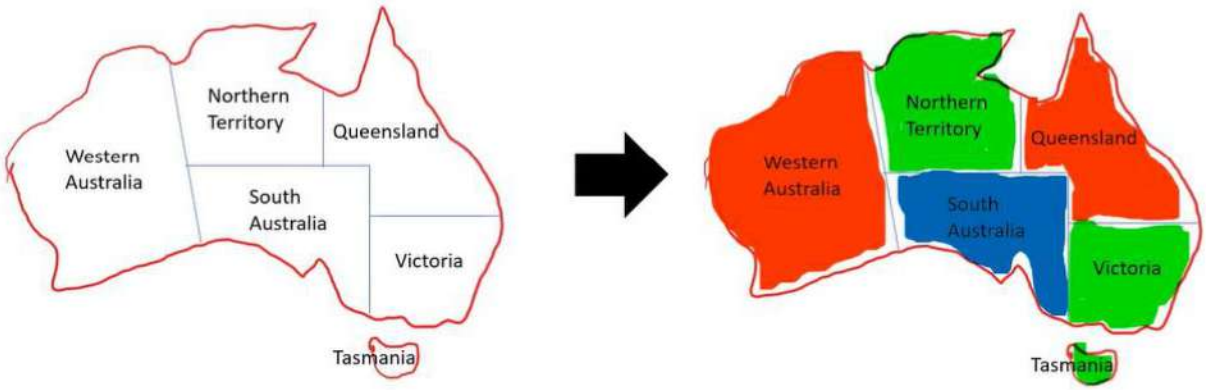
- ***Variables***: Variables are used to represent entities or components of a problem for which values are to be assigned. Set of variables are represented as $X = \{x_1, x_2, \cdots, x_n\}$
- ***Domains:*** Domain specifies the range or set of values a variable can take. Each variable in the CSP is associated with a domain. $D = \{D_1, D_2, \cdots, D_n\}$
- ***Constraints:*** Constraints specify the conditions to be satisfied by the variables.

$$C = \{C_1, C_2, \cdots, C_n\}$$

Examples

1. ***Map Coloring:***

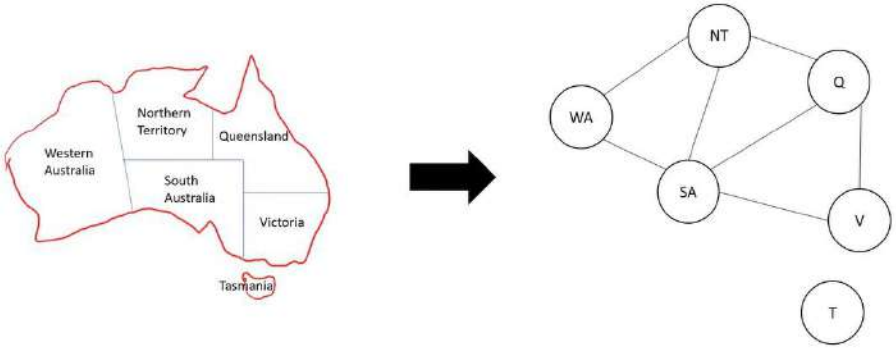
   Map coloring is a problem where given a map and a set of colors, you color the regions of the map such that no adjacent regions have the same color.



   CSP formulation:

   - ***Variables:*** Regions on a map. E.g. {WA, NT, Q, SA, V, T}
   - ***Domain:*** The list of colors. E.g. {red, green, blue}
   - ***Constraints:*** Adjacent regions cannot have the same color. E.g. WA ≠ NT
   - ***Goal:*** Assign colors to each region such that no two neighboring regions have the same color. E.g. {WA = red, NT = green, Q = red, SA = blue, V = green, T = green}

   Constraint Graph: nodes represent the variables and edges represent the constraint.



2. ***N-Queen Problem:***

   N-Queen is a problem where given N-by-N board, you want to place N queens such that no two queens are in the same row, column or diagonal.

   CSP formulation:

   - ***Variables:*** Positions of queens on an $N \times N$ chessboard.
   - ***Domains:*** 1 to N
   - ***Constraints:*** No two queens can be on the same row, column or diagonal.
   - ***Goal:*** Place N queens on the chessboard so that none attack each other.

3. ***Job Scheduling:***

   - ***Variables****:* Jobs to be scheduled.
   - ***Domain****:* Time slots for each job.
   - ***Constraints:***
     1. Some jobs must be completed before others.
     2. No two jobs can use the same resource simultaneously.
     3. Each job has a specific duration.
   - ***Goal:*** Assign jobs to time slots while meeting all constraints.

Constraint Satisfaction Problem solving techniques

Constraint Satisfaction Problems (CSPs) are solved using various algorithms that improve the efficiency of the search process. These algorithms help narrow down the solution space, ensuring that constraints are satisfied while minimizing computational effort. Popular CSP algorithms include backtracking, forward-checking, and constraint propagation, each offering unique strategies for handling constraints effectively.

1. ***Backtracking method:*** It is a systematic search algorithm that explores possible assignments for variables. Its picks a variable, setting a value for it, and then recursively scanning through other variables. In the event of conflict, i.e. when it encounters constraints that cannot be satisfied it backtracks and tries a different value for preceding variable. Backtracking is simple and effective for small problems, but it can be inefficient for large or complex CSPs because it explores every possible solution without any heuristic guidance.
2. ***Forward checking method:*** Forward-checking improves upon backtracking by reducing the search space. After each variable is assigned a value, forward-checking checks the remaining variables to ensure that there are still valid values available for them. If a variable is found to have no valid values left, the algorithm backtracks immediately, avoiding unnecessary exploration of invalid solutions. This technique significantly reduces the number of solutions that need to be explored, making it more efficient than basic backtracking.
3. ***Constraint propagation method:*** Constraint propagation algorithms, such as the ***Arc Consistency Algorithm (AC-3)***, enforce constraints during the search process by ensuring that each variable is consistent with the constraints before proceeding. AC-3 works by iteratively checking the constraints between variables and removing inconsistent values from their domains. This process reduces the search space and eliminates values that cannot lead to valid solutions, making the algorithm more efficient at finding solutions.

Benefits of Constraint Satisfaction Problems (CSPs)

- ***Structured Problem Representation***: Provides a clear framework to represent problems using variables, domains, and constraints, simplifying the problem-solving process.
- ***Manageable Problem-Solving***: Breaks complex problems into smaller components, making them easier to handle and solve.
- ***Flexibility across Domains***: Applicable to diverse areas like scheduling, resource allocation, puzzle-solving, and automated reasoning.
- ***Efficient Search***: Reduces the search space and focuses on promising solutions, making problem-solving more efficient.
- ***Handling Multiple Constraints***: Capable of managing and solving problems with conflicting or multiple simultaneous constraints, essential for real-world applications.

Challenges in Solving Constraint Satisfaction Problems (CSPs)

- ***Scalability Issues***: The problem size grows significantly with the number of variables and constraints, making it difficult to solve efficiently.
- ***Over-Constrained Problems***: In cases where all constraints cannot be satisfied, finding feasible solutions becomes challenging, often requiring techniques like constraint relaxation or optimization.
- ***Computational Complexity***: The exponential growth of the search space with increasing variables and constraints makes it hard to find solutions within a reasonable timeframe.
- ***Efficient Algorithm Design***: Developing effective heuristic approaches and hybrid algorithms is necessary to address scalability and efficiency challenges.
- ***Integration with AI Methods***: Combining CSP techniques with methods such as machine learning and optimization to enhance performance is complex but essential for solving large-scale problems.

Cryptarithmetic Problems

Cryptarithmetic is a class of ***constraint satisfaction problems*** which includes making mathematical relations between meaningful words using simple arithmetic operators like `plus' in a way that the result is conceptually true, and assigning digits to the letters of these words and generating numbers in order to make correct arithmetic operations as well.

Constraints for Cryptarithmetic problems

- A digit (0-9) is assigned to each alphabet.
- Each different alphabet is assigned a unique digit.
- All occurrences of the same alphabet represent the same digit.
- The result should satisfy the predefined arithmetic rule, i.e. 2+2=4
- The problem can be solved from both sides i.e., left-hand side (L.H.S), or right-hand side (R.H.S).

Example

Given a cryptarithmetic problem: ***SEND + MORE = MONEY***

$$\begin{array}{cccccc} C_4 & C_3 & C_2 & C_1 & & \\ & S & E & N & D & \\ + & M & O & R & E & \\ \hline M & O & N & E & Y & \end{array}$$

Carry $(C_i) = \{0, 1\}$

Set of variable (X) = $\{S, E, N, D, M, O, R, Y\}$

Set of domains (D) = $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Constraints equations are:

$D + E = Y + 10C_1$ ---------- (i)
$C_1 + N + R = E + 10C_2$ ---------- (ii)
$C_2 + E + O = N + 10C_3$ ---------- (iii)
$C_3 + S + M = O + 10C_4$ ---------- (iv)
$C_4 = M$ ---------- (v)
$X_i \neq X_j$ (Every variable should have unique value.)

We can easily see that M has to be non-zero digit, so the value of carry C_4 should be 1 and hence $M = 1$.

$$\begin{array}{cccccc} 1 & C_3 & C_2 & C_1 & & \\ & S & E & N & D & \\ + & 1 & O & R & E & \\ \hline 1 & O & N & E & Y & \end{array}$$

Here, to get $C_4 = 1$, $S = 9$ (maximum value).

For C_3, we have two possibilities: $C_3 = 1$ or $C_3 = 0$.

If $C_3 = 1$: $1 + S + 1 = O + 10$
$S = O + 8$
$9 = O + 8$ *(Substituting $S = 9$)*
$O = 1$

If $C_3 = 0$: $S + 1 = O + 10$
$S = O + 9$
$9 = O + 9$ *(Substituting $S = 9$)*
$O = 0$

Here we must have $O = 0$, as 1 is already used by M, so $C_3 = 0$.

$$\begin{array}{cccccc} 1 & 0 & C_2 & C_1 & & \\ & 9 & E & N & D & \\ + & 1 & 0 & R & E & \\ \hline 1 & 0 & N & E & Y & \end{array}$$

For C_2, we have two possibilities: $C_2 = 1$ or $C_2 = 0$.

If $C_2 = 0$, then $E = N$ which is impossible (since all variables must be unique). Therefore, we choose $C_2 = 1$, and thus we have:

$1 + E = N$ ---------- (vi)

From eq. (ii);

$C_1 + N + R = E + 10C_2$
$C_1 + N + R = E + 10$ $[C_2 = 1]$
$C_1 + (1 + E) + R = E + 10$ [from (vi)]
$C_1 + R = 9$

Two Cases for C_1:

If $C_1 = 1$, then $R = 8$.

If $C_1 = 0$, then $R = 9$, which is impossible since 9 is already assigned.

Therefore, $C_1 = 1$, and $R = 8$.

$$\begin{array}{cccc} 1 & 0 & 1 & 1 \\ & 9 & E & N & D \\ + & 1 & 0 & 8 & E \\ \hline 1 & 0 & N & E & Y \end{array}$$

From eq. (i);

$D + E = Y + 10C_1$
$D + E = Y + 10$ $[C_1 = 1]$

To produce $C_1 = 1$, D + E must have carryover. Since Y cannot be 0 or 1 as it already assigned, we need $D + E \geq 12$. Since 9 and 8 are taken for S and R, we can have $5 + 7 = 12$ or $6 + 7 = 13$. So either D = 7 or E = 7.

If $E = 7$, then $E + 1 = N$ so $N = 8$ which is not possible since $R = 8$. So we must have $D = 7$, meaning E is either 5 or 6.

If $E = 6$, then $N = 7$ which is not possible as D = 7. So we must have $E = 5$ and $N = 6$. This means $D + E = 7 + 5 = 12$, and thus $Y = 2$.

$$\begin{array}{cccc} 1 & 0 & 1 & 1 \\ & 9 & 5 & 6 & 7 \\ + & 1 & 0 & 8 & 5 \\ \hline 1 & 0 & 6 & 5 & 2 \end{array}$$

Therefore, the solution to the given Crypt-arithmetic problem is:

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

Q. Solve the following crypto-arithmetic problems, where different letters denote different integers and identical letters denote same integer.

1. LOGIC + LOGIC = PROLOG
2. TWO + TWO = FOUR
3. ODD + ODD = EVEN
4. EAT + THAT = APPLE
5. BASE + BALL = GAMES
6. RIGHT + RIGHT = WRONG
7. LETS + WAVE = LATER