National University of Computer & Emerging Sciences, Peshawar Campus

Computer Science Department
Spring 2025, Lab Manual

Course Code: AI-2002	Course: Artificial Intelligence Lab
Instructor:	**Muhammad Saood Sarwar**

Content :

1. Adversarial Search
2. Elements of game Playing Search
3. Types of Algorithms in Adversarial Search
    A. MinMax Algorithm
    B. Alpha-Beta Pruning(Conditions and working)

4.Constraint Satisfaction Problems

5.Solving Constraint Satisfaction Problem.

Objective:

1. An Introduction to Game theory and Adversarial Search Algorithm.
2. Constraint Satisfaction Problem In AI with multiple examples
3. Tasks

Introduction to Game Theory:

Game Theory is a branch of mathematics used to model the strategic interaction between different players in a context with predefined rules and outcomes. According to game theory, a game is played between two players. To complete the game, one has to win the game, and the other loses automatically.

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. Here, game-playing means discussing those games where human intelligence and logic factor is used, excluding other factors such as luck factor. Tic-tac-toe, chess, checkers, etc., are such types of games where no luck factor works, only the mind works.

Elements of Game Playing search

To play a game, we use a game tree to know all the possible choices and to pick the best one out.

There are the following elements of game-playing:

So: It is the initial state from which a game begins.

PLAYER (s): It defines which player is having the current turn to make a move in the state.

ACTIONS (s): It defines the set of legal moves to be used in a state.

RESULT (s, a): It is a transition model which defines the result of a move.

TERMINAL-TEST (s): It defines that the game has ended and returns true.

UTILITY (s,p): It defines the final value with which the game has ended. This function is also known as the Objective function or Payoff function. The price which the winner will get i.e.

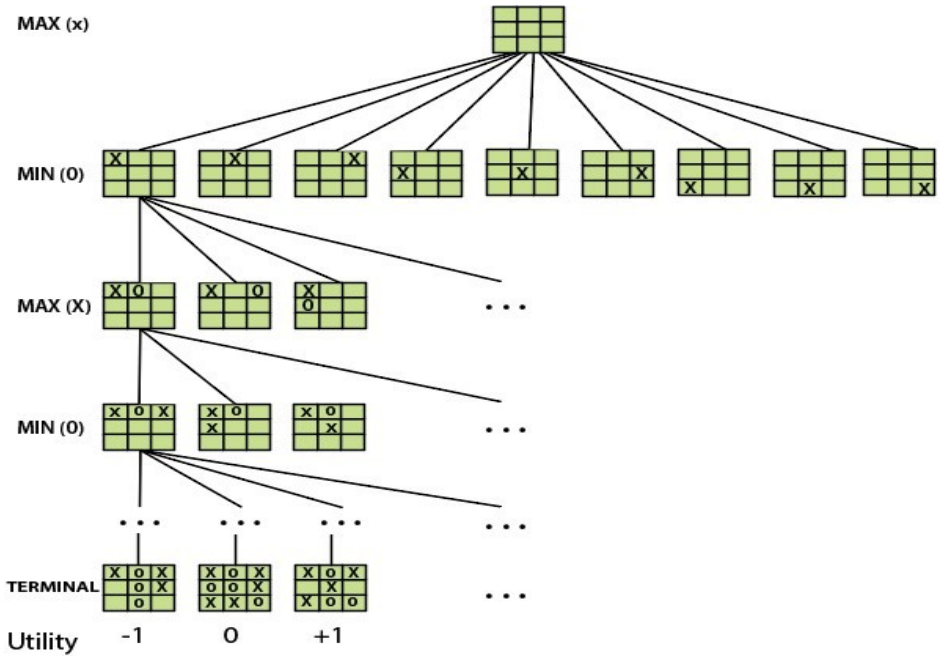
(-1): If the PLAYER loses.

(+1): If the PLAYER wins.

(0): If there is a draw between the PLAYERS.

For example, in chess, tic-tac-toe, we have two or three possible outcomes. Either to win, to lose, or to draw the match with values +1,-1, or 0.

Let’s understand the working of the elements with the help of a game tree designed for tic-tac-toe. Here, the node represents the game state and the edges represent the moves taken by the players.



A game tree for tic-tac-toe

i) **INITIAL STATE (So)**: 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 placing X in the empty square box.

ACTIONS (s): Both the players can make moves in the empty boxes chance by chance.

RESULT (s, a): The moves 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:

Unlike Normal Search, 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 the following types of adversarial search:

i) **Minimax Algorithm**
ii) **Alpha-beta Pruning**

MiniMax Algorithm:

Minimax is a decision-making strategy, which is used to minimize the losing chances in a game and to maximize the winning chances. This strategy is also known as 'Minmax'. It is a two-player game strategy where if one wins, the other loses the game. We can easily understand this strategy via a **game tree** where the nodes represent the states of the game and the edges represent the moves made by the players in the game. Players will be two namely:

- **MIN:** Decrease the chances of **MAX** winning the game.
- **MAX:** Increases his chances of winning the game.

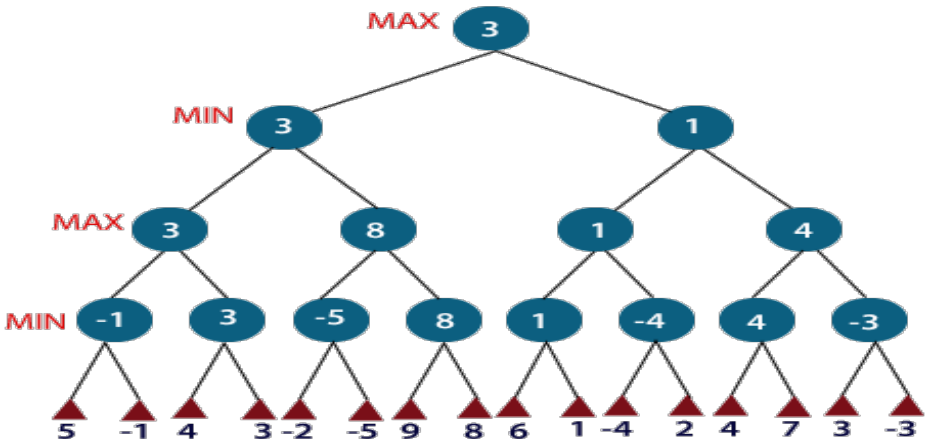
In the minimax strategy, the result of the game or the utility value is generated by a **heuristic function** by propagating from the initial node to the root node. It follows the **backtracking technique** and backtracks to find the best choice. MAX will choose that path that will increase its utility value and MIN will choose the opposite path which could help it to minimize MAX's utility value.

MINIMAX Algorithm:

The MINIMAX strategy follows the **DFS (Depth-first search)** concept. Here, we have two players **MIN and MAX,** and the game is played alternatively between them, i.e., when **MAX** made a move, then the next turn is of **MIN.** It means the move made by MAX is fixed and, he cannot change it. The same concept is followed in DFS strategy, i.e., we follow the same path and cannot change in the middle. That's why in MINIMAX algorithm, instead of BFS, we follow DFS.

- Keep on generating the game tree/ search tree till a limit **d.**
- Compute the move using a heuristic function.

- Propagate the values from the leaf node to the current position following the minimax strategy.
- Make the best move from the choices.



For example, in the above figure, the two players **MAX** and **MIN** are there. **MAX** starts the game by choosing one path and propagating all the nodes of that path. Now, **MAX** will backtrack to the initial node and choose the best path where his utility value will be the maximum. After this, its **MIN** chance. **MIN** will also propagate through a path and again will backtrack, but **MIN** will choose the path which could minimize **MAX** winning chances or the utility value. So, if the level is minimizing, the node will accept the minimum value from the successor nodes. If the level is maximizing, the node will accept the maximum value from the successor.

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

Alpha-Beta Pruning:

Alpha-beta pruning is an advanced version of the MINIMAX algorithm. The drawback of the minimax strategy is that it explores each node in the tree deeply to provide the best path among all the paths. Alpha-beta pruning reduces this drawback of the minimax strategy by less exploring the nodes of the search tree.

The method used in alpha-beta pruning is that it **cut off the search** by exploring less number of nodes. It makes the same moves as a minimax algorithm does, but it prunes the unwanted branches using the pruning technique (discussed in adversarial search). Alpha-beta pruning works on two threshold values, i.e. $-\infty$**(alpha)** and $+\infty$**(beta).**

- $-\infty$: It is the best highest value, a MAX player can have. It is the lower bound, which represents a negative infinity value.
- $^{+}\infty$: It is the best lowest value, a MIN player can have. It is the upper bound which represents positive infinity.

Condition for Alpha-Beta Pruning:

1. α>=β

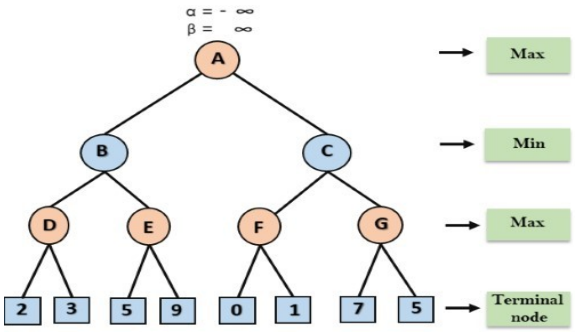
Pseudo-code for Alpha-beta Pruning:

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

Working of Alpha-Beta Pruning:

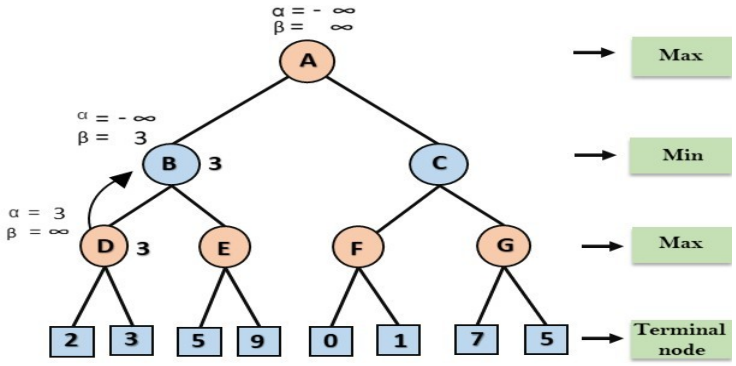
Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

Step 1: At the first step the, Max player will start first move from node A where 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.



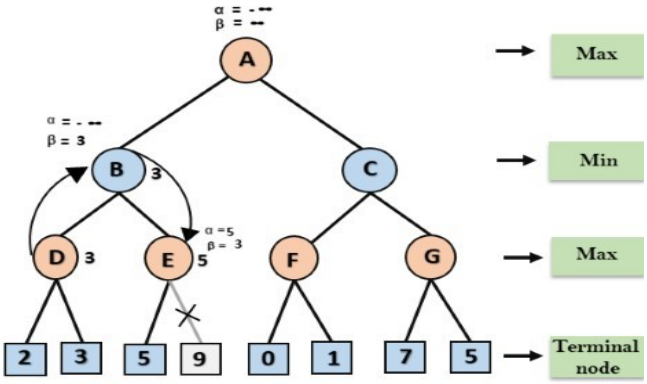
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

Step 3: Now algorithm 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.

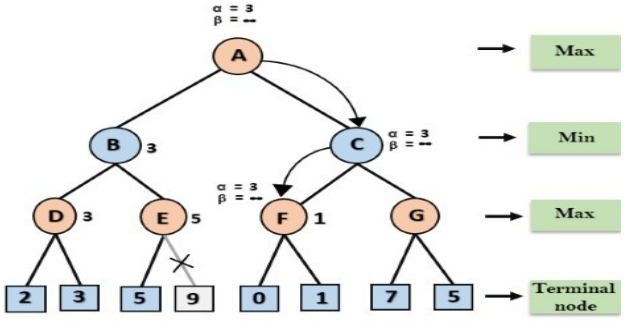
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will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



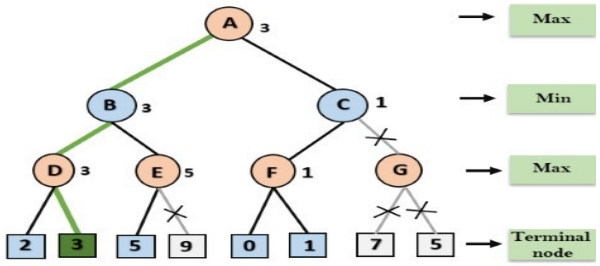
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

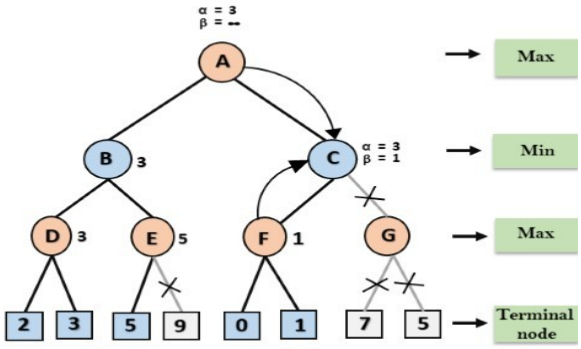
At node C, α=3 and β= +∞, and the same values will be passed on to node F.

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 been computed. Hence the optimal value for the maximizer is 3 for this example.

Constraint Satisfaction Problems:

Constraint satisfaction is a technique where a problem is solved when its values satisfy certain constraints or rules of the problem. Such type of technique leads to a deeper understanding of the problem structure as well as its complexity.
Constraint satisfaction depends on three components, namely:

- X: It is a set of variables.
- D: It is a set of domains where the variables reside. There is a specific domain for each variable.
- C: It is a set of constraints which are followed by a set of variables.

In constraint satisfaction, domains are the spaces where the variables reside, following the problem-specific constraints. These are the three main elements of a constraint satisfaction technique. The constraint value consists of a pair of {scope, rel}. The scope is a tuple of variables that participate in the constraint and rel is a relation that includes a list of values that the variables can take to satisfy the constraints of the problem.

Solving Constraint Satisfaction Problems
The requirements to solve a constraint satisfaction problem (CSP) are as follows:

- A state-space
- notion of the solution.

A state in state-space is defined by assigning values to some or all variables such as

{X1=v1, X2=v2, and so on…}.
An assignment of values to a variable can be done in three ways:

- **Consistent or Legal Assignment:** An assignment which does not violate any constraint or rule is called Consistent or legal assignment.
- **Complete Assignment:** An assignment where every variable is assigned with a value, and the solution to the CSP remains consistent. Such assignment is known as Complete assignment.
- **Partial Assignment:** An assignment which assigns values to some of the variables only. Such type of assignments are called Partial assignments.

Types of Domains in CSP
There are following two types of domains which are used by the variables :

- **Discrete Domain:** It is an infinite domain which can have one state for multiple variables. **For example,** a start state can be allocated infinite times for each variable.
- **Finite Domain:** It is a finite domain which can have continuous states describing one domain for one specific variable. It is also called a continuous domain.

Constraint Types in CSP
With respect to the variables, basically there are following types of constraints:

- **Unary Constraints:** It is the simplest type of constraints that restricts the value of a single variable.
- **Binary Constraints:** It is the constraint type which relates two variables. A value **x2** will contain a value which lies between **x1** and **x3**.
- **Global Constraints:** It is the constraint type which involves an arbitrary number of variables.

Some special types of solution algorithms are used to solve the following types of constraints:

- **Linear Constraints:** These type of constraints are commonly used in linear programming where each variable containing an integer value exists in linear form only.
- **Non-linear Constraints:** These type of constraints are used in non-linear programming where each variable (an integer value) exists in a non-linear form.

Note: A special constraint that works in the real world is known as the **Preference constraint.**
Constraint Propagation
In local state spaces, the choice is only one, i.e., to search for a solution. But in CSP, we have two choices either:

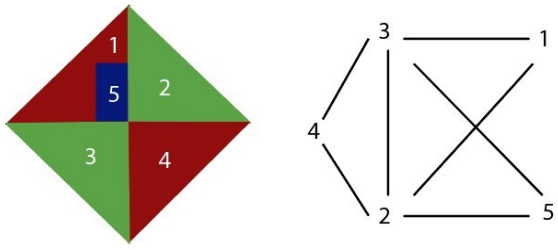
- We can search for a solution or
- We can perform a special type of inference called **constraint propagation**.

Constraint propagation is a special type of inference that helps in reducing the legal number of values for the variables. The idea behind constraint propagation is **local consistency.**
In local consistency, variables are treated as **nodes**, and each binary constraint is treated as an **arc** in the given problem. **There are the following local consistencies which are discussed below:**

- **Node Consistency:** A single variable is said to be node consistent if all the values in the variable's domain satisfy the unary constraints on the variables.
- **Arc Consistency:** A variable is arc consistent if every value in its domain satisfies the binary constraints of the variables.
- **Path Consistency:** When the evaluation of a set of two variables with respect to a third variable can be extended over another variable, satisfying all the binary constraints. It is similar to arc consistency.
- **k-consistency:** This type of consistency is used to define the notion of stronger forms of propagation. Here, we examine the k-consistency of the variables.**CSP Problems**

Constraint satisfaction includes those problems which contain some constraints while solving the problem. CSP includes the following problems:

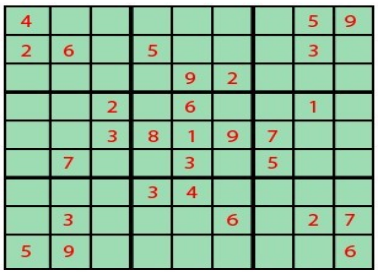
- **Graph Coloring:** The problem where the constraint is that no adjacent sides can have the same color.



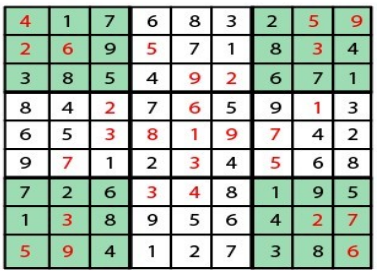
Graph Coloring

- **Sudoku Playing:** The gameplay where the constraint is that no number from 0-9 can be repeated in the same row or column.

SUDOKU



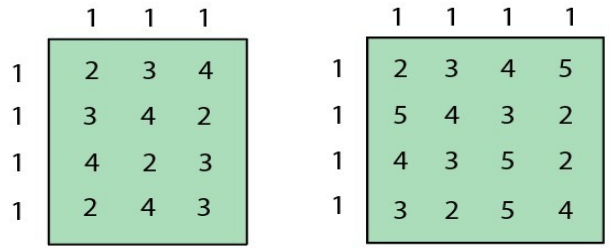
Puzzle



Solution

- **n-queen problem:** In n-queen problem, the constraint is that no queen should be placed either diagonally, in the same row or column.
- **Crossword:** In crossword problem, the constraint is that there should be the correct formation of the words, and it should be meaningful.

- **Latin square Problem:** In this game, the task is to search the pattern which is occurring several times in the game. They may be shuffled but will contain the same digits.



Latin Squence Problem

Cryptarithmetic Problem: This problem has one most important constraint that is, we cannot assign a different digit to the same character. All digits should contain a unique alphabet. Cryptarithmetic Problem is a type of constraint satisfaction problem where the game is about

digits and its unique replacement either with alphabets or other symbols. In a **cryptarithmetic problem,** the digits (0-9) get substituted by some possible alphabets or symbols. The task in the cryptarithmetic problem is to substitute each digit with an alphabet to get the result arithmetically correct.

We can perform all the arithmetic operations on a given cryptarithmetic problem.
The rules or constraints on a cryptarithmetic problem are as follows:

- There should be a unique digit to be replaced with a unique alphabet.
- The result should satisfy the predefined arithmetic rules, i.e., 2+2 =4, nothing else.
- Digits should be from **0-9** only.
- There should be only one carry forward, while performing the addition operation on a problem.
- The problem can be solved from both sides, i.e., **lefthand side (L.H.S), or righthand side (R.H.S)**

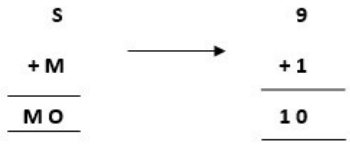
Let's understand the cryptarithmetic problem as well its constraints better with the help of an example:

- Given a cryptarithmetic problem, i.e., **S E N D + M O R E = M O N E Y**

```
  SEND
+MORE
------
 MONEY
------
```

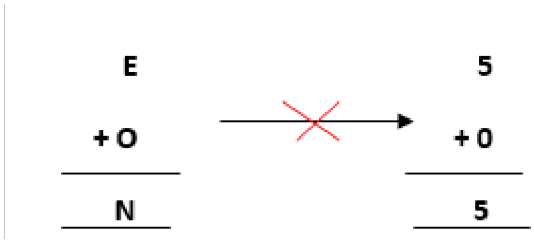
In this example, add both terms **S E N D** and **M O R E** to bring **M O N E Y** as a result.
Follow the below steps to understand the given problem by breaking it into its subparts:

- Starting from the left hand side (L.H.S) , the terms are **S** and **M**. Assign a digit which could give a satisfactory result. Let's assign **S->9** and **M->1**.

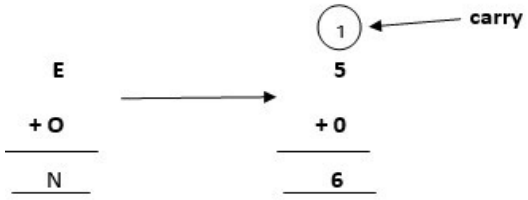


Hence, we get a satisfactory result by adding up the terms and got an assignment for **O** as **O->0** as well.

- Now, move ahead to the next terms **E** and **O** to get **N** as its output.

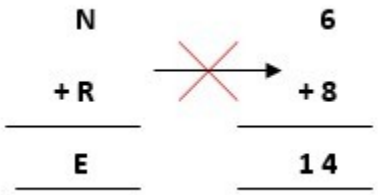


Adding E and O, which means 5+0=0, which is not possible because according to cryptarithmetic constraints, we cannot assign the same digit to two letters. So, we need to think more and assign some other value.



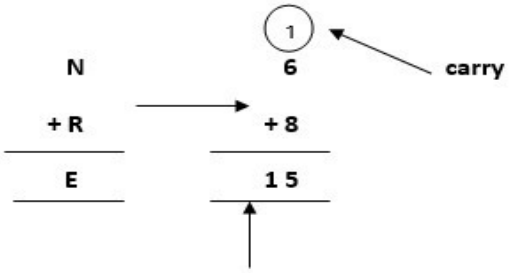
Note: When we will solve further, we will get one carry, so after applying it, the answer will be satisfied.

- Further, adding the next two terms **N** and **R** we get,



But, we have already assigned **E->5**. Thus, the above result does not satisfy the values because we are getting a different value for **E.** So, we need to think more.

Again, after solving the whole problem, we will get a carryover on this term, so our answer will be satisfied.



where 1 will be carry forward to the above term

Let's move ahead.

- Again, on adding the last two terms, i.e., the rightmost terms **D** and **E**, we get **Y** as its result.

where 1 will be carry forward to the above term

- Keeping all the constraints in mind, the final resultant is as follows:

- Below is the representation of the assignment of the digits to the alphabets.

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

More examples of cryptarithmatic problems can be:

1.

```
  BASE
 +BALL
------
 GAMES
```

B	7
A	4
S	8
E	3
L	5
G	1
M	9

2.

```
  YOUR
 +YOU
------
 HEART
```

Y	9
O	4
U	2
R	6
H	1
E	0
A	3
T	8

Similarly, we can also perform multiplication on the cryptarithmetic problems.