INDEX

No.	TITLE	Page No.	Date	Staff Member Signature
1a	Write a program to implement breadth-first search algorithm.		3-8-23	
1b	Write a program to implement depth-first search algorithm		3-8-23	
2	Write a program to simulate 4-Queen/N-Queen Problem		27-9-23	
3	Write a program to simulate the towers of Hanoi problem.		3-8-23	
4	Write a program to implement the Alpha-beta pruning algorithm.		29-9-23	
5	Write a program to solve water jug problem.		15-9-23	
6	Write a program to solve the Missionaries and Cannibals Problem		3-10-23	
7	Write a program to Shuffle a deck of Cards.		8-10-23	
8	Derive the Expressions based on Associative Law.		29-9-23	

INDEX

Code :

```
graph = { 'S' : ['A','B'], 'A' : ['C','D'], 'B' : ['G','H'], 'C' : ['E','F'], 'D' : [], 'G' : ['I'],'H' : [], 'E' : ['K'], 'F' : [], 'I' : [], 'K' : [] }
visited = [] #List of visited nodes.
queue = [] #Initialising the queue.
#function for BFS
def bfs(visited,graph,node):
  visited.append(node)
  queue.append(node)
#Creating loop to visit each node
  while queue:
    m = queue.pop(0)
    print(m,end = " ")

    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)
#Driver code
print("Following is the breadth first search")
bfs(visited,graph,'S')
```

Output :

```
Following is the breadth first search
S A B C D G H E F I K
```

Practical No 1

Write a program to implement breadth first search algorithm

Breadth First Search (BFS) is an essential algorithm used in AI to systematically traverse and explore all the nodes in a graph or a tree data structure. It operates in a breadth-first manner, meaning it explores all the nodes at a given level before moving to the next level. This approach makes BFS particularly suitable for searching for the shortest path or finding solutions to problems where all possible states need to be examined. BFS uses a queue data structure. In BFS, nodes are explored in a First In, First Out (FIFO) manner, which is a characteristic of a queue. This means that the nodes that were added to the queue are explored first.

Algorithm

The BFS algorithm starts from an initial node, often referred to as the root node and proceeds as follows:

1. Initialize a queue data structure
2. Place the initial/root node in the queue and mark it as visited.
3. If the queue is empty return failure and stop
4. If the first element in the queue is a goal node, return success and stop
5. Else Remove the first element (node) from the queue and place its children at the end of the queue
6. Go to step (3)

Code:

```
graph = {'3':['5','8','25'],'5':['1','2'],'8':[],'25':['12','8'],'1':[],'2':[],'12':['6'],'6':['4','9'],'4':[],'9':[]}

visited = set() # set to keep track of visited nodes.
def dfs(visited, graph, node):
  if node not in visited:
    print(node)
    visited.add(node)
    for neighbour in graph[node]:
      dfs(visited, graph, neighbour)

# driver code
dfs(visited, graph, '3')
```

Output:

```
3
5
1
2
8
25
12
6
4
9
```

1b Write a program to implement depth first search algorithm

Depth First Search (DFS) is a ~~fundamental~~ recursive graph traversal algorithm widely used in Artificial Intelligence, to systematically traverse and explore the nodes in a graph or tree datastructuree. DFS algorithm uses a stack to keep track of the nodes that need to be explored and backtracked when necessary. In DFS, nodes are explored in Last In First Out (LIFO) manner, which means that the most recently added node is the first to be explored. DFS operates in a depth-first manner, meaning it explores as deeply as possible along one branch before backtracking. This unique characteristic makes DFS suitable for scenarios where deep exploration is beneficial.

Algorithm

1) Push the root node on the stack and mark it as visited.
2) Do the following until stack is not empty
   a. Pop the first node from the stack
      i. If the popped node is the goal node then stop and return success.
      ii. Else push all of its child nodes onto the stack.

Code:

```
print("Enter the number of queens")
N = int(input())
# chessboard
# NxN matrix with all elements 0
board = [[0] * N for _ in range(N)]

def is_attack(i, j):
    # check if there is a queen in row or column
    for k in range(0, N):
        if board[i][k] == 1 or board[k][j] == 1:
            return True
    # checking diagonals
    for k in range(0, N):
        for l in range(0, N):
            if (k + l == i + j) or (k - l == i - j):
                if board[k][l] == 1:
                    return True
    return False

def N_queen(n):
    # if n is 0, solution found
    if n == 0:
        return True

    for i in range(0, N):
        for j in range(0, N):
            if (not is_attack(i, j)) and (board[i][j] != 1):
                board[i][j] = 1
                if N_queen(n - 1):
                    return True
                board[i][j] = 0
    return False
```

```
if N_queen(N):
    for i in board:
        print(i)
else:
    print("Solution does not exist.")
```

Output:

```
Enter the number of queens
4
[0, 1, 0, 0]
[0, 0, 0, 1]
[1, 0, 0, 0]
[0, 0, 1, 0]
```

Practical No. 2

Write a program to simulate the 4-Queen / N-Queen Problem

The N-Queens problem, is a classic puzzle in Artificial Intelligence. It involves the task of placing N queens on an N×N chessboard in such a way that no two queens threaten each other.
The problem space for the N-Queens problem is defined by the N x N Chessboard. Each square on the board represents a potential position for a queen. The placement of the Queen on the board depends upon certain constraints.

The primary constraints within this space are :-
1) No two queens can share the same row
2) No two queens can occupy the same coloumn
3) No two queens can be placed in positions that form a diagonal line.

The objective is to find a configuration of queens that satisfies all these constraints, and this search is conducted within the defined problem space. The size of the chessboard, denoted by N, determines the complexity of the problem space.

Algorithm

1) Start in the leftmost column
2) If all queens are placed return True
3) Visit all rows in the current column. Do the following for every row:
   i) if the queen can be placed safely in this row
      a) Then mark this [row, column] as part of the solution and recursively check if placing queen here leads to the solution.
      b) If placing the queen in [row, column] leads to the solution then return True.
      c) If placing queen doesn't lead to a solution, then un-mark this [row, column] then backtrack and try other rows.
   ii) If all rows have been tried and valid solution is not found then return false to trigger backtracking

Code:

```
def TowerofHanoi(n,s_pole,d_pole,i_pole):
    if n==1:
        print("Move Disc 1 From Pole",s_pole,"to pole",d_pole)
        return
    TowerofHanoi(n-1,s_pole,i_pole,d_pole)
    print("Move Disc",n,"from pole",s_pole,"to pole",d_pole)
    TowerofHanoi(n-1,i_pole,d_pole,s_pole)

n=3
TowerofHanoi(n,'A','C','B')
```

Output:

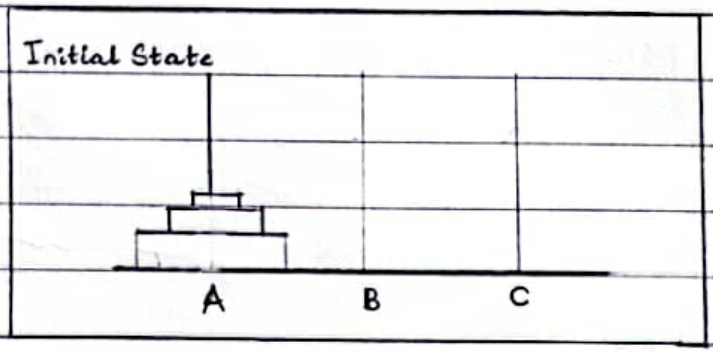
```
Move Disc 1 From Pole A to pole C
Move Disc 2 from pole A to pole B
Move Disc 1 From Pole C to pole B
Move Disc 3 from pole A to pole C
Move Disc 1 From Pole B to pole A
Move Disc 2 from pole B to pole C
Move Disc 1 From Pole A to pole C
```

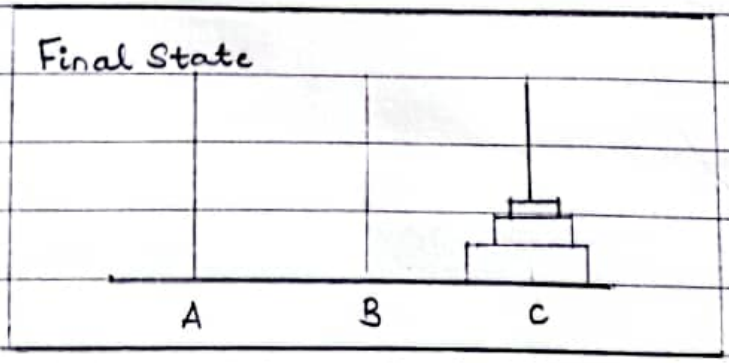
Practical No.3

Write a program to solve tower of Hanoi problem.

Tower of Hanoi is a mathematical puzzle which consists of three rods (A,B,C) and N number of disks of different sizes which can slide on to any rod. Initially, all the disks are stacked in decreasing value of diameter i.e the smaller disks are placed on top of the larger ones. The objective of this puzzle is to move the entire stack to another rod (here considered C), obeying the following simple rules:

1. Only one disk can be moved at a time
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e a disk can only be moved if it is the uppermost disk on a stack
3. No large disk may be placed on top of a smaller disk.





Code :

```
MAX, MIN = 1000, -1000
def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
  if depth == 3:
    return values[nodeIndex]
  if maximizingPlayer:
    best = MIN
    # Recur for left & right children
    for i in range(0, 2):
      val = minimax(depth + 1, nodeIndex * 2 + i, False, values, alpha, beta)
      best = max(best, val)
      alpha = max(alpha, best)
      # Alpha-Beta Pruning
      if beta <= alpha:
        break
    return best
  else:
    best = MAX
    # Recur for left and right children
    for i in range(0, 2):
      val = minimax(depth + 1, nodeIndex * 2 + i, True, values, alpha, beta)
      best = min(best, val)
      beta = min(beta, best)
      # Alpha-Beta Pruning
      if beta <= alpha:
        break
    return best

# Driver Code
if __name__ == "__main__":
  values = [3, 5, 6, 9, 1, 2, 0, -1]
  print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

Output:

```
The optimal value is : 5
```

Practical No. 4

Write a program to implement the alpha beta pruning algorithm.

Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm. It is primarily used in two-player, zero-sum games. The goal is to significantly reduce the number of nodes evaluated in the search tree without affecting the final decision. In game playing scenarios, the Minimax algorithm is employed to find the optimal move by exhaustively evaluating all possible moves in a search tree. However, this exhaustive search can become computationally expensive as the game tree grows. Alpha-Beta Pruning addresses this issue by discarding branches of the tree that cannot influence the final decision, leading to substantial performance improvements. This is known as Pruning. This involves two threshold parameters:

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 $-\infty$

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 $+\infty$

Condition for Alpha-beta Pruning

$$\alpha >= \beta$$

Code:

```
capacity = (12, 8, 5)
# Maximum capacities of 3 jugs -->x,y,z
x = capacity[0]
y = capacity[1]
z = capacity[2]
# to mark visited states
memory = {}
# store solution path
ans = []

def get_all_states(state):
    # Let the 3 jugs be called a,b,c
    a = state[0]
    b = state[1]
    c = state[2]

    if (a == 6 and b == 6):
        ans.append(state)
        return True
    # if current state is already visited earlier
    if ((a, b, c) in memory):
        return False
    memory[(a,b,c)]=1
    #empty jug a
    if a>0:
        #empty a into b
        if(a+b<=y):
            if(get_all_states((0,a+b,c))):
                ans.append(state)
                return True
        else:
            if (get_all_states((a-(y-b),y,c))):
                ans.append(state)
        #empty a into c
        if (a+c<=z):
            if(get_all_states((0,b,a+c))):
                ans.append(state)
                return True
        else:
            if (get_all_states((a-(z-c),b,z))):
                ans.append(state)
                return True

    #empty jug b
    if (b>0):
        #empty b into a
        if (a + b <= x):
            if (get_all_states((a + b, 0, c))):
                ans.append(state)
                return True
        else:
            if (get_all_states((x, b - (x - a), c))):
                ans.append(state)
                return True
        #empty b into c
        if(b+c<=z):
            if(get_all_states((a,0,b+c))):
                ans.append(state)
                return True
        else:
            if(get_all_states((a,b-(z-c),z))):
                ans.append(state)
                return True

    #empty jug c
    if(c>0):
        #empty c into a
        if(a+c<=x):
            if(get_all_states((a+c,b,0))):
                ans.append(state)
                return True
        else:
            if(get_all_states((x,b,c-(x-a)))):
                ans.append(state)
                return True
        #empty c into b
        if (b + c <= y):
            if (get_all_states((a, b + c, 0))):
                ans.append(state)
                return True
        else:
            if (get_all_states((a, y, c - (y - b)
                ans.append(state)
                return True
    return False

initial_state=(12,0,0)
print("starting work...\n")
get_all_states(initial_state)
ans.reverse()
for i in ans:
    print(i)
```

Output:

```
starting work...

(12, 0, 0)
(4, 8, 0)
(0, 8, 4)
(8, 0, 4)
(8, 4, 0)
(3, 4, 5)
(3, 8, 1)
(11, 0, 1)
(11, 1, 0)
(6, 1, 5)
(6, 6, 0)
```

Practical No 5

Write a program to solve water-jug problem

The water jug problem is a classic puzzle/problem in artificial intelligence. In this problem you are given an 'm' liter jug and a 'n' litre jug. Both the Jugs are initially empty. The jugs don't have markings to allow measuring smaller quantities of water. You have to use the jugs to measure 'd' litres of water where $d < n$. (X, Y) corresponds to a state where X refers to the amount of water in Jug 1 and Y refers to the amount of water in Jug 2. Determine the path from e initial state(xi, yi) to the final state (xf, yf), where (xi, yi) is (0, 0) indicating both Jugs are initially empty and (xf, yf) indicates a state which could be (0, d) or (d, 0)

The operations you can perform are:

1) Empty a jug (x, 0) → (0, 0) Empty Jug 1
2) Fill a Jug (0, 0) → (x, 0) Fill Jug 1
3) Pour water from one jug to another until one of the jugs is either empty or full (x, Y) → ((x-d), (Y+d))

Code

```
from collections import deque

initial_state = (3, 3, 1)
goal_state = (0, 0, 0)

def is_valid(state):
    missionaries_left, cannibals_left, boat_location = state

    if (
        missionaries_left < 0
        or cannibals_left < 0
        or missionaries_left > 3
        or cannibals_left > 3
    ):
        return False

    if (
        (missionaries_left < cannibals_left and missionaries_left > 0)
        or (3 - missionaries_left < 3 - cannibals_left and 3 - missionaries_left > 0)
    ):
        return False

    return True

def get_next_states(state):
    states = []
    missionaries_left, cannibals_left, boat_location = state

    passengers = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]

    for m, c in passengers:
        new_missionaries_left = missionaries_left - m * boat_location
        new_cannibals_left = cannibals_left - c * boat_location
        new_boat_location = 1 - boat_location

        new_state = (new_missionaries_left, new_cannibals_left, new_boat_location)

        if is_valid(new_state):
            states.append(new_state)

    return states

def solve():
    visited = set()
    queue = deque([(initial_state, [])])

    while queue:
        current_state, path = queue.popleft()

        if current_state == goal_state:
            return path + [current_state]

        visited.add(current_state)
        next_states = get_next_states(current_state)

        for next_state in next_states:
            if next_state not in visited:
                queue.append((next_state, path + [current_state]))

    return None

solution = solve()

if solution:
    print("Solution:")
    for i, state in enumerate(solution):
        print(f"Step {i + 1}: {state}")
else:
    print("No solution found")
```

Output:

```
Solution:
Step 1: (3, 3, 1)
Step 2: (3, 1, 0)
Step 3: (3, 1, 1)
Step 4: (1, 1, 0)
Step 5: (1, 1, 1)
Step 6: (0, 0, 0)
```

Practical No 6

Write a program to solve the Missionaries and Cannibals Problem.

The Missionaries and Cannibals problem is a classic puzzle in the field of Artificial Intelligence. The problem involves three missionaries and 3 cannibals which are on one side of a river. They have access to a boat that can carry atmost two people at a time. The goal is to get everyone to the other side of the river. However there are constraints:

1.) If cannibals outnumber missionaries on either side of the river, the cannibals will eat the missionaries

2.) The boat cannot cross the river by itself with no people on board.

The problem is to find a sequence of crossings that successfully gets all the missionaries and cannibals to the other side of the river without violating these constraints

Code:

```
import random

ranks =['2','3','4','5','6','7','8','9','10','jack','queen','king','ace']
suits=['hearts','diamonds','clubs','spades']
deck=[{'rank':rank,'suit':suit} for rank in ranks for suit in suits]

def shuffle_deck(deck):
    random.shuffle(deck)

def display_deck(deck):
    for card in deck:
        print(f"{card['rank']} of {card['suit']}")

shuffle_deck(deck)
print("shuffles deck:")
display_deck(deck)
```

Output:

```
shuffles deck:
2 of hearts
7 of diamonds
6 of hearts
10 of clubs
10 of hearts
8 of hearts
queen of diamonds
9 of diamonds
ace of diamonds
10 of diamonds
6 of spades
8 of spades
9 of hearts
4 of clubs
king of diamonds
8 of diamonds
ace of spades
king of clubs
7 of spades
jack of clubs
5 of spades
8 of clubs
4 of hearts
king of hearts
queen of spades
2 of clubs
3 of diamonds
10 of spades
3 of hearts
jack of diamonds
3 of spades
9 of spades
jack of hearts
3 of clubs
5 of hearts
7 of hearts
jack of spades
6 of clubs
```

```
queen of hearts
5 of clubs
ace of hearts
4 of diamonds
4 of spades
ace of clubs
9 of clubs
2 of spades
queen of clubs
2 of diamonds
5 of diamonds
6 of diamonds
7 of clubs
king of spades
```

Practical No 7

Write a Program to Shuffle a deck of cards.

Shuffling is a procedure used to randomize a deck of playing cards to provide an element of chance in card games. Shuffling is often followed by a cut, to help ensure that the shuffler has not manipulated the outcome. The purpose of shuffling is to mix the cards thoroughly, preventing any predictability in the order of the cards and ensuring a fair distribution during gameplay.

Code:

associative.pl

```
associative_addition(A,B,C,Result):- Result is A+(B+C).
associative_multiplication(A,B,C,Result):-Result is A*(B*C).
```

Output:

```
% c:/Users/chimb_sfqb37j/Desktop/AI Pracs/associa
?-
|    associative_addition(2,3,4,Result).
Result = 9.

?- associative_multiplication(2,3,4,Result).
Result = 24.
```

Practical No. 8

Derive Expressions based on Associative Law.

Associative Law

In Artificial Intelligence, the associative law stands as a foundational principle, originating from the mathematical concepts of grouping and order of operations. This law states that the grouping of elements in a computational operation does not impact the final result. On the basis of this law, if there are three x, y and z, then the following relation consists between these numbers.

$A + (B + C) = (A + B) + C$

$A * (B * C) = (A * B) * C$

Code :

associative.pl | distributive.pl

```
distributive_multiplication_over_add(A,B,C,Result):-Result is A*(B+C).

distributive_add_over_multiplication(A,B,C,Result):-Result is A+(B*C).
```

Output:

```
% c:/users/chimb_sfqb37j/desktop/ai pracs/distributive compi
?-
|    distributive_multiplication_over_add(3,4,5,Result).
Result = 27.

?- distributive_add_over_multiplication(3,4,5,Result).
Result = 23.
```

Practical No. 9

Derive the Expressions based on Distributive Law

Distributive Law

Distributive Law, also called distributive property in mathematics, the law relating the operations of multiplication and addition, is stated symbolically as:

$$a(b+c) = ab + ac$$

The monomial factor a is distributed, or seperately applied, to each term of the binomial factor b+c, resulting in the product ab + ac. From this law it is easy to show that the result of first adding several numbers and then multiplying the sum by some number is the same as first multiplying each seperately by the number and then adding the products.

Code:

```
% Define facts
batsman(sachin).
cricketer(batsman).

%rules
cricketer(X):- batsman(X).
```

Output:

```
% c:/users/chimb_sfqb37j/downloads/cr
?-
|    cricketer(sachin).
true.

?- cricketer(virat).
false.
```

Practical No. 10

a Write a Program to derive the predicate. (for eg: Sachin is batsman, batsman is cricketer) → Sachin is Cricketer.

Prolog

Prolog is a logic programming language. In prolog logic is expressed as relations (called as Facts and Rules). Facts constitute the knowledge Base of the system. We can query against the knowledge base to find solution or derive a conclusion
Facts contain predicates followed by an argument.
Rules consist of head (conclusion) on the left hand side and body (conditions) on the right hand side
If the condition is satisfied you can infer the conclusion
Eg:- ancestor (X, Y) :- parent (X, Y)

Code :

```
%Defining Facts
female(pam).
female(liz).
female(pat).
female(ann).

male(tom).
male(bob).
male(tom).
male(bob).
male(peter).

parent(pam, bob).
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).
parent(tom,peter).

%Defining Rules
mother(X,Y) :- parent(X,Y), female(X).
father(X,Y) :- parent(X,Y), male(X).

sister(X,Y) :- parent(Z,X), parent(Z,Y), female(X), X \== Y.
brother(X,Y) :- parent(Z,X), parent(Z,Y), male(X) , X\==Y.

grandmother(X,Z) :- mother(X,Y), parent(Y,Z).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).

uncle(X,Z) :- brother(X,Y), parent(Y,Z).
aunt(X, Z) :- sister(X, Y), parent(Y, Z).
```

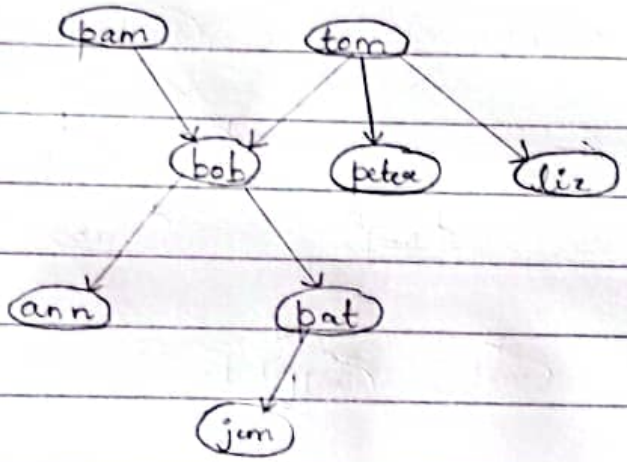
Output :

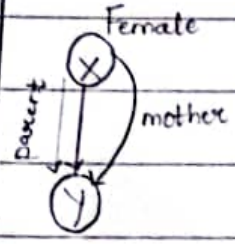
```
% c:/users/chimb_sfqb37j/document
?-
|    mother(pam,bob).
true.

?- father(tom,peter).
true .

?- sister(ann,pat).
true.

?- brother(bob,liz).
true .

?- grandmother(pam,ann).
true.

?- grandfather(tom,ann).
true .

?- aunt(pat,liz).
false.

?- aunt(liz,pat).
true .

?- uncle(peter,ann).
true .

?-
```

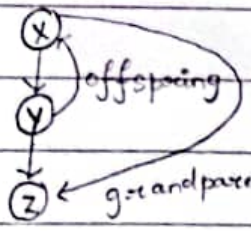
b. Write a program which contains three predicates: male, female, parent. Make rules for the following family relations: father, mother, grandfather, grandmother, brother, sister, uncle, aunt, nephew and niece, cousin.

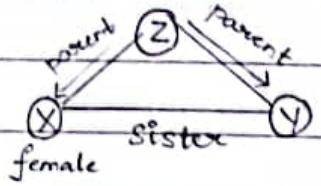
Question: i.) Draw Family Tree

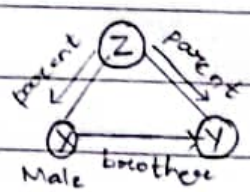
ii.) Define Clauses, Facts, Predicates and Rules with conjunction and disjunction.











Code:

```
class BlockWorld:
    def __init__(self):
        self.state = ['A', 'B', 'C'] #Initial state: Three blocks on table

    def move(self, source, destination):

        if source == destination or source not in self.state:
            return False #Invalid move

        block = self.state.pop(self.state.index(source))  #Remove block from source

        self.state.append(block)  #place the block on destination
        return True

    def solve(self):
        #Move A to C
        self.move('A','C')

        #Move A to B
        self.move('A','B')

        #Move C to B
        self.move('C','B')

        #Move A to C
        self.move('A','C')

        #Move B to A
        self.move('B','A')

        #Move B to C
        self.move('B','C')
```

```
# Create a BlockWorld instance
block_world = BlockWorld()

#Solve the problem
final_state = block_world.solve()

print("Final state:", final_state)
```

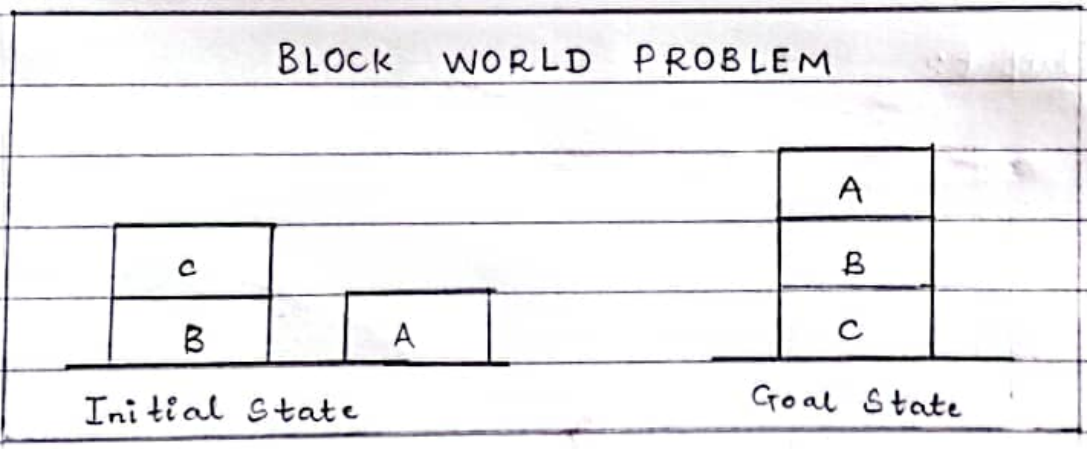
Output:

```
Final state: ['C', 'B', 'A']
```

Practical No. 11

Write a Program to Solve the Block World Problem

The Block World is a classic example used in the field of Artificial Intelligence and planning. It involves manipulating blocks in a world with specific rules and goals. The task is to bring the system from initial state to final state. The Block World Problem involves a set of Blocks placed on a flat surface. The blocks can be stacked on one another, and the goal is to reach a specified arrangement through a sequence of actions. Actions typically include moving a block from one position to another, stacking blocks and unstacking blocks.



Code :

```
from simpleai.search import SearchProblem, astar

GOAL = 'HELLO WORLD'

class HelloProblem(SearchProblem):
    def actions(self, state):
        if len(state) < len(GOAL):
            return list(' ABCDEFGHIJKLMNOPQRSTUVWXYZ')
        else:
            return []

    def result(self, state, action):
        return state + action
    def is_goal(self, state):
        return state == GOAL

    def heuristic(self, state):
        wrong = sum([1 if state[i] != GOAL[i] else 0 for i in range(len(state))])
        missing = len(GOAL) - len(state)
        return wrong + missing
problem = HelloProblem(initial_state='')
result = astar(problem)
print(result.state)
print(result.path())
```

Output:

```
HELLO WORLD
[(None, ''), ('H', 'H'), ('E', 'HE'), ('L', 'HEL'), ('L', 'HELL'), ('O', 'HELLO'), (' ', 'HELLO '),
('W', 'HELLO W'), ('O', 'HELLO WO'), ('R', 'HELLO WOR'), ('L', 'HELLO WORL'), ('D', 'HELLO WORLD')]
```

Practical No. 12

Write a Program to implement A* Search

A* Search is one of the best and popular technique used in path-findings and graph traaversals. A* combines the elements of uniform-cost search and greedy best-first search, employing a heuristic function to guide its exploration efficiently. The A* search algorithm operates by maintaining a priority queue of states to explore, prioritized based on the combination of the cost incurred so far and heuristic estimate of the remaining cost to reach the goal. The key formula is:

$$f_n = g(n) + h(n),$$

where f(n) is the total estimated cost, g(n) is the cost of the path from start node to node n, and h(n) is the heuristic estimate from node n to the goal. The algorithm iteratively selects and expands the state with the lowest f(n) value until the goal state is reached.

A* search maintains two sets of nodes - the open set, representing nodes to be considered for exploration, and the closed set, containing nodes already evaluated. This ensures that the algorithm explores each state only once and prevents redundant computations.