SAHYADRI
COLLEGE OF ENGINEERING & MANAGEMENT
An Autonomous Institution
MANGALURU

LABORATORY COMPONENTS			
Exp. No.	**Experiment Description**	**CO No.**	**Bloom's Taxonomy Level**
1	Design and implement Tic-Tac-Toe game using Python programming.	CO1	CL3
2	Demonstrate Nim game using Python programming.	CO2	CL3
3	Write a Pyhton Pprogram to implement A* Algorithm.	CO2	CL3
4	Write a python program to demonstrate the working of Alpha-Beta Pruning.	CO2	CL3
5	Demonstrate the Union and Intersection of two fuzzy Sets using python programming.	CO3	CL3
6	Write a program in Prolog to implement simple arithmetic.	CO4	CL3
7	Design and implement a Cross word puzzle using Python programming.	CO4	CL3
8	Demonstrate a simple Chatbot with minimum of 10 conversations.	CO5	CL3

PROGRAMS

1. Design and implement Tic-Tac-Toe game using Python programming

```
import random


def print_board(board):
    print("  1 2 3")
    for i in range(3):
        print(f"{i+1} {' '.join(board[i])}")


def check_winner(board, player):
    # Check for row wins
    for row in board:
        if all(cell == player for cell in row):
```

```
            return True
    # Check for column wins
    for col in range(3):
        if all(board[row][col] == player for row in range(3)):
            return True
    # Check for diagonal wins
    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)):
        return True
    return False


def is_board_full(board):
    return all(cell != ' ' for row in board for cell in row)


def get_available_moves(board):
    return [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']


def player_move(board):
    while True:
        try:
            row, col = map(int, input("Enter your move (row and column, e.g., 1 2): ").split())
            if 1 <= row <= 3 and 1 <= col <= 3 and board[row - 1][col - 1] == ' ':
                return row - 1, col - 1
            else:
                print("Invalid move. Try again.")
        except ValueError:
            print("Invalid input. Please enter two integers separated by a space.")


def computer_move(board, computer, player):
    available_moves = get_available_moves(board)

    # Try to win
```

```
    for move in available_moves:
        board[move[0]][move[1]] = computer
        if check_winner(board, computer):
            return move
        board[move[0]][move[1]] = ' '  # Undo move

    # Block player from winning
    for move in available_moves:
        board[move[0]][move[1]] = player
        if check_winner(board, player):
            board[move[0]][move[1]] = computer
            return move
        board[move[0]][move[1]] = ' '  # Undo move

    # Otherwise, make a random move
    return random.choice(available_moves)

def play_game():
    #Creating a 2D array board with string value ' ' in it.
    board = [[' ' for i in range(3)] for j in range(3)]

    #board = [[  ,  ,  ]
    #         [  ,  ,  ]
    #         [  ,  ,  ]]

    player = 'X'
    computer = 'O'

    while True:
        print_board(board) #initial board
```

```
        if check_winner(board, player):
            print("Congratulations! You win!")
            break
        elif check_winner(board, computer):
            print("Computer wins!")
            break
        elif is_board_full(board):
            print("It's a tie!")
            break

        if player == 'X':
            row, col = player_move(board)
            board[row][col] = 'X'
            player = 'O'
            computer = 'X'
        else:
            row, col = computer_move(board, computer, player)
            board[row][col] = 'O'
            player = 'X'
            computer = 'O'

if __name__ == "__main__":
    play_game()
```

2.Demonstrate Nim game using Python programming.

```
def print_board(heap):
    print(f"Current heap: {'|' * heap}")

def get_user_move(heap):
    while True:
```

```
        try:
            sticks_to_remove = int(input(f"Enter the number of sticks to remove (minimum 1, maximum {min(heap, heap // 2)}): "))
            if 1 <= sticks_to_remove <= min(heap, heap // 2):
                break
            else:
                print(f"Invalid number of sticks. Please enter a number between 1 and {min(heap, heap // 2)}.")
        except ValueError:
            print("Invalid input. Please enter a valid number.")

    return sticks_to_remove


def get_computer_move(heap):
    xor_sum = heap
    for i in range(heap):
        xor_sum ^= i

    if xor_sum == 0:
        max_sticks = min(heap, heap // 2)
        sticks_to_remove = random.randint(1, max_sticks)
    else:
        max_sticks = min(heap // 2, heap)
        sticks_to_remove = max(1, min(max_sticks, heap - xor_sum))

    return sticks_to_remove


def nim_game():
    heap = 16
    player_turn = 1

    while heap > 1:
```

```
        print_board(heap)

        if player_turn == 1:
            player_name = "Player 1"
            sticks_to_remove = get_user_move(heap)
        else:
            player_name = "Computer"
            sticks_to_remove = get_computer_move(heap)

        heap -= sticks_to_remove
        print(f"{player_name} removes {sticks_to_remove} sticks.")

        player_turn = 3 - player_turn  # Switch player (1 -> 2, 2 -> 1)

    print_board(heap)
    winner = "Player 1" if player_turn == 2 else "Computer"
    print(f"Player {player_turn} picks the last stick ")
    print(f"\n{winner} is the winner!")

if __name__ == "__main__":
    import random
    nim_game()
```

3.Write a Pyhton Pprogram to implement A* Algorithm

```
def aStarAlgo(start_node, stop_node):

    open_set = set(start_node)
    closed_set = set()
    g = {} #store distance from starting node
    parents = {}# parents contains an adjacency map of all nodes

    #ditance of starting node from itself is zero
    g[start_node] = 0
    #start_node is root node i.e it has no parent nodes
    #so start_node is set to its own parent node
    parents[start_node] = start_node
```

```
while len(open_set) > 0:
    n = None

    #node with lowest f() is found
    for v in open_set:
        if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
            n = v


    if n == stop_node or Graph_nodes[n] == None:
        pass
    else:
        for (m, weight) in get_neighbors(n):
            #nodes 'm' not in first and last set are added to first
            #n is set its parent
            if m not in open_set and m not in closed_set:
                open_set.add(m)
                parents[m] = n
                g[m] = g[n] + weight


            #for each node m,compare its distance from start i.e g(m) to the
            #from start through n node
            else:
                if g[m] > g[n] + weight:
                    #update g(m)
                    g[m] = g[n] + weight
                    #change parent of m to n
                    parents[m] = n

                    #if m in closed set,remove and add to open
                    if m in closed_set:
                        closed_set.remove(m)
                        open_set.add(m)

    if n == None:
        print('Path does not exist!')
        return None

    # if the current node is the stop_node
    # then we begin reconstructin the path from it to the start_node
    if n == stop_node:
        path = []

        while parents[n] != n:
            path.append(n)
            n = parents[n]
```

```
            path.append(start_node)

            path.reverse()

            print('Path found: {}'.format(path))
            return path


          # remove n from the open_list, and add it to closed_list
          # because all of his neighbors were inspected
          open_set.remove(n)
          closed_set.add(n)

        print('Path does not exist!')
        return None

#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
#for simplicity we ll consider heuristic distances given
#and this function returns heuristic distance for all nodes
def heuristic(n):
        H_dist = {
            'A': 11,
            'B': 6,
            'C': 99,
            'D': 1,
            'E': 7,
            'G': 0,

        }

        return H_dist[n]

#Describe your graph here
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],

}
aStarAlgo('A', 'G')
```

4.Write a python program to demonstrate the working of Alpha-Beta Pruning

```python
import math

# Represents the game tree nodes
class Node:
    def __init__(self, value=None):
        self.value = value
        self.children = []

# Minimax algorithm with alpha-beta pruning
def minimax_alpha_beta(node, depth, alpha, beta, maximizingPlayer):
    if depth == 0 or len(node.children) == 0:
        return node.value

    if maximizingPlayer:
        value = -math.inf
        for child in node.children:
            value = max(value, minimax_alpha_beta(child, depth - 1, alpha, beta, False))
            alpha = max(alpha, value)
            if alpha >= beta:
                break
        return value
    else:
        value = math.inf
        for child in node.children:
            value = min(value, minimax_alpha_beta(child, depth - 1, alpha, beta, True))
            beta = min(beta, value)
            if beta <= alpha:
                break
        return value

# Example usage
if __name__ == "__main__":
    # Construct a simple game tree
    root = Node()
    root.value = 3

    node1 = Node()
    node1.value = 5
    root.children.append(node1)

    node2 = Node()
    node2.value = 6
    root.children.append(node2)

    node3 = Node()
    node3.value = 9
    node1.children.append(node3)
```

```python
node4 = Node()
node4.value = 1
node1.children.append(node4)

node5 = Node()
node5.value = 2
node2.children.append(node5)

node6 = Node()
node6.value = 0
node2.children.append(node6)

# Perform minimax with alpha-beta pruning
result = minimax_alpha_beta(root, 3, -math.inf, math.inf, True)
print("Optimal value:", result)
```