```
import copy


class Node:
    def __init__(self, data, level, fval, player):
        self.data = data
        self.level = level
        self.fval = fval
        self.player = player  # 'X' or 'O'

    def generate_child(self):
        children = []
        for i in range(len(self.data)):
            for j in range(len(self.data[i])):
                if self.data[i][j] == '_':  # If the cell is empty
                    # Make a move and generate the child state
                    new_state = self.make_move(i, j)
                    child_node = Node(new_state, self.level + 1, 0, 'O' if self.player == 'X' else 'X')
                    children.append(child_node)
        return children

    def make_move(self, i, j):
        new_data = copy.deepcopy(self.data)
        new_data[i][j] = self.player
        return new_data

    def is_winner(self, player):
        for i in range(3):
            # Check rows and columns
            if all([self.data[i][j] == player for j in range(3)]) or all([self.data[j][i] == player for j in
range(3)]):
                return True
        # Check diagonals
        if self.data[0][0] == player and self.data[1][1] == player and self.data[2][2] == player:
            return True
        if self.data[0][2] == player and self.data[1][1] == player and self.data[2][0] == player:
            return True
        return False

    def find_empty_cells(self):
        return [(i, j) for i in range(3) for j in range(3) if self.data[i][j] == '_']


class TicTacToe:
    def __init__(self):
        self.open = []
        self.closed = []

    def f(self, node):
        return self.h(node) + node.level

    def h(self, start):
        # The heuristic will check for potential wins by 'X' or 'O'
        if start.is_winner('X'):  # Check if 'X' wins
            return 0
        if start.is_winner('O'):  # Check if 'O' wins
            return 0
        # A simple heuristic: number of empty cells or something else to estimate the distance
to goal
        return len(start.find_empty_cells())

    def process(self):
        # Use a default board state instead of user input
        start = [
            ['O', 'X', '_'],
            ['_', 'O', 'X'],
            ['_', '_', 'O']
        ]
        # Print initial state
        print("Initial Board State:")
        for row in start:
            print(' '.join(row))

        start_node = Node(start, 0, 0, 'O')  # 'X' starts
        self.open.append(start_node)

        while self.open:
            # Sort open list to get the node with the smallest f value
            self.open.sort(key=lambda x: x.fval)
            current_node = self.open.pop(0)

            # Print current board state
            for row in current_node.data:
                print(' '.join(row))
            print()

            # If the current state is a goal (winner), we stop
            if current_node.is_winner('X') or current_node.is_winner('O'):
                print(f"Player {current_node.player} wins!")
                break

            # Generate child nodes and add them to the open list
            children = current_node.generate_child()
            for child in children:
                child.fval = self.f(child)  # Calculate f for the child
                self.open.append(child)

            # Add current node to closed list to avoid re-exploration
            self.closed.append(current_node)

# Execute the Tic-Tac-Toe game
puz = TicTacToe()
```

```
puz.process()
```

OUTPUT

```
PS C:\Users\Lenovo\Desktop\VSC1> python -u "c:\Users\Lenovo\Desktop\VSC1\ai-final\ass-2 mod.py"
Initial Board State:
O X _
_ O X
_ _ O
O X _
_ O X
_ _ O
Player O wins!
PS C:\Users\Lenovo\Desktop\VSC1> python -u "c:\Users\Lenovo\Desktop\VSC1\ai-final\ass-2 mod.py"
Initial Board State:
X O _
_ X O
_ _ X
X O _
_ X O
_ _ X

Player X wins!
```