python-daily-coding

September 15, 2024

[1]:

```python
from collections import deque

# Goal state for the 8-puzzle
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

# Function to find the position of the blank tile (0)
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

# Function to check if the current state is the goal state
def is_goal(state):
    return state == goal_state

# Function to generate possible moves (up, down, left, right)
def generate_moves(state):
    moves = []
    i, j = find_blank(state)
    if i > 0:  # Move blank tile up
        new_state = [row[:] for row in state]
        new_state[i][j], new_state[i-1][j] = new_state[i-1][j], new_state[i][j]
        moves.append(new_state)
    if i < 2:  # Move blank tile down
        new_state = [row[:] for row in state]
        new_state[i][j], new_state[i+1][j] = new_state[i+1][j], new_state[i][j]
        moves.append(new_state)
    if j > 0:  # Move blank tile left
        new_state = [row[:] for row in state]
        new_state[i][j], new_state[i][j-1] = new_state[i][j-1], new_state[i][j]
        moves.append(new_state)
    if j < 2:  # Move blank tile right
        new_state = [row[:] for row in state]
        new_state[i][j], new_state[i][j+1] = new_state[i][j+1], new_state[i][j]
        moves.append(new_state)
    return moves
```

```python
# BFS to solve the 8-puzzle problem
def bfs_solve(start_state):
    queue = deque([start_state])
    visited = set()
    while queue:
        current_state = queue.popleft()
        visited.add(tuple(map(tuple, current_state)))
        if is_goal(current_state):
            print("Solved!")
            return
        for move in generate_moves(current_state):
            if tuple(map(tuple, move)) not in visited:
                queue.append(move)
    print("No solution found.")

# Initial state (can be changed)
initial_state = [[1, 2, 3], [4, 0, 6], [7, 5, 8]]

# Run the BFS solver
bfs_solve(initial_state)
```

```
Solved!
```

[1]:

```python
# 8 Queens problem
def is_safe(board, row, col):
    # Check this row on the left side
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on the left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on the left side
    for i, j in zip(range(row, len(board), 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solve_n_queens(board, col):
    # Base case: If all queens are placed, return True
    if col >= len(board):
        return True
```

```python
    # Consider this column and try placing this queen in all rows one by one
    for i in range(len(board)):
        if is_safe(board, i, col):
            # Place this queen in board[i][col]
            board[i][col] = 1

            # Recur to place the rest of the queens
            if solve_n_queens(board, col + 1):
                return True

            # If placing the queen in board[i][col] doesn't lead to a solution,
            # then remove the queen (backtrack)
            board[i][col] = 0

    # If the queen cannot be placed in any row in this column, return False
    return False

def print_board(board):
    for row in board:
        print(" ".join("Q" if x == 1 else "." for x in row))
    print()

# Driver code
def solve_8_queens():
    board = [[0 for _ in range(8)] for _ in range(8)]  # Initialize an 8x8
 ↪board with all 0s

    if not solve_n_queens(board, 0):
        print("No solution exists")
        return

    print_board(board)

# Solve the 8-Queens Problem
solve_8_queens()
```

```
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .
```

[5]:
```python
# Water Jug Problem
from collections import deque

# Function to check if the current state (amount in jug1, amount in jug2) has␣
↪been visited
def is_visited(visited, state):
    return state in visited

# BFS function to solve the water jug problem
def water_jug_bfs(jug1, jug2, target):
    # To store visited states (amount of water in each jug)
    visited = set()

    # Queue for BFS, initialized with the starting state (0, 0)
    queue = deque([(0, 0)])

    while queue:
        # Get the current state (amount of water in jug1 and jug2)
        current_jug1, current_jug2 = queue.popleft()

        # If the target amount is found in jug1, return the solution
        if current_jug1 == target:
            print(f"Solution found: ({current_jug1}, 0)")
            return True

        # Check if the target amount is found in jug2, then pour it into jug1␣
↪to make (2,0)
        if current_jug2 == target:
            # Pour jug2 into jug1 to get (2, 0)
            print(f"Solution found: ({target}, 0)")
            return True

        # If this state has already been visited, skip it
        if is_visited(visited, (current_jug1, current_jug2)):
            continue

        # Mark this state as visited
        visited.add((current_jug1, current_jug2))

        # Generate all possible moves (fill, empty, pour)
        possible_states = [
            (jug1, current_jug2),  # Fill jug1
            (current_jug1, jug2),  # Fill jug2
            (0, current_jug2),     # Empty jug1
            (current_jug1, 0),     # Empty jug2
            # Pour jug1 to jug2
```

```
            (current_jug1 - min(current_jug1, jug2 - current_jug2),␣
↪current_jug2 + min(current_jug1, jug2 - current_jug2)),
            # Pour jug2 to jug1
            (current_jug1 + min(current_jug2, jug1 - current_jug1),␣
↪current_jug2 - min(current_jug2, jug1 - current_jug1))
        ]

        # Add all valid moves (states) to the queue for further exploration
        for state in possible_states:
            if not is_visited(visited, state):
                queue.append(state)

    # If the queue is exhausted and no solution is found
    print("No solution exists")
    return False

# Driver code to test the solution
if __name__ == "__main__":
    jug1_capacity = 4  # Capacity of the first jug
    jug2_capacity = 3  # Capacity of the second jug
    target_amount = 2  # Target amount of water to measure

    water_jug_bfs(jug1_capacity, jug2_capacity, target_amount)
```

```
Solution found: (2, 0)
```

[9]:

```
# CryptArithmetic Problems SEND + MORE = MONEY
from itertools import permutations

# Function to check if the current assignment of letters to digits satisfies␣
↪the equation
def solve_cryptarithmetic():
    # List of letters in the puzzle
    letters = 'SENDMOREY'

    # Iterate over all permutations of digits (0-9) for the letters
    for perm in permutations(range(10), len(letters)):
        # Create a dictionary to map letters to digits
        mapping = dict(zip(letters, perm))

        # Check if the first letters (S, M) of SEND and MORE are not mapped to 0
        if mapping['S'] == 0 or mapping['M'] == 0:
            continue  # Skip this permutation since it would lead to leading␣
↪zeros

        # Calculate the numerical values of SEND, MORE, and MONEY
```

```python
        send = mapping['S']*1000 + mapping['E']*100 + mapping['N']*10 +␣
↪mapping['D']
        more = mapping['M']*1000 + mapping['O']*100 + mapping['R']*10 +␣
↪mapping['E']
        money = mapping['M']*10000 + mapping['O']*1000 + mapping['N']*100 +␣
↪mapping['E']*10 + mapping['Y']

        # Check if the equation SEND + MORE = MONEY holds true
        if send + more == money:
            print(f"SEND = {send}, MORE = {more}, MONEY = {money}")
            print(f"Solution found: {mapping}")
            return mapping

    # If no solution is found
    print("No solution exists")

# Driver code to solve the puzzle
solve_cryptarithmetic()
```

```
SEND = 9567, MORE = 1085, MONEY = 10652
Solution found: {'S': 9, 'E': 5, 'N': 6, 'D': 7, 'M': 1, 'O': 0, 'R': 8, 'Y': 2}
```

[9]:
```
{'S': 9, 'E': 5, 'N': 6, 'D': 7, 'M': 1, 'O': 0, 'R': 8, 'Y': 2}
```

[1]:
```python
def is_valid(state):
    m, c, boat = state
    if m < 0 or c < 0 or m > 3 or c > 3:  # Out of bounds check
        return False
    if (m > 0 and m < c) or (3 - m > 0 and 3 - m < 3 - c):  # Missionaries eaten
        return False
    return True

def dfs(state, path):
    if state == (0, 0, 0):  # Goal state
        return path + [state]
    m, c, boat = state
    moves = [(1, 0), (0, 1), (1, 1), (2, 0), (0, 2)]  # Possible moves
    for m_move, c_move in moves:
        new_state = (m - boat * m_move, c - boat * c_move, 1 - boat)
        if is_valid(new_state) and new_state not in path:
            solution = dfs(new_state, path + [state])
            if solution:
                return solution
    return None

# Initial state: (missionaries, cannibals, boat_side)
solution = dfs((3, 3, 1), [])
```

```python
if solution:
    for step in solution:
        print(step)
else:
    print("No solution found!")
```

```
(3, 3, 1)
(3, 2, 0)
(3, 2, 1)
(2, 2, 0)
(2, 2, 1)
(1, 1, 0)
(1, 1, 1)
(0, 1, 0)
(0, 1, 1)
(0, 0, 0)
```

[11]:
```python
# Vacuum cleaner problem
class VacuumCleanerAgent:
    def __init__(self, environment):
        self.environment = environment
        self.position = [0, 0]  # Start at the top-left corner (0, 0)
        self.moves = []         # To track the sequence of actions

    # Function to check if the current cell is dirty
    def is_dirty(self):
        return self.environment[self.position[0]][self.position[1]] == 'dirty'

    # Function to clean the current cell
    def clean(self):
        if self.is_dirty():
            print(f"Cleaning cell at {self.position}")
            self.environment[self.position[0]][self.position[1]] = 'clean'
            self.moves.append(f"Clean at {self.position}")

    # Function to move the vacuum cleaner
    def move(self, direction):
        if direction == 'up' and self.position[0] > 0:
            self.position[0] -= 1
        elif direction == 'down' and self.position[0] < 1:
            self.position[0] += 1
        elif direction == 'left' and self.position[1] > 0:
            self.position[1] -= 1
        elif direction == 'right' and self.position[1] < 1:
            self.position[1] += 1
        else:
            return False  # Invalid move
```

```python
        print(f"Moved {direction} to {self.position}")
        self.moves.append(f"Move {direction} to {self.position}")
        return True

    # Function to execute the cleaning task
    def clean_environment(self):
        # Simple strategy: move around and clean
        for i in range(2):   # Two rows
            for j in range(2):  # Two columns
                # Clean current position
                self.clean()

                # Move to the next position
                if j == 0 and i == 0:  # At (0,0), move right
                    self.move('right')
                elif j == 1 and i == 0:  # At (0,1), move down
                    self.move('down')
                elif j == 1 and i == 1:  # At (1,1), move left
                    self.move('left')

        # Ensure the vacuum cleaner finishes at the starting point (0,0)
        if self.position != [0, 0]:
            self.move('up')

        print("Environment cleaned!")
        return self.moves

# Environment setup
# 2x2 grid: Each cell can be 'clean' or 'dirty'
environment = [['dirty', 'dirty'],  # Row 0
               ['clean', 'dirty']]   # Row 1

# Initialize the agent
vacuum = VacuumCleanerAgent(environment)

# Execute the cleaning task
actions = vacuum.clean_environment()

# Print the final environment state and sequence of actions
print("\nFinal Environment State:")
for row in environment:
    print(row)

print("\nSequence of Actions:")
for action in actions:
    print(action)
```

```
Cleaning cell at [0, 0]
Moved right to [0, 1]
Cleaning cell at [0, 1]
Moved down to [1, 1]
Cleaning cell at [1, 1]
Moved left to [1, 0]
Moved up to [0, 0]
Environment cleaned!

Final Environment State:
['clean', 'clean']
['clean', 'clean']

Sequence of Actions:
Clean at [0, 0]
Move right to [0, 1]
Clean at [0, 1]
Move down to [1, 1]
Clean at [1, 1]
Move left to [1, 0]
Move up to [0, 0]
```

[12]:
```python
# BFS Implementation
from collections import deque

def bfs(graph, start):
    visited = set()          # Set to keep track of visited nodes
    queue = deque([start])  # Initialize the queue with the start node
    result = []             # List to store the order of nodes visited

    while queue:
        node = queue.popleft()  # Dequeue a node
        if node not in visited:
            visited.add(node)   # Mark the node as visited
            result.append(node) # Add the node to the result list

            # Add all unvisited neighbors to the queue
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)

    return result

# Example graph as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
```

```
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

# Driver code
if __name__ == "__main__":
    start_node = 'A'
    traversal_order = bfs(graph, start_node)
    print("BFS Traversal Order:", traversal_order)
```

```
BFS Traversal Order: ['A', 'B', 'C', 'D', 'E', 'F']
```

[13]:
```
# DFS implementation
def dfs_recursive(graph, node, visited):
    # Mark the current node as visited
    visited.add(node)
    print(node, end=' ')  # Print or process the node

    # Explore all unvisited neighbors
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs_recursive(graph, neighbor, visited)

# Example graph as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

# Driver code
if __name__ == "__main__":
    start_node = 'A'
    visited = set()  # Set to keep track of visited nodes
    print("DFS Traversal Order (Recursive):")
    dfs_recursive(graph, start_node, visited)
```

```
DFS Traversal Order (Recursive):
A B D E F C
```

[14]:
```
# Travelling Salesman Problem
import itertools
```

```python
def calculate_total_distance(permutation, distance_matrix):
    total_distance = 0
    for i in range(len(permutation) - 1):
        total_distance += distance_matrix[permutation[i]][permutation[i + 1]]
    # Add the distance to return to the starting city
    total_distance += distance_matrix[permutation[-1]][permutation[0]]
    return total_distance

def traveling_salesman_bruteforce(distance_matrix):
    n = len(distance_matrix)
    cities = list(range(n))
    min_distance = float('inf')
    best_route = None

    # Generate all permutations of cities
    for perm in itertools.permutations(cities):
        current_distance = calculate_total_distance(perm, distance_matrix)
        if current_distance < min_distance:
            min_distance = current_distance
            best_route = perm

    return best_route, min_distance

# Example distance matrix (symmetric)
# Distance between cities (0-indexed)
distance_matrix = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

# Driver code
if __name__ == "__main__":
    best_route, min_distance = traveling_salesman_bruteforce(distance_matrix)
    print("Best route:", best_route)
    print("Minimum distance:", min_distance)
```

```
Best route: (0, 1, 3, 2)
Minimum distance: 80
```

[15]:

```python
# A* Search
import heapq

def heuristic(a, b):
    # Using Manhattan distance as the heuristic
```

```python
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def astar(grid, start, goal):
    rows, cols = len(grid), len(grid[0])
    open_list = []  # Priority queue for nodes to explore
    heapq.heappush(open_list, (0 + heuristic(start, goal), 0, start, []))

    g_costs = {start: 0}  # Cost from start to node
    visited = set()      # Set of visited nodes

    while open_list:
        _, g, current, path = heapq.heappop(open_list)

        if current in visited:
            continue

        visited.add(current)
        path = path + [current]

        if current == goal:
            return path

        x, y = current
        for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] == 0:
                neighbor = (nx, ny)
                new_g = g + 1
                if neighbor not in g_costs or new_g < g_costs[neighbor]:
                    g_costs[neighbor] = new_g
                    f_cost = new_g + heuristic(neighbor, goal)
                    heapq.heappush(open_list, (f_cost, new_g, neighbor, path))

    return None  # No path found

# Example grid (0 = free space, 1 = obstacle)
grid = [
    [0, 0, 0, 0, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0]
]

# Driver code
if __name__ == "__main__":
    start = (0, 0)
```

```
    goal = (4, 4)
    path = astar(grid, start, goal)

    if path:
        print("Path found:", path)
    else:
        print("No path found")
```

```
Path found: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (4,
4)]
```

[1]:
```
# MAP COLORING TO IMPLEMENT CSP
# A simple function to check if coloring is safe for a given node
def is_safe(node, color, colors, graph):
    for neighbor in graph[node]:
        if colors[neighbor] == color:
            return False
    return True

# Backtracking function to solve the map coloring problem
def map_coloring(graph, colors, color_domain, node=0):
    if node == len(graph):
        return True  # All nodes have been colored successfully

    for color in color_domain:
        if is_safe(node, color, colors, graph):
            colors[node] = color  # Assign color to the current node

            if map_coloring(graph, colors, color_domain, node + 1):
                return True  # If coloring the rest of the map is successful,␣
 ↪return True

            colors[node] = None  # Backtrack if coloring the next node fails

    return False  # If no color can be assigned, return False

# Driver code
if __name__ == "__main__":
    # Define the graph (adjacency list representation of a map)
    graph = {
        0: [1, 2],  # Node 0 is adjacent to nodes 1 and 2
        1: [0, 2, 3],  # Node 1 is adjacent to nodes 0, 2, and 3
        2: [0, 1, 3],  # Node 2 is adjacent to nodes 0, 1, and 3
        3: [1, 2]   # Node 3 is adjacent to nodes 1 and 2
    }

    # Domain of colors (3 colors: Red, Green, Blue)
```

```
    color_domain = ['Red', 'Green', 'Blue']

    # Initialize colors for each node as None (uncolored)
    colors = {node: None for node in graph}

    if map_coloring(graph, colors, color_domain):
        print("Solution found:")
        for node, color in colors.items():
            print(f"Node {node}: {color}")
    else:
        print("No solution found")
```

```
Solution found:
Node 0: Red
Node 1: Green
Node 2: Blue
Node 3: Red
```

[4]:

```
# TIC TAC TOE
# Function to print the Tic-Tac-Toe board
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 5)

# Function to check if there is a winner
def check_winner(board, player):
    # Check rows, columns, and diagonals
    for row in board:
        if all([cell == player for cell in row]):
            return True
    for col in range(3):
        if all([board[row][col] == player for row in range(3)]):
            return True
    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

# Function to check if the board is full (draw)
def is_draw(board):
    return all([cell != " " for row in board for cell in row])

# Minimax algorithm to find the best move for AI
def minimax(board, depth, is_maximizing):
    if check_winner(board, "O"):
        return 1
```

```python
    if check_winner(board, "X"):
        return -1
    if is_draw(board):
        return 0

    if is_maximizing:
        best_score = -float('inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == " ":
                    board[i][j] = "O"
                    score = minimax(board, depth + 1, False)
                    board[i][j] = " "
                    best_score = max(score, best_score)
        return best_score
    else:
        best_score = float('inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == " ":
                    board[i][j] = "X"
                    score = minimax(board, depth + 1, True)
                    board[i][j] = " "
                    best_score = min(score, best_score)
        return best_score

# Find the best move for AI
def best_move(board):
    best_score = -float('inf')
    move = None
    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                board[i][j] = "O"
                score = minimax(board, 0, False)
                board[i][j] = " "
                if score > best_score:
                    best_score = score
                    move = (i, j)
    return move

# Main function to play Tic-Tac-Toe
def play_tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]

    while True:
        print_board(board)
```

```python
        # Player X's turn (human)
        row, col = map(int, input("Enter your move (row and column): ").split())
        if board[row][col] == " ":
            board[row][col] = "X"
        else:
            print("Invalid move, try again.")
            continue

        if check_winner(board, "X"):
            print_board(board)
            print("Player X wins!")
            break
        if is_draw(board):
            print_board(board)
            print("It's a draw!")
            break

        # AI's turn (Player O)
        row, col = best_move(board)
        board[row][col] = "O"

        if check_winner(board, "O"):
            print_board(board)
            print("AI wins!")
            break
        if is_draw(board):
            print_board(board)
            print("It's a draw!")
            break

# Start the game
play_tic_tac_toe()
```

```
  |   |
-----
  |   |
-----
  |   |
-----

Enter your move (row and column):  0 2

  |   | X
-----
  | O |
-----
  |   |
```

```
-----
Enter your move (row and column):  0 0
X | O | X
-----
  | O |
-----
  |   |
-----
Enter your move (row and column):  2 1
X | O | X
-----
O | O |
-----
  | X |
-----
Enter your move (row and column):  1 2
X | O | X
-----
O | O | X
-----
  | X | O
-----
Enter your move (row and column):  2 0
X | O | X
-----
O | O | X
-----
X | X | O
-----
It's a draw!
```

[5]:

```python
#MINIMAX Implementation
# Minimax function to evaluate the best score for maximizing and minimizing␣
↪player
def minimax(position, depth, is_maximizing):
    # Base case: if depth is 0 or end of game (no moves left)
    if depth == 0:
        return position

    if is_maximizing:
        best_score = -float('inf')  # Initialize the worst possible score for␣
↪maximizing player
        for move in possible_moves:  # Loop through all possible moves
```

```
            score = minimax(position + move, depth - 1, False)  # Recursively␣
↪calculate the score
            best_score = max(best_score, score)  # Choose the move with the␣
↪best score for maximizing player
        return best_score
    else:
        best_score = float('inf')  # Initialize the worst possible score for␣
↪minimizing player
        for move in possible_moves:  # Loop through all possible moves
            score = minimax(position + move, depth - 1, True)  # Recursively␣
↪calculate the score
            best_score = min(best_score, score)  # Choose the move with the␣
↪worst score for minimizing player
        return best_score

# Example to test Minimax algorithm
if __name__ == "__main__":
    position = 0  # Starting position (initial score)
    depth = 3  # Depth of the game tree (number of turns remaining)
    possible_moves = [1, -1]  # Moves that a player can make (increase or␣
↪decrease score by 1)

    # Start the game with the maximizing player
    result = minimax(position, depth, True)

    print("Best score for maximizing player:", result)
```

```
Best score for maximizing player: 1
```

[6]:

```
# Alpha-Beta Pruning function
def alphabeta(position, depth, alpha, beta, is_maximizing):
    # Base case: if the depth is 0, return the position (score)
    if depth == 0:
        return position

    if is_maximizing:
        max_eval = -float('inf')  # Start with the worst possible score
        for move in possible_moves:
            eval = alphabeta(position + move, depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval)  # Get the maximum score
            alpha = max(alpha, eval)  # Update alpha (best score so far for␣
↪maximizing)
            if beta <= alpha:  # Prune the branch
                break
        return max_eval
    else:
        min_eval = float('inf')  # Start with the worst possible score
```

```
        for move in possible_moves:
            eval = alphabeta(position + move, depth - 1, alpha, beta, True)
            min_eval = min(min_eval, eval)  # Get the minimum score
            beta = min(beta, eval)  # Update beta (best score so far for␣
 ↪minimizing)
            if beta <= alpha:  # Prune the branch
                break
        return min_eval

# Example to test Alpha-Beta Pruning
if __name__ == "__main__":
    position = 0  # Initial position (score)
    depth = 3  # Depth of the tree (number of turns remaining)
    possible_moves = [1, -1]  # Example moves (increase or decrease score)

    # Alpha and Beta values start as worst-case scenarios
    alpha = -float('inf')
    beta = float('inf')

    # Start the game with the maximizing player
    result = alphabeta(position, depth, alpha, beta, True)

    print("Best score for maximizing player:", result)
```

```
Best score for maximizing player: 1
```

[7]:

```
# Import necessary libraries
import math

# Define the dataset
# The dataset is a list of lists, where the last element of each list is the␣
 ↪label (class).
dataset = [
    [1, 1, 'Yes'],
    [1, 0, 'Yes'],
    [0, 1, 'No'],
    [0, 0, 'No']
]

# Function to calculate the Gini Index
def gini_index(groups, classes):
    # Total number of samples
    total_samples = sum([len(group) for group in groups])

    gini = 0.0
    # Loop through each group
    for group in groups:
```

```
        size = len(group)
        if size == 0:
            continue
        score = 0.0
        # Count the proportion of each class in the group
        for class_val in classes:
            proportion = [row[-1] for row in group].count(class_val) / size
            score += proportion ** 2
        gini += (1.0 - score) * (size / total_samples)

    return gini

# Function to split the dataset based on an attribute
def split_dataset(index, value, dataset):
    left, right = [], []
    for row in dataset:
        if row[index] == value:
            left.append(row)
        else:
            right.append(row)
    return left, right

# Function to choose the best split point
def get_best_split(dataset):
    class_values = list(set(row[-1] for row in dataset))
    best_index, best_value, best_score, best_groups = 999, 999, float('inf'), None
    for index in range(len(dataset[0]) - 1):
        for row in dataset:
            groups = split_dataset(index, row[index], dataset)
            gini = gini_index(groups, class_values)
            if gini < best_score:
                best_index, best_value, best_score, best_groups = index, row[index], gini, groups
    return {'index': best_index, 'value': best_value, 'groups': best_groups}

# Function to create a terminal node (i.e., leaf node)
def to_terminal(group):
    outcomes = [row[-1] for row in group]
    return max(set(outcomes), key=outcomes.count)

# Function to recursively split the dataset and build the tree
def split(node, max_depth, min_size, depth):
    left, right = node['groups']
    del(node['groups'])

    # If either left or right group is empty, create a terminal node
```

```python
    if not left or not right:
        node['left'] = node['right'] = to_terminal(left + right)
        return

    # Check for maximum depth
    if depth >= max_depth:
        node['left'], node['right'] = to_terminal(left), to_terminal(right)
        return

    # Process left child
    if len(left) <= min_size:
        node['left'] = to_terminal(left)
    else:
        node['left'] = get_best_split(left)
        split(node['left'], max_depth, min_size, depth+1)

    # Process right child
    if len(right) <= min_size:
        node['right'] = to_terminal(right)
    else:
        node['right'] = get_best_split(right)
        split(node['right'], max_depth, min_size, depth+1)

# Function to build a decision tree
def build_tree(train, max_depth, min_size):
    root = get_best_split(train)
    split(root, max_depth, min_size, 1)
    return root

# Function to make predictions with the decision tree
def predict(node, row):
    if row[node['index']] == node['value']:
        if isinstance(node['left'], dict):
            return predict(node['left'], row)
        else:
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']

# Example of building and using the decision tree
if __name__ == "__main__":
    # Build the tree
    tree = build_tree(dataset, max_depth=3, min_size=1)
```

```
    # Print the tree structure
    print(tree)

    # Test predictions
    for row in dataset:
        prediction = predict(tree, row)
        print('Expected=%s, Got=%s' % (row[-1], prediction))
```

```
{'index': 0, 'value': 1, 'left': {'index': 0, 'value': 1, 'left': 'Yes',
'right': 'Yes'}, 'right': {'index': 0, 'value': 0, 'left': 'No', 'right': 'No'}}
Expected=Yes, Got=Yes
Expected=Yes, Got=Yes
Expected=No, Got=No
Expected=No, Got=No
```

[9]:
```
import numpy as np

# Define the sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Define the FeedForwardNN class
class FeedForwardNN:
    def __init__(self, input_size, hidden_size, output_size):
        # Initialize weights and biases
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        self.weights_input_hidden = np.random.rand(self.input_size, self.
↪hidden_size)
        self.weights_hidden_output = np.random.rand(self.hidden_size, self.
↪output_size)
        self.bias_hidden = np.random.rand(1, self.hidden_size)
        self.bias_output = np.random.rand(1, self.output_size)

    def forward(self, X):
        # Forward pass
        self.hidden_input = np.dot(X, self.weights_input_hidden) + self.
↪bias_hidden
        self.hidden_output = sigmoid(self.hidden_input)
        self.final_input = np.dot(self.hidden_output, self.
↪weights_hidden_output) + self.bias_output
        self.final_output = sigmoid(self.final_input)
```

```
        return self.final_output

# Example usage of FeedForwardNN
if __name__ == "__main__":
    # Define network parameters
    input_size = 2  # Number of input neurons
    hidden_size = 4  # Number of hidden neurons
    output_size = 1  # Number of output neurons

    # Create a FeedForward Neural Network instance
    nn = FeedForwardNN(input_size, hidden_size, output_size)

    # Example input data
    X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

    # Perform a forward pass
    predictions = nn.forward(X)

    print("Predictions:")
    print(predictions)
```

```
Predictions:
[[0.90385993]
 [0.92443813]
 [0.92374768]
 [0.93636514]]
```

[2]:
```
def vacuum(world, position):
    moves = 0
    for i in range(len(world)):
        if world[position] == "dirty":
            world[position] = "clean"
        position = (position + 1) % len(world)
        moves += 1
    return moves

world = ["clean", "dirty", "clean", "dirty"]
position = 1  # starting position
print(vacuum(world, position))
```

```
4
```

[3]:
```
class AlphaBeta:
    def __init__(self):
        self.pruned_branches = 0
```

```
    def alpha_beta_pruning(self, depth, nodeIndex, maximizingPlayer, values,␣
↪alpha, beta):
        if depth == 3:  # terminal node
            return values[nodeIndex]

        if maximizingPlayer:
            maxEval = float('-inf')
            for i in range(2):
                eval = self.alpha_beta_pruning(depth + 1, nodeIndex * 2 + i,␣
↪False, values, alpha, beta)
                maxEval = max(maxEval, eval)
                alpha = max(alpha, eval)
                if beta <= alpha:
                    self.pruned_branches += 1
                    break  # prune
            return maxEval
        else:
            minEval = float('inf')
            for i in range(2):
                eval = self.alpha_beta_pruning(depth + 1, nodeIndex * 2 + i,␣
↪True, values, alpha, beta)
                minEval = min(minEval, eval)
                beta = min(beta, eval)
                if beta <= alpha:
                    self.pruned_branches += 1
                    break  # prune
            return minEval

# Example usage:
values = [3, 5, 6, 9, 1, 2, 0, -1]  # terminal values of the game tree
ab = AlphaBeta()
optimal_value = ab.alpha_beta_pruning(0, 0, True, values, float('-inf'),␣
↪float('inf'))
print(f"Optimal value: {optimal_value}")
print(f"Branches pruned: {ab.pruned_branches}")
```

```
Optimal value: 5
Branches pruned: 2
```

[ ]: