VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Pratik Jana - (1BM22CS356)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled “Artificial Intelligence (23CS5PCAIN)” carried out by **Pratik Jana (1BM22CS356),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Sunayana S Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link: pratik03092003/AI_PYTHON

Program 1

Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:

11/1/24 lab 1

Implementing a tictac-toe problem

→ Minimax algorithm ** (NO)

algorithm.

① → 2D array using list comprehension method and print array as strings

→ board = [[' ' for _ in range(3)] for _ in range(3)]

→ def print board
    for i in board:
        print ("|". join(i))
        print ("-" * 14)

ex

X	O	X
	O	X
	X	O

② # def check_win (board):
    for i in range (3):
        if {board[i][0] = board[i][1] = board[i][2] != ' ':
            return True}
        ~~if { board [0][i] = Board [1][i] == board [2][i] != ' ':~~
            return True }
        if{ board [0][0] == bord [1][1] = board [2][2]
        or [0][2] == [1][1] = [2][0] != ' ' }

fn checks all winning condition

→ getall available moves ( )

```
def. get_available_moves (board)
    return [(r,c) for r in range (3) for c in
            range (3) if board [r][c] == ' ']
```

AI ( )

① for each available move (r,c)
- temporarily place 'O' at (r,c):
  - If check_wins(board) : return (r,c)
  - undo move (set back to ' ').

② temporarily place 'x' at (r,c) :
- if check_win(board) : return (r,c) (blocking move)
- → undo move :

③ If the center (1,1) is empty return 1,1

(4) check each corner

(5) choose randomly

play game ()

→ Initialize the board

→ Game loop

→ print_board (board)

→ if current_player == 'X':
    r, c = map(int, input ("Enter your move")
    .split()

else
    r, c = ai_move (board)

→ update the board with current player move

→ check for win:

→ check for draw, if board is full print it declare a draw exit,

→ switch current player for next turn.

24/9/2024

Code:

```
count = 0
def rec(state, loc):
  global count
  if state['A'] == 0 and state['B'] == 0:
    print("Turning vacuum off")
    return

  if state[loc] == 1:
    state[loc] = 0
    count += 1
    print(f"Cleaned {loc}.")
    next_loc = 'B' if loc == 'A' else 'A'
    state[loc] = int(input(f"Is {loc} clean now? (0 if clean, 1 if dirty): "))
    if(state[next_loc]!=1):
     state[next_loc]=int(input(f"Is {next_loc} dirty? (0 if clean, 1 if dirty): "))
  if(state[loc]==1):
   rec(state,loc)
  else:
   next_loc = 'B' if loc == 'A' else 'A'
   dire="left" if loc=="B" else "right"
   print(loc,"is clean")
   print(f"Moving vacuum {dire}")
   if state[next_loc] == 1:
     rec(state, next_loc)
```

```
state = {}

state['A'] = int(input("Enter state of A (0 for clean, 1 for dirty): "))

state['B'] = int(input("Enter state of B (0 for clean, 1 for dirty): "))

loc = input("Enter location (A or B): ")

rec(state, loc)

print("Cost:",count)

print(state)
```

```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
Turning vacuum off
Cost: 0
{'A': 0, 'B': 0}
```

```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A
A is clean
Moving vacuum right
Cleaned B.
Is B clean now? (0 if clean, 1 if dirty): 0
Is A dirty? (0 if clean, 1 if dirty): 0
B is clean
Moving vacuum left
Cost: 1
{'A': 0, 'B': 0}
```

```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
Cleaned A.
Is A clean now? (0 if clean, 1 if dirty): 0
Is B dirty? (0 if clean, 1 if dirty): 0
A is clean
Moving vacuum right
Cost: 1
{'A': 0, 'B': 0}
```

```
def check_win(board, r, c):

    if board[r - 1][c - 1] == 'X':

        ch = "O"

    else:

        ch = "X"

    if ch not in board[r - 1] and '-' not in board[r - 1]:

        return True

    elif ch not in (board[0][c - 1], board[1][c - 1], board[2][c - 1]) and '-' not in (board[0][c - 1],
board[1][c - 1], board[2][c - 1]):

        return True

    elif ch not in (board[0][0], board[1][1], board[2][2]) and '-' not in (board[0][0], board[1][1],
board[2][2]):

        return True

    elif ch not in (board[0][2], board[1][1], board[2][0]) and '-' not in (board[0][2], board[1][1],
board[2][0]):

        return True

    return False


def displayb(board):

  print(board[0])

  print(board[1])

  print(board[2])
```

```
      break
   else :
     print("enter position to place O:")
     x=int(input())
     y=int(input())
     if(x>3 or y>3):
       print("invalid position")
       continue
     if(board[x-1][y-1]=='-'):
       board[x-1][y-1]='O'
       xo=1
       displayb(board)
     else:
       print("invalid position")
       continue
     if(check_win(board,x,y)):
         print("0 wins")
         flag=1
         break
if flag==0:
  print("Draw")
print("Game Over")
```

```
['-', '-', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place X:
1
1
['X', '-', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place O:
2
2
['X', '-', '-']
['-', 'O', '-']
['-', '-', '-']
enter position to place X:
3
3
['X', '-', '-']
['-', 'O', '-']
['-', '-', 'X']
enter position to place O:
1
2
['X', 'O', '-']
['-', 'O', '-']
['-', '-', 'X']
enter position to place X:
3
2
['X', 'O', '-']
['-', 'O', '-']
['-', 'X', 'X']
enter position to place O:
3
1
['X', 'O', '-']
['-', 'O', '-']
['O', 'X', 'X']
```

```
['-', '-', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place X:
1
1
['X', '-', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place O:
1
2
['X', 'O', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place X:
2
1
['X', 'O', '-']
['X', '-', '-']
['-', '-', '-']
enter position to place O:
2
2
['X', 'O', '-']
['X', 'O', '-']
['-', '-', '-']
enter position to place X:
3
1
['X', 'O', '-']
['X', 'O', '-']
['X', '-', '-']
X wins
Game Over
```

Program 2

Implement 8 puzzle problems using Depth FirstSearch (DFS)
Implement Iterative deepening search algorithm

Algorithm:

8 puzzle algorithm (BFS)

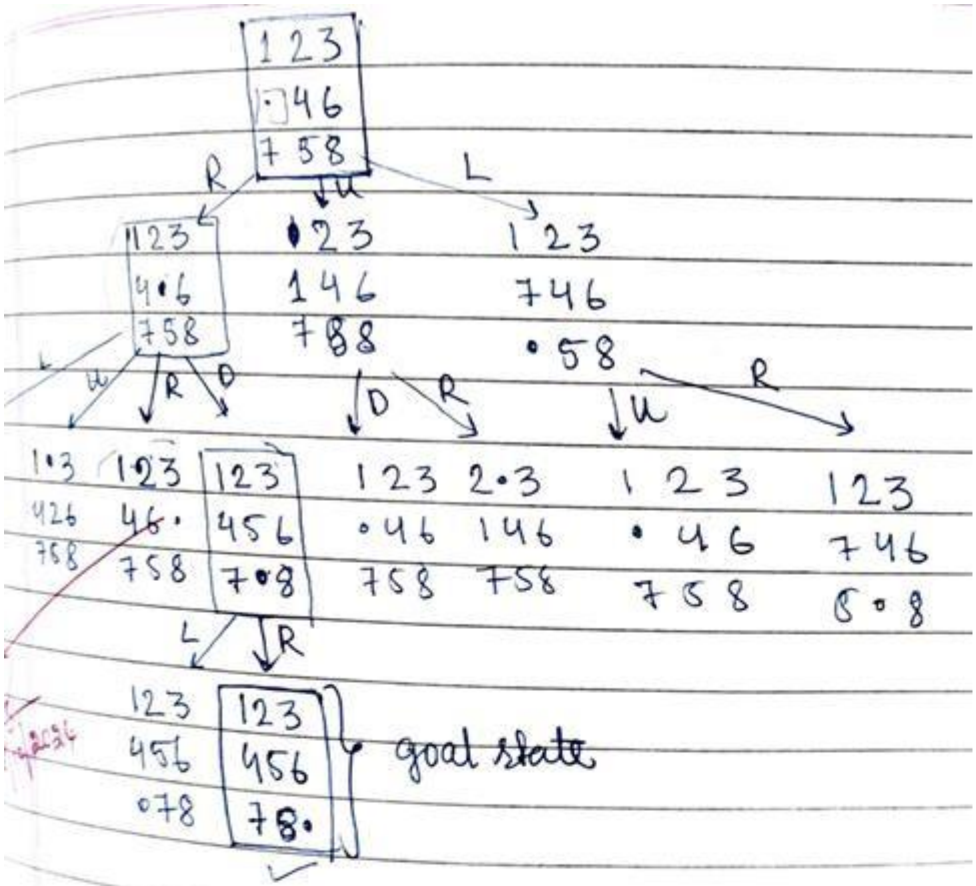
① Define goal state
Set goal state as (1,2,3,4,5,6,7,8,0)

② Input starting state
prompt user for initial config of the of the puzzle as 9 number (0-8)

③ Initialize BFS
- create a queue and add the starting state with empty path
- create a set to track visited states

④ BFS loop:-
- while Q is notempty
  → Deque current state
  → If the current state matches the goal state return the path

⑤ output Result -
If goal state is found, print sequence of moves
Q is empty and goal is not reached. indicate no sol. exist


goal state

Code:

4. 8 puzzle Manhattan distance heuristic

```
def manhattan_distance(state, goal):
  distance = 0
  for i in range(3):
    for j in range(3):
      tile = state[i][j]
      if tile != 0:  # Ignore the blank space (0)
        # Find the position of the tile in the goal state
        for r in range(3):
          for c in range(3):
            if goal[r][c] == tile:
              target_row, target_col = r, c
              break
        # Add the Manhattan distance (absolute difference in rows and columns)
        distance += abs(target_row - i) + abs(target_col - j)
  return distance


def findmin(open_list, goal):
  minv = float('inf')
  best_state = None
  for state in open_list:
    h = manhattan_distance(state['state'], goal)  # Use Manhattan distance here
    f = state['g'] + h
```

```
    if f < minv:
      minv = f
      best_state = state
  open_list.remove(best_state)
  return best_state

def operation(state):
  next_states = []
  blank_pos = find_blank_position(state['state'])
  for move in ['up', 'down', 'left', 'right']:
    new_state = apply_move(state['state'], blank_pos, move)
    if new_state:
      next_states.append({
        'state': new_state,
        'parent': state,
        'move': move,
        'g': state['g'] + 1
      })
  return next_states

def find_blank_position(state):
  for i in range(3):
    for j in range(3):
      if state[i][j] == 0:
```

```
        if f < minv:
            minv = f
            best_state = state
    open_list.remove(best_state)
    return best_state


def operation(state):
    next_states = []
    blank_pos = find_blank_position(state['state'])
    for move in ['up', 'down', 'left', 'right']:
        new_state = apply_move(state['state'], blank_pos, move)
        if new_state:
            next_states.append({
                'state': new_state,
                'parent': state,
                'move': move,
                'g': state['g'] + 1
            })
    return next_states


def find_blank_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
```

```
goal_state = [[1,2,3], [8,0,4], [7,6,5]]


# Open list and visited states
open_list = [{'state': initial_state, 'parent': None, 'move': None, 'g': 0}]
visited_states = []


while open_list:
  best_state = findmin(open_list, goal_state)


  print("Current state:")
  print_state(best_state['state'])


  h = manhattan_distance(best_state['state'], goal_state)  # Using Manhattan distance here
  f = best_state['g'] + h
  print(f"g(n): {best_state['g']}, h(n): {h}, f(n): {f}")


  if best_state['move'] is not None:
    print(f"Move: {best_state['move']}")
  print()
  if h == 0:  # Goal is reached if h == 0
    goal_state_reached = best_state
    break


  visited_states.append(best_state['state'])
```

```
    next_states = operation(best_state)

    for state in next_states:
        if state['state'] not in visited_states:
            open_list.append(state)

# Reconstruct the path of moves
moves = []
while goal_state_reached['move'] is not None:
    moves.append(goal_state_reached['move'])
    goal_state_reached = goal_state_reached['parent']
moves.reverse()

print("\nMoves to reach the goal state:", moves)
print("\nGoal state reached:")
print_state(goal_state)
```

```
Current state:
2 8 3
1 6 4
7 0 5
g(n): 0, h(n): 5, f(n): 5

Current state:
2 8 3
1 0 4
7 6 5
g(n): 1, h(n): 4, f(n): 5
Move: up

Current state:
2 0 3
1 8 4
7 6 5
g(n): 2, h(n): 3, f(n): 5
Move: up

Current state:
0 2 3
1 8 4
7 6 5
g(n): 3, h(n): 2, f(n): 5
Move: left

Current state:
1 2 3
0 8 4
7 6 5
g(n): 4, h(n): 1, f(n): 5
Move: down
```

```
Current state:
1 2 3
8 0 4
7 6 5
g(n): 5, h(n): 0, f(n): 5
Move: right


Moves to reach the goal state: ['up', 'up', 'left', 'down', 'right']

Goal state reached:
1 2 3
8 0 4
7 6 5
```

Program 3

Implement A* search algorithm

Algorithm:

8 puzzle problem using A* search It is used for finding the shortest path from start to the end

Step:1 → Initialize

create two lists : openlist. (for nodes to explore) and closed list (for nodes already explored)

Add the start node to openlist

Step:2 :→ Set costs

for each node calculate

g:- depth of node

Then find f = g+h

Step3 :- Process nodes

→ choose the node in open list with lowest of value

→ Move it from open list to close list

→ If this node is the goal Stop, this is the shortest path

Enta the start state : 28316470
No of state visited 34
Sol found at depth : 5 with cost 5

2 8 3   2 0 3   0 2 3   1 2 3
1 0 4 → 1 8 4 → 1 8 4 → 0 8 4
7 6 5   7 5 5   7 6 5   7 6 5

→ 1 2 3
8 0 4
7 6 5

Output (Missing Tiles)
Start ~~state~~: 2 8 3 1 6 4 7 0 5
no of state visited : 12
Sol found at depth: 5 with cost 5

2 8 3   2 0 3   0 2 3   1 2 3   1 2 3
1 0 4 → 1 8 4 → 1 8 4 → 0 8 4 → 8 0 4
7 6 4   7 6 5   7 6 5   7 6 5   7 6 5

Code:

```
from collections import deque

GOAL_STATE = (1, 2, 3, 4, 5, 6, 7, 8, 0)

def find_empty(state):
    return state.index(0)

def get_neighbors(state):
    neighbors = []
    empty_index = find_empty(state)
    row, col = divmod(empty_index, 3)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for dr, dc in directions:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            new_state = list(state)
            new_state[empty_index], new_state[new_index] = new_state[new_index],
new_state[empty_index]
            neighbors.append(tuple(new_state))
    return neighbors

def bfs(initial_state):
```

```
    queue = deque([(initial_state, [])])
    visited = set()
    visited.add(initial_state)
    visited_count = 1  # Initialize visited count
    while queue:
        current_state, path = queue.popleft()
        if current_state == GOAL_STATE:
            return path, visited_count  # Return path and count
        for neighbor in get_neighbors(current_state):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, path + [neighbor]))
                visited_count += 1  # Increment visited count
    return None, visited_count  # Return count if no solution found

def input_start_state():
    print("Enter the starting state as 9 numbers (0 for the empty space):")
    input_state = input("Format: 1 2 3 4 5 6 7 8 0\n")
    numbers = list(map(int, input_state.split()))
    if len(numbers) != 9 or set(numbers) != set(range(9)):
        print("Invalid input. Please enter numbers from 0 to 8 with no duplicates.")
        return input_start_state()
    return tuple(numbers)
```

```
def print_matrix(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])


if _name_ == "_main_":
    initial_state = input_start_state()
    print("Initial state:")
    print_matrix(initial_state)
    solution, visited_count = bfs(initial_state)
    print(f"Number of states visited: {visited_count}")
    if solution:
        print("\nSolution found with the following steps:")
        for step in solution:
            print_matrix(step)
            print()
    else:
        print("No solution found.")
```

```
Enter the starting state as 9 numbers (0 for the empty space):
Format: 1 2 3 4 5 6 7 8 0
2 3 5 1 6 4 8 0 7
Initial state:
(2, 3, 5)
(1, 6, 4)
(8, 0, 7)
Number of states visited: 24445

Solution found with the following steps:
(2, 3, 5)
(1, 0, 4)
(8, 6, 7)

(2, 3, 5)
(1, 4, 0)
(8, 6, 7)

(2, 3, 5)
(1, 4, 7)
(8, 6, 0)

(2, 3, 5)
(1, 4, 7)
(8, 0, 6)

(2, 3, 5)
(1, 0, 7)
(8, 4, 6)

(2, 3, 5)
(1, 7, 0)
(8, 4, 6)

(2, 3, 0)
(1, 7, 5)
(8, 4, 6)
```

Program 4
Implement Hill Climbing search algorithm tosolve N-Queens problem

Algorithm:

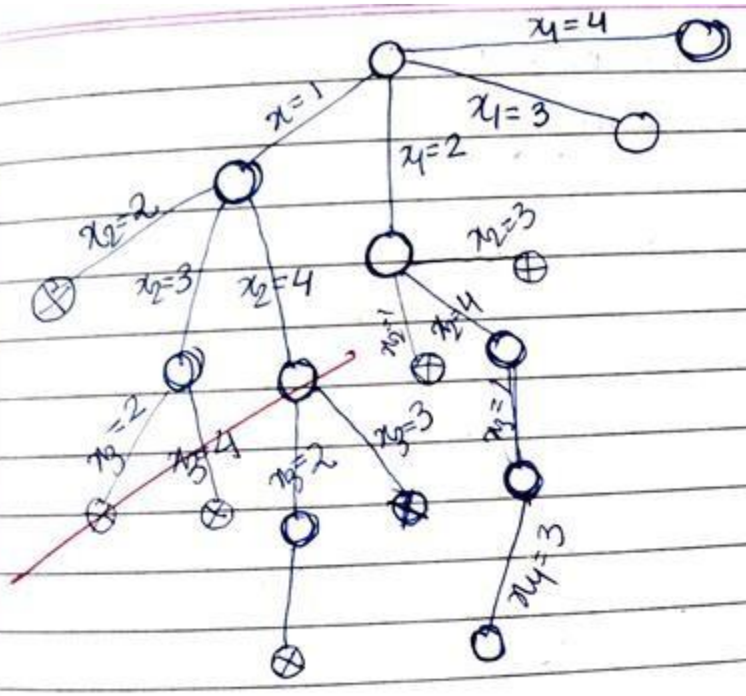
Hill climbing search algo

fn Hill_climbing (N):
current_state = random_configuration()
while true:
neighbour_states = generate_neighbours(current_state)
best_neighbour = find_neighbour_with_lowest_cost(neighbour_state)

if cost(best_neighbor) < cost(current_state)
current_state = best_neighbor
else:
return current_state # No better neighbor, terminate

output

Initial board	final state
0 0 1 0	0 0 0 1
0 0 1 0	0 1 0 0
0 1 0 0	0 0 1 0
0 1 0 0	1 0 0 0



$x_1 = 4$
$x = 1$
$x_1 = 3$
$x_1 = 2$
$x_2 = 2$
$x_2 = 3$
$x_2 = 3$
$x_2 = 4$
$x_2 = 1$
$x_2 = 4$
$x_3 = 2$
$x_3 = 4$
$x_3 = 2$
$x_3 = 3$
$x_3 = 1$
$x_4 = 3$

Code:

4 queens using hill climbing

```
import random


def calculate_conflicts(board):
  conflicts = 0
  n = len(board)
  for i in range(n):
    for j in range(i + 1, n):
      if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
        conflicts += 1
  return conflicts


def hill_climbing(n):
  cost=0
  while True:
    # Initialize a random board
    current_board = list(range(n))
    random.shuffle(current_board)
    current_conflicts = calculate_conflicts(current_board)

    while True:
      # Generate neighbors by moving each queen to a different position
      found_better = False
```

```
for i in range(n):
  for j in range(n):
    if j != current_board[i]:  # Only consider different positions
      neighbor_board = list(current_board)
      neighbor_board[i] = j
      neighbor_conflicts = calculate_conflicts(neighbor_board)
      if neighbor_conflicts < current_conflicts:
        print_board(current_board)
        print(current_conflicts)
        print_board(neighbor_board)
        print(neighbor_conflicts)
        current_board = neighbor_board
        current_conflicts = neighbor_conflicts
        cost+=1
        found_better = True
        break
  if found_better:
    break

# If no better neighbor found, stop searching
if not found_better:
  break

# If a solution is found (zero conflicts), return the board
```

```
  for i in range(n):
    for j in range(n):
      if j != current_board[i]:  # Only consider different positions
        neighbor_board = list(current_board)
        neighbor_board[i] = j
        neighbor_conflicts = calculate_conflicts(neighbor_board)
        if neighbor_conflicts < current_conflicts:
          print_board(current_board)
          print(current_conflicts)
          print_board(neighbor_board)
          print(neighbor_conflicts)
          current_board = neighbor_board
          current_conflicts = neighbor_conflicts
          cost+=1
          found_better = True
          break
    if found_better:
      break

  # If no better neighbor found, stop searching
  if not found_better:
    break

# If a solution is found (zero conflicts), return the board
```

```
===============
Q . . .
. . . Q
. . Q .
. Q . .

4
Q . . .
Q . . .
. . Q .
. Q . .

3
Q . . .
Q . . .
. . Q .
. Q . .

3
. . Q .
Q . . .
. . Q .
. Q . .

2
. . Q .
Q . . .
. . Q .
. Q . .

2
. . . Q
Q . . .
. . Q .
. Q . .

1
Final Board Configuration:
. Q . .
. . . Q
Q . . .
. . Q .
```

Program 5

Simulated Annealing to Solve 8-Queensproblem

Algorithm:

(1) import mlrose as — Annealing
impor numpy — pogram

(2) define fitness fn for 8 queens
queen-max (position)
ip : position
op : no of non-attacking Quee

set queen not-attacking to 0 (to count non attacking queen)
check rows
colums
diagnols

return queen not-attacking

(3) setup optimization problem

→ create a custom fitness fn using queens-max
→ define Discrete optimization problem
→ length = 8 (for 8 queens)
→ fitness fn = queen-max
→ maximize = True (We want to maximize non-attacking Queens)

(

→ max_value = 8 (each Queen can be placed in any row from 0 to 7).

→ Define temperature Decay Schedule for simulated annealing

→ Set Initial position
Define Initial guess for q position on board

→ Run simulated annealing

→ print results

~~output~~

best position: [4 6 1 5 2 0 3 7]
no of queens not attacking each other
8.0

7/10/2024

Code:

Simulated annealing 8 queens

```
import numpy as np
from scipy.optimize import dual_annealing


def queens_max(position):
  # This function calculates the number of pairs of queens that are not attacking each other
  position = np.round(position).astype(int)  # Round and convert to integers for queen positions
  n = len(position)
  queen_not_attacking = 0

  for i in range(n - 1):
    no_attack_on_j = 0
    for j in range(i + 1, n):
      # Check if queens are on the same row or on the same diagonal
      if position[i] != position[j] and abs(position[i] - position[j]) != (j - i):
        no_attack_on_j += 1
    if no_attack_on_j == n - 1 - i:
      queen_not_attacking += 1
  if queen_not_attacking == n - 1:
    queen_not_attacking += 1
  return -queen_not_attacking  # Negative because we want to maximize this value


# Bounds for each queen's position (0 to 7 for an 8x8 chessboard)
bounds = [(0, 8) for _ in range(8)]
```

```
# Use dual_annealing for simulated annealing optimization

result = dual_annealing(queens_max, bounds)


# Display the results

best_position = np.round(result.x).astype(int)

best_objective = -result.fun  # Flip sign to get the number of non-attacking queens


print('The best position found is:', best_position)

print('The number of queens that are not attacking each other is:', best_objective)
```

```
The best position found is: [0 8 5 2 6 3 7 4]
The number of queens that are not attacking each other is: 8
```

Program 6

Create a knowledge base using propositional logic and showthat the given query entails the knowledge base or not.

Algorithm:

propotional logic (Either true or false, not both

syntax Semantic

Atomic Complex

¬ negation

∨ disjuction

∧ conjuction

→ If then

↔ iff

A	B	∧	∨	→	↔
0	0	0	0	T	T
0	1	1	0	T	F
1	0	1	0	F	F
1	1	1	1	T	T

fn TT-Entails? (KB, α) return true or false

inputs: KB, the knowledge base, a sentence in propotional logic α, the query, a sentence in propositional logic

symbols ← a list of the proposition symbols in KB and α.

returns TT-CHECK-ALL (KB, α, symbols, { })

function TT-check-All (KB, α, symbols, model)
return True or false
if Empty? (symbols) Then
if PL-True ? (KB, model) return
TL-True?

else return true

else do
$P \leftarrow$ first (symbols)
rest - Rest (symbols)
return (TT check-AL (KB, α, symbols
-model U True) }

$\alpha = A \vee B$ $KB = (A \vee C)$

→ goal is to Determine if a given
statement (query) logically follows from
a KB
→ $KB = \{A \wedge B\}$
$\alpha = B$?

4/11/2024

Code:

Propositional logic

```
import itertools

# Function to evaluate an expression

def evaluate_expression(a, b, c, expression):

  # Use eval() to evaluate the logical expression

  return eval(expression)


# Function to generate the truth table and evaluate a logical expression

def truth_table_and_evaluation(kb, query):

  # All possible combinations of truth values for a, b, and c

  truth_values = [True, False]

  combinations = list(itertools.product(truth_values, repeat=3))


  # Reverse the combinations to start from the bottom (False -> True)

  combinations.reverse()


  # Header for the full truth table

  print(f"{'a':<5}   {'b':<5} {'c':<5} {'KB':<20}{'Query':<20}")


  # Evaluate the expressions for each combination

  for combination in combinations:

    a, b, c = combination
```

```
    # Evaluate the knowledge base (KB) and query expressions
    kb_result = evaluate_expression(a, b, c, kb)
    query_result = evaluate_expression(a, b, c, query)

    # Replace True/False with string "True"/"False"
    kb_result_str = "True" if kb_result else "False"
    query_result_str = "True" if query_result else "False"

    # Convert boolean values of a, b, c to "True"/"False"
    a_str = "True" if a else "False"
    b_str = "True" if b else "False"
    c_str = "True" if c else "False"

    # Print the results for the knowledge base and the query
    print(f"{a_str:<5} {b_str:<5} {c_str:<5} {kb_result_str:<20} {query_result_str:<20}")

# Additional output for combinations where both KB and query are true
print("\nCombinations where both KB and Query are True:")
print(f"{'a':<5}   {'b':<5} {'c':<5} {'KB':<20}{'Query':<20}")

# Print only the rows where both KB and Query are True
for combination in combinations:
    a, b, c = combination
```

```
    # Evaluate the knowledge base (KB) and query expressions
    kb_result = evaluate_expression(a, b, c, kb)
    query_result = evaluate_expression(a, b, c, query)

    # Replace True/False with string "True"/"False"
    kb_result_str = "True" if kb_result else "False"
    query_result_str = "True" if query_result else "False"

    # Convert boolean values of a, b, c to "True"/"False"
    a_str = "True" if a else "False"
    b_str = "True" if b else "False"
    c_str = "True" if c else "False"

    # Print the results for the knowledge base and the query
    print(f"{a_str:<5} {b_str:<5} {c_str:<5} {kb_result_str:<20} {query_result_str:<20}")

# Additional output for combinations where both KB and query are true
print("\nCombinations where both KB and Query are True:")
print(f"{'a':<5}   {'b':<5} {'c':<5} {'KB':<20}{'Query':<20}")

# Print only the rows where both KB and Query are True
for combination in combinations:
    a, b, c = combination
```

```
a       b      c      KB                    Query
False False False False                    False
False False True  False                    False
False True  False False                    True
False True  True  True                     True
True  False False True                     True
True  False True  False                    True
True  True  False True                     True
True  True  True  True                     True

Combinations where both KB and Query are True:
a       b      c      KB                    Query
False True  True  True                     True
True  False False True                     True
True  True  False True                     True
True  True  True  True                     True
```

Program 7

Implement unification in first orderlogic

Algorithm:

Unification Algorithm

Algorithm: unify (Ψ_1, Ψ_2)

Step 1: if Ψ_1 or Ψ_2 is a variable or constant, then:

a) Ψ_1 or Ψ_2 are Identical then return NIL

b) Else if Ψ_1 is a variable a, then if Ψ_1 occurs in Ψ_2 then return failure

c) Else return $\{(\Psi_2/\Psi_1)\}$

d) Else if Ψ_2 is variable,

    a) Ψ_2 occurs in Ψ_2 then return failure

    b) Else return $\{(\Psi_1/\Psi_2)\}$

e) Else return Failure

Step 2: If the Initial predicate symbol in Ψ_1 and Ψ_2 are not same, then return failure

Step 3: If Ψ_1 and Ψ_2 have a different number of arguments, then return failure

Step4: Set Substitution Set (Subset) to NIL

Step5: For $i=1$ to the no of elements in Ψ_1

a) call unify function with the i^{th} element of Ψ_1 and i^{th} element of Ψ_2 and put the result into S

b) If S = failure then return failure

c) If $S \neq NIL$ then do,

a) Apply S to the remainder of both L_1 and L_2

Subset = Append(S, Subset)

Step6: return Subset :-

26/11/24

Code:

Program 7:Implement unification in first order logic

```
import re


def occurs_check(var, x):
  """Checks if var occurs in x (to prevent circular substitutions)."""
  if var == x:
    return True
  elif isinstance(x, list):  # If x is a compound expression (like a function or predicate)
    return any(occurs_check(var, xi) for xi in x)
  return False



def unify_var(var, x, subst):
  """Handles unification of a variable with another term."""
  if var in subst:  # If var is already substituted
    return unify(subst[var], x, subst)
  elif isinstance(x, (list, tuple)) and tuple(x) in subst:  # Handle compound expressions
    return unify(var, subst[tuple(x)], subst)
  elif occurs_check(var, x):  # Check for circular references
    return "FAILURE"
  else:
    # Add the substitution to the set (convert list to tuple for hashability)
    subst[var] = tuple(x) if isinstance(x, list) else x
    return subst
```

```
def unify(x, y, subst=None):
  """
  Unifies two expressions x and y and returns the substitution set if they can be unified.
  Returns 'FAILURE' if unification is not possible.
  """
  if subst is None:
    subst = {}  # Initialize an empty substitution set

  # Step 1: Handle cases where x or y is a variable or constant
  if x == y:  # If x and y are identical
    return subst
  elif isinstance(x, str) and x.islower():  # If x is a variable
    return unify_var(x, y, subst)
  elif isinstance(y, str) and y.islower():  # If y is a variable
    return unify_var(y, x, subst)
  elif isinstance(x, list) and isinstance(y, list):  # If x and y are compound expressions (lists)
    if len(x) != len(y):  # Step 3: Different number of arguments
      return "FAILURE"

    # Step 2: Check if the predicate symbols (the first element) match
    if x[0] != y[0]:  # If the predicates/functions are different
      return "FAILURE"
```

```
def unify(x, y, subst=None):
  """
  Unifies two expressions x and y and returns the substitution set if they can be unified.
  Returns 'FAILURE' if unification is not possible.
  """
  if subst is None:
    subst = {}  # Initialize an empty substitution set

  # Step 1: Handle cases where x or y is a variable or constant
  if x == y:  # If x and y are identical
    return subst
  elif isinstance(x, str) and x.islower():  # If x is a variable
    return unify_var(x, y, subst)
  elif isinstance(y, str) and y.islower():  # If y is a variable
    return unify_var(y, x, subst)
  elif isinstance(x, list) and isinstance(y, list):  # If x and y are compound expressions (lists)
    if len(x) != len(y):  # Step 3: Different number of arguments
      return "FAILURE"

    # Step 2: Check if the predicate symbols (the first element) match
    if x[0] != y[0]:  # If the predicates/functions are different
      return "FAILURE"
```

```
    # Step 5: Recursively unify each argument
    for xi, yi in zip(x[1:], y[1:]):  # Skip the predicate (first element)
        subst = unify(xi, yi, subst)
        if subst == "FAILURE":
            return "FAILURE"
    return subst
  else:  # If x and y are different constants or non-unifiable structures
    return "FAILURE"


def unify_and_check(expr1, expr2):
  """
  Attempts to unify two expressions and returns a tuple:
  (is_unified: bool, substitutions: dict or None)
  """
  result = unify(expr1, expr2)
  if result == "FAILURE":
    return False, None
  return True, result


def display_result(expr1, expr2, is_unified, subst):
  print("Expression 1:", expr1)
```

```
# Main function to interact with the user

def main():

  while True:

    # Get the first and second terms from the user

    expr1_input = input("Enter the first expression (e.g., p(x, f(y))): ")

    expr2_input = input("Enter the second expression (e.g., p(a, f(z))): ")



    # Parse the input strings into the appropriate structures

    expr1 = parse_input(expr1_input)

    expr2 = parse_input(expr2_input)



    # Perform unification

    is_unified, result = unify_and_check(expr1, expr2)



    # Display the results

    display_result(expr1, expr2, is_unified, result)



    # Ask the user if they want to run another test

    another_test = input("Do you want to test another pair of expressions? (yes/no):
").strip().lower()

    if another_test != 'yes':
```

```
        break


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

```
Enter the first expression (e.g., p(x, f(y))): p(b,x,f(g(z)))
Enter the second expression (e.g., p(a, f(z))): p(z,f(y),f(y))
Expression 1: ['p', '(b', 'x', ['f', '(g(z)))']]
Expression 2: ['p', '(z', ['f', '(y)'], ['f', '(y))']]
Result: Unification Successful
Substitutions: {'(b': '(z', 'x': ['f', '(y)'], '(g(z)))': '(y))'}
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(x,h(y))
Enter the second expression (e.g., p(a, f(z))): p(a,f(z))
Expression 1: ['p', '(x', ['h', '(y))']]
Expression 2: ['p', '(a', ['f', '(z))']]
Result: Unification Failed
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(f(a),g(y))
Enter the second expression (e.g., p(a, f(z))): p(x,x)
Expression 1: ['p', '(f(a)', ['g', '(y))']]
Expression 2: ['p', '(x', 'x)']
Result: Unification Successful
Substitutions: {'(f(a)': '(x', 'x)': ['g', '(y))']}
Do you want to test another pair of expressions? (yes/no): no
```

Program 8

Create a knowledge base consisting of first order logicstatements and prove the given query using forward
reasoning.

Algorithm:

1) $\Psi_1 = P(b, x, f(g(z)))$ — ①
$P(z, f(y), f(y))$ — ②

Replace z in ② with b

$p(b, f(y), f(y))^{b/z}$

replace x in ① with f(y)

$p(b, f(y), f(g(z)))^{f(y)/x}$

replace y in ② with g(z)

$p(b, f(y), f(g, z))^{f(z)/y}$

$= p(b, f(y), f(g(z)))$

$p = b, f(y), f(g(z))$

unification possible

$\Psi_1 = p(f(a), g(y))$

$\Psi_2 = p(x, x)$

unification not possible as x cannot be replaced

output

$\Psi_1 - D(b, x, f(g(z)))$

$\Psi_2 - p(z, f(y), f(y))$

$\Psi_1 = p(f(a), g(x))$

$\Psi_2 - P(x, x)$

verification failde : Unification not possible
as a cannot be replaced.

Code:

Program 7:Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

```
# Define the knowledge base (KB) as a set of facts

KB = set()


# Premises based on the provided FOL problem

KB.add('American(Robert)')

KB.add('Enemy(America, A)')

KB.add('Missile(T1)')

KB.add('Owns(A, T1)')


# Define inference rules

def modus_ponens(fact1, fact2, conclusion):

  """ Apply modus ponens inference rule: if fact1 and fact2 are true, then conclude conclusion
"""

  if fact1 in KB and fact2 in KB:

    KB.add(conclusion)

    print(f"Inferred: {conclusion}")


def forward_chaining():

  """ Perform forward chaining to infer new facts until no more inferences can be made """

  # 1. Apply: Missile(x) → Weapon(x)

  if 'Missile(T1)' in KB:

    KB.add('Weapon(T1)')

    print(f"Inferred: Weapon(T1)")
```

```
    # 2. Apply: Sells(Robert, T1, A) from Owns(A, T1) and Weapon(T1)
    if 'Owns(A, T1)' in KB and 'Weapon(T1)' in KB:
        KB.add('Sells(Robert, T1, A)')
        print(f"Inferred: Sells(Robert, T1, A)")

    # 3. Apply: Hostile(A) from Enemy(A, America)
    if 'Enemy(America, A)' in KB:
        KB.add('Hostile(A)')
        print(f"Inferred: Hostile(A)")

    # 4. Now, check if the goal is reached (i.e., if 'Criminal(Robert)' can be inferred)
    if 'American(Robert)' in KB and 'Weapon(T1)' in KB and 'Sells(Robert, T1, A)' in KB and
'Hostile(A)' in KB:
        KB.add('Criminal(Robert)')
        print("Inferred: Criminal(Robert)")

    # Check if we've reached our goal
    if 'Criminal(Robert)' in KB:
        print("Robert is a criminal!")
    else:
        print("No more inferences can be made.")

# Run forward chaining to attempt to derive the conclusion
forward_chaining()
```

```
Inferred: Weapon(T1)
Inferred: Sells(Robert, T1, A)
Inferred: Hostile(A)
Inferred: Criminal(Robert)
Robert is a criminal!
```

Program 9

Create a knowledge base consisting of first order logic statements and prove the given query using Resolution

Algorithm:

20/11/24 Lab 9

Representation Forward reasoning Algorithm

fn FOL FC ASK(KB, a) returns a substitution of false

inputs: KB, the knowledge base, a set of first order definite clauses

a, the query an atomic sentence

repeat until new is empty

new ← { }

for each rule in KB do

for each θ such that subst(θ, p1 - pn) = SUBST(θ, P1 ∧ ... ∧ Pn')

for some p1 ... pn in KB

q - subst(θ, q)

if q doesnot unify with some sentence already in KB

q' - SUBST(Q, q)

if q doesnot unify with some sentence already in KB of new then

θ ← unify(q, a)

if φ is not fail then return

add networks

return false

output

final inferred facts
American (Robert) is True
Missile (T1) is True
Enemy (A, America) is True
owns (A, T1) is True.
Hostile (A) is True
weapon (T1) is True
sells (Robert, T1, A) is True
Criminal (Robert) is True.

26-11-24

Code:

Program 9:Create a knowledge base consisting of first order logic statements and prove the given query using Resolution

```
# Define the knowledge base (KB)
KB = {
    "food(Apple)": True,
    "food(vegetables)": True,
    "eats(Anil, Peanuts)": True,
    "alive(Anil)": True,
    "likes(John, X)": "food(X)",  # Rule: John likes all food
    "food(X)": "eats(Y, X) and not killed(Y)",  # Rule: Anything eaten and not killed is food
    "eats(Harry, X)": "eats(Anil, X)",  # Rule: Harry eats what Anil eats
    "alive(X)": "not killed(X)",  # Rule: Alive implies not killed
    "not killed(X)": "alive(X)",  # Rule: Not killed implies alive
}

# Function to evaluate if a predicate is true based on the KB
def resolve(predicate):
    # If it's a direct fact in KB
    if predicate in KB and isinstance(KB[predicate], bool):
        return KB[predicate]

    # If it's a derived rule
    if predicate in KB:
        rule = KB[predicate]
```

```
    if " and " in rule:  # Handle conjunction
        sub_preds = rule.split(" and ")
        return all(resolve(sub.strip()) for sub in sub_preds)
    elif " or " in rule:  # Handle disjunction
        sub_preds = rule.split(" or ")
        return any(resolve(sub.strip()) for sub in sub_preds)
    elif "not " in rule:  # Handle negation
        sub_pred = rule[4:]  # Remove "not "
        return not resolve(sub_pred.strip())
    else:  # Handle single predicate
        return resolve(rule.strip())

# If the predicate is a specific query (e.g., likes(John, Peanuts))
if "(" in predicate:
    func, args = predicate.split("(")
    args = args.strip(")").split(", ")
    if func == "food" and args[0] == "Peanuts":
        return resolve("eats(Anil, Peanuts)") and not resolve("killed(Anil)")
    if func == "likes" and args[0] == "John" and args[1] == "Peanuts":
        return resolve("food(Peanuts)")

# Default to False if no rule or fact applies
return False
```

```
# Query to prove: John likes Peanuts

query = "likes(John, Peanuts)"

result = resolve(query)


# Print the result

print(f"Does John like peanuts? {'Yes' if result else 'No'}")
```

```
Does John like peanuts? Yes
```

Program 10

Implement Alpha-Beta Pruning.

Algorithm:

3/12/24 Lab 10

~~search techniques~~

→ Implement α, β Running Algorithm

α-β process to find the optimal path. without look at every node in the same time

In both min and MAX node we return α ≥ β which compare with parent node only

Both minimax Alpha(α) - Beta(β) cutoff give score

Alpha (α) - Beta (β) gives optimal sol as it takes time to get the value for the root node

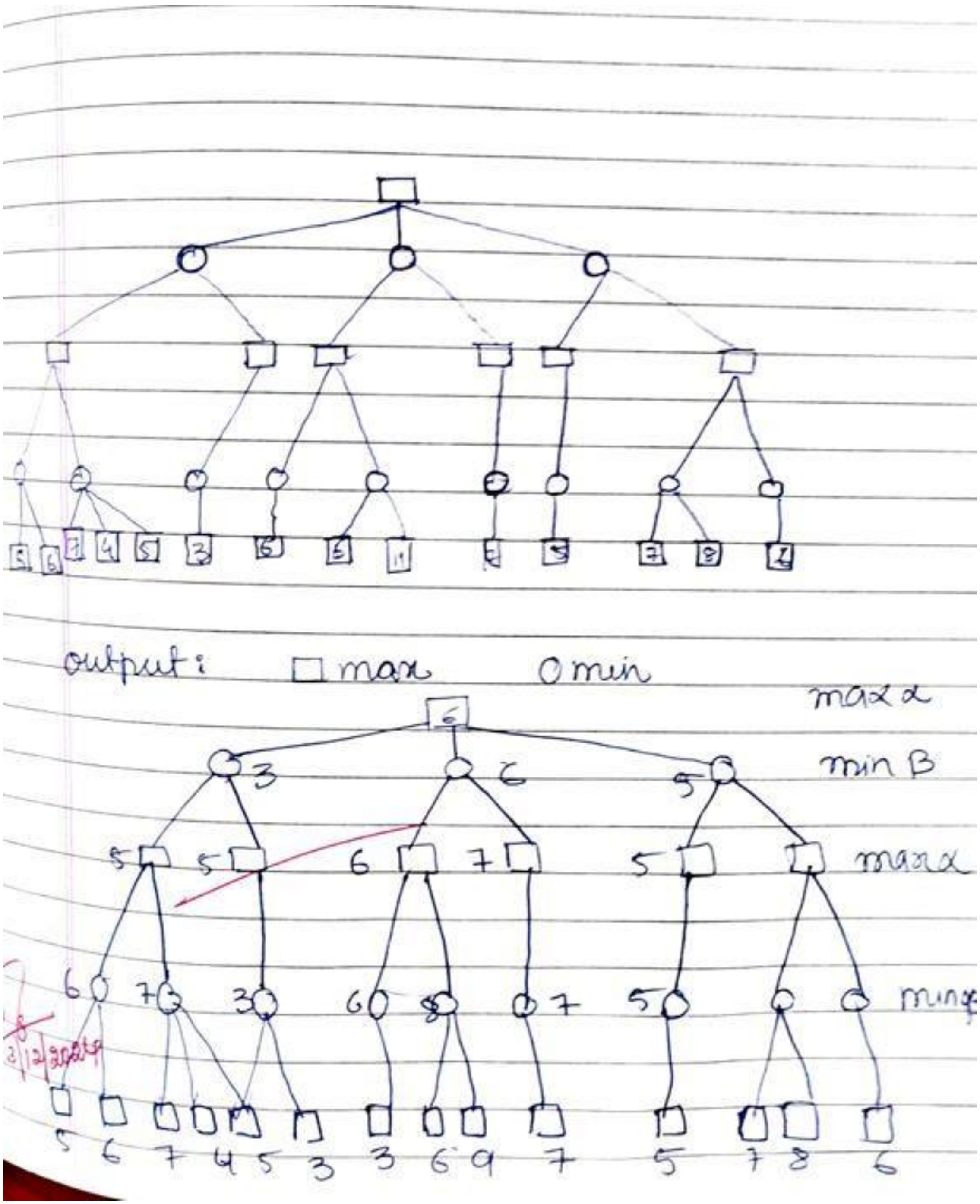
□ max ○ min

☆ output:

Enter no for the gave for the game tree (space - generated)

10 9 14 8 5 4 50 3

final result of Alpha beta Proning : 50



Code:

Program 10:Implement Alpha-Beta Pruning.

```
# Alpha-Beta Pruning Implementation

def alpha_beta_pruning(node, alpha, beta, maximizing_player):

  # Base case: If it's a leaf node, return its value (simulating evaluation of the node)

  if type(node) is int:

    return node



  # If not a leaf node, explore the children

  if maximizing_player:

    max_eval = -float('inf')

    for child in node:  # Iterate over children of the maximizer node

      eval = alpha_beta_pruning(child, alpha, beta, False)

      max_eval = max(max_eval, eval)

      alpha = max(alpha, eval)  # Maximize alpha

      if beta <= alpha:  # Prune the branch

        break

    return max_eval

  else:

    min_eval = float('inf')

    for child in node:  # Iterate over children of the minimizer node

      eval = alpha_beta_pruning(child, alpha, beta, True)

      min_eval = min(min_eval, eval)

      beta = min(beta, eval)  # Minimize beta

      if beta <= alpha:  # Prune the branch
```

```
            break
        return min_eval


# Function to build the tree from a list of numbers
def build_tree(numbers):
    # We need to build a tree with alternating levels of maximizers and minimizers
    # Start from the leaf nodes and work up
    current_level = [[n] for n in numbers]

    while len(current_level) > 1:
        next_level = []
        for i in range(0, len(current_level), 2):
            if i + 1 < len(current_level):
                next_level.append(current_level[i] + current_level[i + 1])  # Combine two nodes
            else:
                next_level.append(current_level[i])  # Odd number of elements, just carry forward
        current_level = next_level

    return current_level[0]  # Return the root node, which is a maximizer


# Main function to run alpha-beta pruning
def main():
    # Input: User provides a list of numbers
    numbers = list(map(int, input("Enter numbers for the game tree (space-separated): ").split()))
```

```python
    # Build the tree with the given numbers
    tree = build_tree(numbers)

    # Parameters: Tree, initial alpha, beta, and the root node is a maximizing player
    alpha = -float('inf')
    beta = float('inf')
    maximizing_player = True  # The root node is a maximizing player

    # Perform alpha-beta pruning and get the final result
    result = alpha_beta_pruning(tree, alpha, beta, maximizing_player)

    print("Final Result of Alpha-Beta Pruning:", result)

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

```
Enter numbers for the game tree (space-separated): 10 9 14 18 5 4 50 3
Final Result of Alpha-Beta Pruning: 50
```