CSE-241 Artificial Intelligence

Name: Shruti T. Avhad
Roll No.: 20074030
Branch: Computer Science and Engineering (IDD)
Email Id: shruti.tavhad.cse20@itbhu.ac.in

Q1.) Explain how Simulated Annealing is better than Hill Climbing.
Ans-

Using Hill Climbing, we reach an optimum value by checking if its current state has the best cost/score in its neighborhood.This method will quickly find a local maximum/minimum but the local optimum may not be the global optimum. So as we move for the value which is better than the current state, this makes it prone to getting stuck in local optima.

Simulated Annealing is considered as an optimized way of hill-climbing because, in this method, we attempt to overcome this problem by choosing a "bad" move every once in a while. The probability of choosing a "bad" depends on the temperature of the current state. Temperature decreases as time moves on, and eventually, Simulated Annealing becomes Hill Climbing.

Generally, simulated Annealing has better chances of finding global optima than hill-climbing, whereas hill-climbing sticks on an optima nearest to the starting state.

Q2.) Solve the nQueen's problem using Hill Climbing algorithm.

Ans-

(n-queen_hill-climbing.cpp file is also submitted with this file)

```cpp
#include <bits/stdc++.h>
#define Max 500
using namespace std;
int n;
void Board_print(int board[][Max])
{
   for (int i = 0; i < n; i++)
   {
       cout << " ";
       for (int j = 0; j < n; j++)
      {
           cout << board[i][j] << " ";
       }
       cout << "\n";
   }
}
bool Compare_States(int *state1, int *state2)
{

   for (int i = 0; i < n; i++)
   {
       if (state1[i] != state2[i])
       {
           return false;
       }
   }
   return true;
}
```

```
void randomConfig(int board[][Max], int *state)
{

    for (int i = 0; i < n; i++)
    {
        state[i] = rand() % n;
        board[state[i]][i] = 1;
    }
}

void fill(int board[][Max], int value)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            board[i][j] = value;
        }
    }
}

int calculateObjective(int board[][Max], int *state)
{
    int attacking = 0;

    int row, col;

    for (int i = 0; i < n; i++)
    {

        row = state[i], col = i - 1;
        while (col >= 0 && board[row][col] != 1)
        {
            col--;
        }
        if (col >= 0 && board[row][col] == 1)
        {
            attacking++;
        }

        row = state[i], col = i + 1;
        while (col < n && board[row][col] != 1)
        {
            col++;
        }
        if (col < n && board[row][col] == 1)
        {
            attacking++;
        }

        row = state[i] - 1, col = i - 1;
        while (col >= 0 && row >= 0 && board[row][col] != 1)
        {
            col--;
            row--;
        }
        if (col >= 0 && row >= 0 && board[row][col] == 1)
        {
            attacking++;
        }

        row = state[i] + 1, col = i + 1;
        while (col < n && row < n && board[row][col] != 1)
        {
            col++;
            row++;
        }
        if (col < n && row < n && board[row][col] == 1)
        {
            attacking++;
        }

        row = state[i] + 1, col = i - 1;
        while (col >= 0 && row < n && board[row][col] != 1)
        {
            col--;
            row++;
        }
        if (col >= 0 && row < n && board[row][col] == 1)
        {
            attacking++;
        }

        row = state[i] - 1, col = i + 1;
        while (col < n && row >= 0 && board[row][col] != 1)
        {
            col++;
            row--;
        }
        if (col < n && row >= 0 && board[row][col] == 1)
        {
            attacking++;
        }
    }

    return (int)(attacking / 2);
}

void generateBoard(int board[][Max], int *state)
{

    fill(board, 0);
    for (int i = 0; i < n; i++)
    {
        board[state[i]][i] = 1;
    }
}

void Copy_State(int *state1, int *state2)
{

    for (int i = 0; i < n; i++)
    {
        state1[i] = state2[i];
    }
}

void Get_Neighbours(int board[][Max], int *state)
{

    int board_op[Max][Max];
    int state_op[n];

    Copy_State(state_op, state);
    generateBoard(board_op, state_op);

    int opObjective = calculateObjective(board_op, state_op);

    int NeighbourBoard[Max][Max];
    int NeighbourState[n];

    Copy_State(NeighbourState, state);
    generateBoard(NeighbourBoard, NeighbourState);

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (j != state[i])
            {

                NeighbourState[i] = j;
                NeighbourBoard[NeighbourState[i]][i] = 1;
                NeighbourBoard[state[i]][i] = 0;

                int temp = calculateObjective(NeighbourBoard, NeighbourState);

                if (temp <= opObjective)
                {
                    opObjective = temp;
                    Copy_State(state_op, NeighbourState);
                    generateBoard(board_op, state_op);
                }

                NeighbourBoard[NeighbourState[i]][i] = 0;
                NeighbourState[i] = state[i];
                NeighbourBoard[state[i]][i] = 1;
            }
        }
    }

    Copy_State(state, state_op);
    fill(board, 0);
    generateBoard(board, state);
}

void hillClimbing(int board[][Max], int *state)
{

    int neighbourBoard[Max][Max] = {};
    int neighbourState[n];

    Copy_State(neighbourState, state);
    generateBoard(neighbourBoard, neighbourState);

    while (true)
    {

        Copy_State(state, neighbourState);
        generateBoard(board, state);

        Get_Neighbours(neighbourBoard, neighbourState);

        if (Compare_States(state, neighbourState))
        {

            Board_print(board);
            break;
        }
        else if (calculateObjective(board, state) == calculateObjective(neighbourBoard,
neighbourState))
        {
            neighbourState[rand() % n] = rand() % n;
            generateBoard(neighbourBoard, neighbourState);
        }
    }
}

int main()
{
    cout << "Enter the board size: ";
    cin >> n;
    if (n < 4)
    {
        cout << "\nNo possible solution\n";
        return 0;
    }

    int board[Max][Max] = {};
    int state[Max] = {};


    randomConfig(board, state);


    hillClimbing(board, state);


    return 0;
}
```