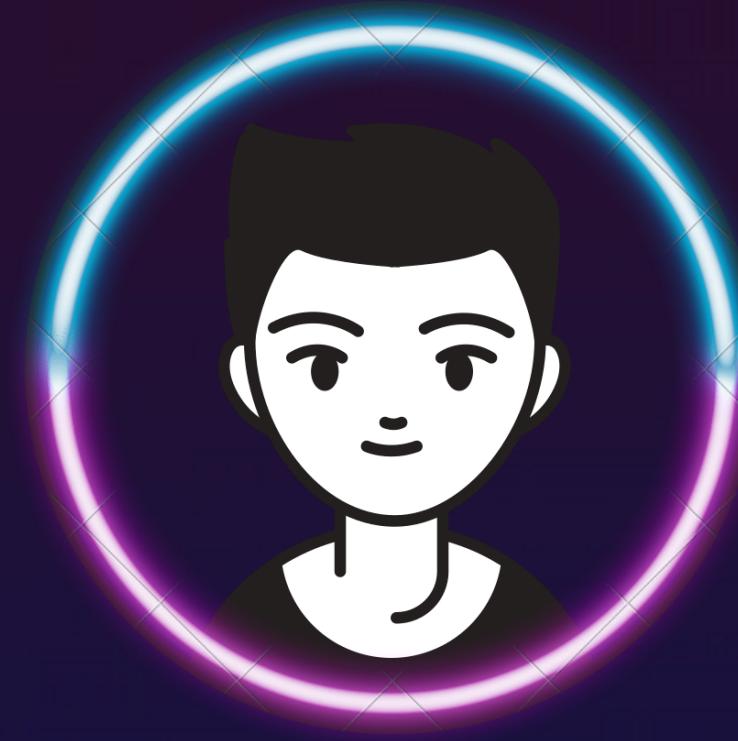


8-QUEENS PROBLEM WITH LOCAL SEARCH

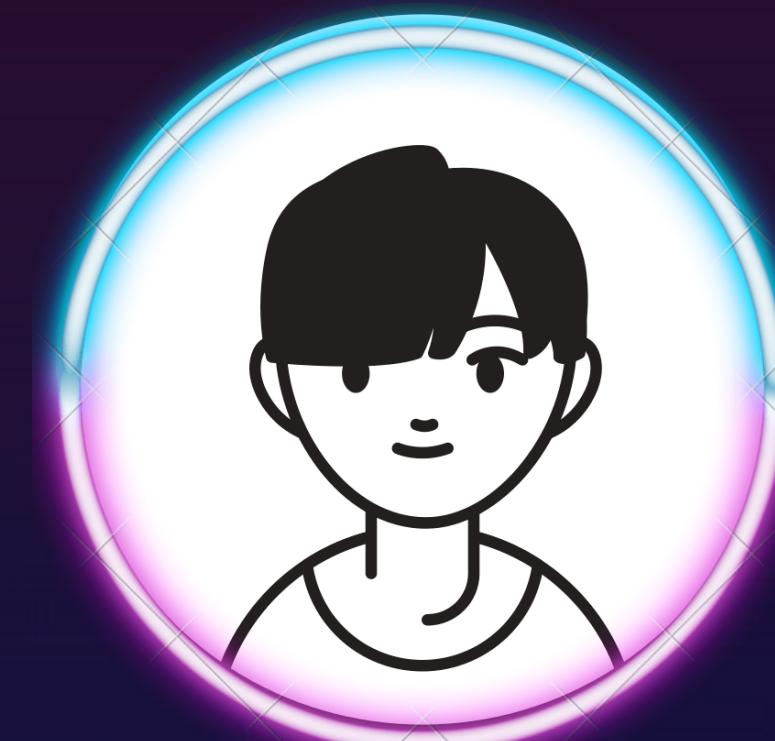
SIGN IN



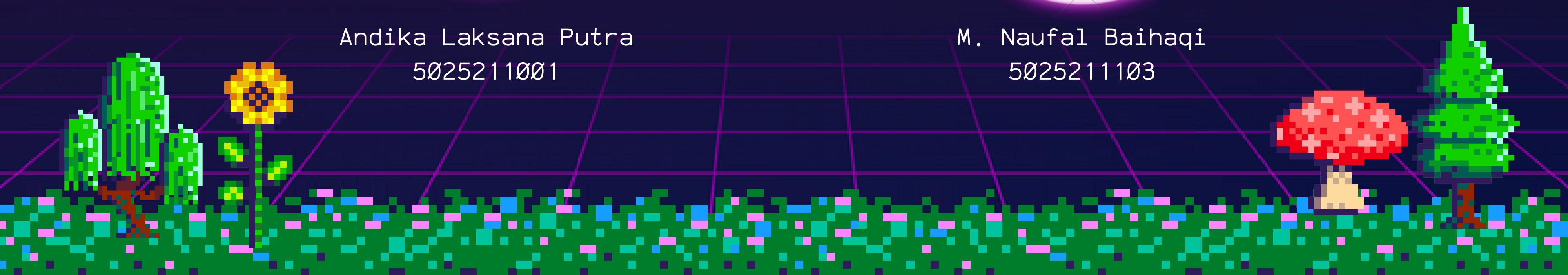
OUR AMAZING TEAM



Andika Laksana Putra
5025211001



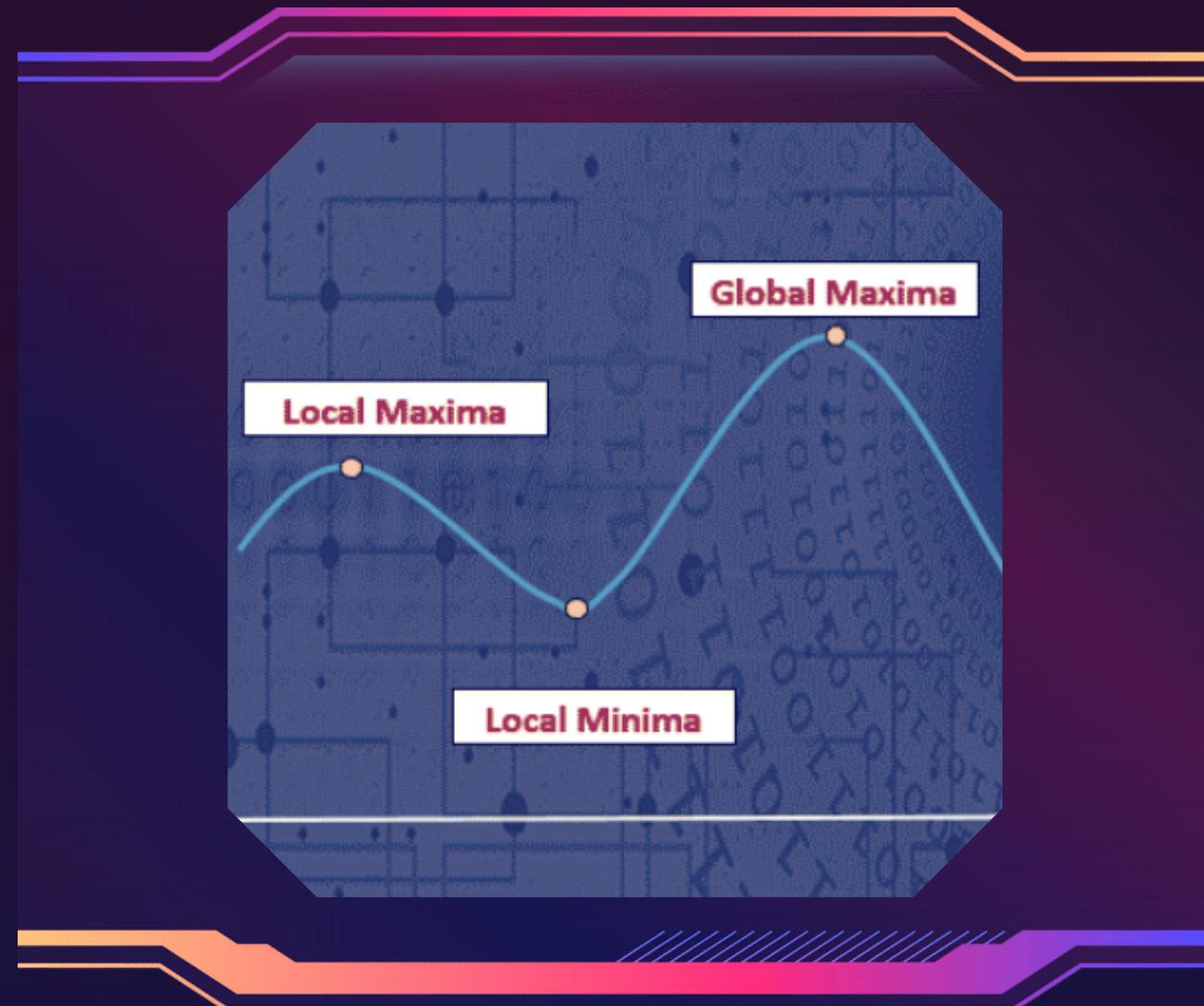
M. Naufal Baihaqi
5025211103



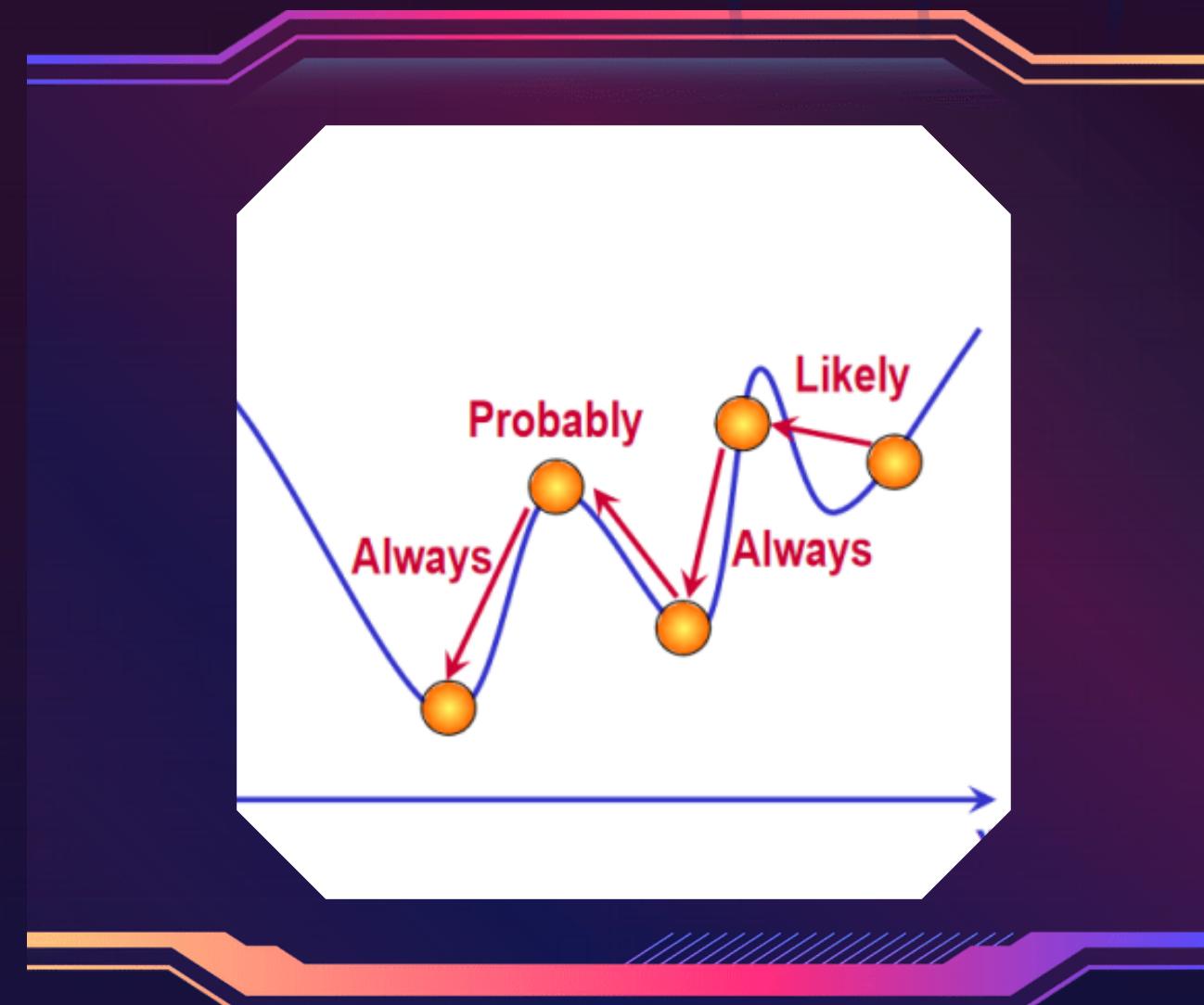
LOCAL SEARCH

Algoritma Local Search merupakan sebuah teknik dalam pemecahan masalah yang berfokus pada memperbaiki solusi secara iteratif dengan membuat perubahan kecil pada solusi yang sedang digunakan hingga tidak mungkin lagi memperbaiki solusi dengan perubahan kecil.

JENIS LOCAL SEARCH

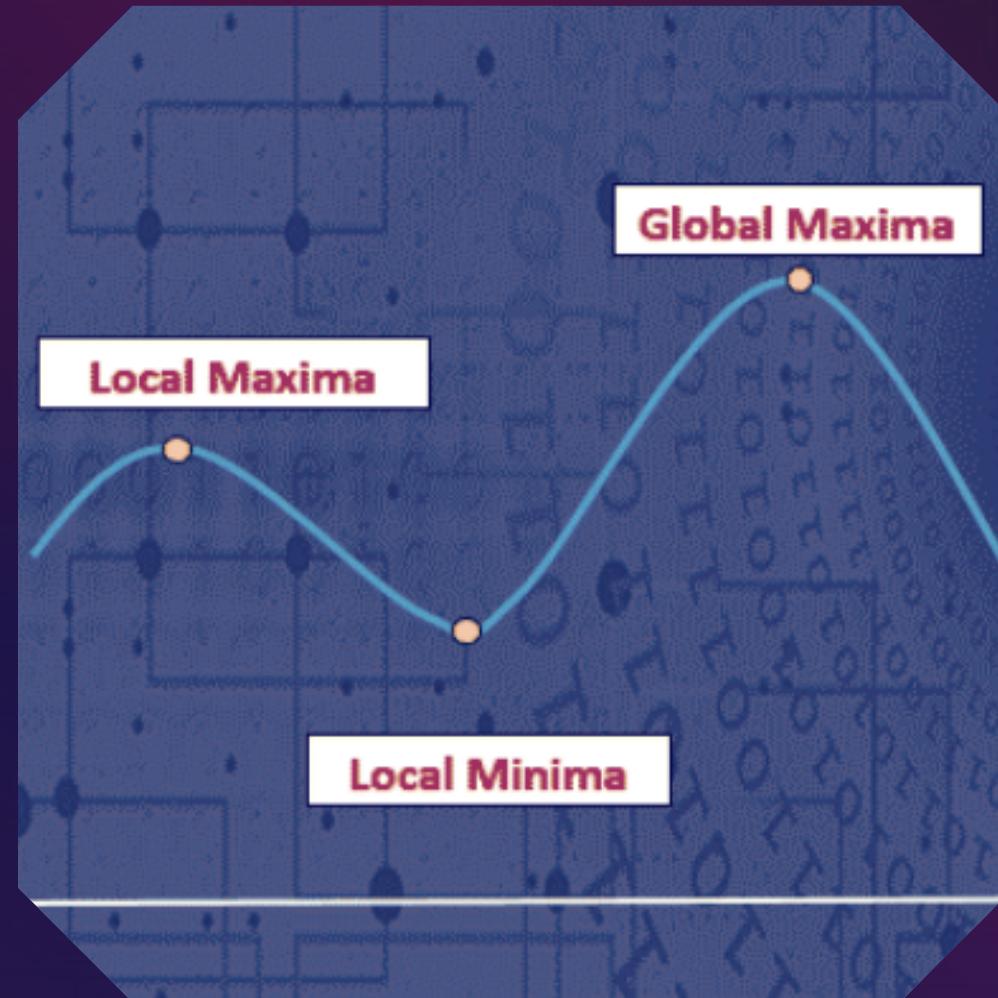


HILL CLIMBING
ALGORITHM



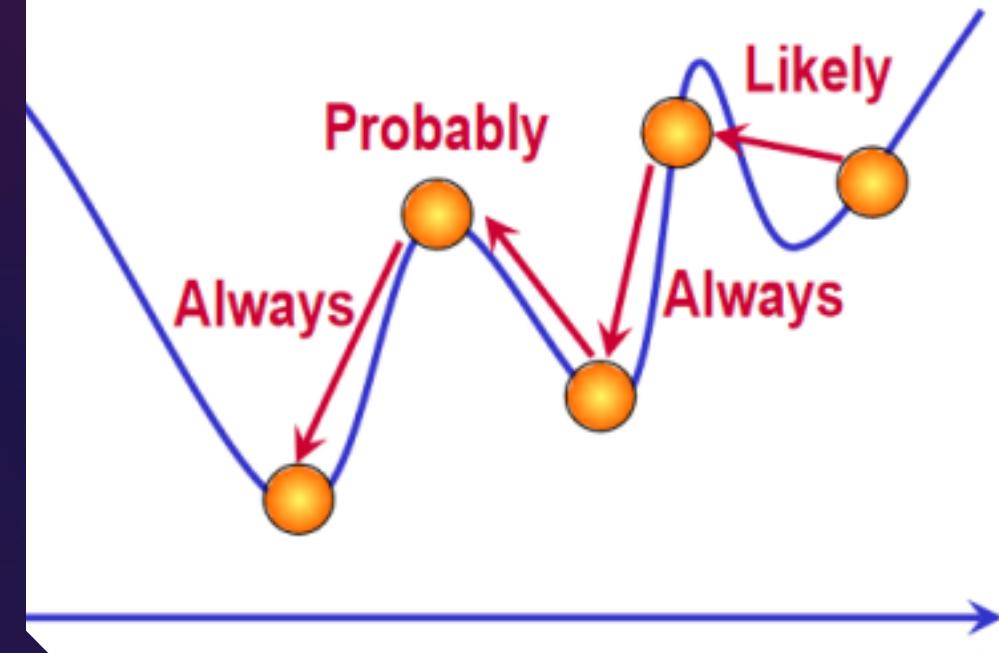
SIMULATED ANNEALING
ALGORITHM

HILL CLIMBING



Hill climbing adalah salah satu algoritma optimasi yang digunakan untuk mencari solusi terbaik dalam masalah yang didefinisikan dalam ruang pencarian. Algoritma ini bekerja secara iterasi dengan kontinu bergerak kearah peningkatan nilai (mendaki) dan berhenti ketika mencapai puncak.

SIMULATED ANNEALING



Simulated Annealing adalah salah satu algoritma optimasi yang digunakan untuk menemukan solusi terbaik dalam ruang pencarian yang kompleks. Algoritma ini bekerja dengan cara meniru proses annealing (penyusunan ulang) pada logam, di mana logam dipanaskan terlebih dahulu dan kemudian didinginkan secara perlahan-lahan untuk mengurangi energi dan meningkatkan struktur kristalnya.

IMPLEMENTASI CODE HILL CLIMBING

START



configureRandomly()

```
void configureRandomly(int board[][N],int *state) {  
    srand(time(0));  
  
    for (int i = 0; i < N; i++) {  
        state[i] = rand() % N;  
        board[state[i]][i] = 1;  
    }  
}
```

menginisialisasi keadaan awal (`state`) secara acak. Pada fungsi ini, setiap queen ditempatkan secara acak di kolom pada baris tertentu sehingga tidak ada dua ratu yang berada pada baris dan kolom yang sama. Fungsi ini menggunakan fungsi `srand()` dan `rand()` dari library `<cstdlib>` dan `<ctime>` untuk memastikan bahwa angka acak yang dihasilkan selalu berbeda pada setiap program dijalankan.

Utilities

```
void printState(int *state) {  
  
    for (int i = 0; i < N; i++) {  
        cout << " " << state[i] << " ";  
    }  
    cout << endl;  
}  
  
bool compareStates(int *state1, int *state2) {  
  
    for (int i = 0; i < N; i++) {  
        if (state1[i] != state2[i]) {  
            return false;  
        }  
    }  
    return true;  
}  
  
void fill(int board[][N], int val)  
{  
    for (int i = 0; i < N; i++) {  
        for (int j = 0; j < N; j++) {  
            board[i][j] = val;  
        }  
    }  
}  
  
void generateBoard(int board[][N], int *state) {  
    fill(board, 0);  
    for (int i = 0; i < N; i++) {  
        board[state[i]][i] = 1;  
    }  
}  
  
void copyState(int *state1, int *state2) {  
  
    for (int i = 0; i < N; i++) {  
        state1[i] = state2[i];  
    }  
}
```

- **printState()** digunakan untuk mencetak state ke layar.
- **compareStates()** digunakan untuk membandingkan dua state dan mengembalikan nilai boolean.
- **fill()** digunakan untuk mengisi papan catur dengan nilai tertentu
- **printBoard()** digunakan untuk mencetak papan catur ke layar
- digunakan untuk menghasilkan papan catur dari suatu keadaan (state) dan menggunakan fungsi **fill()**
- **copyState()** : digunakan untuk menyalin keadaan (state) dari satu array ke array lainnya



```
int calculateObjective(int board[][N], 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++;  
        }  
    }  
}
```

... AND MORE

calculateObjective()

Fungsi ini digunakan untuk menghitung jumlah "attack" yang terjadi pada keadaan (state) tertentu. Fungsi ini menggunakan perulangan for untuk menghitung setiap queen pada setiap baris dan kolom pada keadaan tersebut, dan mencari jumlah serangan yang terjadi pada queen tersebut dengan menghitung jumlah queen lain yang berada pada baris yang sama, kolom yang sama, atau diagonal dengan queen tersebut.

getNeighbour()

```
void getNeighbour(int board[][], int *state) {  
  
    int opBoard[N][N];  
    int opState[N];  
    copyState(opState, state);  
    generateBoard(opBoard, opState);  
  
    int opObjective = calculateObjective(opBoard, opState);  
  
    int NeighbourBoard[N][N];  
    int NeighbourState[N];  
  
    copyState(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,  
                if (temp <= opObjective) {  
  
                    opObjective = temp;  
                    copyState(opState, NeighbourState);  
                    generateBoard(opBoard, opState);  
                }  
            }  
        }  
    }  
}
```

... AND MORE

menghasilkan solusi-solusi yang berbeda dari solusi sebelumnya. fungsi ini melakukan looping pada setiap kolom pada papan catur dan mencoba untuk memindahkan ratu ke kolom lain pada baris yang sama. Jika setelah dipindahkan, posisi ratu yang baru tidak menyebabkan serangan horizontal atau diagonal terhadap ratu yang lain, maka solusi yang dihasilkan akan menjadi nilai "State" yang baru. Jika tidak maka nilai "state" akan sama dengan sebelumnya

hillClimbing()

```
void hillClimbing(int board[][N], int *state) {  
  
    int neighbourBoard[N][N] = {};  
    int neighbourState[N];  
  
    copyState(neighbourState, state);  
    generateBoard(neighbourBoard, neighbourState);  
  
    do {  
  
        copyState(state, neighbourState);  
        generateBoard(board, state);  
  
        getNeighbour(neighbourBoard, neighbourState);  
  
        if (compareStates(state, neighbourState)) {  
            printBoard(board);  
            break;  
        } else if (calculateObjective(board, state) ==  
                   calculateObjective(neighbourBoard, neighbourState)) {  
  
            neighbourState[rand() % N] = rand() % N;  
            generateBoard(neighbourBoard, neighbourState);  
        }  
    } while (true);  
}
```

menemukan solusi yang lebih baik dari solusi sebelumnya dengan menggunakan metode hill climbing. Fungsi akan melakukan looping sebanyak mungkin untuk mencari solusi terbaik. Dalam setiap iterasi, fungsi akan memanggil fungsi "getNeighbour()" untuk mendapatkan "Neighbour" dari solusi sebelumnya dan membandingkan jumlah serangan pada "Neighbour" terbaru dengan jumlah serangan pada solusi sebelumnya. Jika "Neighbour" memiliki jumlah serangan yang lebih sedikit daripada solusi sebelumnya, maka hillClimbing() akan memperbarui "state" dan board dengan "Neighbour" yang baru ditemukan

IMPLEMENTASI CODE SIMULATED ANNEALING

START

init, calculateConflict, generate neighbor

```
1 #include <iostream>
2 #include <cstdlib>
3 #include <cmath>
4 #include <ctime>
5 #include <cstring>
6 using namespace std;
7
8 const int N = 8; // Size of the board
9 const int MaxIterations = 100000; // Maximum number of iterations
10 const double InitialTemperature = 1000.0; // Initial temperature
11 const double CoolingRate = 0.99; // Cooling rate
12
13 // Initialize the board with a random placement of the queens
14 void InitializeBoard(int board[])
15 {
16     for (int i = 0; i < N; i++) {
17         board[i] = rand() % N;
18     }
19 }
20
21 // Calculate the number of conflicts (pairs of queens that attack each other)
22 int CalculateConflicts(int board[])
23 {
24     int conflicts = 0;
25     for (int i = 0; i < N; i++) {
26         for (int j = i + 1; j < N; j++) {
27             if (board[i] == board[j] || abs(i - j) == abs(board[i] - board[j])) {
28                 conflicts++;
29             }
30         }
31     }
32     return conflicts;
33 }
34
35 // Generate a new candidate solution by randomly moving one queen to a new row in its column
36 void GenerateNeighbor(int board[])
37 {
38     int column = rand() % N;
39     int row = rand() % N;
40     board[column] = row;
41 }
```

- menginisialisasi keadaan awal (state) secara acak.
- menghitung conflict yang terjadi sebagai pertimbangan efektivitas pada fungsi simulated annealing
- men-generate kandidat solusi terbaru

Fungsi Simulated Annealing

```
// Simulated Annealing algorithm to solve the 8-Queen problem
void SimulatedAnnealing(int board[])
{
    double temperature = InitialTemperature;
    int currentConflicts = CalculateConflicts(board);
    for (int i = 0; i < MaxIterations && currentConflicts > 0; i++) {
        int newBoard[N];
        memcpy(newBoard, board, N * sizeof(int));
        GenerateNeighbor(newBoard);
        int newConflicts = CalculateConflicts(newBoard);
        int delta = newConflicts - currentConflicts;
        if (delta < 0 || exp(-delta / temperature) > (double)rand() / RAND_MAX) {
            memcpy(board, newBoard, N * sizeof(int));
            currentConflicts = newConflicts;
        }
        temperature *= CoolingRate;
    }
}
```

- Dalam setiap iterasi, buat sebuah solusi baru dengan memindahkan satu ratu ke baris acak pada kolom yang sama. Lalu menghitung conflict pada solusi terbaru tersebut
- Membandingkan conflict dengan solusi sebelumnya untuk memutuskan apakah solusi terbaru akan dipakai. Bukan hanya itu, dipertimbangkan juga nilai dari perbandingan probabilitas solusi baru dengan nilai random.
- temperature terus diupdate untuk memastikan nilai probabilitas semakin mengecil

PRINTBOARD & MAIN FUNCTION

- melakukan output dari setiap board yang dipanggil
- dimulai dengan
- srand(time(NULL)) pada main() digunakan untuk memberikan nilai awal pada generator bilangan acak (rand()) agar menghasilkan deret bilangan acak yang berbeda-beda setiap kali program dijalankan.

```
62 // Print the board
63 void PrintBoard(int board[])
64 {
65     for (int i = 0; i < N; i++) {
66         for (int j = 0; j < N; j++) {
67             if (board[j] == i) {
68                 cout << "Q ";
69             } else {
70                 cout << ". ";
71             }
72         }
73         cout << endl;
74     }
75 }
76
77 int main()
78 {
79     srand(time(NULL));
80     int board[N];
81     InitializeBoard(board);
82     cout << "Initial board:" << endl;
83     PrintBoard(board);
84     SimulatedAnnealing(board);
85     cout << "Final board:" << endl;
86     PrintBoard(board);
87     return 0;
88 }
```



THANK YOU