

بسم الله الرحمن الرحيم



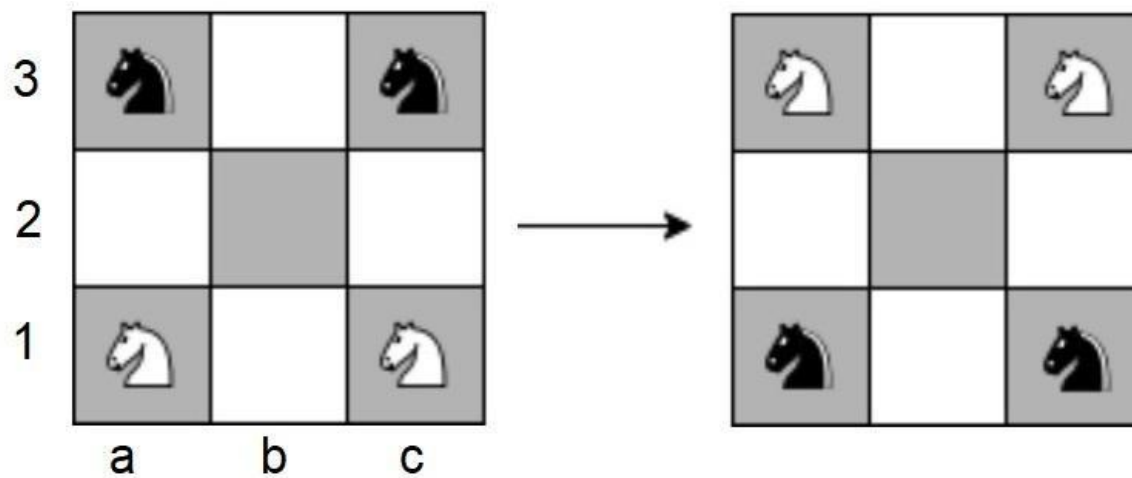
Abdalrahim Sawalha 211081

Ahmad Zaqeeq 211043

Course : Intelligent Systems .

Dr. Hashem Altamime.

GA for the 4 knights problem



### 1. What is the representation of a chromosome, and what is its length? Why?

The length of the chromosome will be 16 genes, because the minimum possible movements to solve this problem is 16 different movements (we took into account that the possible movement is in the form of the letter L and also the number of squares "the number of spaces and the general shape consisting of 3 x 3" we took all of them into account) and for this reason it turned out that the least possible number or the minimum possible movements (genes "chromosome length") is 16 genes, and they are in the following form represented in the chromosome below:

A3	A1	C1	C3	C2	B3	A2	B1	A1	C1	C3	A3	B3	A2	B1	C2
to	to	To	to	to	to	to	to	to	to	to	to	to	to	to	to
C2	B3	A2	B1	A1	C1	C3	A3	B3	A2	B1	C2	C1	C3	A3	A1

### Parent 1:

A3	A1	C3	C1	C3	B3	A2	B1	A1	C1	C3	A3	B1	A2	B1	C3
to	to	To	to	to	to	to	to	to	to	to	to	to	to	to	to
C3	B2	A1	B1	A3	C2	C3	A3	B3	A1	B2	C2	C1	C3	A1	A1

Parent 2:

A1	C1	C2	A2	A3	B3	A2	B1	A1	A1	C3	A3	B1	A2	A2	C3
to	to	To	to	to	to	to	to	to	to	to	to	to	to	to	to
B3	A2	B1	B3	C3	C2	C3	A3	B3	B3	B2	C2	C1	C3	C3	A1

2. Give an example of two parents and their children after crossover. Explain.

Parent 1:

A3	A1	C3	C1	C3	B3	A2	B1	A1	A1	C3	A3	B1	A2	A2	C3
to	to	To	to	to	to	to	to	to	to	to	to	to	to	to	to
C3	B2	A1	B1	A3	C2	C3	A3	B3	B3	B2	C2	C1	C3	C3	A1

Parent 2:

A1	C1	C2	A2	A3	B3	A1	C3	C1	A3	B1	A3	B1	A3	B1	A1
to	to	To	to	to	to	to	to	to	to	to	to	to	to	to	to
B3	A2	B1	B3	C3	C2	B3	B1	A1	C3	A2	C3	C1	C2	A1	C3

After Crossover:

Child 1 :

A3	A1	C3	C1	C3	B3	A1	C3	C1	A3	B1	A3	B1	A3	B1	A1
to	to	To	to	to	to	to	to	to	to	to	to	to	to	to	to
C3	B2	A1	B1	A3	C2	B3	B1	A1	C3	A2	C3	C1	C2	A1	C3

Child 2 :

A1	C1	C2	A2	A3	B3	A1	C3	C1	A3	B1	A3	B1	A3	B1	A1
to	to	To	to	to	to	to	to	to	to	to	to	to	to	to	to
B3	A2	B1	B3	C3	C2	B3	B1	A1	C3	A2	C3	C1	C2	A1	C3

# Mutation → swap :

From child 2 :- A2→B3 will swap with B1→A3 , so the new chromosome will look like this:

A1	C1	C2	A2	A3	B3	A2	B1	A1	A1	C3	A3	B1	A2	A2	C3
to	to	To	to	to	to	to	to	to	to	to	to	to	to	to	to
B3	A2	B1	B3	C3	C2	C3	A3	B3	B3	B2	C2	C1	C3	C3	A1

After Mutation :-

A1	C1	C2	B1	A3	B3	A2	A2	A1	A1	C3	A3	B1	A2	A2	C3
to	to	To	to	to	to	to	to	to	to	to	to	to	to	to	to
B3	A2	B1	A3	C3	C2	C3	B3	B3	B3	B2	C2	C1	C3	C3	A1

Explain:-

We do the mutation process in order to get new children with new characteristics other than those inherited by their parents. Therefore, we do this process because it is very important in reaching the correct solution, as it may lead me to the best result and the best performance of the algorithm, so that we may have reached the solution through this mutation that we are doing.

# Fitness Function :-

```
int Fitness(string movements[]) {  
  
    const int targetConfiguration[9] = { 1, 0, 1, 0, 0, 0, 2, 0, 2 };  
    int currentConfiguration[9] = { 2, 0, 2, 0, 0, 0, 1, 0, 1 };  
    int totalFitnessScore = 0;  
    int moveIndex = 0;  
    while (moveIndex < 16) {  
        // extract starting and destination positions from movements array  
        char startColumn = movements[moveIndex][5]; // get starting column  
        char startRow = movements[moveIndex][6];    // get starting row  
        string startingflosition = string(1, startColumn) + startRow; // combine  
column with row  
  
        char targetColumn = movements[moveIndex][11]; //target column  
        char targetRow = movements[moveIndex][12];    //target row  
        string destinationflosition = string(1, targetColumn) + targetRow; // combine  
column with row  
  
        // transform starting position to index  
        int startColIndex = startColumn - 'a'; // 'a' corresponds to 0  
        int startRowIndex = 3 - (startRow - '1' + 1); // example: '1' corresponds to  
2  
  
        int startIndex = startRowIndex * 3 + startColIndex;  
  
        // transform destination position to index  
        int endColIndex = targetColumn - 'a'; // 'a' corresponds to 0  
        int endRowIndex = 3 - (targetRow - '1' + 1); // example: '3' corresponds to  
0  
  
        int endIndex = endRowIndex * 3 + endColIndex;  
  
        // assess row and column differences  
        int rowDifference = abs(startIndex / 3 - endIndex / 3);  
        int columnDifference = abs(startIndex % 3 - endIndex % 3);  
  
        // check move according to chess rules  
        if (((rowDifference == 2 fifi columnDifference == 1) || (rowDifference == 1 fifi  
columnDifference == 2)) fifi  
        currentConfiguration[endIndex] == 0 fifi currentConfiguration[startIndex]  
!= 0) {  
  
            currentConfiguration[endIndex] = currentConfiguration[startIndex];  
            currentConfiguration[startIndex] = 0; // clr initial position  
        }  
        else {  
            totalFitnessScore -= 2;  
        }  
  
        moveIndex++; // inc index for next move  
    }  
  
    int i = 0;  
    while (i < 9) {  
        if (currentConfiguration[i] == targetConfiguration[i]) {  
            totalFitnessScore++; // inc score for correctly placed pieces  
        }  
        i++;  
    }  
  
    return totalFitnessScore; // Return final fitness score  
}
```

# Explain :-

## Definitions and Initialization:

**targetConfiguration[9]:** An array that defines how the final desired state of the board should look. Each index corresponds to a position on the 3x3 board, where specific numbers may represent specific pieces.

**currentConfiguration[9]:** An array that holds the current state of the board. It initializes with specific values representing where the pieces are placed at the start.

**totalFitnessScore:** A variable to keep track of the score as movements are validated and assessed. It starts at zero.

**moveIndex:** An index variable initialized to zero to track how many moves have been processed.

## Movement Processing Loop:

The while (`moveIndex < 16`) loop processes up to 16 moves (assumed from the movements array). For each move, the function performs the following steps:

**Extract Starting and Destination Positions:**

The starting position is obtained using `movements[moveIndex][5]` and `movements[moveIndex][6]`, which represent the column and row of the piece being moved. The target position is similarly extracted from `movements[moveIndex][11]` and `movements[moveIndex][12]`.

**Transform Positions to Indices:**

The starting and destination positions, which are in the format like "a1" or "b3", are converted to indices that can be used in the array representation of the board.

The transformation takes into account the mapping of columns ('a', 'b', 'c') to 0, 1, and 2 respectively and maps the rows accordingly, adjusting for the inverted row numbering.

**Assess Row and Column Differences:**

It calculates how many rows and columns the piece moves, which is used to validate the move against chess-like rules.

### Check Move Validity:

The function checks if the move follows a knight's movement pattern (L-shape: two squares in one direction and one square perpendicular). It also checks if the start position has a piece (non-zero value in `currentConfiguration`) and if the end position is empty (zero in `currentConfiguration`).

If the move is valid, it updates the `currentConfiguration` and clears the previous position of the moved piece. If invalid, it penalizes the score by subtracting 2.

### Fitness Score Calculation:

After processing all movements, the function evaluates how many pieces in the `currentConfiguration` match the `targetConfiguration`.

It uses a second while loop to iterate through the 9 positions. For each position that matches the target, the `totalFitnessScore` is incremented by 1.

### Return Value:

The function returns the total fitness score, reflecting how effectively the movements have aligned the current state with the desired state.