

Ministry of Science and Higher Education of the Russian Federation

National Research University ITMO

Evolutionary calculations

Spring

2024

Laboratory work No. 5

DESIGN OF AN EVOLUTIONARY ALGORITHM FOR THE PROBLEM OF QUEEN PLACEMENT

Steps of implementing the work:

1. Select a framework (you can continue with Watchmaker) for the algorithm or make your own implementation.

After implementing the previous labs, we saw that the watchmaker is a good framework that provides a lot of utilities for building the project, so it will be chosen for implementing the work.

2. Select an evolutionary algorithm (GA or other).

Since the class is called (genetic algorithms), then we will choose this kind of evolutionary algorithm to solve the problem.

3. Analyze the problem, highlight optimization criteria or limitations.

- Analyzing the problem:

According to the requirements, we are supposed to build a system that finds a solution for the queen placement problem using an Evolutionary algorithm.

The problem is about placing the maximum possible number of queens in a way that none of them threaten any other queen on a board of size $n \times n$.

- Optimization criteria and limitations:

We know that queens can attack vertically, horizontally, or diagonally along the board, and therefore, queens should not be on the same row, column, or diagonal line. So we can clearly see that at each row (and each column) of the board, there will be at maximum one queen.

4. Design a solution to the problem, type of solution, ways of variations and handling possible restrictions.

- Since the board is a chess board with an organized grid, where queens can be placed on squares, then it will be a good choice to represent the board with a 2D-array $n \times n$. And at each cell, there will be a value showing whether there will be a queen or not.

- But as mentioned in the limitations, at each row of the board, there will be at maximum one queen, so it will be better and less space consuming to choose to represent the placement of queens using a 1D-array of length n , where at the index i of this array, we will have a value v . This value v is the number of column where the queen is located on the i 'th row (queen at the index $[i][v]$), so the value v can vary between 1 and n .

- in some cases, there might be no suitable place for the queen on the row (when $n=2$ for example, we can place only one queen on the board), so this means that we will need another value at the index i to show that there won't be any queen on this row, and therefore, values will vary between 0 and n . So if $v_i=0$, then there won't be any queen at the i 'th row, otherwise, it will be at column $v[i]$. So as a conclusion for the previous analysis, the representation of solutions (individuals) will be a 1D array of length n , and values of each element of this array will be integers in the range $(0, n)$.

5. Generate conditions for termination or convergence to the algorithm.

- First, we will start by the generating the class named **QueenFactory**, which is simply responsible for creating the candidate individuals for the first generation, so the only restriction so far is to create individuals that will have random integer genes values between 0 and boardSize included (boardSize+1).

```
for (int i = 0; i < boardSize; i++) {  
    // Generate a random column index for placing the queen  
    candidate[i] = random.nextInt((boardSize+1));  
}
```

- Next, we will implement the **QueenMutation** class, that will mutate individuals to create the new generation, in this code we also need only to put the same restriction that was mentioned before, which sets limit for new values of genes.

```
int rowIndex = random.nextInt(boardSize);  
int newColumnIndex = random.nextInt( bound: boardSize + 1);  
candidate[rowIndex] = newColumnIndex;  
}
```

- The other operator is **QueenCrossover**. By taking a look on the problem we're having, we can see that the arithmetic crossover or any other type that includes changing genes is not a good choice, because we can notice that the genes values are depending on each other, so a good choice is the Partially Mapped Crossover (PMX), which applies the following:

- A) Two crossover points are randomly selected.
- B) Genes between these crossover points are swapped between the parents to create offspring.
- C) Any duplicate genes are swapped according to a mapping between the parents' chromosomes (I thought that this point will give much better performance, but surprisingly, it wasn't. so this one is not applied in the code).

```
for (int i = crossoverPoint1; i <= crossoverPoint2; i++) {  
    while (child1[index1] != null) {  
        index1 = (index1 + 1) % boardSize; // Wrap around if needed.  
    }  
    while (child2[index2] != null) {  
        index2 = (index2 + 1) % boardSize; // Wrap around if needed.  
    }  
    child1[index1] = parent2[i];  
    child2[index2] = parent1[i];  
}
```

6. Establish characteristics for measuring the algorithm. efficiency

Moving to fitness function, the class **QueenFitness** was built to calculate fitness of individuals according to the following considerations between all possible pairs of queens:

- A) If two elements of the 1D-array are having the same value, this means that they are being on the same columns, so this raises a conflict.
- B) If two elements are on the same diagonal line, then the difference between their row indices will equal the difference of the column indices, and therefore, a conflict.
- C) If an element of the array is equal to zero, then the row doesn't contain a queen, which will make us avoid possible conflicts, but since we want the algorithm to maximize number of queens, then we will put penalty for missing a queen by adding penalty of $(1 + \text{boardSize})/4.5$ which is less than the penalty of a conflict when boardSize is 2 or 3 (these calculations are not precised but approximated).

Since the maximum number of conflicts is $\text{boardSize} * (\text{boardSize} - 1) / 2$, so the following equation will be used to calculate fitness:

$$\text{fitness} = 1 - \frac{2 * (\text{number of conflicts} + \text{penalty of zeros})}{\text{boardSize} * (\text{boardSize} - 1)}$$

But running the algorithm with the same parameters, we noticed that the algorithm is giving higher fitness with higher boardSize, and the reason can be seen in the previous equation, where $\text{fitness} \rightarrow 1$ when $\text{boardSize} \rightarrow \infty$ unless number of conflicts became so huge too, so we put another restriction to the fitness function, where it becomes as following:

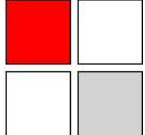
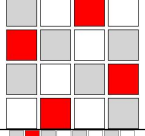
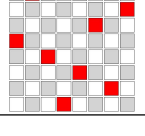
$$\text{fitness} = (1 - \frac{2 * (\text{number of conflicts} + \text{penalty of zeros})}{\text{boardSize} * (\text{boardSize} - 1)})^{\frac{\text{boardSize}}{\log(\text{boardSize})}}$$

This will guarantee us having more realistic results.

7. Configure the algorithm parameters.

So after choosing mutation and crossover techniques, the algorithm, and the fitness function, we have the following parameters: board size, population size, number of generations, mutation probability, crossover probability.

8. Carry out a series of runs at different values and analyze the effectiveness.

Board size	Population size	Generations	Mutation probability	Crossover probability	fitness	visualization
2	2	3	0.5	0.5	1	
4	2	10	0.5	0.5	1	
8	10	100	0.5	0.9	1	
16	50	700	0.5	0.9	1	
32	60	700	0.5	0.9	1	
64	200	4500	0.7	0.5	0.985	
64	200	4500	0.5	0.9	1	
128	200	4500	0.5	0.9	1	
128	200	4000	0.5	0.5	1	
256	200	1300	0.5	0.5	1	
512	200	2000	0.5	0.5	1	
512	100	4000	0.5	0.2	1	
512	100	3000	0.5	0.5	1	
512	100	2000	0.5	0.9	1	

Based on the previous results, we can notice few points here:

- 1- In this problem, the crossover operator is more important than mutation, since the placement of queens depends on each other.
- 2- With the increase of the problem size, we need more crossover operations.
- 3- The bigger the value of board size is, the more possible solutions become available.

Now after implementing the code and conducting experiments, we can answer the following questions:

Apologies for the oversight. You're correct. Let's revisit the observations and adjust the answers accordingly:

1. Is the problem optimization or constrained?

The problem is both optimization and constrained. It's optimization because we aim to find the best solution that maximizes the fitness function, representing the optimal placement of queens on a board. Besides, it's also constrained because the solution must adhere to the specific rules that we mentioned before (queens shouldn't share the same row, column, or diagonal).

2. How does the complexity of the problem increase with increasing dimension?

Initially, as the dimension (board size) increases, the complexity of the problem also increases due to the exponential growth in the search space. However, at some point with higher board sizes, we may observe that the number of feasible solutions also increases significantly. This increase in feasible solutions can lead to a phenomenon where the evolutionary algorithm finds satisfactory solutions more quickly, often requiring fewer generations to converge. This occurs because the larger solution space offers more diverse and potentially better solutions, making it easier for the algorithm to explore and exploit the search space effectively. Therefore, while the problem complexity still increases with dimension, the effectiveness of the algorithm in finding solutions may improve due to the abundance of viable options available for exploration.