8 Queens Problem

Genetic Algorithm

NOVEMBER 16

Saeed77t@gmail.com / student number: 40160957
Authored by: Reza Tahmasebi

Table of Contents

.

.

.

.

.

.

.

.

.

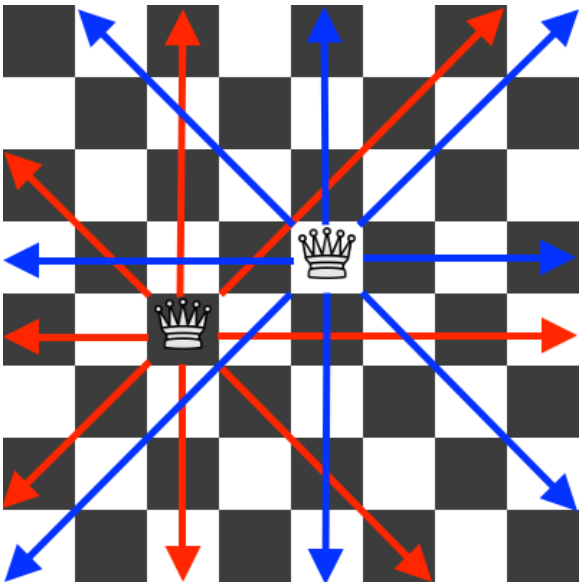
.

.

.

About 8 queens problem

This problem is about how to put eight queens on a chess board that they can not check on each other vertically, horizontally and diagonally.



Picture1: 8 queens problem

What is a genetic algorithm?

The genetic algorithm may be a strategy for solving both compelled and unconstrained optimization problems based on common choice, the method that drives natural advancement. The hereditary calculation over and over alters a populace of person arrangements. At each step, the genetic analysis chooses people from the current crowd to be guardians and employments them to deliver the children for another era. Over progressive ages, the public "advances" toward an ideal arrangement. You'll apply the hereditary calculation to unravel an assortment of optimization issues that are not well suited for standard optimization calculations, problems counting in which the objective work is spasmodic, nondifferentiable, stochastic, or profoundly nonlinear. The genetic analysis can address issues of blended numbers programming, where a few components are limited to be integer-valued.

How does code work?

Importing libraries :

To start, the *numpy* and *random* libraries have been imported. *Numpy* is used for calculating some statistics, and *random* is used for generating random numbers that we need in the code.

InitialPopulation function :

This function gets the population size we want to create and the number of queens we want to solve the problem with as arguments. It will make the initial population and genes with n chromosomes, and at the end of the chromos, it adds a 0 value for the amount of fitness at the end of the gene, so the genes it creates look like the shape below

1	8	7	2	5	6	3	4	0

The above shape looks like this on a chessboard

*							
							*
						*	
	*						
				*			
					*		
		*					
			*				

FitSort function :

This function gets the population and number of queens as arguments, and it uses a simple temp technique to sort the population from top to bottom based on the last value (fitness) of each gene.
Eventually, it returns the sorted population.

Fitness function :

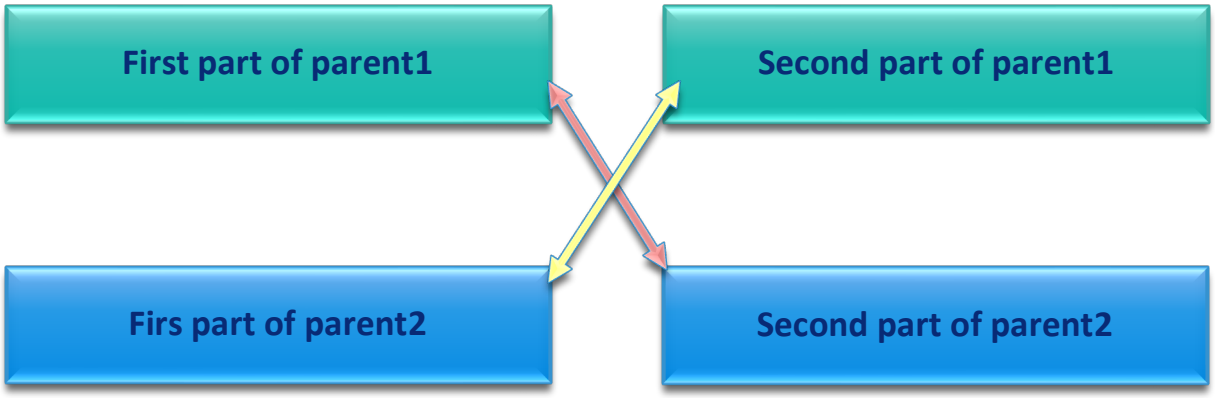
It gets the population and number of queens as arguments and calculates the fitness of each gene; the best fitness is 0 , which means that no queens check on each other, so in this algorithm, we are trying to minimize the fitness.
This function works with two conditions if a queen checks on other queens in a column or diagonally, one fitness is added to the check variable. And *FitSort function has been used to sort the population.*
In the end, it returns a calculated and sorted fitness population.

ParentSelection function :

This function gets the population as the argument, and it randomly chooses five members of the people; it sorts the five chosen members based on fitness and selects two of them that have the best fitness.
And it returns two separate genes as parents.

CrossOver function :

It gets two parents as arguments, and it will break each parent into two parts; it consists of part one of the first parent and part two of the second parent together to make a new child, and also it consists second part of the first parent and first part of the second parent together to make another child

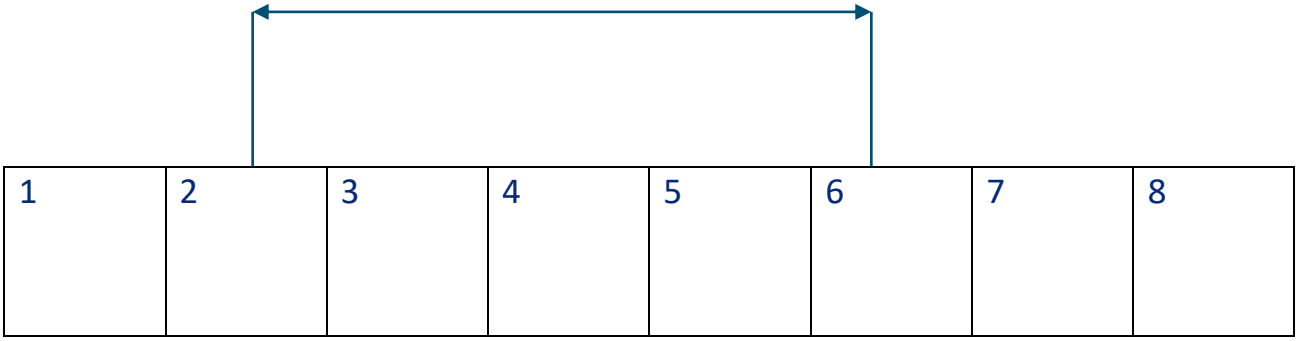


At the end of the code, the function returns two new children.

Mutation Function:

The function gets two children and a mutation rate as arguments, and by the probability of the mutation rate, selects two random chromosomes and swaps their places so that we can have two muted children.

Child :



Muted child :

1	6	3	4	5	2	7	8

SurvivalSelection function :

This function gets the population and children and the number of queens as the arguments and will add the children to the end of the people.
And then, the function uses the *FitSort process to sort the population based on fitness, and after sorting the population, it will remove the two last members of people with the least fitness.*
And it returns the whole population.

Calling the functions :

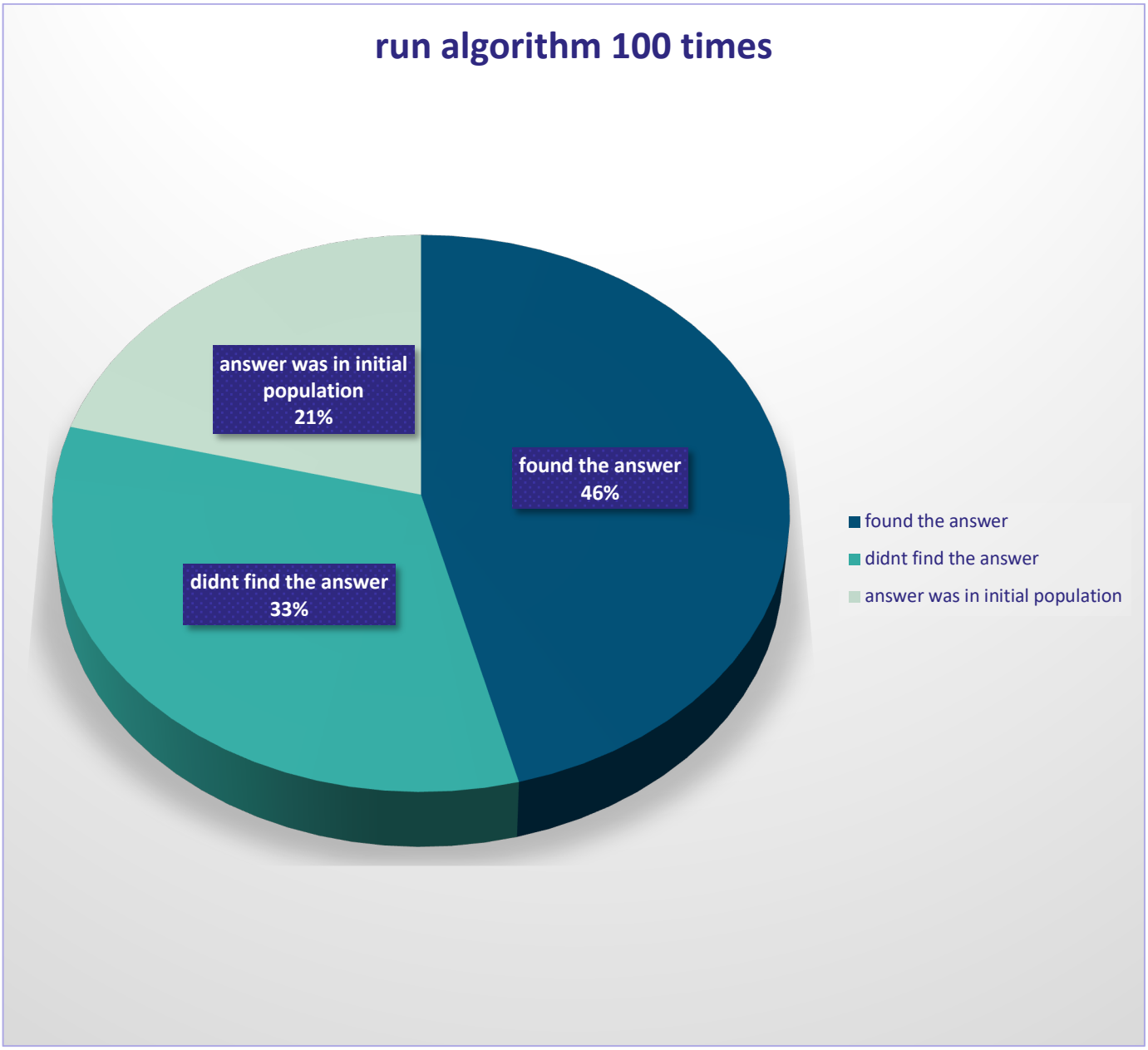
After creating the first population and calculating each member's fitness, we check if there is any answer in the initial population.
After that, for 10000 iterations, we let the rest of the program work and see if there were any answers.

Run the whole algorithm 100 times

After implementing the algorithm, I ran it 100 times to check how the algorithm worked.

The results are in the table below :

The answer was in the initial population	**21 times**
Algorithm founds the answer	46 times
Algorithm couldn’t find the answer	33 times


run algorithm 100 times
answer was in initial population 21%
found the answer 46%
didnt find the answer 33%
found the answer
didnt find the answer
answer was in initial population

The table below shows the average number and variance and the standard deviation of iterations when the algorithm found the answer:

The average number of iterations when the algorithm reaches the answer	**4351.58**
The variance of iterations when the algorithm reaches the answer	**8230845.37**
The standard deviation of iterations when the algorithm reaches the answer	**2868.94**