Genetic Algorithms

Carlos Mougan Navarro

N Queens

Submitted To:
Lluis Alseda
Research and Innovation
Mathematics Department

Submitted By :
Carlos Mougan
Msc in Modelling
CRM

Contents

1 Project Description **2**

2 Genetic Algorithms **3**

3 Code **5**

4 Results **8**

4.1 Future Improvements 8

5 Conclusions **9**

1 Project Description

The **eight queens** puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other [Wikipedia]. Thus, a solution requires that no two queens share the same row, column, or diagonal. The eight queens puzzle is an example of the more general n queens problem of placing n non-attacking queens on an n×n chessboard, for which solutions exist for all natural numbers n with the exception of n=2 and n=3.

The problem of finding all solutions to the 8-queens problem can be quite computationally expensive, as there are 4,426,165,368 possible arrangements of eight queens on an 8×8 board, but only 92 solutions. [Alseda]

In order to solve this problem we will implement a Genetic Algorithm. A GA is a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.

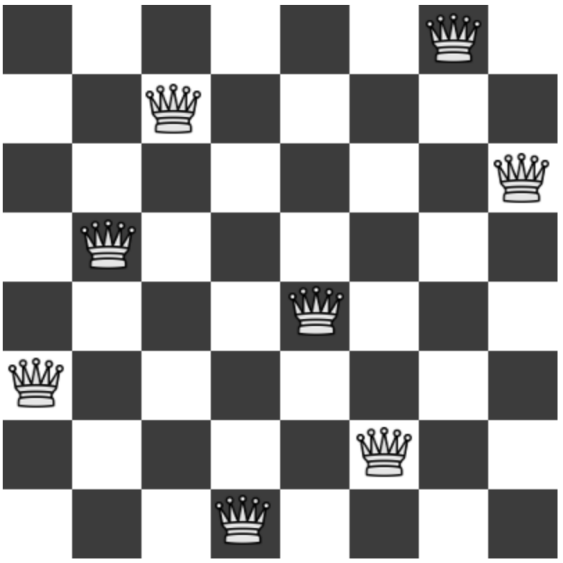

Figure 1: 8 Queens solution

2 Genetic Algorithms

Genetic Algorithms (GAs) are adaptive methods which may be used to solve search and optimization problems. They are based on the genetic processes of biological organisms. Over many generations, natural populations evolve according to the principles of natural selection and survival of the fittest", stated by Charles Darwin in The Origin of Species . By mimicking this process, genetic algorithms are able to *evolve* solution to real world problems.

GAs use a direct analogy of natural behaviour. They work with a population of **individuals**, each representing a possible solution to a given problem. Each individual is assigned a **fitness score** according to how good a solution to the problem is. The highly fit individuals are given opportunities to reproduce, by **crossover** and **mutation** with other individuals in the population. A genetic algorithm is expected

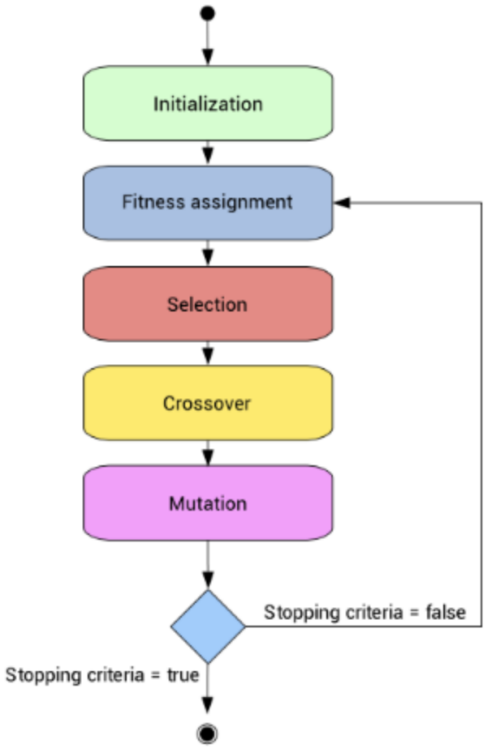

Figure 2: Graphical representation of genetic algorithms

to make evolve the population of individuals over generations, in such a way that the average fitness score tends to *the optimal fitness score*, so that the individuals tend to the optimal solution of the problem.

Fittest individuals are given chance to reproduce, passing useful genes to their offspring. Thus the gene pool constantly improves, i.e. evolution is converging to better and better solutions. There are very many candidate solutions evaluated all the time. This is the behaviour that we try to roughly mimic with genetic algorithms.

Once we have understood what is the idea of GA, we can go deeper in the algorithm. Normally a GA works as it follows, [Uddalok Sarkar, 2010]:

- **1-** Start with a randomly generated population of n l−bit chromosomes (candidate solutions to a problem).
- **2-** Calculate the fitness function of each chromosome x in the population.
- **3-** Repeat the following steps until n offspring have been created:
  - **a)** Select a pair of parent chromosomes from the current population, the probability of selection being an increasing function of fitness. Selection is done "with replacement," meaning that the same chromosome can be selected more than once to become a parent.
  - **b)** With probability pc (the "crossover probability" or "crossover rate"), cross over the pair at a randomly chosen point (chosen with uniform probability) to form two offspring. If no crossover takes place, form two offspring that are exact copies of their respective parents. (Note that here the crossover rate is defined to be the probability that two parents will cross over in a single point. There are also "multi−point crossover" versions of the GA in which the crossover rate for a pair of parents is the number of points at which a crossover takes place.)
  - **c)** Mutate the two offspring at each locus with probability pm (the mutation probability or mutation rate), and place the resulting chromosomes in the new population. [mut] If n is odd, one new population member can be discarded at random.
- **4-** Replace the current population with the new population.
- **5-** Go to step 2.

3 Code

In this section I will try to give an overview of how the code is build. For deeper analysis the Jupyter Notebook is attached.

To initiate a **random population** we build the following functions that create a random population o N size individuals. After some thought, we found the conclusion that the best way to represent an individual is as a N size vector wich doesn't have any repeated numbers.

```
def random_individual(N):
    return random.sample(range(N),len(range(N)))
out=[0, 2, 6, 7, 3, 1, 5, 4]
def initialize_pop(N,pop_size):
    pop=[]
    for i in range(pop_size):
        pop.append(random_individual(N))
    return pop
```

As for the **fitness function**, since we don't have any repeated numbers, there won't be any repeated row or column and we will only take into account the diagonal

```
def individual_fitness(individual):
    N=len(individual)
    diag=0.
    for i in range(N-1):
        for j in range(N-1):
            if (int(abs(individual[j]-individual[i]))
                            ==int(abs(j-i)) and i is not j):
                diag+=1
    return diag
```

In order to **select** two individual we have make two different selection criteria:

- We make the population biased toward the fittest individual but giving a chance to the less fit individual to reproduce and then selecting the best individuals
- Tournament selection involves running several "tournaments" among a few individuals (or "chromosomes") chosen at random from the population.

```
def selectFromPopulation(population):
    pop_performance=population_fitness(population)
```

```
    leng=int(np.round(len(pop_performance)-1))
    for i in range(len(pop_performance)):
        pop_performance[i][0]=(pop_performance[i][0]**2)*random.random()
    list=sorted(pop_performance,reverse=False)
    ind1=list[random.randint(0,3)][1]
    for i in range(len(pop_performance)):
        pop_performance[i][0]=pop_performance[i][0]*random.random()
    ind2=list[random.randint(0,3)][1]
    return ind1,ind2
```

For obtaining global optimum for a non-convex function the **cross over** function performs a very important role for obtaining better offspring. As we are considering each of the chromosomes actually a random permutation of (1, 2, 3... N), so it was needed to design a permutation crossover. Here we are using the order 1 crossover. Order 1 Crossover is one sort of very simple permutation crossover in which 2 random points are selected from parent-1 and the alleles between these points are carried over to the child and the other alleles from parent-2 which are absent in child are carried and placed in the child in the order which they appear in parent-2.

$$\text{Father} = 6\ 5\ \underline{4\ 3\ 7}\ 1\ 0\ 2$$
$$\text{Mother} = 0\ 1\ 2\ \cancel{3}\ \cancel{4}\ 5\ 6\ \cancel{7}$$
$$\text{Child} = 0\ 1\ 4\ 3\ 7\ 2\ 5\ 6$$

Mutation is very important in genetic algorithms for not to stuck the process in local optimum. We defined mutation as the exchange of genome order inside the chromosome. We decide to also define a chance for double mutation with an increasing probability of mutation, so after the first mutation is giving, higher order mutations are more probable.

```
def mutation(individual):
    N=len(individual)
    copy_ind=individual.copy()
    rand1=0
    rand2=0
    while rand1==rand2:
        rand1,rand2=random.sample(range(0, N), 2)
    individual[rand1]=copy_ind[rand2]
    individual[rand2]=copy_ind[rand1]
    return individual
mutation([5, 4, 3, 0, 1, 2, 7, 6])
```

```
out=[5, 4, 6, 0, 1, 2, 7, 3]
def mutate_population(population,probability):
    for i in range(len(population)):
        if random.random()*100<probability:
            population[i]=mutation(population[i])
            if random.random()*100<2*probability:
                population[i]=mutation(population[i])
                if random.random()*100<4*probability:
                    population[i]=mutation(population[i])
    return population
```

To finalize the algorithm we have to create the next generation, we will do it by measuring the performance, selecting individuals, cross overing them and applying mutation the population. We drop the last two individuals and provide two new random ones, to make the algorithm more exploratory.

```
def next_generation(population):
    next_gen=[]
    size=len(population)
    elite=int(np.round(size/5))
    for i in range(size):
        ind1,ind2=selectFromPopulation(population)
        next_gen.append(crossover_pointed(ind1,ind2))
    next_gen.pop()
    next_gen.pop()
    next_gen.append(random_individual(len(population[0])))
    next_gen.append(random_individual(len(population[0])))
    return next_gen
```

4 Results

In this section we will only present the results for some of the parameters since there has been a lot of time spent tuning parameters trying to find the optimal performance. Since our goal is not to only find a single solution for the N-queens

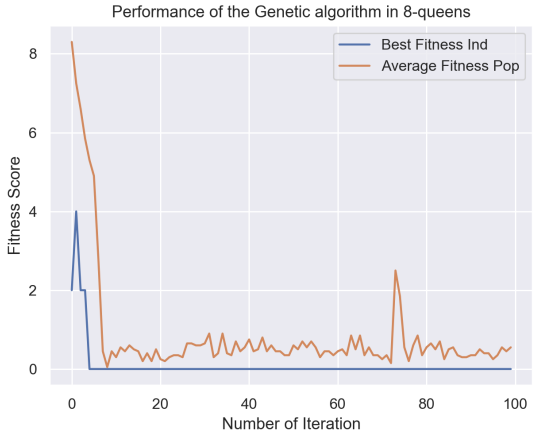

Figure 3: Evolution of the genetic algorithm with 8 queens

but to make the population converge. The algorithm has to be balance betwen exploitatory and exploratory. This means, that it has to evolve towards the best solution but in the other hand it has to be exploratory for the case in that the rearrangement has a low score but we are still far away in the problem functional space. We found that there is a fight between exploitatory and exploratory in order to find the optimal perfomance of the genetic algorithm. We have represented the graphs of $N = 8$ and $N = 20$ in which we can appreciate how is the evolution of the algorithm.

We can notice that the average performance doesn't drop to zero, that's due to the brand new random individual that are created in every generation.

4.1 Future Improvements

In order to achieve better results, there are some suggested approaches that could make the algorithm work better.

- Using different methods of crossover as uniform crossover, 2 point crossover or using problem specific knowledge to design crossover operators for this specific task

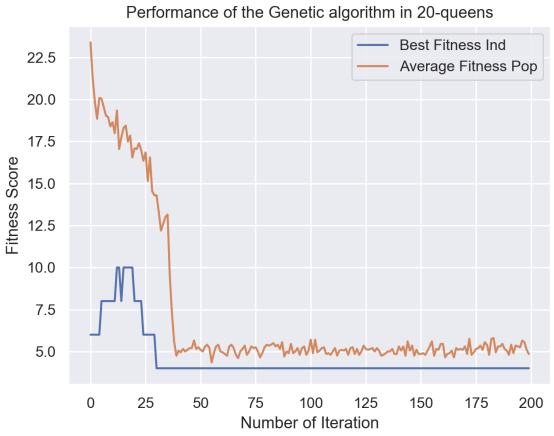

Figure 4: Evolution of the genetic algorithm with 20 queens

- Applying different methods of selection will enable to pick childs with different distributions

5 Conclusions

In this project we have developed a genetic algorithm that is able to solve the 8 queens problem. We have also faced the algorithm to bigger tasks by making it run for a more general approach N-queens.

We have been able to solve the initial problem with more than 8 queens; we have obtained a solution for a 20 and 100 queens problem. We can appreciate that the general idea of genetic algorithms is powerful technique to optimize complex systems in a very intuitive way.

One of the weakness of genetic algorithms it's the tuning, some parameters can work for certain purposes but they won't for others. That's why it is important to understand Genetic Algorithms and their parameters, so we can build the solutions ad-hoc.

References

Mutation methods. URL https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm.

Luis Alseda. A list of genetic operations (crossover and mutation).

Sayan Nag Uddalok Sarkar. An adaptive genetic algorithm for solving n- 1. introduction: Queens problem. *Department of Electrical Engineering*, 2010.

Wikipedia. General information for the 8 queens probelm.