Universitat Autònoma de Barcelona

N Queens Problem
An implementation using Genetic Algorithms and Python

Alfredo Hernández

30th November 2017

Contents

1 The N Queens Problem

This problem consists on placing N queens on a $N \times N$ chessboard so that no queen can attack another. In the usual fashion, an attack is defined when two or more queens are in the same row, column, or diagonal (as depicted in figure 1). We will expand on how to compute if a queen attacks another in § 2.4.

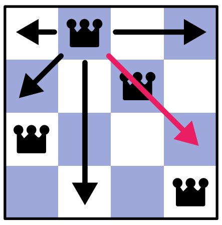

Figure 1: Example of a queen attacking another queen on a 4×4 board

The problem can be quite expensive from a computational point of view, as there are N^N possible arrangements of N queens on an $N \times N$ board, but only a small amount of solutions (relative to the total possible arrangements). It is possible, however, to use shortcuts that reduce computational requirements of brute-force techniques.

1.1 Computational representation

Given the nature of this problem, we can program our algorithm to work with queens so that no queen shares the same row or column. The simplest way is to think of each board as an array of non-repeated consecutive numbers, where the index represents the row, and the number itself is the column; this is called *Permutation Representation*. For instance, the board in figure 1 would be represented as

$$\texttt{[1 2 0 3]}. \tag{1}$$

This consideration alone makes the complexity of a brute-force algorithm much simpler, as now we only have $N!$ possible permutations of N queens on an $N \times N$ board (we will expand on this later).

1.2 Solutions of the problem

Table 1 summarises the number of solutions for the N Queens Problem, both fundamental and all (or non-fundamental) solutions, where fundamental solutions are those that exclude mirrored images as well as rotations of a solution.

There is currently no known formula for the exact number of solutions, or even for its asymptotic behaviour for larger values of N.

N	1	2	3	4	5	6	7	8	9	10	...
fundamental	1	0	0	1	2	1	6	12	46	92	...
all	1	0	0	2	10	4	40	92	352	724	...

Table 1: List of solutions for the N Queens Problem

All non-fundamental solutions up to $N = 10$ can be easily obtained using the script `brute-queens.py`. We will not discuss the details of the implementation, for it is a really simple algorithm, and brute-force methods are not the main focus of this text.

As an illustration, in figure 2 we depict the only two non-fundamental solutions for $N = 4$. It is worth to notice that the two solutions are the same fundamental solution, as they are the mirrored image of each other.

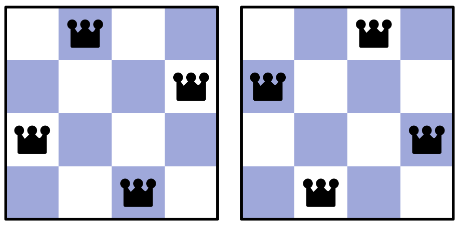

Figure 2: The two non-fundamental solutions of the 4 Queens Problem

In figure 3 we compare the complexity of using an algorithm that calculates all possible arrangements and another that only calculates all the possible permutations against the amount of non-fundamental solutions for the N Queens Problem.

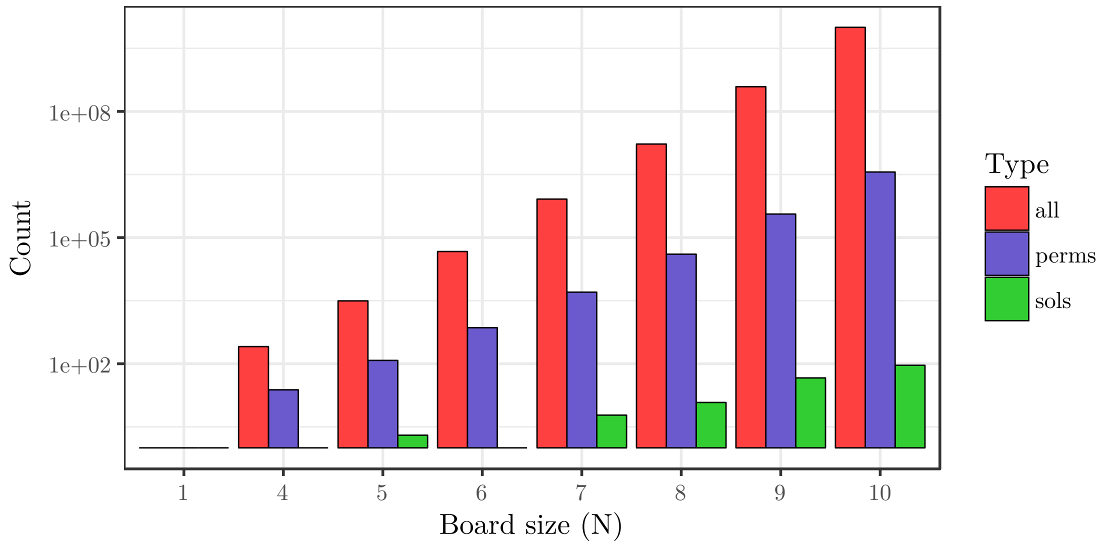

Figure 3: Comparison of brute-force complexity for the N Queens Problem

1.3 Role of metaheuristics

One may think that using only permutations gives us a good enough brute-force method to solve this problem. This may be true for small values of N, but $N > 10$ presents quite a computational barrier in terms of memory requirements. Using $N = 11$ requires calculating $11! = 39\,916\,800$ different permutations; this means using $\sim 3.5\,\text{GB}$ of RAM[1] just to store the numbers on the boards (the boards' structure would be another thing altogether).

It is for this reason that heuristics are used to solve this kind of problem, as they can be used to speed up the process of finding a solution. In particular, we will solve the N Queens Problem using a Genetic Algorithm, a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms. We will discuss this implementation in § 2.

[1]On a 64-bit platform (8 bits per integer).

2 Implementation with a Genetic Algorithm

2.1 Methodology of a Genetic Algorithm

The Genetic Algorithm was invented by John Holland at the University of Michigan in the 1970s. This evolutionary algorithm consists on iterating through fitness assessment, recombination, and population reassembly.

In the recombination process, we select two parents from the original population, cross them over with one another, and mutate the results. The children are then reassembled into the original population (the way to do this varies from author to author).

Zäpfel et al. [3] go into great detail on how each component of the algorithm affects the solution of the problem. In figure 4 we can see a simple diagram on how the Genetic Algorithm is structured and how each operation depends on the specific problem one wants to tackle.

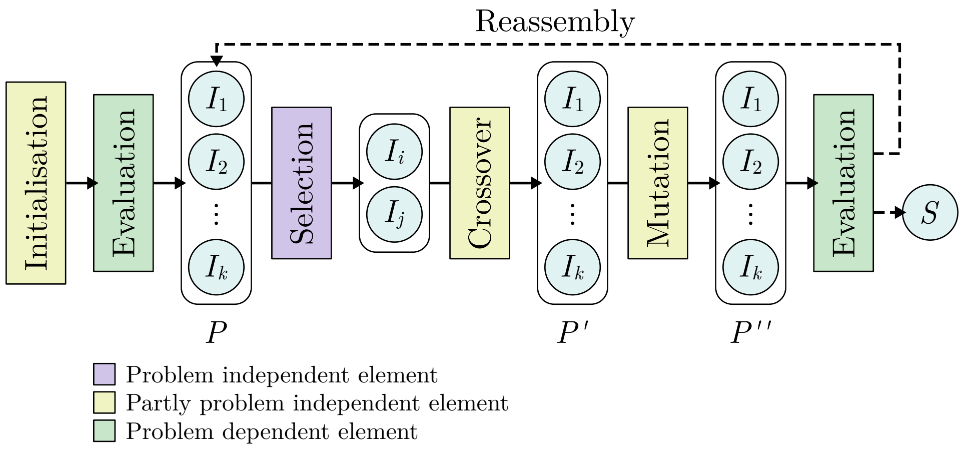

Figure 4: Structure of a Genetic Algorithm

One might wonder why initialisation, crossover, and mutation are marked as partly problem dependent here. The reason is that in Genetic Algorithms, different encodings can be used to represent individuals. By using simple or generic encodings, like binary or permutation, one can use standard genetic operators for crossover and mutation. Hence these operators are referred to as partly problem dependent.

2.2 Computational structure of the program

First of all, we need to decide the programming language we will use to implement the Genetic Algorithm. I decided to use Python 3, as I am more experienced with it, and because it would allow me to tweak the program and play with the different parameters without focusing too much on the technical side of things.

One of the main advantages of using an object-oriented programming language like Python is the ability to use classes to group each individual with its properties (sequence and fitness) in an organised and straightforward way. In snippet 1 we can see the `BoardPermutation` class we will use to store each one of the individuals or `boards`.

Snippet 1: Definition of the `BoardPermutation` class

```
class BoardPermutation:
  def __init__(self):
    self.sequence = None
    self.fitness = None
  def set_sequence(self, val):
    self.sequence = val
  def set_fitness(self, fitness):
    self.fitness = fitness
  def get_attr(self):
    return { 'sequence' : sequence, 'fitness' : fitness }
```

In the following sections we will see how `sequence` and `fitness` are defined.

2.3 Initialisation of the population

Now that we know the computational structure of the individuals, we can work out a function to generate each of the individuals that constitute the initial population. In snippet 2 we can see how we define our `generate_chromosome()` function. What the function does is generate an ordered array from `0` to `N_QUEENS` (as in `[0 1 2 3 ... N_QUEENS]`), and then shuffle the elements to create different permutations. For the sake of academic rigour, as is usual with Genetic Algorithms, we call each individual a chromosome.

Snippet 2: Function to generate a single individual

```
def generate_chromosome():
  # Randomly generates a sequence of board states.
  init_distribution = np.arange(N_QUEENS)
  np.random.shuffle(init_distribution)
  return init_distribution
```

Notice that we are using Python's `numpy` library, which makes our program run almost as fast[2] as if it used native C.

All we have to do now is to generate the initial population. The implementation is straightforward: we create a list of `population_size` objects of class `BoardPermutation`. Then all we need to do is to set the sequence and the fitness of each board using the `generate_chromosome()` and `assess_fitness()` functions, respectively. This is clearly illustrated in snippet 3.

[2]Provided we know how to make an efficient program (which is probably not the case).

Snippet 3: Population initialization function

```python
def generate_population(population_size = 100):
  # Create the different boards
  population = [BoardPermutation() for i in range(population_size)]
  for i in range(population_size):
    population[i].set_sequence(generate_chromosome())
    population[i].set_fitness(assess_fitness(population[i].sequence))
  return population
```

2.4 Evaluation: fitness function

The evaluation function determines the quality of a candidate solution. In metaheuristics this is usually called fitness function. The fitness value of a proposed solution is needed for the selection process and eventually finding the final solution of our problem.

As we mentioned before, the assessment of the fitness of a solution is a problem dependent element. In § 1.1 and § 2.3 we discussed how we only worked with permutations of an array of non-repeated consecutive numbers. This eases the calculation of the fitness quite a bit.

Since the queens are by definition in different rows and columns, we only have to check if a queen attacks another one diagonally. We could do this by iterating on the diagonal cells of any given queen and check if there is a queen in any of the cells, but this is a bit complex to program. We found that the easiest way to check if two queens attack each other is to check if

$$|x_i - x_j| = |y_i - y_j|, \tag{2}$$

that is, when the two queens' vertices define a square, as depicted in figure 5.

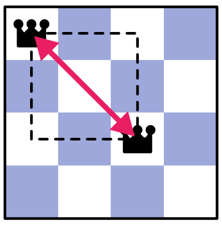

Figure 5: Example of a queen attacking another queen

In snippet 4 we can see our implementation of the fitness function. In it we count the amount of clashes produced in any given board, then we return `MAX_FIT - clashes`. In this problem, `MAX_FIT` is $N(N-1)$; the reason behind this is really simple. We can think of the queens as a set of N vertices; in a K_N graph, the amount of edges is $v(v-1)/2$, where edges connect queens that do not clash with one another. The reason behind the $\times 2$ factor is that we count clashes twice in our loop (Q_i attacks Q_j and Q_j attacks Q_i); this does not matter as long as our fitness function is well defined and behaves in a controlled way.

Snippet 4: Fitness function

```
def assess_fitness(chromosome = None):
  clashes = 0
  # Calculate diagonal clashes
  for i in range(len(chromosome)):
    for j in range(len(chromosome)):
      if ( i != j):
        dx = abs(i-j)
        dy = abs(chromosome[i] - chromosome[j])
        if(dx == dy):
          clashes += 1
  return (MAX_FIT - clashes)
```

2.5 Reassembly of the population

Before we explain the recombination process, we need to define how the reassembly of the population will be done in our program. This is very important, because it will directly define the way we implement the parent selection and crossover process.

The reassembly process consists on replacing old individuals with new ones. There are several replacement strategies, the following being the most popular:

- Generational Replacement.
- Steady-State Replacement.

In Generational Replacement the new generation supersedes the old generation. But this replacement scheme has a big disadvantage. If one replaces the whole generation the risk of dismissing a very promising solution is high.

In contrast, Steady-State Replacement consists on dynamically replacing old individuals by new ones. But whether only a single individual or multiple individuals are replaced, so one must define which individuals are replaced (worst, random, parents, ...)

In our implementation we will perform a Steady-State Replacement of the parents.

2.6 Recombination process

This is probably the most important part of the whole Genetic Algorithm. This is where we need to choose if we want to focus on exploitation (corresponding to intensification) or exploration (corresponding to diversification). This is not an easy task, because in an ideal scenario, we want the balance between both; as Stützle [4] puts it

A metaheuristic will be successful on a given optimization problem if it can provide a balance between the exploitation of the accumulated search experience and the exploration of the search space to identify regions with high quality solution in a problem specific, near optimal way.

Not only that, but using a population of permutation individuals has a trade-off of having an easier fitness function: the crossover function has to be modified to avoid getting an invalid children when recombining two parents (e.g., `[1 1 0 3]`). Although most texts deal with crossovers in the most generalised manner, some do make this distinction and tweak the classical algorithms to impede such children, such as Gendreau and Potvin [5], and Goldberg [6]. The same is true for the mutation function, although the workaround is fairly straightforward.

Parent selection

As mentioned before, the parent selection is trivial, as it does not depend on the structure of our individuals; the only important parameter is their fitness.

The usual selection methods involve ranking in some fashion all individuals by their fitness. The most widely used method is the Fitness-Proportionate Selection, also known as Roulette Selection. This method is discussed in detail by Sean [2] and Gendreau and Potvin [5]; in short it selects individuals with higher fitness more often (this involves defining a selection pressure, or a survival function).

The method we will use, however, is the Tournament Selection. In this selection method a set of τ (this is called the Tournament Size) chromosomes are chosen and compared, the best one being selected for parenthood. Our implementation can be seen in snippet 5.

Snippet 5: Parent selection using a Tournament Selection

```python
def get_parent(population = None):
  # Get parent using a Tournament Selection algorithm
  best_board = random.randint(0, len(population) - 1)
  for i in range(1,TOURN_SIZE):
    next_board = random.randint(0, len(population) - 1)
    if ( population[next_board].fitness >
        population[best_board].fitness ):
      best_board = next_board
  return best_board
```

Using a Tournament Selection over a Fitness-Proportionate Selection has quite a number of advantages. First of all, the Tournament Selection can deal with situations where there is no formal objective fitness function (needed to define the selection pressure), as it is not sensitive to how the fitness is defined; it only needs the fitness value itself. Second, it is not a complex method, leading to very fast computational times. Last but not least, it can be finely tuned to our liking; allowing us to be more exploratory or exploitatory.

In the Genetic Algorithm, the most popular setting is $\tau = 2$; this approach has similar properties to Fitness-Proportionate Selection. This value for the Tournament Size leads to a fairly exploratory algorithm, but using bigger values makes the algorithm too exploitatory, which can lead to very slow (or very fast, depending on the initial population[3]) convergence.

[3]This can be understood in terms of Lyapunov's instability principle.

In § 3.1 we will discuss in more detail how we will choose the optimal value for τ as well as the other parameters involved in the Genetic Algorithm.

Crossover process

Under a standard Two-Point Crossover, two parents are aligned and two crossing points are picked uniformly at random. These two points define a matching region, which are swapped between the two parents, giving place to two children:

$$\begin{matrix} P_1: & \texttt{[0 4 | 2 3 1 | 5]} \\ P_2: & \texttt{[5 0 | 2 1 4 | 3]} \end{matrix} \mapsto \begin{matrix} P_1': & \texttt{[0 4 | 2 1 4 | 5]} \\ P_2': & \texttt{[5 0 | 2 3 1 | 3]} \end{matrix}. \tag{3}$$

The first thing we notice is that this simple crossover has led to two invalid individuals under our Permutation Representation, just as we warned in § 2.6. This will surely be the case for most crossovers. So let us find a better suited crossover function for the problem in our hands.

The Partially-Matched Crossover, or PMX, tries to solve this problem. Its philosophy is very similar to that of the Two-Point Crossover: two points are randomly selected to define a matching region. The difference is that the matching region defines a position-by-position interchange mapping. Let us see this with an example:

$$\begin{matrix} P_1: & \texttt{[0 4 | 2 3 1 | 5]} \\ P_2: & \texttt{[5 0 | 2 1 4 | 3]} \end{matrix} \mapsto \begin{matrix} P_1': & \texttt{[0 1 2 4 3 5]} \\ P_2': & \texttt{[5 0 2 3 1 4]} \end{matrix}, \tag{4}$$

where $\{2 \leftrightarrow 2, 3 \leftrightarrow 1, 1 \leftrightarrow 4\}$ is our set of interchange mappings.

Even so, not everything is perfect with this crossover: in some situations one may end up with a set of interchange mappings that contain a number more than once (just as the example above). In my opinion[4], this crossover has an unclear order of operations (in the example above we did the interchange mappings from left to right, but we could have done them in any order) and inevitably leads to redundant calculations.

Goldberg [6] explores several crossover algorithms similar to the PMX that can be easily applied to problems with Permutation Representation. In particular, we will focus on the Ordered Crossover, or OX.

The OX starts in a similar fashion as the PMX: two points are randomly selected to define a matching region. Like PMX, each string maps to constituents of the matching section of its mate. Instead of using point-by-point interchange mapping, OX uses a sliding motion to fill the holes left by transferring the mapped positions. For example

[4]I am sure the PMX works just fine, but I am not particularly fond of it from a philosophical and programming point of view.

when parent P_2 maps to parent P_1; queens 2, 3, and 1 will leave holes (marked by an `H`) in the board:

$$P_2 : \; \texttt{[5 0 | H H 4 | H].} \tag{5}$$

These holes are filled with a sliding motion that starts following the second crossing site:

$$P_2 : \; \texttt{[5 0 | H H H | 4].} \tag{6}$$

The holes are then filled with the matching region taken from the other parent. Performing this operation and completing the reciprocal cross we obtain the offspring P_1' and P_2' as follows:

$$\begin{array}{l} P_1' : \; \texttt{[0 3 | 2 1 4 | 5]} \\ P_2' : \; \texttt{[5 0 | 2 3 1 | 4]} \end{array}. \tag{7}$$

The Ordered Crossover is slightly more complicated to program than the Partially-Matched Crossover, but I find it a much more elegant method and surely is more fun to do. In snippet 6 we can see the way we implement OX:

Snippet 6: Ordered crossover function, as described by Goldberg [6]

```python
def ordered_crossover(ind1 = None, ind2 = None):
  # Ordered crossover
  a, b = random.sample(range(N_QUEENS), 2)
  if a > b:
      a, b = b, a
  holes1, holes2 = [True]*N_QUEENS, [True]*N_QUEENS
  for i in range(N_QUEENS):
    if i < a or i > b:
      holes1[ind2[i]] = False
      holes2[ind1[i]] = False
  # We must keep the original values somewhere before scrambling
     everything
  temp1, temp2 = ind1, ind2
  k1 , k2 = b + 1, b + 1
  for i in range(N_QUEENS):
    if not holes1[temp1[(i + b + 1) % N_QUEENS]]:
      ind1[k1 % N_QUEENS] = temp1[(i + b + 1) % N_QUEENS]
      k1 += 1
    if not holes2[temp2[(i + b + 1) % N_QUEENS]]:
      ind2[k2 % N_QUEENS] = temp2[(i + b + 1) % N_QUEENS]
      k2 += 1
  # Swap the content between a and b (included)
  for i in range(a, b + 1):
    ind1[i], ind2[i] = ind2[i], ind1[i]
```

Although PMX and OX are very similar, they preserve different kinds of similarities. PMX tends to respect absolute queen position, whereas OX tends to respect relative queen position.

Mutation process

In the case of binary strings, mutation can be simply performed by swapping bits by their complementary (Bit-Flip Mutation) with a small probability p. This is usually done by applying an uniform mask to the desired individual:

$$[\ 0\ 0\ 0\ 0\ 1\ 1\ ] \xrightarrow{010110} [\ 0\ 1\ 0\ 1\ 0\ 1\ ]. \tag{8}$$

More generally, when there are several allele possibilities for each gene (in our problem, this means several possible columns for each row) instead of just `0` and `1`, when we mutate a particular allele, we must find a way to decide its new value.

Since we are dealing with Permutation Representation, the easiest way to perform the mutation is to uniformly select two swapping points at random. This means we can use a mask to define the swapping points:

$$[\ 0\ 3\ 2\ 1\ 4\ 5\ ] \xrightarrow{010001} [\ 0\ 5\ 2\ 1\ 4\ 3\ ]. \tag{9}$$

In snippet 7 we can see our implementation of the mutation function. Notice we make sure the two selected swapping points are different from each other, otherwise the p of performing a mutation would not be well defined (actually it would, but it would not be the one we explicitly specify in the program).

Snippet 7: Mutation function

```python
def mutate(board = None):
  # Mutate a board using a mask
  if random.random() < MUTATE_PROB :
    a, b = random.sample(range(N_QUEENS), 2)
    while (b == a): # To ensure a true mutation
      b = random.sample(range(N_QUEENS), 1)
    population[board].sequence[a], population[board].sequence[b] =
        population[board].sequence[b], population[board].sequence[a]
```

It is often suggested that mutation has a somewhat secondary function, that of helping to preserve a reasonable level of population diversity to make sure the algorithm can escape from sub-optimal regions of the solution space, preventing a premature convergence, but not all authors seem to agree.

For academic purposes, our algorithm includes a boolean `MUTATE_FLAG` that can be changed if one wishes to explore the difference on performance of the algorithm when mutation is used or not.

2.7 Revaluation and stop condition

After crossing the two parents we need to reassess their fitness; this is straightforward since we are using objects. All we are left with is the process of finding out whether we have a good solution in our population or if we need to rerun the Genetic Algorithm with the newly computed generation. The function in which we have implemented this process can be seen in snippet 8.

Snippet 8: Function to stop the program when we find a good solution

```
def find_good_queen(population = None):
  global good
  # Look for a good board
  for board in range(len(population)):
    if ( population[board].fitness == MAX_FIT ):
      if VERBOSE_FLAG:
        print("Found␣a␣non-fundamental␣solution␣on␣generation",
            iteration)
        print( "Q",  board, ":", population[board].sequence,
            "fitness:", population[board].fitness )
      good = True
      return True
```

Since we are using a Steady-State Replacement of the parents, most of our functions just do operations within the population itself; the *crucial* return value in the algorithm is the return of the `find_good_queen()` function. As we will see later, the Genetic Algorithm stops when `find_good_queen()` returns the boolean `True`.

The global `good` variable in this function is used to tell the program whether we have found a solution or not, although it is not essential for the algorithm to work.

2.8 The Genetic Algorithm

Now we have all the ingredients to define our main `genetic_algorithm()` function. This function just calls all the other functions in the usual order (cf. figure 4), and sets `good` to `True` when/if it finds a solution. The implementation in snippet 9 is self-explanatory and quite easy to understand.

Snippet 9: The Genetic Algorithm

```
def genetic_algorithm(MAX_ITER):
  global iteration
  for iteration in range(MAX_ITER):
    # Select parents
    parent1 = get_parent(population)
    parent2 = get_parent(population)
    ordered_crossover(population[parent1].sequence,
        population[parent2].sequence)
    # Mutate children
    if(MUTATE_FLAG):
      mutate(parent1)
      mutate(parent2)
    # Reassess fitness
    population[parent1].set_fitness(
        assess_fitness(population[parent1].sequence))
    population[parent2].set_fitness(
        assess_fitness(population[parent2].sequence))
    # Stop the algorithm
    if find_good_queen(population):
      if(WRITE_FLAG):
        with open('results.csv', 'a') as csvfile:
          writer = csv.writer(csvfile)
          writer.writerow([N_QUEENS, POPULATION, MUTATE_PROB,
              TOURN_SIZE, iteration])
      break
```

To run the Genetic Algorithm we just need to do three simple things: (i) start our initial population, (ii) tell the program it has not found a `good` solution yet, (iii) execute the algorithm until `MAX_ITER` is reached or a valid solution is found:

Snippet 10: Instructions needed to perform the Genetic Algorithm

```
population = generate_population(POPULATION)
good = False
genetic_algorithm(MAX_ITER)
if VERBOSE_FLAG and not good:
  print("Couldn't␣find␣any␣solution␣in", MAX_ITER, "generations.")
```

Below we can see a typical output of our Python program. In particular, we have run `python n-queens.py 8 200 0.05 2` to solve the 8 Queens Problem with an initial population of 200 individuals, $p = 0.05$, and $\tau = 2$:

```
Found a non-fundamental solution on generation 106
Q 92 : [6 2 0 5 7 4 1 3] fitness: 56
```

We found a valid solution (depicted in figure 6) in third generation of the algorithm.

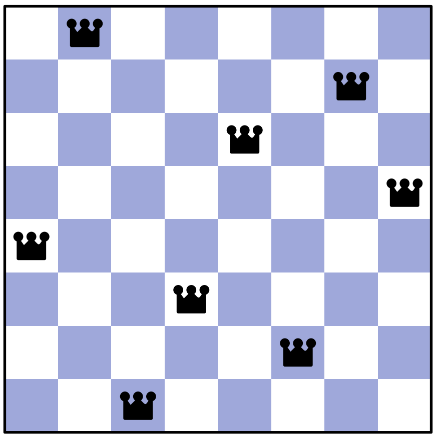
Figure 6: A single solution for the 8 Queens Problem

3 Results and Discussion

3.1 Parameter control

As Michalewicz and Fogel [7] put it, typically, only one parameter is tuned at a time, which may result in some less-than-optimal choices because parameters interact in complex ways. The simultaneous tuning of more parameters, however, leads to an enormous amount of experimentation. The technical drawbacks of parameter tuning based on experimentation are:

- The parameters are not independent, but trying all possible combinations is practically impossible.
- The process of parameter tuning is time consuming, even if parameters are optimised one at a time.
- The results are often disappointing, even when a significant expenditure of time is made.

Following this reasoning, and keeping in mind we want to find the balance between exploitation and exploration, we will try to find the right parameters for our N Queens Problem. Ideally, our aim is to find a general value for the tournament size τ and the mutation probability p, independent problem size N; in contrast, the initial population size should depend on N.

Therefore, we have to find the minimum of a function f such that:

$$\begin{array}{rccc} f: & \mathbb{N}\times\mathbb{R}\times\mathbb{N} & \to & \mathbb{N} \\ & (\tau, p, \texttt{POPULATION}) & \mapsto & \texttt{iter}. \end{array} \tag{10}$$

Obviously, this function f is an abstract one with an unpredictable behaviour. So, one wonders how to find the best parameters for a given N. Michalewicz and Fogel [7] thoroughly discuss previous literature on the topic of parameter control, only to conclude that even if one assumes for a moment that there is a perfect configuration, finding it is really a hopeless task.

To ease the hard task of finding somehow optimal parameters, we will stick with $\tau = 2$ (the most popular setting in Genetic Algorithms), and try to tune our algorithm playing with other parameters of the algorithm. This means our algorithm will probably lean slightly towards an exploratory approach, as we will not have a tunable variable to make the algorithm more exploitatory.

Our analysis will consist on playing with the following settings:

- Mutation: $p \in [0.005, 0.01]$ with a step of $\Delta p = 0.001$.
- Population size: $\texttt{POPULATION} \in [50, 50N]$ with a step of $\Delta\texttt{POPULATION} = 50$.

For each tuple of parameters, we will run the algorithm 100 times, to have a statistically significant result (we will be running the algorithm around 16 200 times). We can easily do this using `bash` (see snippet 11), setting the flags `WRITE_FLAG` and `VERBOSE_FLAG` in our Python program to `True` and `False` respectively.

Snippet 11: Bash script to find the *optimal* parameters for the N Queens Problem

```
# Looping over N
for (( N = 8; N <= 10; N ++ )); do
  echo 'Running the Genetic Algorithm for N =' $N
  # Looping over POPULATION
  for (( P = 50; P <= 50 * N ; P += 50 )); do
    # Looping over MUTATE_PROB
    # for (( M = 5; M <= 10; M ++ )); do
    for M in 0.005 0.006 0.007 0.008 0.009 0.01; do
      # 100 runs for each tuple
      for (( i = 0; i < 100; i++ )); do
        python n-queens.py $N $P $M 2
      done
    done
  done
done
```

3.2 Best parameters

In figure 7, we can see a heatmap of the required iterations to find a solution of the 8 Queens Problem[5], with the combination of parameters we discussed before.

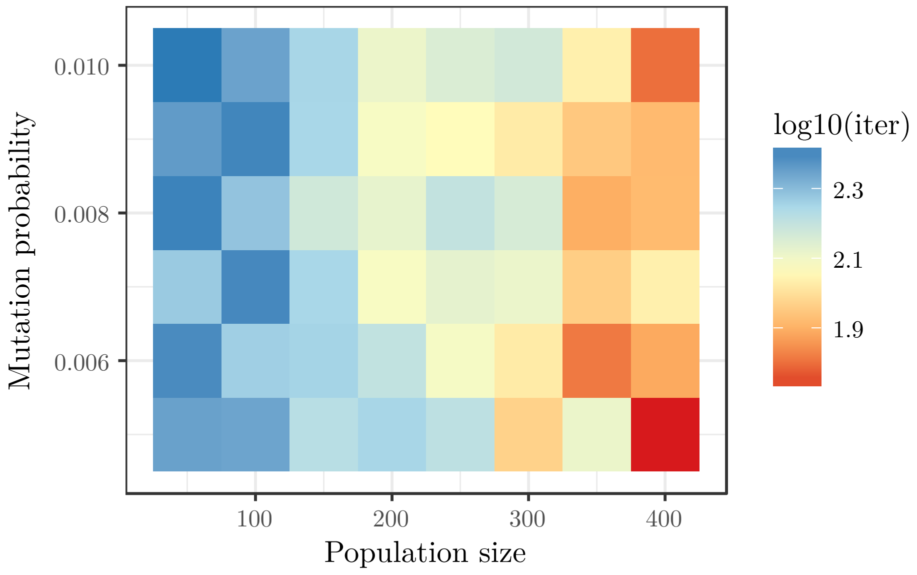

Figure 7: Required iterations to find a solution to the 8 Queens Problem

As we can see, it seems the mutation probability p does not play a significant role on the iterations needed to find a solution. To make sure this is really the case, we decided to

[5]The analysis was initially made with $N = 8, 9, 10$, but the results were very much the same and we decided to exclude the figures for $N = 9, 10$. The raw results can be found in `results_all.csv`.

focus on $N = 8$, and decided to run a more granular version of snippet 11. The results of these 75 600 tests can be seen in figure 8.

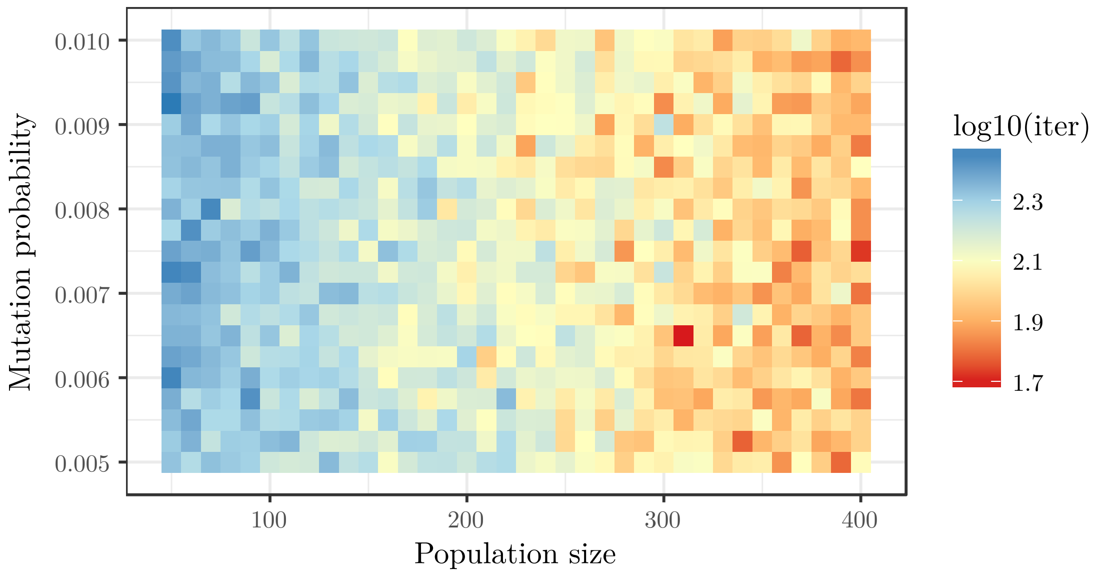

Figure 8: Required iterations to find a solution to the 8 Queens Problem, using a more granular analysis

Looking at the figure above, our hypothesis seems confirmed. Whilst the dependence on the required iterations with the population size `POPULATION` is more than clear, the mutation probability p seems to be statistically irrelevant on the performance of the Genetic Algorithm.

For this reason, we will retract our previous train of thought and will consider once again the tournament size τ as variable to be optimised. For the next analysis, we will fix `POPULATION`, and play with the following settings:

- Mutation: $p \in [0.005, 0.01]$ with a step of $\Delta p = 0.00025$.
- Tournament size: $\tau \in [2, 7]$ with a step of $\Delta\tau = 1$.

In figures 10 and 9 we can see the results of tuning our Genetic Algorithm playing with the mutation probability p and the tournament size τ, with `POPULATION` $= 100, 350$. These two population sizes were chosen to see if we could make an improvement on a slow algorithm (`POPULATION` $= 100$) and a fast one (`POPULATION` $= 350$).

Tuning both parameters do seem to give us a small improvement on the iterations required to find a solution (see table 2). Yet, it is difficult to say for sure that the results follow a robust pattern; one could argue that there is a sweet-spot for $\tau = 3$, $p \sim 0.009$, debatably the *best configuration of parameters*.

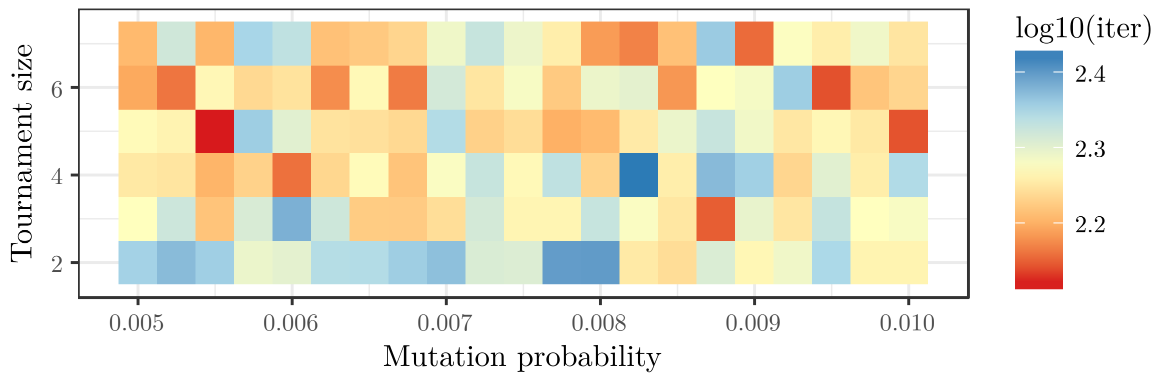

Figure 9: Required iterations to find a solution to the 8 Queens Problem, with `POPULATION` $= 100$

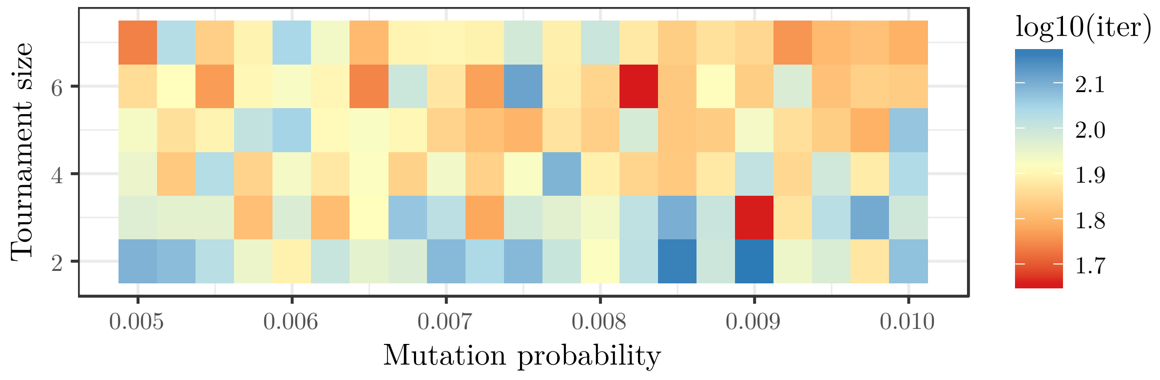

Figure 10: Required iterations to find a solution to the 8 Queens Problem, with `POPULATION` $= 350$

`POPULATION`	100	350
$\min f(p)$	150	70
$\min f(p, \tau)$	130	45

Table 2: Best mean iteration time to find a solution to the 8 Queens Problem

3.3 Discussion

Throughout this text and analysis, our approach has always been quite exploratory compared to algorithms implemented with more elitist selection methods, or using $p \leq 1$. The reason being that if we really cared for having a fast-converging algorithm, we could have used better suited algorithms for the N Queens Problem, such as Recursive Backtracking.

After running the algorithm more than 115 000 times, I can only say Michalewicz and Fogel [7] are unquestionably right; finding a consistently perfect configuration of parameters for a given problem can indeed feel like a hopeless task.

Notwithstanding, this problem and its implementation using a Genetic Algorithm has given us an insightful introduction into the world of metaheuristics, whilst seeing its advantages and limitations.

References

[1] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* 1984, pages 4–6. isbn: 0201055945. doi: `10.1016/0306-4573(85)90100-1`.

[2] L. Sean. *Essentials of Metaheuristics: A Set of Undergraduate Lecture Notes.* 2010, pages 29–52. isbn: 9780557148592. doi: `10.1007/s10710-011-9139-0`.

[3] G. Zäpfel, R. Braune and M. Bögl. *Metaheuristic Search Concepts: A Tutorial with Applications to Production and Logistics.* Springer, Heidelberg : 2010, pages 127–133. isbn: 9783642113420.

[4] T. G. Stützle. Local Search Algorithms for Combinatorial Problems - Analysis, Algorithms and New Applications:33, 1999.

[5] M. Gendreau and J.-Y. Potvin. *Handbook of Metaheuristics*, volume 146. Springer, New York : 2010, pages 123–127. isbn: 978-1-4419-1663-1. arXiv: `0102188v1`. url: `http://link.springer.com/10.1007/978-1-4419-1665-5`.

[6] D. E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning.* 1989, pages 170–172.

[7] Z. Michalewicz and D. B. Fogel. *How to Solve It: Modern Heuristics.* 2000, pages 277–295. isbn: 978-3-540-22494-5. doi: `10.1145/568438.568443`. arXiv: `arXiv:1011.1669v3`.