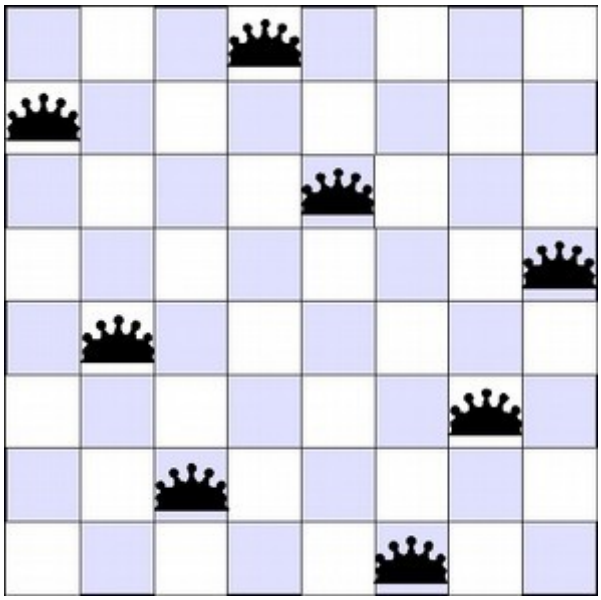


Report 1 (N-Queen)

Artificial Intelligence Module 2

Marco Roetta (876884) in collaboration with Christian Bernabe Cabrera (843382)

Project structure

The assignment is written in Python 3 language and is provided in multiple files.

The main script to run is called *main.py* and output all relevant information in the terminal.

The code has been written together with Christian while the report is individual.

Project goals

The goal of the assignment is create solvers for the n-queen problem for any N.

I will provide solvers using three different methods:

- Backtracking (depth first search standard and optimized)
- Local optimization (hill climbing with and without restart)
- Global optimization (genetic algorithm)

The next chapters will describe the problem and the used approaches, the representation/modeling choices made, and a comparison of the behavior, ending with pro/cons of the various approaches.

The N-Queen problem

The n-queens problem consist in placing N queens on a chess board, randomly in each row with only one queen in each column. The goal is to move the queens into positions where no queen is attacking any other queen.

All solutions to the problem can be identified using a depth-first search algorithm. With this search strategy, a queen is placed in any square in the first column. For each of these states, a queen is then placed in any rows in the second column for which the two queens will not be attacking each other. Queens are sequentially added to each column to generate all possible states.

Project background

Backtracking is the first method used in this project and consist of a recursive function that will try every possible solution, starting from the first column in the matrix of the problem space, up to the last one. The first solution found is shown in the console.

To improve the running time, some constraint have been placed:

- For every column, first check if a queen is present on the current row, if yes, skip to next solution;
- Then check only the upper left diagonal, then the lower left since there are no queens yet on the right part of the matrix;

The algorithm will place the first queen always on the first available cell.

It’s possible to obtain different solutions by randomly placing queens in the columns instead of picking always the first one starting form the top and a speed up if using an optimized class for constraint checks. This has been implemented in the **optimized Backtracking** code section.

The algorithm works in the same way as the previous one, but with the difference that it checks if a position is already occupied or not, instead of looking for every position for each queen.

The second algorithm implemented is **Hill Climbing**. This is implemented in two versions since there are more strategies available.

The first one is the **min conflicts hill climbing** where all current state successors are compared and the closest to the solution is chosen. The action chosen at each iteration is the one that most decreases the number of attacking queens, so the name *min conflicts*. It terminates at a minima, when no single move from the present state can reduce the number of attacking queens.

Both algorithms fail if there is no closer node, which may happen if there are *flat areas* in the search space. This is why we use the next Hill Climbing technique.

The second version is **random-restart hill climbing.** It will simply compute the previous hill-climbing, each time with a random initial condition (the initial column where to start) for the same problem. The best run is kept: if a new run of hill climbing produces a better result than the stored one, it replaces the stored result.

The last algorithm implemented is **Genetic** from the global optimization family.

The strategy is to create a population and run on every individual a fitness function that will determine the best mother and father to be used for the next generation run.

Those two individuals are combined with a crossover probability and with a probability of random mutation before being used to generate a new population.

Problem data set decisions

Backtracking uses an empty board as starting state.

For the other search strategies, a starting state is generated by placing one queen randomly in each column. Random restart hill climbing will use the same board but will start from a different position in the search state and try to find a better solution.

The board is implemented as a matrix object with nodes representing a queen [cell=1] or an empty space [cell=0]. In some cases it is converted into a list[N] where the position in the list is the column and the integer value inside the row position. This way I can improve performance during the search phase and print the solution in a compact manner.

Results analysis

A simple brute force solution, as stated in the textbook, will have a complexity from N^N to N! and will be completely impractical in a real scenario.

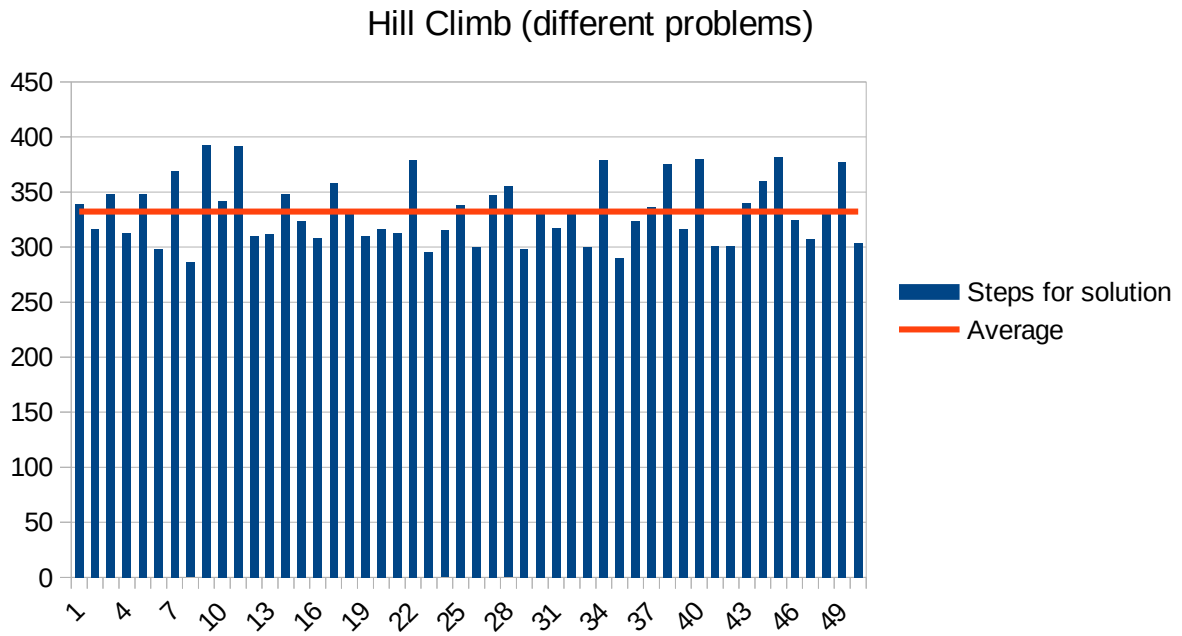
If I use backtracking without a specific optimization pattern, I achieve a solution with complexity from N^3 to N^2. By applying the techniques described above I can improve the execution time and approach the lower complexity limit.

Using our first proposed solution in the code, I achieved the positioning of 27 queens in 14 seconds. Optimized version does better with 256 queens in less than 10 seconds.

Hill Climbing instead, provides an even better improvement in the execution time per number of queen placed correctly.

The estimated code complexity is in the range N^2 to N^1.5. The randomness of the local search starting point allows to find a solution faster. In case of random restart it also increases the probability to find a lower number of steps required for the solution.

The algorithm count the steps to reach the solution from the provided problem. A lower number of steps is considered an improvement for this minimize problem.

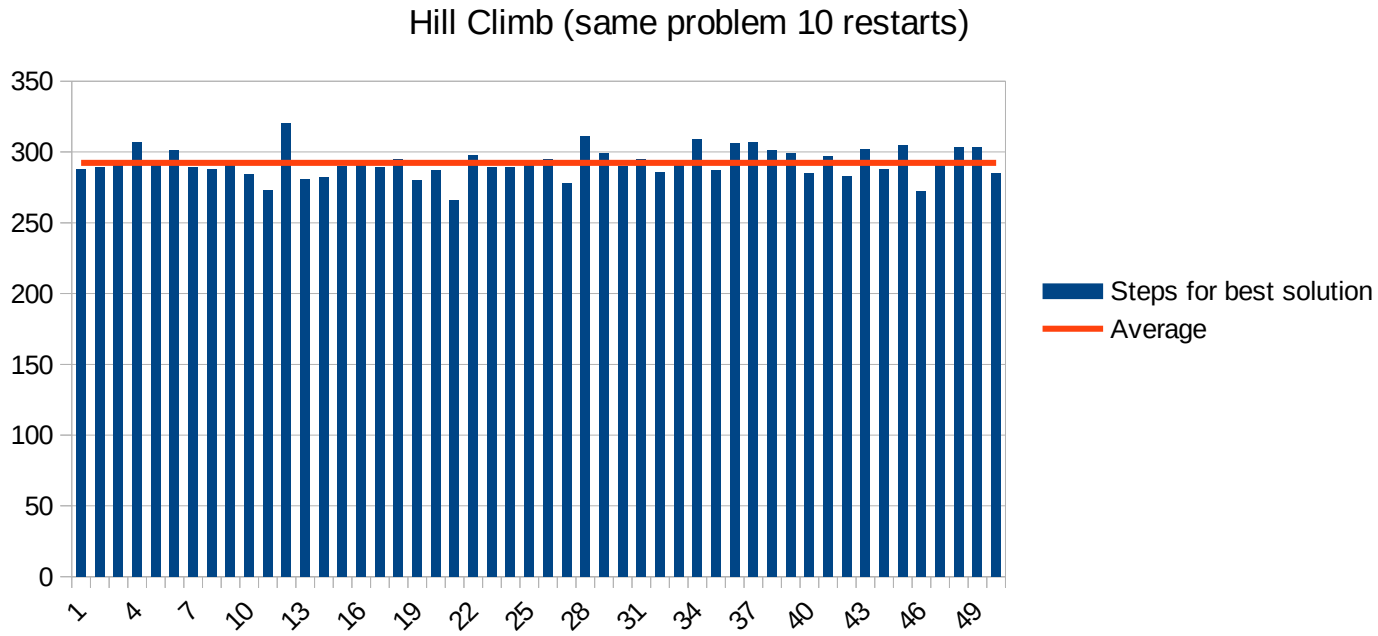


Using the hill climbing algorithm we can position 400 queens in around 15 seconds. Rounded average on 50 runs. This is a 14x improvement over standard backtracking and 2x over the optimized version.

The number of steps required to reach the solution is 332 on average of 50 runs, as visible in the graph above. Note that each run calculate the solution for a new problem.

Can I improve also the *quality* of the solution using an arbitrary number of random restarts for the same problem?

Yes. In the test I used 10 random restarts per run (again, to position 400 queens) and the average steps for the best run dropped from 332 to 292. An improvement of 14% with more consistent results.



The time required instead increased from 15 to 167 seconds per run on average, 10% more than expected. Probably due to object handling by python.

Basic Genetic Algorithm Tests

As last solver, we have first implemented a standard genetic algorithm as available in pseudo-code on the textbooks with no optimizations nor changes.

I used the recommended settings for a generic GA algorithms for the first run: 60% crossover with 10% probability of mutation and N*2 population size.

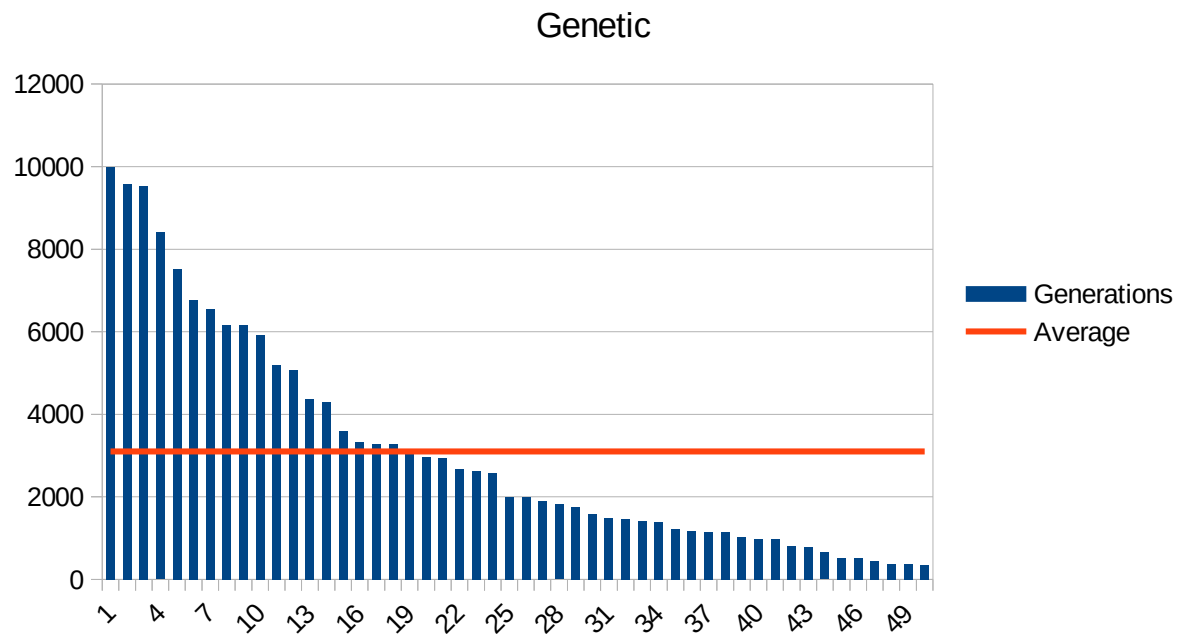
With those settings I run the program 50 times asking to position 20 queens.

The results was very variable and ranged from 340 steps to our arbitrary upper limit of 10000 steps.

The average of 50 runs is 3100 generations to find a solution as shown in the graph. The results have been ordered descending for simplicity. It is possible to see that results distribution favor the smaller values and this is a good result so far.

Generations is the number of generations created before a complete solution is found.

Only one run did not terminate before the generation limit.

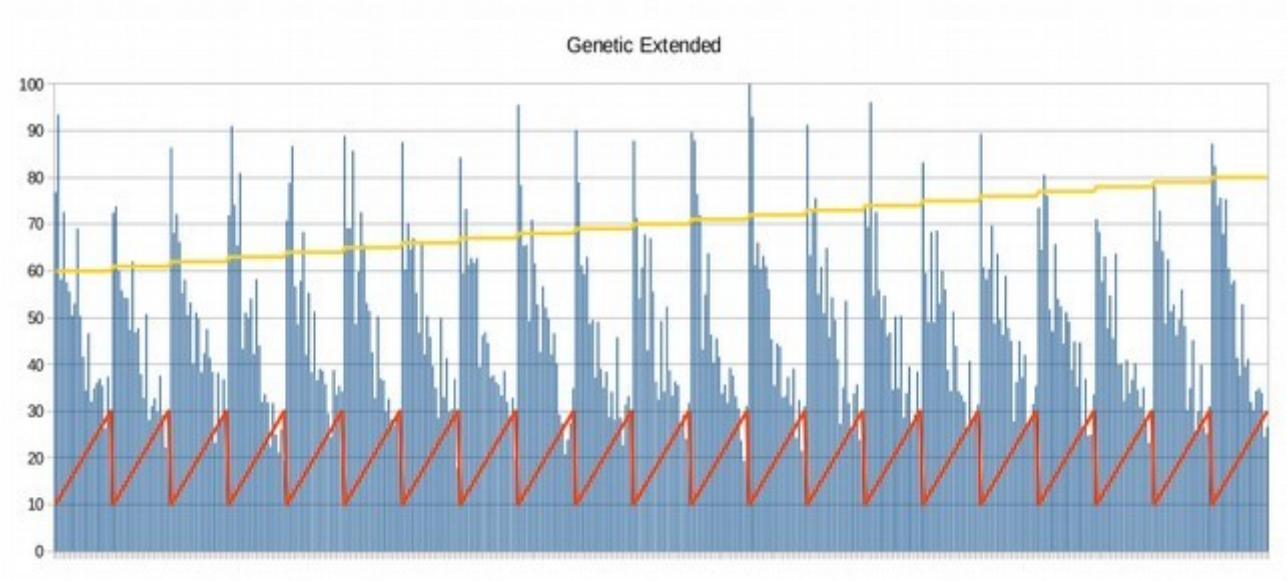


By tuning some parameters and choices of the genetic algorithm I can probably achieve better results.

Extended Genetic Algorithm Tests

One of the first things I tried was to plot how much of an impact have the crossover and mutation probabilities on the generations, before a valid solution is found.

Fixing the number of queens to position at 8, and population size at N*2 = 16 boards, the results are as following, after around 7 hours of computation time.



Yellow Line: probability of crossover, chosen between 60% and 80% from a precedent rough run;

Red Line: probability of mutation in each generation, chosen from 10% and 30%;

Blu Bars: average generation before solution on 100 runs with the same parameters. Data is normalized to 0-100%. 100% equals to 3150 generations.

I can see that an increase of mutation probability impacts a lot the number of iteration needed for a solution. This is expected, since it's required to minimize it.

But this could means also that the population size is too low to handle correctly the crossover between individuals or an error on father / mother choice.

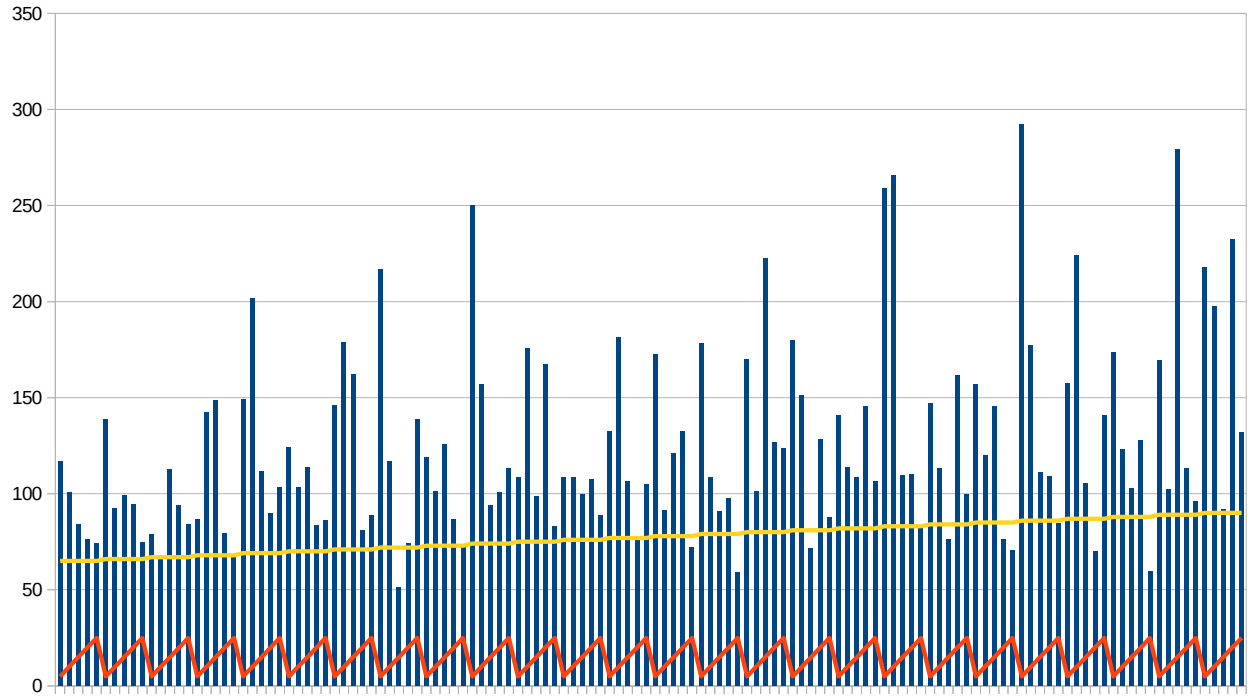
Crossover variance do not show particularly interesting changes in the results, only that around 78% it produces slightly more consistent results.

To try to improve the situation we have rewritten the code as following:

- Each generation is composed by (N^2)*2 individuals. More population will help picking different kinds of solutions in the first generations;
- For each generation, divide the population in N*2 groups of N individuals. This way I will pick the best of each group without leaving out someone.
- Run the fitness selection on all of those groups and select N*2 leaders.
- For each child of the new generation I randomly pick mother and father from the leader pool and apply crossover and mutation functions;
- Each time the father and mother solutions are equal (usually near convergence to solution) I increase the probability mutation of this generation only, by 5% per equal couple found up to 100% max. This way I will explore much faster the space around the solution.

Despite the increased population per generation, the speed up is good, from 7 hours it dropped to around 3 hours with a lower generations number before solution. This allowed me to do the following tests in a reasonable amount of time.

Results for 8 queens (100 runs average)



Yellow Line: probability of crossover, chosen between 60% and 90%;

Red Line: probability of mutation in each generation, chosen from 5% and 25%;

Blu Bars: average generation before complete solution on 100 runs with the same parameters.

The search space for best parameters has been set to 60-90% crossover and 5-25% of global mutation probability as this should be a good range to minimize the global mutation probability for this specific algorithm.

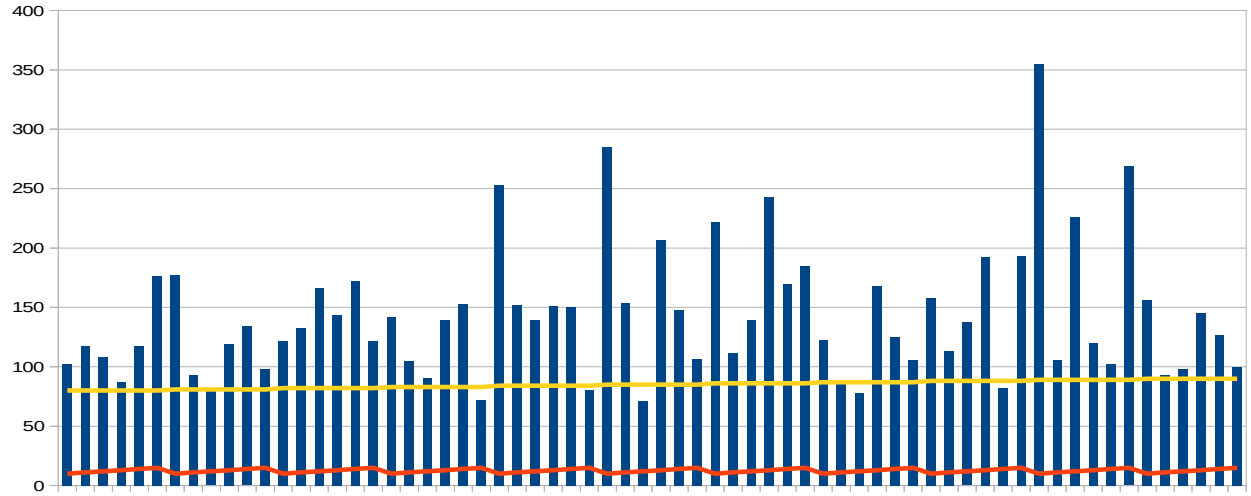
The graph shows a high speed up, required generations dropped from 3100 to 142 on average on all the search space.

Meanwhile, the progressive mutation increase after few generations makes the mutation probability parameter useless by design.

The crossover variance show that the best range is again around 80% but with a very small margin.

Let’s run it again in a focused range and see if we can determine which is the best value of crossover.

Results for 8 queens focused region (100 runs average)



Yellow Line: probability of crossover, chosen between 80% and 90%;

Red Line: starting probability of mutation in each generation, chosen from 10% and 15%;

Blu Bars: average generation before complete solution on 100 runs with the same parameters.

The best average number of generations required for a solution is 123 using 81% crossover probability, down from the previous 142.

The best parameters for this solution are then: mutation p. at any value and crossover p. at 81%.

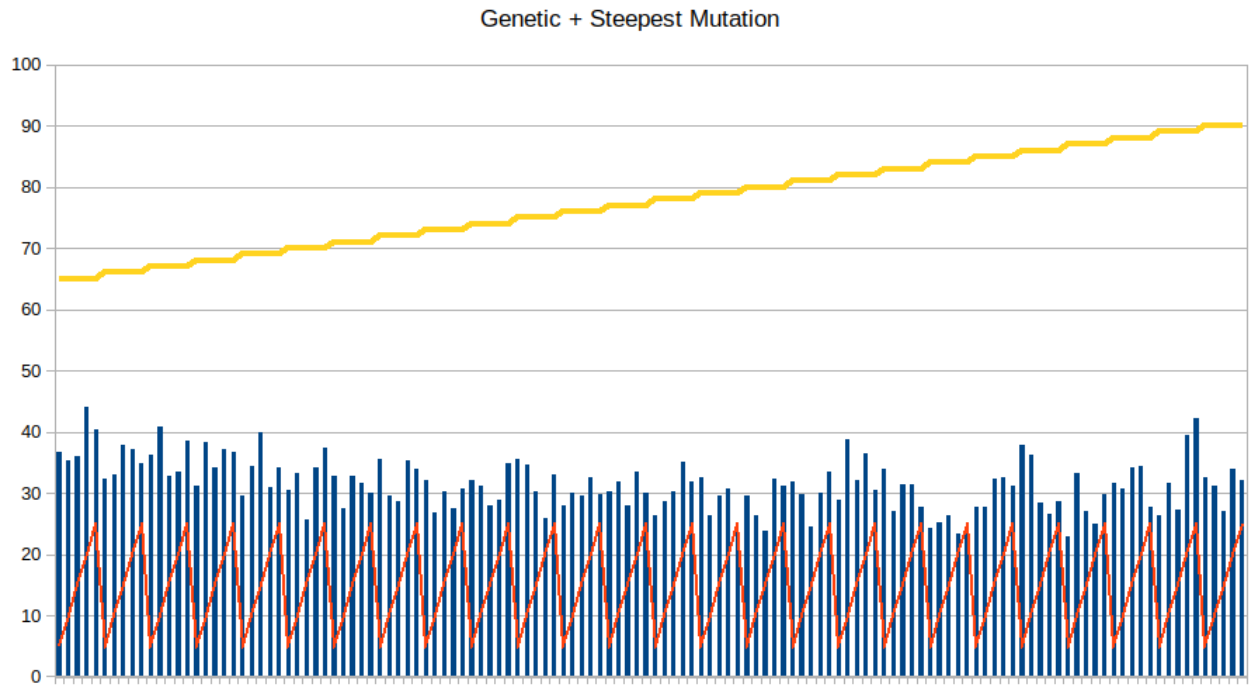
From this test I learned that using a progressively increasing mutation, activated each time the same solution is randomly picked to breed the next generation, leads to a faster search.

So, what happens if I just set the mutation at 100% probability the first time I pick mother and father equals?

Since they are randomly chosen from the mating pool, the probability of them being equals will increase only when approaching the complete solution (when fitness function will select two equal solution from two different sections of the current generation).

As soon as this happens, generating an almost whole new generation that differ by only a single queen position from the best solution so far, should greatly increase the probability of finding the complete solution or one with increased fitness.

Results for 8 queens + steep mutation (100 runs average)



Yellow Line: probability of crossover, chosen between 60% and 90%;

Red Line: probability of mutation in each generation, chosen from 5% and 25%;

Blu Bars: average generation before complete solution on 100 runs with the same parameters.

This version simply set mutation probability to 100% when the first “mother == father” occurs, during the new generation creation. The mutation is then set to the default value when starting the next generation.

It yields the best results overall, from 123 it now finds a solution in just 31 generations.

The mutation probability is again not very significant since we are going to always mutate every child each time mother and father are equal in a generation. I set it to 20% in the final code because it’s one of the value that leads to the best solution so far for all tried versions of the algorithm.

The option to mutate every child I used here is activated by placing the variable “nuke = True” in the *genetic_lib.py* file, inside the *new_generation* function.

The crossover probability shows that the best value for this version of the genetic algorithm is around 85%. I set this value in the final code.

As last test I run the whole code with a variable number of queens and see as the execution time changes at increasing values of N.

Results for all algorithms at increasing N

N	4	6	8	16	24	32
Backtracking	0.0	0.0	0.001	0.152	10.452	NaN
Opt. Back.	0.0	0.0	0.0	0.033	1.764	494.517
Hill Climb	0.0	0.0	0.001	0.01	0.019	0.094
Rand. Res. H. C.	0.001	0.023	0.001	0.081	0.2	0.422
Genetic Prog. Muta.	0.002	0.039	0.352	0.835	9.167	103.751
Genetic Ste. Muta.	0.001	0.016	0.014	2.848	29.076	62.449

Conclusions

- Backtracking (optimized):
  PRO: easy to implement, small on memory
  CON: higher compute time than hill climb
- Hill Climb:
  PRO: very fast with low variations, small on memory
  CON: easily stops on local flat solutions from time to time
- Hill Climb w/ restarts:
  PRO: fast, consistent, small on memory, better quality of results
  CON: require at least 10x time over Hill Climb for a good consistency
- Genetic:
  PRO: faster than optimized backtracking
  CON: complex to implement and slower than hill climb, requires good parameters selection (pop size, mutation prob and crossover prob)
- Genetic + Progressive Mutation:
  PRO: much faster than standard Genetic
  CON: complex to implement and still slower than hill climb, requires good parameters selection (pop size and crossover prob)
- Genetic + Steepest Mutation:
  PRO: a bit faster, independent from mutation parameter by design
  CON: still slow due to generation management, heavy on memory