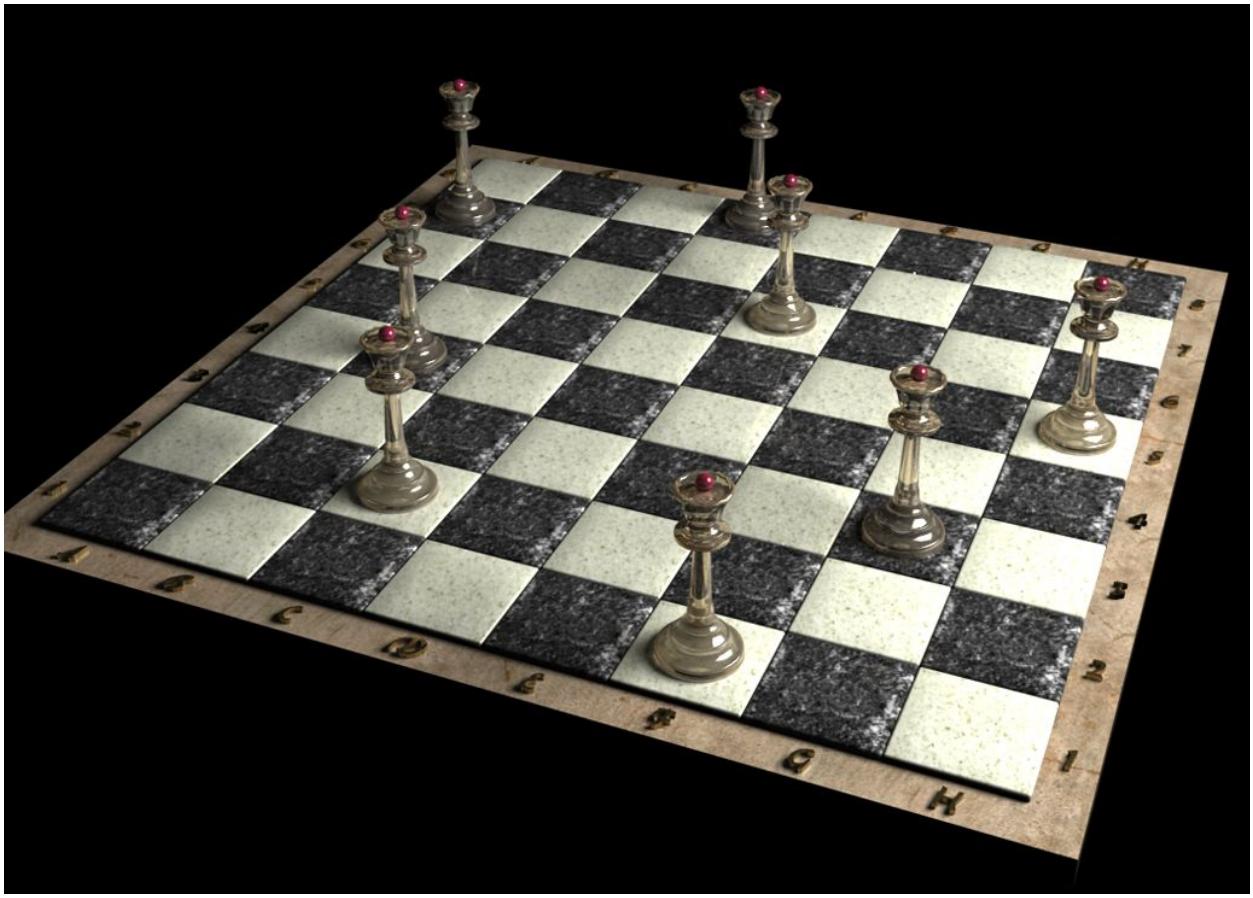
Artificial Intelligence: Knowledge Representation

N -Queen Problem

Cabrera Christian Bernabe 843382 - Roetta Marco 876884



Introduction

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other., *i.e.*, there is never more than one queen for every row, column and diagonal on the board.

Language and structure

The task has been done in Python. For every algorithm there is a single file. To execute them a main.py script has been provided to test the solutions.

Assignment goal

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

You should provide solves using three different methods:

- Constraint Propagation and Backtracking
- Local optimization (hill climbing)
- Global optimization

For Global optimization use your choice between simulated annealing or genetic algorithms.

Turn in the code and a report describing problem and approaches, the representation/modeling choices you have made, and a comparison of the behavior, pros and cons of the various approaches.

How work was divided

To do this assignment both of us worked together. The reports have been written separately.

Implemented algorithms

For this project, five different methods have been implemented, two for the first task and for the second, and one for the final one:

- Backtracking Algorithm
- Optimized Backtracking
- Hill-climbing (min-conflicts version) and Hill-climbing (min-conflicts) with random restart
- Genetic Algorithm

Backtracking Algorithm (depth-first search)

The idea for the backtracking algorithm is simple: it places queens one by one in different columns, starting from the leftmost column.

Everytime before placing a queen in a column, it checks for clashes with already placed queens. To do this some basic constraints have been used through an utility function that is called when the Queens have already been placed from [0 , column -1]:

- Given a column and a row, it checks the row on the left side. If a queen was already present, then it tries the next row
- If it doesn't fail it, it'll check the upper diagonal on the left side
- Then if the previous check passed, it'll check the lower diagonal on the left side
- If all the checks were passed, it'll return True indicating that it's safe to place a Queen in that position

After that the Queen has been placed, tries recursively to place the missing queens. If the returning recursion, didn't lead to a solution, then it removes the current placed Queen and proceeds to find the other possible solutions. Eventually it'll return false if the current col-Queen cannot be placed in any row.

In the current state the backtracking will always lead to the same solution, since the rows are checked in a progressive order. This could be changed by shuffling the order of the rows when looking if the position is safe. However we thought that it wasn't necessary for this assignment.

Optimized Backtracking Algorithm (depth-first search)

The idea behind the implementation of this algorithm was to try to improve the running time from the previous algorithm. In particular I notice, through research, that an optimization could be achieved by improving the "check if position is safe" which complexity is O(N) in the previous algorithm.

In order to to improve this, it was necessary to save the row, column, the upper left and the lower left diagonal that each Queen occupies.

The diagonals are accessed with [row - column] and [row + column]

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

Hill-Climbing - (with min-Conflicts)

For the second strategy we used Hill-climbing min-conflicts algorithm. The point of local search is to eliminate the violated constraints and can easily be extended to constraint optimization problems. Because of that, all the techniques for hill climbing can be applied.

This algorithm starts with all the Queens already placed, each one in a different column.

Once the Queens are placed, it tries to improve the current situation by choosing randomly a conflicting Queen and then placing it in the row with the least number of conflicts (if there're are more than one possible rows, then it'll select randomly one of them).

Sometimes the algorithm can have a problem of local minima, when the current arrangement cannot be improved.

There are a few possible ways to eliminate these situations. For example one could have been to implement a way to detect if we're in a situation of local minima and if we're, then randomly choose one of the conflicted Queens and place it in a random row, to create new conflicts and get out of the situation. With these method and the number of steps sufficiently large, a solution could be guaranteed most of the time.

Another way to guarantee a solution, instead could be the following.

Random Restart Hill-Climbing - (with min-Conflicts)

With this technique we are just executing the same algorithm, for a fixed number of times, also known as restarts. Although usually random restart is executed with a randomly generated initial state, we opted to use the same board each time to find a better solution with the same initial state. This is because the min-conflicts hill climbing will randomly choose a conflicted Queen, giving out different results.

For every run the solution found is confronted with the best one found so far, and if the new one is the best, then it just replaces it.

Genetic Algorithm

For global optimization, the choice was to implement a genetic algorithm. The method of genetic algorithm is probabilistic; whereas, traditional algorithms use deterministic methods.

The general strategy applied is the following:

- Step 1: Random generation of an initial population given a fixed size;
- Step 2: Calculate the maximum fitness value;
- Step 3: Start a while cycle until a solution is found or a maximum number of generations is reached;
- Step 4: Evaluate each solution in the population by calculating its fitness value;
- Step 5: Generate a new population from the current one by applying:
    - Selection of parents (with a small modification of the tournament method):
        - The population is being divided in N * 2 parts
        - For each sub-list the best candidate is selected and added to a list of possible Leaders
        - Choose randomly two individuals from the leader list to become parents
        - When approaching convergence, we increase the probability of mutation of the childs , by escalating the mutation rate by 5% if the two possible parents are equal
    - Crossover with a fixed probability and childs generation
    - Mutation with a fixed probability
    - Repeat Selection until the new population size is the same as the old one
    - Return new population

- Step 6: substitute the old population with the new one
- Step 7: check if in the new population there is a suitable solution (there might be more than one, we just look for one). If there's a solution, stop and return it . Otherwise if the maximum number of generations has been reached just stop.
- Step 8: Go to step 4 and repeat

System Setup

I7 8700k	16GB Ram	ssd 850 evo	python 3.7	Windows 10

Tests and Results

Backtracking vs Optimized version

N	Backtracking	Optimized
8	0.0193 seconds	0.0084 seconds
16	0.245 seconds	0.0574 seconds
20	6.1423 seconds	1.037 seconds
24	18.372 seconds	2.719 seconds
28	172.12 seconds	21.985 seconds
32	NaN	302.247 seconds

By confronting the two versions of the backtracking algorithm on my system 10 times, on average gave me these results.

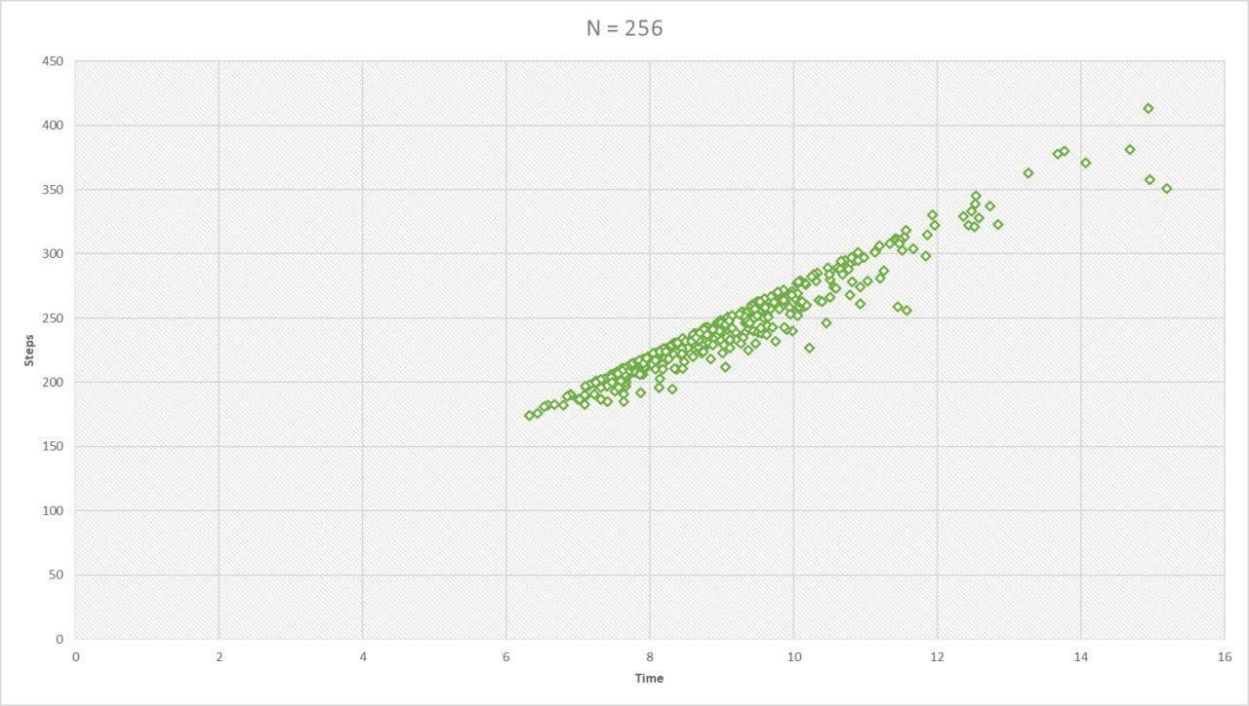
As expected there's a substantial difference between the simple one and the optimized version. However the complexity is still exponential for the backtracking method and while it can be indeed useful to find, for example, of the possible solutions given a fixed N, it is not suitable for big problems.

Although I recognize that this program is limited by the python interpreter, which is using only the 9-10% of my CPU power because it only runs as one single process by default. This limitation could easily avoided, by producing a parallel version, increasing the speed significantly by starting multiple threads.

N	Hill-Climbing Min-Conflicts avg. Time	Average Steps	Failed Tests
8	0.0042 seconds	61	35
16	0.0119 seconds	82	3
32	0.0582 seconds	90	0
64	0.267 seconds	107	0
128	1.505 seconds	153	0
256	9.074 seconds	240	0
1024	8 minutes 54 seconds	689	0

For the Hill Climbing min-conflicts I decided to test on average 500 runs, with a max number of iterations of 10000 for every N (not for 1024 which I test it only 5 times).

The results were quite interesting. First of all, I could notice, as suspected, that the algorithm could get stuck in local minima. However I thought that it will always get stuck, independently of the dimension of N. Instead, it seems that it had more problems when N was small. For example, with N = 8 it failed 7% of the time, which it's quite big. While with N = 16 the failure rate went down to 0.006% on average. While for N big enough, the algorithm never got stuck in local minima, probably due to a larger number of possible choices, despite the overall number of steps increased. In the following graph we can notice how the average results were obtained in more or less 9 seconds. We can also notice that there are a few results that took longer time, almost 5 seconds more of the average time.



Another thing I notice, is that despite in the text-book was written, that this algorithm could solve very large problems in a sufficiently small number of steps, the results stated in the book were quite more impressive than ours.
However this it's just a nit pick, since without doubt, for scientific researches this algorithm was written first of all in C++, secondly with probably far better auxiliary methods than ours. For example, a tabu search, with a small list of recently visited

states could have been implemented, to forbid the Queens to return to those states temporarily. This could have help to reduce significantly the number of steps, so the computation time.

Overall this algorithm resulted to be extremely good.

We also set up the version with **Random Restarts** to try to improve the number of computed steps given a fixed initial state.

N	Worst n°Steps	Best n°Steps	Restarts	Total Time
128	219	118	25	37.172 seconds

The sacrifice of applying random restarts is of course the global time lost in execution, which would be more or less equal to : Average Time of N * Number of Restarts.

The last implementation was the **Genetic Algorithm**. It's structure it's more or less a classical one, with some little variations that we made for the selection of the best fitted parents for the next generation as explained in the overall structure.

We made the following choices:

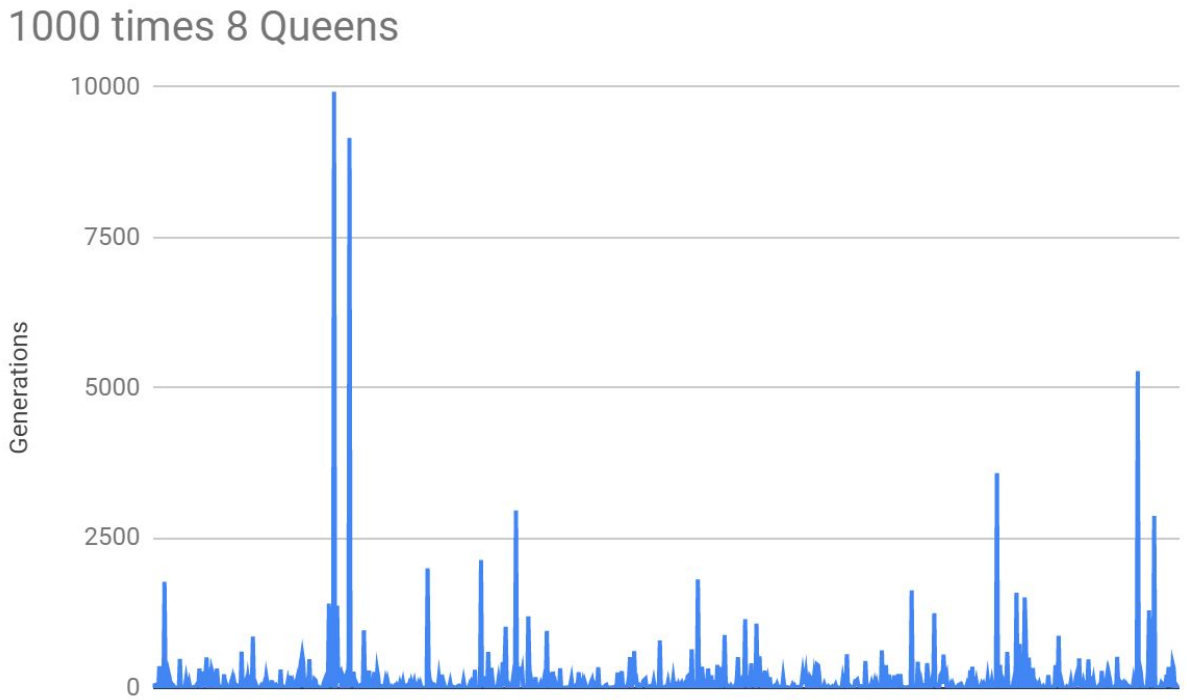
The size of the population is equal to N^2*2. This resulted in a reasonable size for each N problem that we tested (even though for big problems probably a fixed population is better, otherwise the computation time will increase).

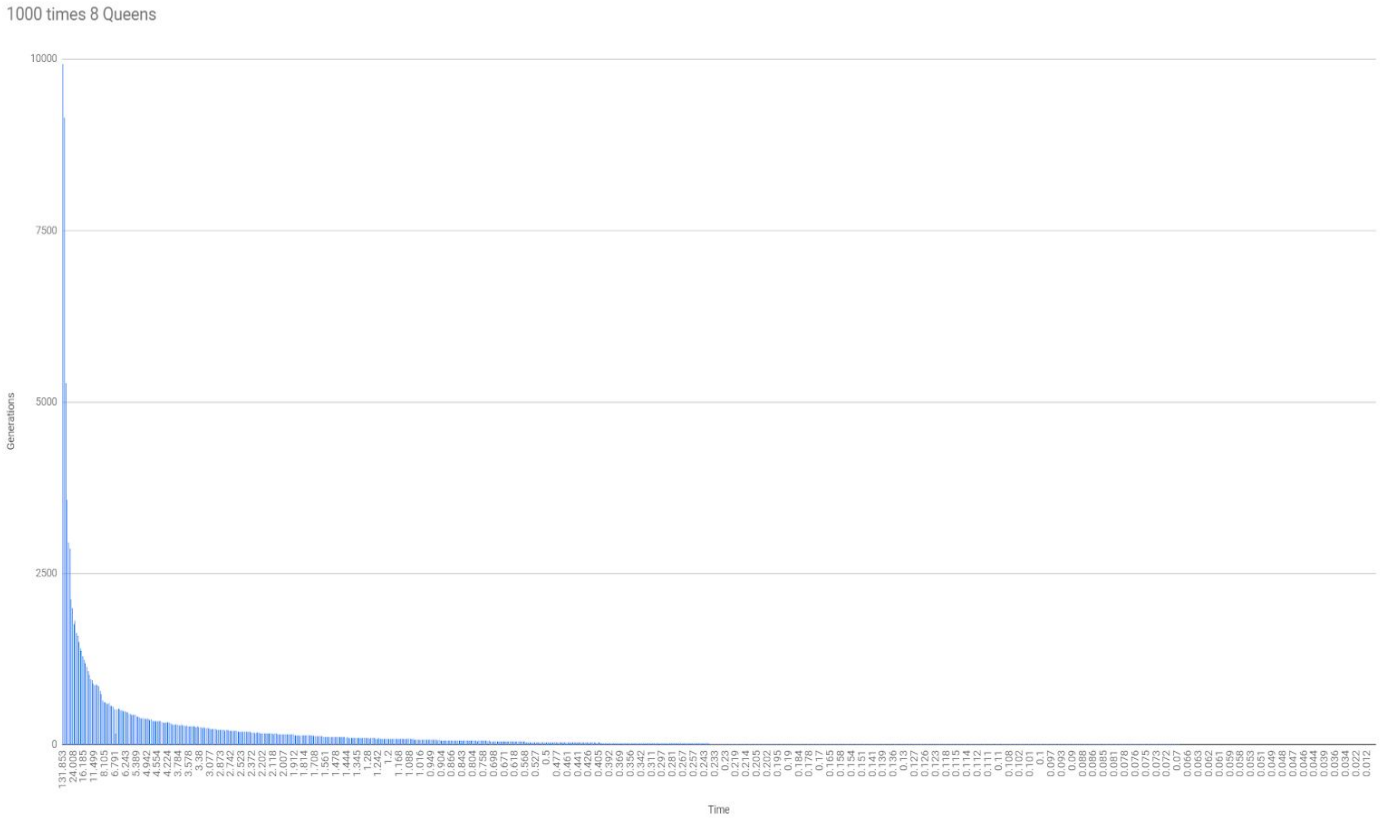
We tried to keep the probability of crossover high enough. A good compromise was to set it at 90% while we wanted the probability of mutation as low and reasonable as possible. We ended up with 15%.

The maximum fitness is obtained by this formula = N*(N-1)/2, which represent the pairs of non attacking Queens.

I conducted two times an exhaustive tests for 8 Queens. The first time with probability of crossover at 90% and probability of mutation equal to 10%. The second one with probability of mutation equal to 15%. The first test took more generations to complete, so I kept the second test. Running the algorithm 1000 times on average I found out that on average it found out a solution around 130 generations, were the minimum generation that took it to find a solution was 1 (extremely lucky case) up to the upper limit of 10000 generations.

In the next two graphs I represented, in the first the data as it was, once all the 1000 runs were done. In the second I represented the data in terms of time it took each solution to be found in a decreasing order.





As we can notice, the execution time it's actually good enough.

Also the bad runs (the ones that took it more than 400 generations to find a solution) are only the 0.058%. Which in my opinion is impressive.

Despite that, sometimes the algorithm can found itself in an unlucky run and A: not end, B: ends taking a significant amount of time in respect of the average time.

The main problem with genetic algorithms probably is that it takes time to find the perfect parameters for a given problem, for example the size of the population, the probabilities of mutation and crossover ect.

Also a possible optimization for this algorithm was to implement a better version of the crossover to reduce possible conflicts generated by it.

Conclusions

Overall I think the best solution was the Hill-Climbing algorithm. It simply works better than the other solutions.

The optimized backtracking resulted to be much better than the plain solution.

The Genetic algorithm however, turned out to be the most fascinating one. Being able to even able to solve a N=100 in with a population size of 200 in around 5 to 15 minutes (I executed it only two times).

It would have taken to the backtracking algorithm around a day, but it took the Hill-climbing only a second and a half on average.