Comparing and analysing the performance of n-queen problem solving algorithms.

Word Count: 4000

Extended Essay

Computer Science

Table of contents

1. Introduction

N-queen problem is a general version of 8 queens puzzle, which was originally created by a German chess player and puzzle composer Max Bezzel. The objective is to place 8 queens on an 8x8 chess board in such way that no queen is attacking any other. Queens can move any amount of tiles on horizontal and vertical lines or diagonals. The problem quickly caught the attention of mathematicians all over the world, who wanted to find all the possible solutions and prove that there could not be more. The first solution was proposed in 1850 by mathematician Franz Nauck[1], who also expanded the problem for a board with n^2 cells and n queens. Figure 1 shows a possible solution to the problem for n = 8.

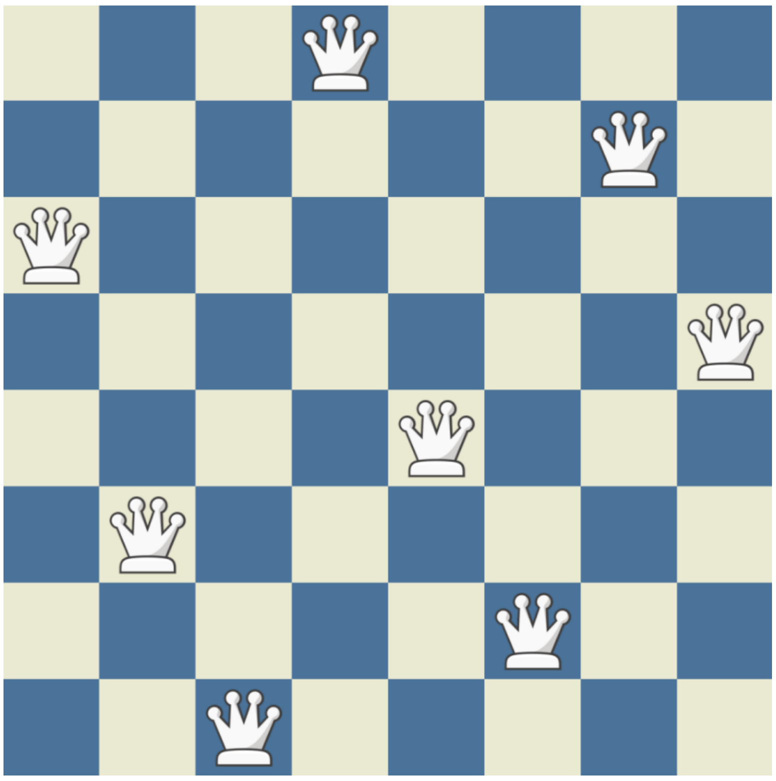

Figure 1 one of the solutions for the 8 queen puzzle.
Image created in analysis tool on https://www.chess.com/analysis?tab=analysis (Accessed 05.03.2024)

In modern mathematics the puzzle is still unsolved for bigger sizes as it is non-trivial and no formula for getting solutions exists. As it requires an analysis of huge amounts of data and

[1] Ball, Walter William Rouse. (1905) "Mathematical Recreations and Essays," Macmillan, New York, pp. 97–103. Link: https://web.northeastern.edu/seigen/11Magic/Books/Rouse%20Ball.pdf (Accessed 06.03.2024)

solving it by hand is very inefficient, it has turned from a game theory issue to an algorithm design problem.

The aim of this paper is to compare four completely different algorithms designed to find solutions for this puzzle. The first one is a modified brute-force algorithm and will be used mostly as reference, the second one is a backtracking algorithm based on a depth-first approach. The third is an iterative repair algorithm, and the final one is a genetic algorithm. Thus many approaches will be compared including machine learning, heuristic problem solving via greedy algorithm, and turning the problem into a tree. In this research all the algorithms will be compared by the times of finding one solution, as finding multiple is redundant for most real-life uses of applications and would make the topic much closer to mathematics and game theory than an algorithm analysis.

This research could prove useful as an indicator of algorithm choice in many other problems, as in computer science the n-queen problem is very often used as a benchmarking tool, allowing to test algorithms in constraint programming. Its simple rules allow for easy understanding, while finding solutions is very demanding, making this a very good way to approximate how different algorithm designs would work on solving similar problems. Few examples of which are task scheduling or microchip building. In the former the challenge is that tasks assigned to one entity can not overlap, and in the latter the issue is much more physical – micro components can not interfere with each other. The restraints addressed in those problems are congruous to the restraints in the n-queen puzzle. According to Vishal Kesri and Manoj Kumar Mishra “The N-queen problem can be applied in many different areas, like parallel memory storage schemes, VLSI testing, traffic control and deadlock prevention. It is also applicable to find solutions of those problems which require permutations like travelling salesman problem.”[2]

[2] Kesri, Vishal & Mishar, Manoj. (2013). “A new approach to solve n-queens problem based on series,” International Journal of Engineering Research and Applications. 3. 1346-1349.
Link: https://www.researchgate.net/publication/321192822_A_new_approach_to_solve_n-queens_problem_based_on_series (Accessed 06.03.2024)

2. Algorithm Structure

2.1. Brute-force algorithm

A brute-force algorithm works by checking all solutions one by one. In the most basic version there are $\binom{n^2}{n}$ possible queen placements on n by n board, making the programme have a complexity of $O(n^n)$, as in the worst-case scenario, almost every possible position has to be checked. Furthermore, the process of verification of a position requires a double loop.

The improved version of the algorithm excludes all the cases where queens are on the same horizontal and vertical lanes, which allows for only n! combinations. The time complexity is then O(n!). The space complexity is only O(n) as one-dimensional array of integers is used.

Obviously, such solutions are very slow and with the increase of the size of the board, the time increases greatly.

The brute force algorithm was implemented to go through every permutation of the set (1,2,…, n), where x^{th} number in the sequence is the column of the queen in x^{th} row, inputting them into a double loop which checked if any two queens are on the same diagonal. This is easy to do as in such case the absolute values of differences of rows and columns are equal. In order to save time, the second loop ends as soon as it reaches the position of the first loop instead of checking every row, ensuring that each pair of queens is only checked once.

The algorithm switching between positions works on a simple basis.

1. Generate a new board with queens placed on top-left to down-right diagonal.
2. Move the queen in the last row one tile to the left, moving the queen in the row it just moved to go one tile to the right.
3. In case that queen was on the most left column, it goes to the most right one, moving every other queen one tile to the left. If this occurs, Step 2 is repeated with the queen

one row higher. If the same thing happens then, the process is repeated with the next queens as long as they are on the leftmost column.

4. Repeat from Step 2 until a solution is found.

Also, for the sake of the accuracy of the experiment, there was a second version of this algorithm created, which starts from a random position. In case in Step 3 the queen from the highest row would be moved from the most left spot, the board would be reset as in Step 1, allowing the program to go through boards in a cycle. The solution from the random start would show the real efficiency of the algorithm instead of the proximity of the solution to the default starting spot.

2.2. Backtracking algorithm

Backtracking algorithm also searches through every possible position, although in contrast to the brute-force it works by building positions a queen at a time and eliminating them as soon as a conflict occurs instead of finishing the solution.

This is equivalent to turning the game into a tree, in which the root node is an empty board and the children of each node are boards with one queen added in the next row to the one filled in the parent node. A node with q queens placed has n-q children. The algorithm searches through the tree from the top, and if one node has two queens attacking each other, it moves to the next branch of the tree, saving time on the verification of children of that node. This type of searching algorithms is classified as depth-first search.

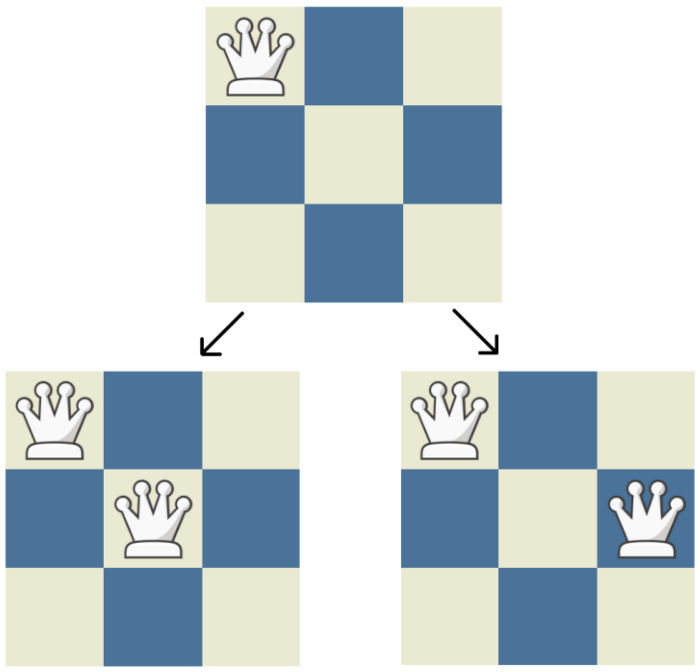

Figure 2 example of children of a node in a tree for n = 3
Image created in analysis tool on https://www.chess.com/analysis?tab=analysis (Accessed 05.03.2024)

Although the backtracking algorithm removes many options in contrast to brute-force algorithm, its time complexity is still O(n!). It also has space complexity of O(n).

The order of positions the algorithm goes through is different than the one in the brute-force. It starts with placing a singular queen on the top-left square of the board. Then it repeats the following scheme:

1. Add the queen in the first spot from the left that is not attacked in the first free row.
2. If there is no such spot, move the queen from the previous row to the right until it finds the next available spot.
3. In case the queen from Step 2 does not find a place, repeat the step with the queen row higher.

The checking algorithm here works in a similar way as in the previous algorithm, but it only compares the spaces for the queen in the first unoccupied row with queens from previous rows. The program outputs the solution when it detects a queen placed in the last row.

Similarly to the brute-force algorithm this one also had a separate variation created that would start with a random position. In this case, the position of the 1st queen would be randomized. To avoid the program ending with no solution, if the 1st queen would be moved through the right edge, it was to be placed in the most left square, looping all positions the program would go through.

2.3. Genetic algorithm

The genetic algorithm is a partially random algorithm that is based on machine learning. The one used in the experiment is based on a paper[3] by Uddalok Sarkar and Sayan Nag. It works in the following steps:

1. Generate a population of many entities, called chromosomes. In this case, each of them is a vector[4] containing a random permutation the set of integers from 1 to n, representing a position on the board, with each queen in unique row and column. Also create a bad chromosome repository, which is not changed at any stage of the program and contains bad chromosomes in order to avoid a local optimum.
2. Rate each chromosome by how close it is to the desired solution, the consensus is that such rating in genetic algorithms is called the fitness of a chromosome. It is equal to 0 when no queens are attacked and (n-1) when all queens are attacked.
3. Eliminate part of the population with worst scores.
4. Using the crossover function fill the newly created empty spots with children of remaining chromosomes.
5. Mutate some of the new chromosomes. The mutation is done in order to avoid local optima and is achieved by randomly changing a small part of the chromosome.
6. Repeat from Step 2 until a solution is found.

As it has much randomness included and a combination of two chromosomes with good fitness can result in a child with a bad one, the algorithm is not guaranteed to give a solution in a finite time. Many genetic algorithms usually aim to find solutions close to optimal, although in this case the algorithm allows for finding an optimal solution with a high chance of the time not being too long.

[3] Sarkar, Uddalok & Nag, Sayan. "An Adaptive Genetic Algorithm for Solving N-Queens Problem," Department of Electrical Engineering, Jadavpur University, Kolkata, India.
Link: https://arxiv.org/ftp/arxiv/papers/1802/1802.02006.pdf (Accessed 06.02.2024)

[4] In mathematics vectors are elements consisting of a single row or column of a matrix. In this case, the board is a n by 1 one array – a structure in computer science analogue to a matrix in mathematics and the vectors here are just the positions of the queen.

Assuming the main population counts m chromosomes, the space complexity is O((m+$\sqrt{n}$)×n).

In order to generate the initial population the method generating a random permutation was created. It works in a loop of n iterations. Each time it finds a random empty index in an array and then fills the position with increasing integers. Using this method an initial population of size 1000 and a bad chromosome repository of size not bigger than $\sqrt{n}$ are created. Then the population was rated via fitness function.

Bad population repository

During each iteration of the algorithm two random chromosomes from this repository are chosen and the results of their crossover might be added to the main population if they have better fitness score than any of the main population chromosomes. The same process is repeated with one random chromosome from bad population repository and one random from the main population.

Fitness function

This works by computing two values for each queen first one being row number minus column number and the second one being row number plus column number. Those values were put into two separate arrays and then sorted by quick sort algorithm.[5] Then a single loop checked for two consecutive values in each of the arrays. If found the fitness value was increased by 1. This makes the fitness value increase by one for each attacked queen except the first one. The only case it is 0 is when no queens are attacked.

Then the main loop starts and iterates until a solution is found. In each iteration the half of the population with the highest fitness scores is replaced by the results of the crossover function ran with remaining half of the population. The chromosomes were randomly paired for crossover. After replacing the worse half of the population the new 500 chromosomes are rated. Then two random chromosomes from the bad population repository are crossed over

[5] Baeldung (2024), "Quicksort Algorithm Implementation in Java."
Link: https://www.baeldung.com/java-quicksort (Accessed 08.02.2024)

along with one from the same repository and one from the bad population. The resulting chromosomes replace the worst chromosomes in the main population if such exist.

Chromosome crossover function

The chromosome crossover function works by finding two random numbers, which set the range of rows in which positions from the 1st chromosome are transferred. Those positions are unchanged. Then the remaining rows are filled with the data from the 2nd chromosome without changing the order, but excluding values that were already copied from the 1st one in order to avoid one column having two queens. This is done by copying a range from the 1st array and then going through the 2nd array, excluding those values that already exist in the new array.

The loop stops when the chromosome in the first position in the array after sorting has a fitness score of 0.

2.4. Minimum conflicts algorithm

The heuristic minimum conflicts algorithm is also randomized. It works on one position by repeatedly finding the worst placed queen and improving its position. It does not guarantee to find a solution in a finite time. Furthermore, it can get stuck in a local optimum, guaranteeing the lack of solution even in the infinite time. Such a thing happens when a sequence of few positions is repeated in a loop. To prevent this the program could restart after the same position is achieved many times. Usually heuristic algorithms aim to finding a solution close to optimum, in this case, similarly to the genetic algorithm, the program will not stop until finding a complete solution. The space complexity is O(n), as the program uses only one array of size n.

The algorithm starts with generating a random position using the method used in genetic algorithm. Then it starts working in a loop.

In each iteration the amount of conflicting queens for each of the queens is calculated, this works in the same loop as the one in the brute-force algorithm, although instead of returning false it is increasing the values in the conflicts number array for both queens that are attacked. If each value in the array is equal to 0 after the loop ends, the solution is deemed valid and the loop breaks. Else, it chooses the random queen among those with the most amounts of conflicts. And then the next algorithm calculated amounts of collisions of the other spots in the same row as if the queen moved here. Importantly, in order to avoid local optimum – a state where algorithm does not make any progress the queen has to move to a different spot. Then out of those spots the one with the least conflicts is selected. In case there are many such spots, one is selected at random.

3. Experiment Methodology

The goal of the experiment is to gather sufficient data for the analysis. In order to do so, the algorithms discussed above were implemented into Java, as it is one of the most commonly used programming languages in the world. The full code is in the appendix.

3.1. Hardware

The computer on which the experiment is performed has the following specifications:

Processor: AMD Ryzen 7 2700 Eight-Core Processor 3.20 GHz

RAM: 16 GB

Motherboard: ASRock B450M Pro4

Graphics Card: NVIDIA GeForce 960 1GB

3.2. Experimental procedure

The experiment aims for collecting times of execution of the algorithms. To ensure that results are accurate the times are an average of many runs of the algorithms. Of course, as the size of the board is increasing, the times will get larger and therefore sample size will decrease. The algorithms will be tested in the range of values of N is from 4 to 30 with increments of 1 and from 35 to 100 with increments of 5. Due to huge increase in the difficulty of the problem with the increase of n, in order to avoid insanely long runtimes, if some algorithm averages more than one minute for at least 5 executions, it will not be tested further, i.e. for higher or equal n. On the other hand, the algorithms that manage to compute for n = 100 in the time will be tested further, until the last value of n that does not exceed the runtime of one minute for such algorithm is found.

The times are to be collected using Java methods System.currentTimeMillis() and System.nanoTime(). Which read system time and return it as a long type variable. Then the times from after and before the execution of the algorithm can be subtracted, giving the accurate runtime.

Furthermore, as mentioned before, the versions with the random starting position of brute-force and backtracking algorithms will be tested.

4. Data presentation and analysis

4.1. Experiment data results

Figures 3, 4, 5, 6 and 7 show the results of the experiment.

n =	4	5	6	7	8	9	10	11	12	13
brute-force	207	225	1 780	2 446	28 530	221 574	338 975	935 391	6 493 851	21 590 703
brute-force random start	392	308	2 204	3 145	14 726	48 844	208 844	972 024	3 526 170	12 249 683
backtracking	128	84	873	225	6 427	2 115	7 120	3 285	29 154	12 640
backtracking random start	339	115	869	410	2 520	4 432	11 459	11 729	44 063	50 396
minimum conflicts	5 111	4 259	96 012	52 780	187 205	354 793	281 725	145 610	131 554	150 836
genetic	672 664	877 719	821 862	857 422	1 088 335	1 804 743	5 649 295	11 806 324	18 094 422	29 021 372

Figure 3 runtimes of algorithms for 4 ≤ n ≤ 13. Times are presented in nanoseconds.

n=	14	15	16	17	18	19	20	21	22
brute-force	323	1 900	3 246	20 275	236 231	over 1 minute, no further data			
brute-force random start	52	200	687	3 383	7 111	20 028	over 1 minute, no further data		
backtracking	0,29	0,21	1,89	0,96	8,44	0,50	45,80	1,98	46,60
backtracking random start	0,24	0,20	0,77	0,85	0,32	1,99	19,70	11,40	135
minimum conflicts	0,17	0,16	0,18	0,19	0,20	0,22	0,23	0,25	0,28
genetic	49	74	115	160	222	272	339	442	522

Figure 4 runtimes of algorithms for 14 ≤ n ≤ 22. Times are presented in milliseconds.

n=	23	24	25	26	27	28	29	30
backtracking	6	111	13	116	128	886	504	20 883
backtracking random start	52	202	359	559	184	4 969	722	34 991
minimum conflicts	0,42	0,86	0,55	0,30	0,70	0,49	0,53	0,35
genetic	557	630	674	781	767	959	838	853

Figure 5 runtimes of algorithms for 23 ≤ n ≤ 30. Times are presented in milliseconds.

n=	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
minimum conflicts	0,351	0,517	0,632	0,756	0,917	1,055	1,227	1,449	1,704	1,886	2,178	2,317	2,885	3,133	3,345
genetic	853	1 380	1 459	1 337	2 109	2 623	2 442	2 978	3 069	4 067	4 353	4 275	5 140	5 983	6 837

Figure 6 runtimes of algorithms for 35 ≤ n ≤ 100. Times are presented in milliseconds.

	highest n	time
backtracking	33	54953
random backtracking	33	30148
genetic	274	57283
minimal conflicts	4313	59197

Figure 7 highest n for each algorithm with runtime under a minute. Times are presented in milliseconds.

4.2. Brute-force algorithm results analysis

Figure 8 shows the results of the experiment transferred onto a graph.

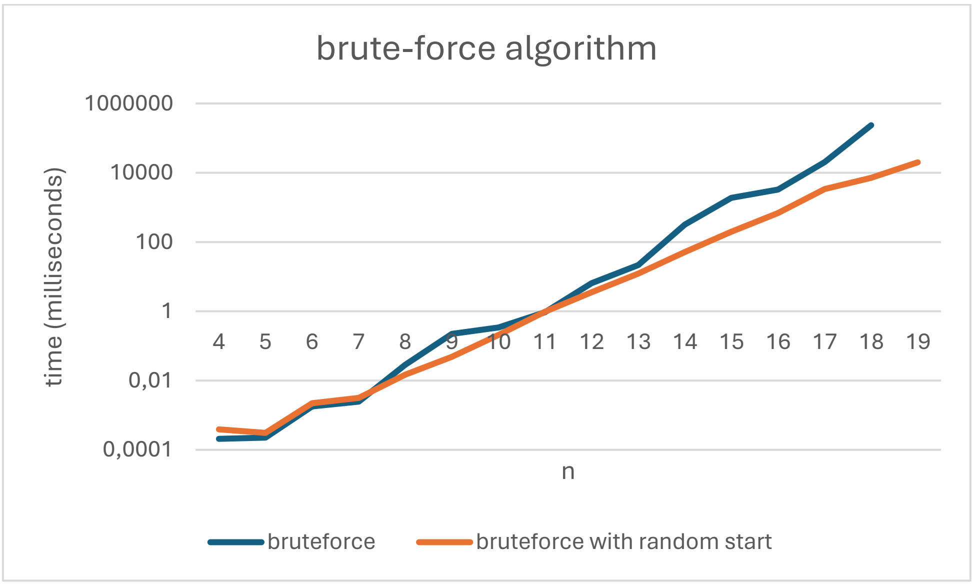

Figure 8 board size to time for the brute-force algorithm with exponential scale.

The time axis on the graph is in exponential scale in order to make the interpretation easier. As can be seen, the representation of values is very similar to linear function, meaning that the time complexity of the execution of the algorithm is increasing in exponential scale, the changes in n needed to increase the time 100 times are 3, 4.5 and 4, indicating that the time of execution of the algorithm is increasing tenfold with increase of n by 2. In the description of the algorithm it was stated that it has a factorial time complexity, which might be true, but an argument for it having only an exponential one is that as the higher the n is, there are also more solutions, which is very convenient for the programme as it has to analyse a smaller percentage of the positions.

Also the version of the algorithm with random starting position seems to result in better times, indicating that the starting position in the default version of the algorithm is far from the closest solution.

4.3. Backtracking algorithm results analysis

Similarly to the previous one, graph in Figure 9 also has an exponential scale. Again the shape of the line reassembles the one of linear function, meaning the exponential time complexity. Again, it might be possible that it has factorial time complexity, although the experimental evidence hints otherwise.

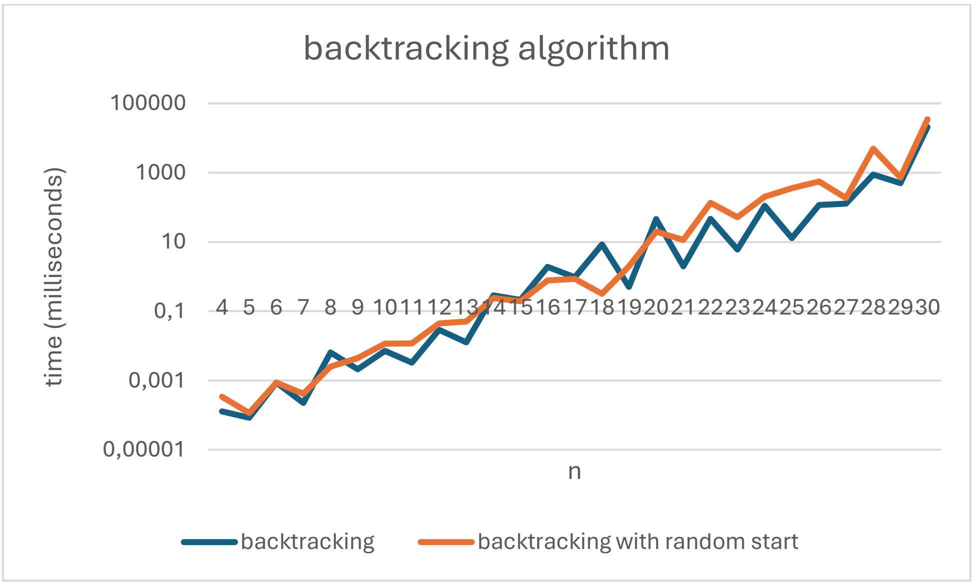

Figure 9 board size to time graph for the backtracking algorithm with exponential scale.

The times are fluctuating – the line is changing very quickly in small scales, indicating that the algorithm has harder times for some values of n, in which, probably, the density of solutions is lower or there are less nodes of the tree that the algorithm can skip – this is not the problem with proximity of the solution to the starting point, as both versions of the algorithm performed very similarly. Neither version is better than the other, as for each n that one has an edge over the other, there is a n where it works vice versa.

4.4. Genetic algorithm results analysis

Figure 10 and 11 show the graphical representation of the results for genetic algorithm. The first graph shows the polynomial time complexity, with a decline at the end, for n ≤ 30. For bigger values the times seem to increase almost linearly, with a very small increase of the slope as the n gets bigger.

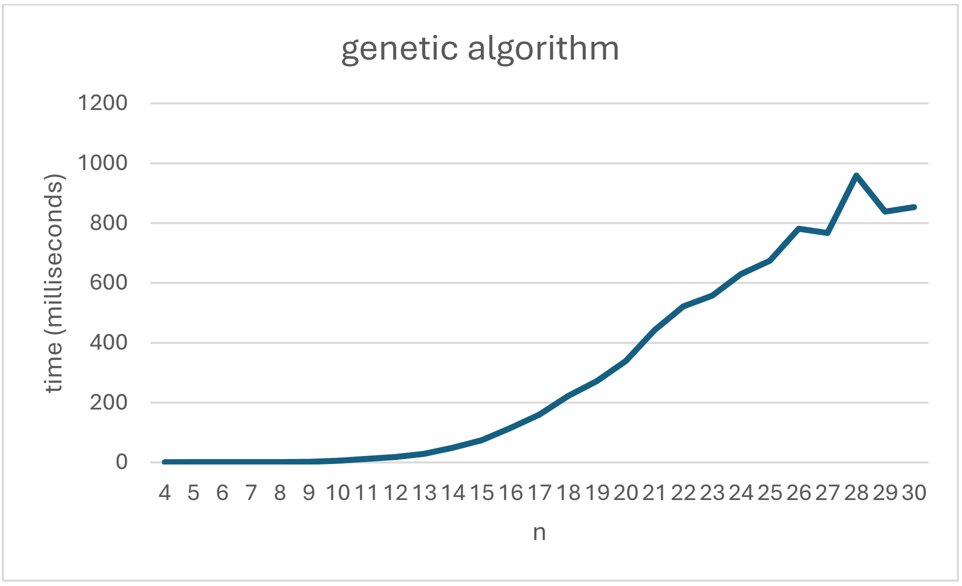

Figure 10 board size to time graph for the genetic algorithm for 4 ≤ n ≤ 30.

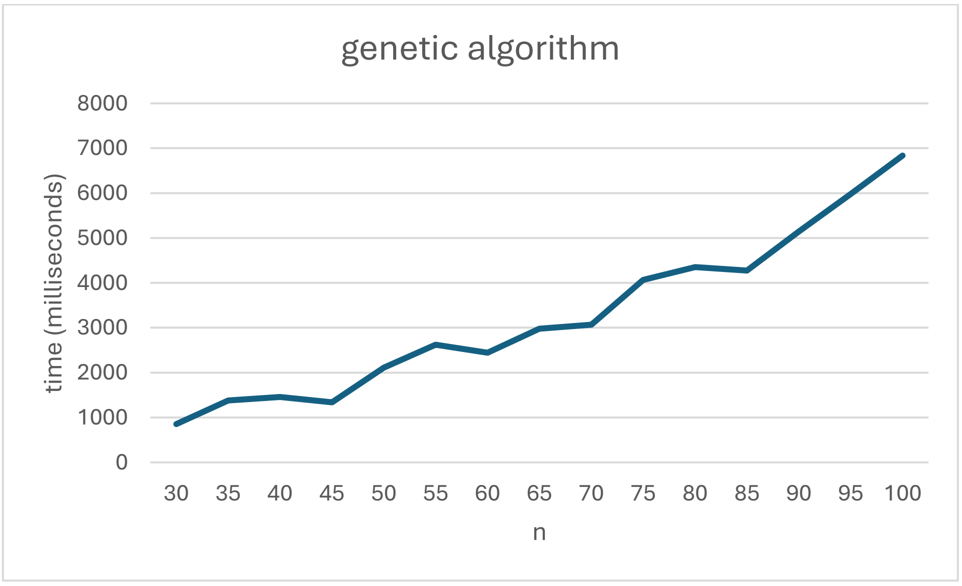

Figure 11 board size to time graph for the genetic algorithm for 30 ≤ n ≤ 100.

Figures 12 and 13 represent the whole range of data, the former showing it on the linear scale and the latter showing it on the exponential scale. Both confirm the observation that the algorithm has less than exponential time complexity. The time complexity must be only a polynomial, as the shape of the line on the first graph is close to linear and one on the second graph reassembles a logarithmic function.

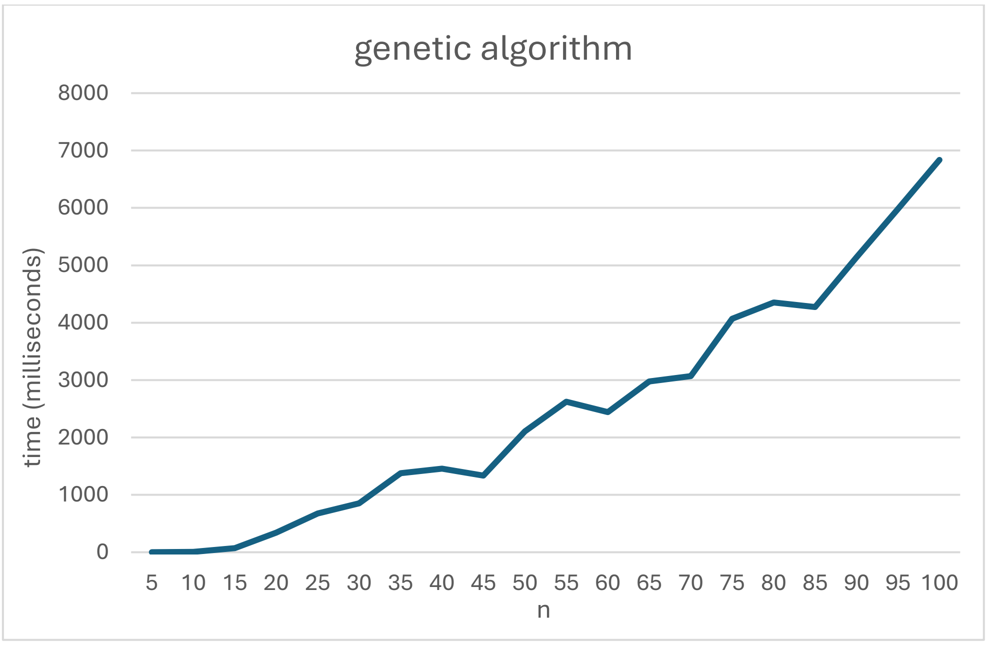

Figure 12 board size to time graph for the genetic algorithm for 5 ≤ n ≤ 100 with linear scale.

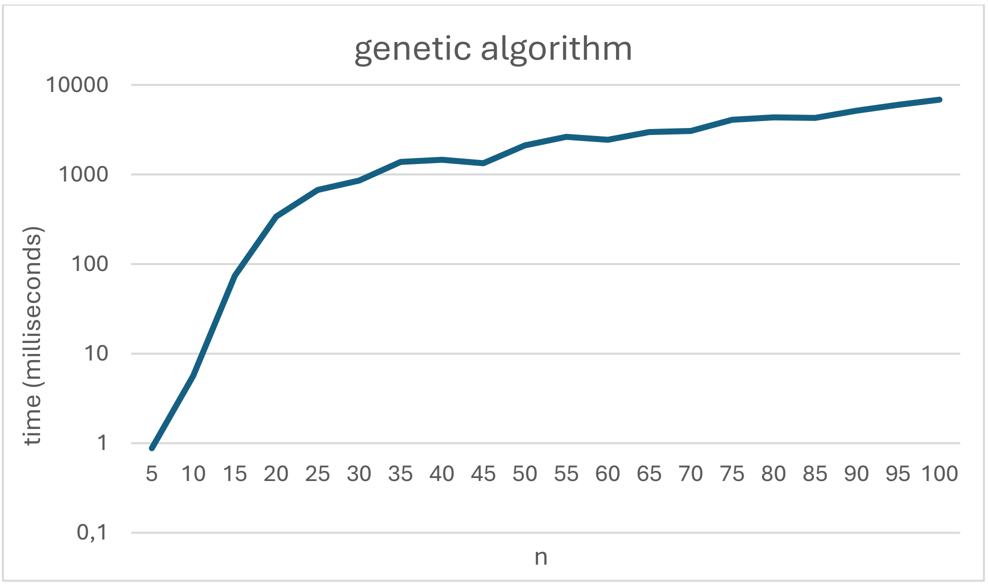

Figure 13 board size to time graph for the genetic algorithm for 5 ≤ n ≤ 100 with exponential scale.

4.5. Minimum conflicts algorithm results analysis

Both Figures 14 and 15 show that the pace of increase is very close to, but not exactly, linear. The first graph shows some fluctuations of the values, which are most certainly not the measurement error, as the experiment was repeated many times. The most probable cause is, as for previous algorithms, the unfavourable distribution of solutions for some values of n, causing the algorithm to get stuck in a local optimum, as all greedy algorithms have such tendency.

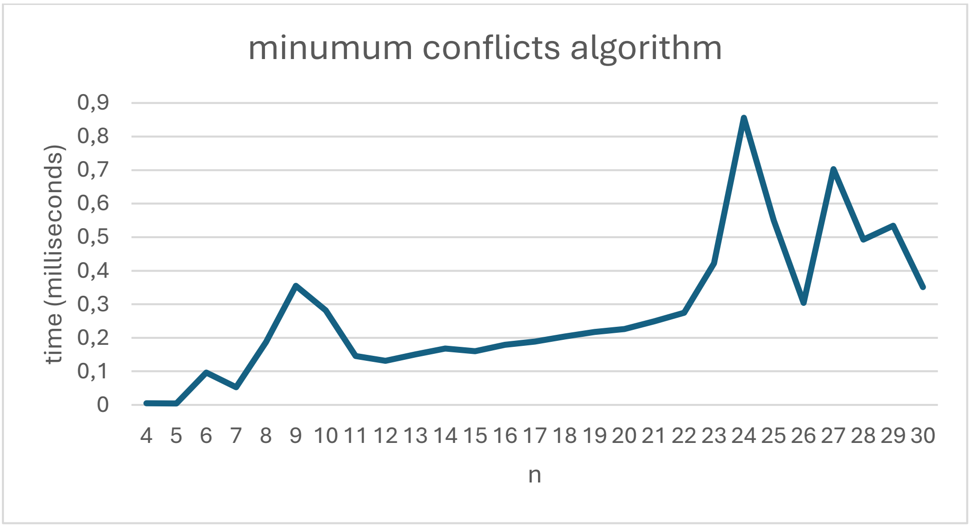

Figure 14 board size to time graph for the genetic algorithm for 4 ≤ n ≤ 30.

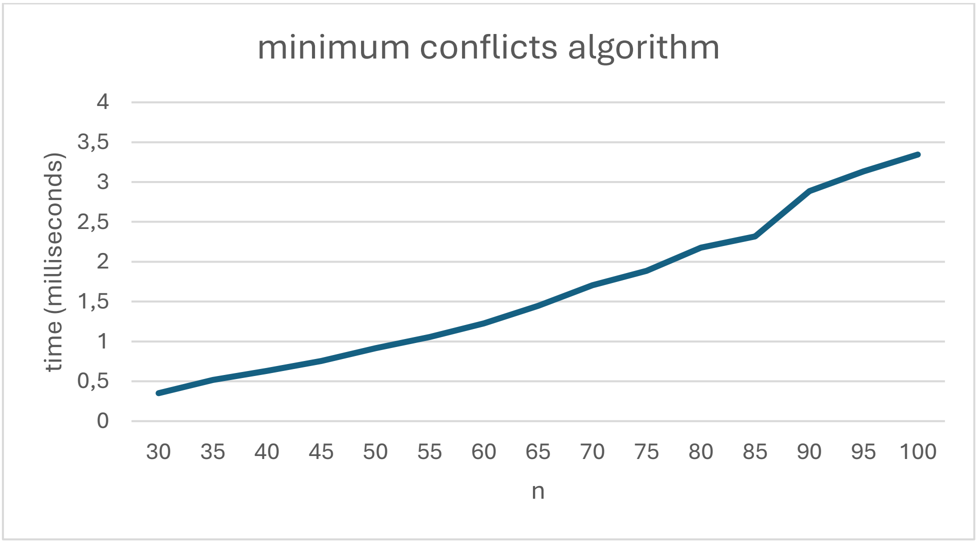

Figure 15 board size to time graph for the genetic algorithm for 30 ≤ n ≤ 100.

Figure 16 confirms that the time complexity is not exponential, as exponential best-fit line rises more quickly than the values on the graph. From this comes the conclusion that the time complexity of this algorithm is polynomial.

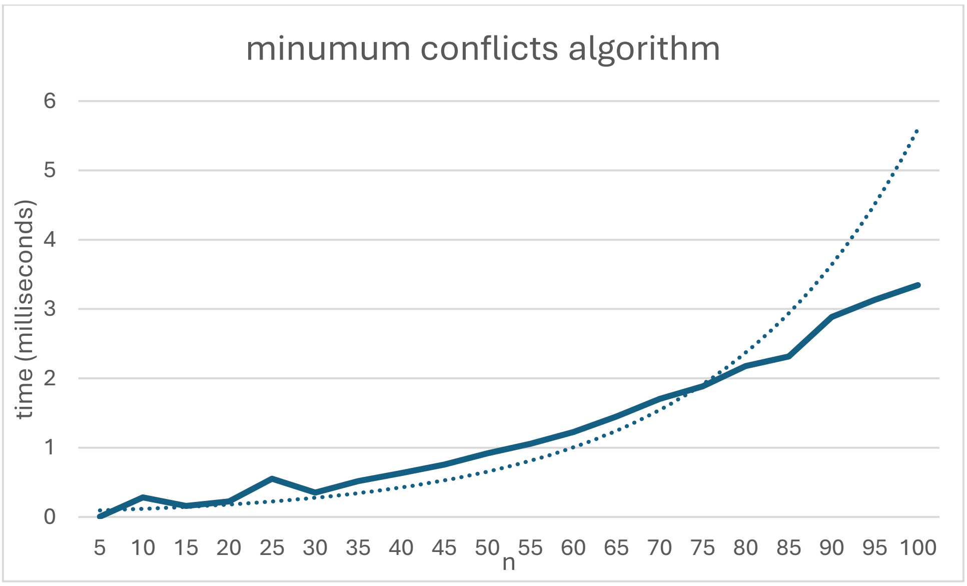

Figure 16 board size to time graph for the genetic algorithm for 5 ≤ n ≤ 100 with linear scale.

4.6. Comparison of the algorithms

The minimum conflicts outperformed every other algorithm for every n ≥ 14. For smaller values of n the backtracking algorithm proved to be the best, which was better than brute-force in every case. Genetic algorithm was the worst for small values of n, but, although always worse than minimum-conflicts, have proven to be the second one for bigger board sizes.

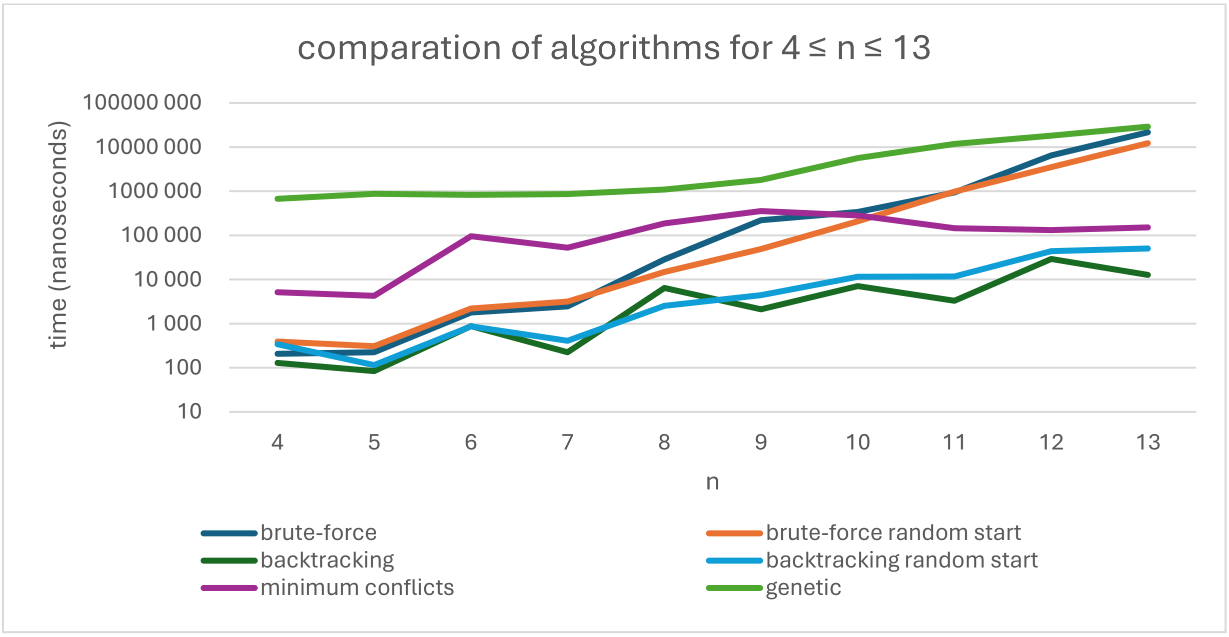

Figure 17 graph portraying the performance of algorithms for 4 ≤ n ≤ 13 with exponential scale.

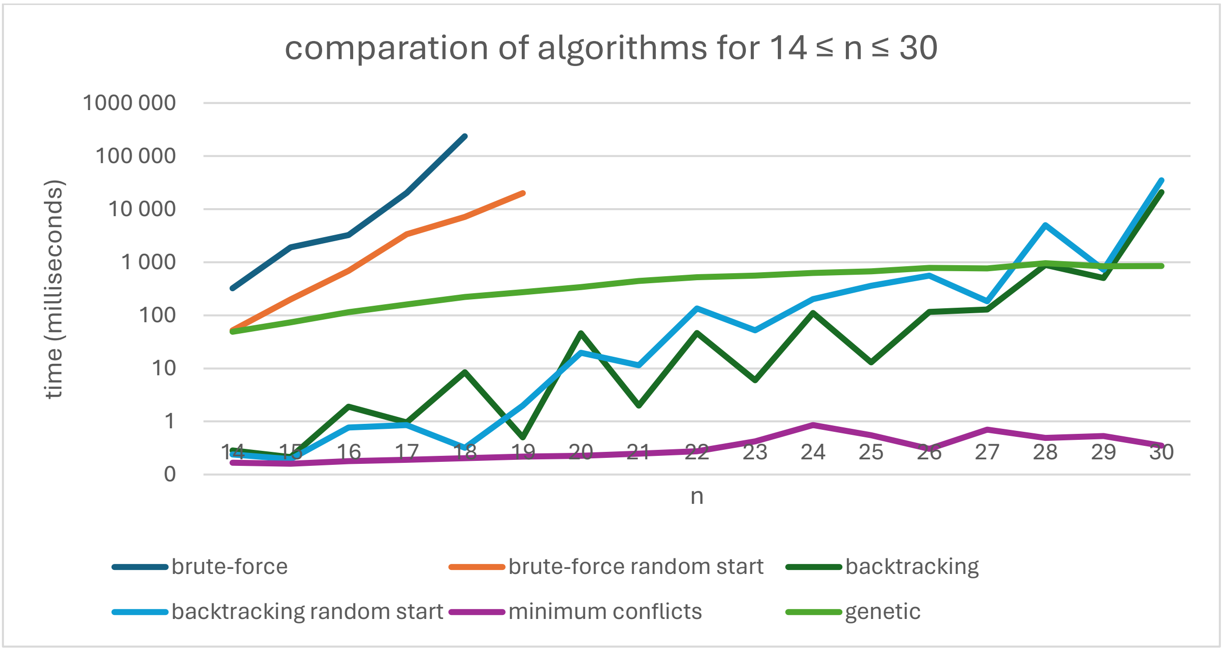

Figure 18 graph portraying the performance of algorithms for 14 ≤ n ≤ 30 with exponential scale.

A mathematical analysis of this data would result in labelling of brute-force and genetic algorithms as strictly dominated by respectively backtracking and minimum conflict ones, resulting in skipping the former two in analysis, as they have always a better counterpart. Backtracking, except being the best for small numbers, also allows for discovery of all solution, which minimum conflicts would struggle to, as with each discovered solution, the chance of algorithm discovering a new one drops. On the other hand, minimum conflicts algorithm has only a polynomial time complexity, while the backtracking one has at least exponential one, making the latter useless for higher board sizes, which the experiment confirms.

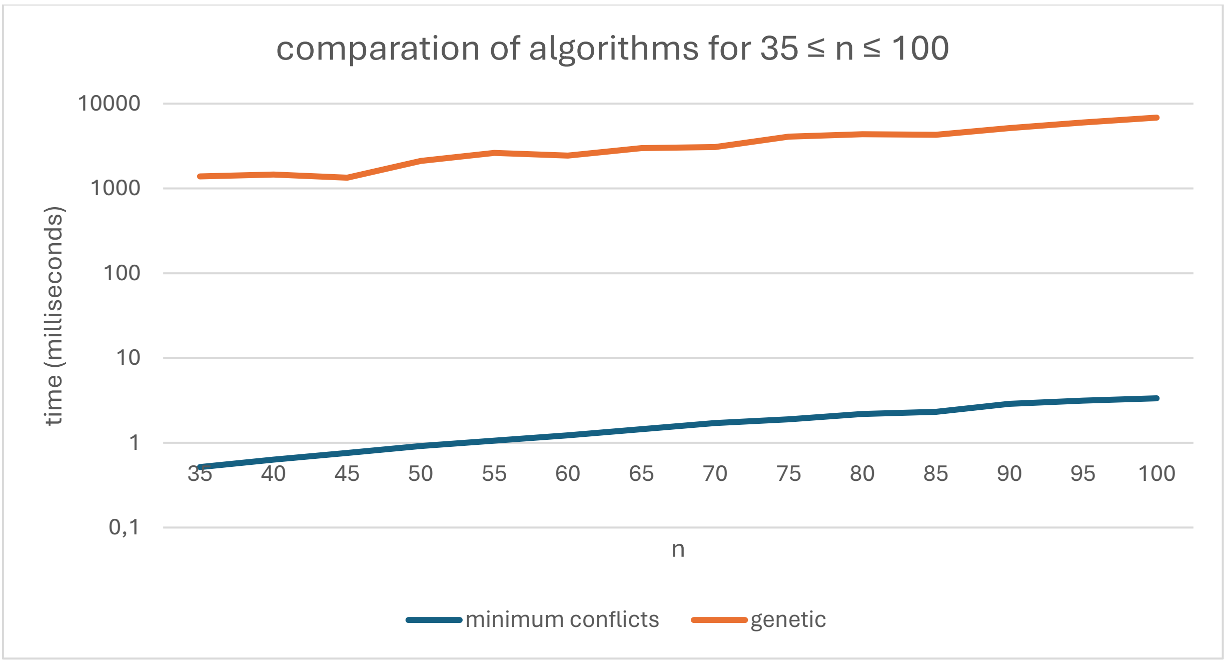

Figure 19 graph portraying the performance of algorithms for 35 ≤ n ≤ 100

From the graphs it can be observed that minimum conflicts averages over 3 magnitudes of order better times than genetic algorithm, which means it is 1000 times faster. Similarly, the backtracking algorithm is at least 100 times faster than brute-force.

5. Conclusion

In this paper different approaches to the problem were compared with satisfying result. Unfortunately no algorithm managed to achieve linear time complexity. The minimum conflicts solution proved to be the best for finding single solutions, indicating that greedy algorithms might be the best for finding singular solutions to the problems like scheduling tasks, a great example of which would be a program making plans for IB students and teachers, as each student can have different courses and the plans are hard to create by hand, or making complex structures with multiple restraints. On the other hand, backtracking algorithm was proven the best for small sized problems, indicating potential usage in cases of many simple calculations, such as pathfinding of an NPC in a game.

6. Further research opportunities

This paper was focused on the basic analysis and comparation of n-queen problem solving algorithms. However the field is much broader than that and allows for many more interesting analyses.

One potential idea is to test more different algorithms, example of one being a recursive algorithm that generates a solution for n based on a solution for (n-1). Such algorithm might be interesting, as it can give question if such generation is possible in every case.

Second possibility is to check how would those algorithms perform with the usage of GPU for computing the calculations instead of the CPU. Of the latter also the test engaging more than one core could be conducted.

The genetic and minimum conflicts algorithms could also be tempered with, as they contain many factors that could be changed, such as population size, mutation chance or the crossover function in the former and the board reset time in the second one, although it is a very wide topic and would require a paper on its own.

7. Works cited

- Ball, Walter William Rouse. (1905) “Mathematical Recreations and Essays,” Macmillan, New York, pp. 97–103.
  Link: https://web.northeastern.edu/seigen/11Magic/Books/Rouse%20Ball.pdf (Accessed 06.03.2024)
- Kesri, Vishal & Mishar, Manoj. (2013). “A new approach to solve n-queens problem based on series,” International Journal of Engineering Research and Applications. 3. 1346-1349.
  Link: https://www.researchgate.net/publication/321192822_A_new_approach_to_solve_n-queens_problem_based_on_series (Accessed 06.03.2024)
- Sarkar, Uddalok & Nag, Sayan. “An Adaptive Genetic Algorithm for Solving N-Queens Problem,” Department of Electrical Engineering, Jadavpur University, Kolkata, India.
  Link: https://arxiv.org/ftp/arxiv/papers/1802/1802.02006.pdf (Accessed 06.02.2024)

Other materials used

- https://www.chess.com/analysis?tab=analysis (Accessed 05.03.2024)
- Baeldung (2024), “Quicksort Algorithm Implementation in Java.”
  Link: https://www.baeldung.com/java-quicksort (Accessed 08.02.2024)

8. Appendix

8.1. Code of the algorithms used in the experiment.

Code of the brute-force algorithm

```
package com.company;


public class BruteForce {
    public static int[] solve(int size){
        int[] board = new int[size];
        for(int i = 0; i < size;i++){
            board[i] = i+1;
        }
        while(!check(board)){
            increase(board,size);
        }

        return board;
    }
    public static int[] solveR(int size){
        int[] board = getRandomPermutation(size);
        while(!check(board)){
            increaseR(board,size);
        }
        return board;
    }

    public static int[] getRandomPermutation(int size){
        int[] data = new int[size];
        for(int i = 0; i < size; i++){
            int index = (int) (Math.random()*(size-i));
            for(int l = 0; l <= index;l++){
                if(data[l] != 0)
                    index++;
            }
            data[index] = i+1;
        }
        return data;
    }

    public static void increase(int[] board, int size){
        if(board[size-1] > 1){
            board[size-1]--;
            for(int j = 0; j < size;j++)
                if(j!=size-1 && board[j] == board[size-1]) board[j]++;
        }else{
            for(int j = size-1;j >=0;j--){
                if(board[j] == 1){
                    board[j] = size+1;
                    for(int k = 0; k < size;k++){
                        board[k]--;
                    }
                }else {
                    board[j]--;
                    for(int k = 0; k < size;k++) if(k!=j && board[k] ==
board[j]) board[k]++;
                    break;
                }
            }
        }
    }
    public static void increaseR(int[] board, int size){
        if(board[size-1] > 1){
            board[size-1]--;
            for(int j = 0; j < size;j++)
                if(j!=size-1 && board[j] == board[size-1]) board[j]++;
        }else{
            for(int j = size-1;j >=0;j--){
                if(board[j] == 1){
                    if(j == 0){
                        for(int i = 0; i < size;i++){
                            board[i] = i+1;
                        }
                        return;
                    }
                    board[j] = size+1;
                    for(int k = 0; k < size;k++){
                        board[k]--;
                    }
                }else {
                    board[j]--;
                    for(int k = 0; k < size;k++) if(k!=j && board[k] ==
board[j]) board[k]++;
                    break;
                }
            }
        }
    }

    public static boolean check(int[] board){
        int size = board.length;
        for(int i = 0; i<size;i++){
            for(int j = 0; j < i; j++){
                if(board[i] + i == board[j] + j)return false;
                if(board[i] - i == board[j] - j)return false;
            }
        }
        return true;
    }
}
```

Code of the backtracking algorithm

```
package com.company;

import java.util.Arrays;

public class Backtracking {
    public static int[] solve(int size){
        int[] current = new int[size];
        while(increase(current,size));
        return current;
    }
    public static int[] solveR(int size){
        int[] current = new int[size];
        current[0] = (int)(Math.random()*size+1);
        while(increaseR(current,size));
        return current;
    }

    public static int[] getRandomPermutation(int size){
        int[] data = new int[size];
        for(int i = 0; i < size; i++){
            int index = (int) (Math.random()*(size-i)); //value from 0 to 
size-i-1
            for(int l = 0; l <= index;l++){
                if(data[l] != 0)
                    index++;
            }
            data[index] = i+1;
        }
        return data;
    }

    public static boolean increase(int[] current, int size){
        for(int i = 0; i < size; i++){
            if(current[i] == 0 || i == size-1){
                current[i]++;
                boolean forceRun = false;
                while (!check(current, i) || forceRun) {
                    forceRun = false;
                    current[i]++;
                    if (current[i] > size){
                        current[i--] = 0;
                        forceRun = true;
                    }
                }
                break;
            }
        }
        return current[size - 1] <= 0;
    }
    public static boolean increaseR(int[] current, int size){
        for(int i = 0; i < size; i++){
            if(current[i] == 0 || i == size-1){
                current[i]++;
                boolean forceRun = false;
                while (!check(current, i) || forceRun) {
                    forceRun = false;
                    current[i]++;
                    if (current[i] > size){
                        current[i--] = 0;
                        forceRun = true;
                        if(i == -1){
                            forceRun = false;
                            Arrays.fill(current,0);
                        }
                    }
                }
                break;
            }
        }
        return current[size - 1] <= 0;
    }

    public static boolean check(int[] current, int y){
        for(int i = 0; i < y;i++){
            if(current[i] == current[y]) return false;
            if(current[i] - i == current[y] - y) return false;
            if(current[i] + i == current[y] + y) return false;
        }
        return true;
    }
}
```

Code of the minimum conflicts algorithm

```
package com.company;

import java.util.Arrays;

public class MinimumConflicts {
    static long time;
    public static int[] solve(int size){
        int[] board = getRandomPermutation(size);
        int[][] history = new int[1000][size];
        time = System.nanoTime();
        while(iterate(board,history));
        return board;
    }

    public static int[] getRandomPermutation(int size){
        int[] data = new int[size];
        for(int i = 0; i < size; i++){
            int index = (int) (Math.random()*(size-i)); //value from 0 to
size-i-1
            for(int l = 0; l <= index;l++){
                if(data[l] != 0)
                    index++;
            }
            data[index] = i+1;
        }
        return data;
    }

    public static void checkForRepetition(int[] board, int[][] history){
        if(System.nanoTime() - time >= 1000000000L *60) {
            time = System.nanoTime();
            int[] fill = getRandomPermutation(board.length);
            System.arraycopy(fill, 0, board, 0, board.length);
            Arrays.fill(history, null);
            return;
        }
        int counter = 0;
        for(int i = 0; i < history.length && history[i] != null;i++){
            if(history[i] == board)counter++;
        }
        if(counter >= 5){
            time = System.nanoTime();
            int[] fill = getRandomPermutation(board.length);
            System.arraycopy(fill, 0, board, 0, board.length);
            Arrays.fill(history, null);
            return;
        }
        for(int i = history.length-2; i >= 0;i--){
            history[i+1] = history[i];
        }
        int[] fill = new int[board.length];
        System.arraycopy(board,0,fill,0,board.length);
        history[0] = fill;
    }

    public static boolean iterate(int[] board, int[][] history){
        int[] conflicts = getConflicts(board);
        if(checkSol(conflicts)) return false;
        int index = findWorstQueen(board, conflicts);
        board[index] = findBestPlace(board,index);
        checkForRepetition(board,history);
        return true;
    }

    public static boolean checkSol(int[] conflicts){
        for(int c : conflicts){
            if(c != 0) return false;
        }
        return true;
    }


    public static int findWorstQueen(int[] board, int[] conflicts){
        int size = board.length;
        int max = 0;
        int amount = 1;
        for(int i = 0; i < size;i++){
            if(conflicts[i] > conflicts[max]){
                max = i;
                amount = 1;
            }else if(conflicts[i] == conflicts[max]){
                amount++;
            }
        }
        int id = (int)(Math.random()*amount+1);
        for(int i = 0; i < size;i++){
            if(conflicts[i] == conflicts[max]){
                id--;
                if(id == 0){
                    max = i;
                    break;
                }
            }
        }
        return max;
    }

    public static int findBestPlace(int[] board, int index){
        int size = board.length;
```

```
int[] value = new int[size];
for(int i = 0; i < board[index]-1; i++){
    for(int j = 0; j < index; j++){
        if(index + i+1 == board[j] + j){
            value[i]++;
        }
        if(i - index+1 == board[j] - j){
            value[i]++;
        }
        if(i+1 == board[j]){
            value[i]++;
        }
    }
    for(int j = index+1; j < size; j++){
        if(index + i+1 == board[j] + j){
            value[i]++;
        }
        if(i - index+1 == board[j] - j){
            value[i]++;
        }
        if(i+1 == board[j]){
            value[i]++;
        }
    }
}
value[board[index]-1] = 8;
for(int i = board[index]; i < size; i++){
    for(int j = 0; j < index; j++){
        if(index + i+1 == board[j] + j){
            value[i]++;
        }
        if(i - index+1 == board[j] - j){
            value[i]++;
        }
        if(i+1 == board[j]){
            value[i]++;
        }
    }
    for(int j = index+1; j < size; j++){
        if(index + i+1 == board[j] + j){
            value[i]++;
        }
        if(i - index+1 == board[j] - j){
            value[i]++;
        }
        if(i+1 == board[j]){
            value[i]++;
        }
    }
}
int min = 0;
int amount = 1;
for(int i = 0; i < size;i++){
    if(value[i] < value[min]){
        min = i;
        amount = 1;
    }else if(value[i] == value[min]){
        amount++;
    }
}
int id = (int)(Math.random()*amount+1);
```

```
        for(int i = 0; i < size;i++){
            if(value[i] == value[min]){
                id--;
                if(id == 0){
                    min = i;
                    break;
                }
            }
        }
        return min+1;
    }


    public static int[] getConflicts(int[] board){
        int size = board.length;
        int[] out = new int[size];
        for(int i = 0; i<size;i++){
            for(int j = 0; j < i; j++){
                if(board[i] + i == board[j] + j){
                    out[i]++;
                    out[j]++;
                }
                if(board[i] - i == board[j] - j){
                    out[i]++;
                    out[j]++;
                }
                if(board[i] == board[j]){
                    out[i]++;
                    out[j]++;
                }
            }
        }
        return out;
    }
}
```

Code of the genetic algorithm

```
package com.company;

public class Genetic {
    final int SIZE;
    int[][] chromosomes = new int[1000][];
    int[][] badChromosomePool;
    int sqrtOfSize;
    int[] fitness = new int[1000];

    public Genetic(int size){
        SIZE = size;
        sqrtOfSize = (int)Math.sqrt(SIZE);
        badChromosomePool = new int[sqrtOfSize][];
    }

    public int[] solve(){
        generatePopulation();
        boolean run = true;
        while (run){
            run = evolve();
        }
        return chromosomes[0];
    }

    public boolean evolve(){

        quickSort(fitness,chromosomes,0,999);
        if(fitness[0] == 0) return false;
        int[] indexes = getRandomPermutation(500);
        for(int i = 0; i < 250; i++){
            chromosomes[500+2*i] =
crossover(chromosomes[indexes[2*i]],chromosomes[indexes[2*i+1]]);
            chromosomes[501+2*i] =
crossover(chromosomes[indexes[2*i+1]],chromosomes[indexes[2*i]]);
            mutate(chromosomes[500+2*i]);
            mutate(chromosomes[501+2*i]);
        }
        for(int i = 500; i < 1000; i++){
            fitness[i] = getFitness(chromosomes[i]);
        }
        int[] indexes2 = getRandomPermutation(sqrtOfSize);
        int[] b1 = crossover(badChromosomePool[indexes2[0]-
1],badChromosomePool[indexes2[1]-1]);
        int[] b2 = crossover(badChromosomePool[indexes2[1]-
1],badChromosomePool[indexes2[0]-1]);
        int[] b3 =
crossover(badChromosomePool[(int)(Math.random()*sqrtOfSize)],chromosomes[(i
nt)(Math.random()*1000)]);
        int f1 = getFitness(b1);
        int f2 = getFitness(b2);
        int f3 = getFitness(b3);
        for(int i = 999; i >= 0;i--){
            if(f1 < fitness[i]){
                int[] temp1 = chromosomes[i];
                chromosomes[i] = b1;
                b1 = temp1;

                int temp2 = fitness[i];
                fitness[i] = f1;
                f1 = temp2;
            }
            if(f2 < fitness[i]){
                int[] temp1 = chromosomes[i];
                chromosomes[i] = b2;
                b2 = temp1;

                int temp2 = fitness[i];
                fitness[i] = f2;
                f2 = temp2;
            }
            if(f3 < fitness[i]){
                int[] temp1 = chromosomes[i];
                chromosomes[i] = b3;
                b3 = temp1;

                int temp2 = fitness[i];
                fitness[i] = f3;
                f3 = temp2;
            }
        }
        return true;
    }

    public int[] getRandomPermutation(int size){
        int[] data = new int[size];
        for(int i = 0; i < size; i++){
            int index = (int) (Math.random()*(size-i)); //value from 0 to
size-i-1
            for(int l = 0; l <= index;l++){
                if(data[l] != 0)
                    index++;
            }
            data[index] = i+1;
        }
        return data;
    }

    public void generatePopulation(){
        for(int i = 0; i < sqrtOfSize;i++){
            badChromosomePool[i] = getRandomPermutation(SIZE);
        }
        for(int j = 0; j < 1000;j++){
            chromosomes[j] = getRandomPermutation(SIZE);
        }
        for(int i = 0; i < 1000; i++){
            fitness[i] = getFitness(chromosomes[i]);
        }
    }

    public int getFitness(int[] chromosome){
        int fitness = 0;
        int[] d1 = new int[SIZE];
        int[] d2 = new int[SIZE];
        for(int i = 0; i < SIZE;i++){
            d1[i] = chromosome[i]-i;
            d2[i] = chromosome[i]+i;
        }
        quickSort(d1,0,SIZE-1);
        quickSort(d2,0,SIZE-1);
        for(int i = 1; i < SIZE;i++){
            if(d1[i] == d1[i-1])fitness++;
            if(d2[i] == d2[i-1])fitness++;
        }
        return fitness;
    }

    public int[] crossover(int[] c1, int[] c2){
        int end,beginning;
        int i1 = (int)(Math.random()*SIZE);
        int i2 = (int)(Math.random()*SIZE);
        if(i1 > i2){
            beginning = i2;
            end = i1;
        }else {
            beginning = i1;
            end = i2;
        }

        int pos = 0;
        int[] output = new int[SIZE];
        for(int i = 0; i < beginning;i++){
            boolean isGood = true;
            for(int j = beginning; j <= end; j++){
                if (c1[j] == c2[pos]) {
                    isGood = false;
                    break;
                }
            }
            if(isGood){
                output[i] = c2[pos];
            }else i--;
            pos++;
        }
        if (end + 1 - beginning >= 0) System.arraycopy(c1, beginning, 
output, beginning, end + 1 - beginning);
        for(int i = end+1; i < SIZE;i++){
            boolean isGood = true;
            for(int j = beginning; j <= end; j++){
                if (c1[j] == c2[pos]) {
                    isGood = false;
                    break;
                }
            }
            if(isGood){
                output[i] = c2[pos];
            }else i--;
            pos++;
        }

        return output;
    }

    public void mutate(int[] chromosome){
        if(Math.random() < 0.8){
            int p1 = (int)(Math.random()*SIZE);
            int p2 = (int)(Math.random()*(SIZE-1));
            if(p2 >= p1) p2++;
            int temp = chromosome[p1];
            chromosome[p1] = chromosome[p2];
            chromosome[p2] = temp;
            if (Math.random() < 0.5){
                p1 = (int)(Math.random()*SIZE);
                p2 = (int)(Math.random()*(SIZE-1));
                if(p2 >= p1) p2++;
                temp = chromosome[p1];
                chromosome[p1] = chromosome[p2];
                chromosome[p2] = temp;
            }
        }

    }


    public static void quickSort(int[] arr, int[][] arr2, int begin, int 
end) {
        if (begin < end) {
            int partitionIndex = partition(arr, arr2, begin, end);

            quickSort(arr, arr2, begin, partitionIndex-1);
            quickSort(arr, arr2,partitionIndex+1, end);
        }
    }
    private static int partition(int[] arr, int[][] arr2, int begin, int
end) {
        int pivot = arr[end];
        int i = (begin-1);

        for (int j = begin; j < end; j++) {
            if (arr[j] <= pivot) {
                i++;

                int swapTemp = arr[i];
                arr[i] = arr[j];
                arr[j] = swapTemp;

                int[] swapTemp2 = arr2[i];
                arr2[i] = arr2[j];
                arr2[j] = swapTemp2;
            }
        }

        int swapTemp = arr[i+1];
        arr[i+1] = arr[end];
        arr[end] = swapTemp;

        int[] swapTemp2 = arr2[i+1];
        arr2[i+1] = arr2[end];
        arr2[end] = swapTemp2;

        return i+1;
    }

    public static void quickSort(int[] arr, int begin, int end) {
        if (begin < end) {
            int partitionIndex = partition(arr, begin, end);

            quickSort(arr, begin, partitionIndex-1);
            quickSort(arr, partitionIndex+1, end);
        }
    }

    private static int partition(int[] arr, int begin, int end) {
        int pivot = arr[end];
        int i = (begin-1);

        for (int j = begin; j < end; j++) {
            if (arr[j] <= pivot) {
                i++;

                int swapTemp = arr[i];
                arr[i] = arr[j];
                arr[j] = swapTemp;
            }
        }

        int swapTemp = arr[i+1];
        arr[i+1] = arr[end];
        arr[end] = swapTemp;

        return i+1;
    }
}
```