Group #9 Assignment Report

An Evolutionary Algorithm for Solving the N-Queens Problem

Mahmut Osmanovic Isac Paulsson Sebastian Tuura
Mohamed Al Khaled Emil Wagman

October 3, 2024

Evolutionary algorithms have been applied in various domains to great success due to their ability to explore large search spaces efficiently. Their stochastic nature means that they are not easily predictable, but it is that inherent randomness that allows them avoid local optima. A problem that benefits greatly from these characteristics is the N-Queens problem, which involves placing N chess queens on a $N \times N$ chess board in such a way that no two queens threaten each other. This problem entails a $N \times N$ search space, which we in this project have reduced to $N!$ by only allowing one single queen per column on the chess board. Methods such as brute force and backtracking have been proven useful, but are more computationally expensive at increasing sizes of N, meaning that a more a scalable approach is needed. In this project, we propose a Genetic Algorithm, implementing partially mapped crossover recombination algorithm with duplication removal. Our measurement of performance was based on the amount of iterations needed to find correct answer to a 8-Queens problem, averaged over 100 randomly generated initial board populations. This measurement approach was then used to perform a combination search to find the best combination of genetic operators (strategies) and a parameter values.

1 Introduction

This report was created as a part of the *Artificial Intelligence* Course at Jönköping University, and details the specific application of genetic algorithms to solve the N-Queens problem. This was done with the goal of learning more about how evolutionary algorithms can be applied to problems, and deepening our understanding of scientific computation and exploration.

The broader area of evolutionary algorithms (EAs) have proven to be effective in tackling a wide range of problems [8]. Unlike deterministic algorithms, evolutionary algorithms rely on a stochastic process, enabling them to avoid local optima and converge towards more optimal global solutions [8]. This characteristic makes them especially suitable for problems that inherently have a larger search space.

The topic of this report relates to one of those problems, namely the well-known N-Queens problem. The task is to find valid placements of N chess queens on $N \times N$ chess board. This means that no queen can be placed in a column, row, or diagonal line, where there already exists another queen. A solution to the N-Queens problem is therefore only valid if all N queens have been placed on the board and none of them are under threat [1].

This problem can be trivially solved via brute force at smaller sizes of N. But as N grows, the problem becomes computationally unfeasible given a naive brute force method. This is where algorithms that can handle large search spaces prove themselves useful. A naive N-queens search space has $N \times N$ amount of possible solutions. In this project we've restricted the search space by limiting queens to one column each, which results in a search space of $N!$ possible solutions.

This project focused on comparing different genetic operators, parameter values, and additional evolutionary strategies, with each other to find the best combination from those tested in this project. Our aim was to systematically compare different combinations through the use of parameter tuning, to generate 500 different combinations. Where combinations included different genetic operators, parameter values, and additional strategies. This resulted in 500 separate Genetic Algorithm setups that we individually evaluated 100 times on randomly generated initial board layouts. Performance across all setup runs were then averaged, and our final setup was the top performing Genetic Algorithm.

From these results came a couple of insights: the first is that duplication removal is crucial for removing many invalid individuals from a population. Additionally, duplication removal in tandem with partially mapped crossover (PMX) performed especially well. Another insight was that adjusting mutation rate (starting low, ending high) and crossover rate (starting high, ending low) variably each generation did not show clear performance benefits. We saw the same thing happening once we introduced genocide into the algorithm (starting high, ending low). Overall we found that evolutionary algorithms are a very powerful tool to solve problems such as the N-Queens problem, and that with a more in depth study into other types of genetic operators, even better results could be achieved.

2 The N-Queens Problem

The N-Queens problem is a well-known combinatorial problem within computer science, where N chess queens are placed on a $N \times N$ chess board in such a way that none of them threaten each other. A chess queen can move in any direction: horizontally, vertically, or diagonally. Therefore a valid solutions requires that no two queens were placed in the same column, row, or diagonal line. This problem can be represented by a one-dimensional array where indices represents rows, and values in the array represents columns [1].

Various evolutionary algorithms have been used to solve the N-Queens problem due to its combinatorial nature. An example of this comes from Sarkar and Nag [7], who propose a Genetic Algorithm that begins with a random population of chromosomes. Their fitness function evaluated the number of conflicts between queens, with the aim of minimizing them. The Order 1 Crossover strategy was used to merge traits from random pairs of parents, while preserving valid queen placements. They additionally made use of a mutation operator with a low probability of causing mutations.

Similarly, Jain and Prasad [6] presents another Genetic Algorithm for solving the N-Queens problem. Unlike Sarkar's and Nag's version, this algorithm uses an advanced mutation operator that specifically targets conflicting queens. Instead of applying random mutations, this strategy swaps the positions of queens that are in conflict with each other, aiming to reduce the overall number of conflicts in the board. This approach ensures that the changes made are more likely to lead to a solution, rather than just random mutations.

Boiiković, Golub, and Budin [2] present a Global Parallel Genetic Algorithm, which operates on a master-slave framework. The master (the main thread) manages population control and mutation, while slaves (secondary threads) perform selection and crossover operations. Their approach uses a 3-way tournament selection process, which allows multiple slaves to make the selection at the same time. This parallel architecture may enhance computational efficiency by allowing the workload to be distributed across multiple processors, enabling faster convergence and the ability to handle larger problem sizes more effectively.

Chauhan, Gonder, and Garg [4] propose an innovative Genetic Algorithm approach utilizing a hybrid crossover operator. Their algorithm aims to eliminate conflicts between queens by employing a fitness function that evaluates conflicts through paired interactions. The fitness score of a solution is determined by the number of diagonal collisions, with the algorithm ranking solutions from highest to lowest fitness value. The parent selection method incorporates a mating probability, which determines whether a solution can be chosen for crossover, resulting in unique parent solutions. For generating new solutions, the algorithm employs the Position-Based Crossover (PBC) technique, which combines traits from selected parents.

As we were novices to the subject of evolutionary algorithms, we saw an opportunity to explore and compare many different evolutionary strategies. Although the papers above specify different Genetic Algorithm approaches, it was not clear to us why certain genetic operators, strategies, and so on, were not considered for this algorithms. Therefore a more comprehensive approach to evaluating genetic algorithms was decided upon.

3 Algorithm

3.1 Solution Representation

Solutions to the N-Queens problem are represented within this project as permutations of 1-Dimensional arrays of length N, filled with integers between 1 and N, with no duplicate values. Indices in the arrays represents each column on the chess board, while the integer values represent each column. This naturally removes all vertical and horizontal conflicts. The following is an example solution in both its NumPy array representation, and on a chess board [5].

```
[6 3 5 8 1 4 2 7]
```

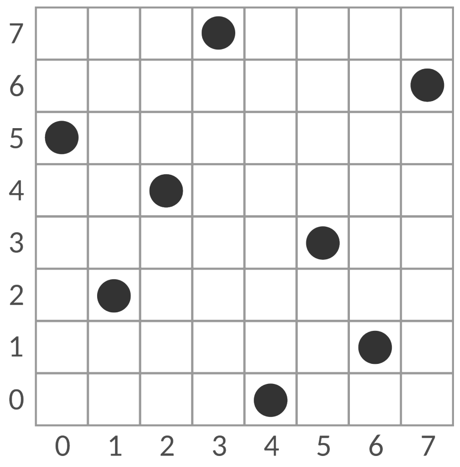

Figure 1: Example array representation in a board view

3.2 Fitness Function

The fitness function selected for the algorithm is based on the number of conflicts counted in given a board state. The maximum number of possible conflicts can be calculated as follows.

$$C_{max} = \binom{N}{2} = \frac{N(N-1)}{2} \tag{1}$$

Where N equals the board size $N \times N$ as well as the number of queens. Using this we normalize the counted number of conflicts in a state.

$$fitness = 1 - \frac{C_{count}}{C_{max}} \tag{2}$$

A valid solution 1 represents no conflicts. A worst case 0 represents C_{max} conflicts.

3.3 Operators

The algorithm combines evolutionary operators on individuals within a population in order to reach the goal state, (i.e. the solution to the problem). There are three main evolutionary operators: **selection**, **recombination** and **mutation**, which will be used later on in the literate code section 3.4.

3.3.1 Selection

The **selection operator** is generally based on the "survival of the fittest" concept from Darwinian evolution theory. The goal is to use probability to pick the most fit candidate solutions in the current population, to be used either passed on to the next generation or be recombined. More precisely, given a population of size N, the selection operator outputs m individuals, where $m \subseteq N$, chosen based on a heuristic function h. The heuristic function can be defined in numerous ways. For parental selection, we used **tournament selection**. This selection method first defines a tournament group size of m as a subset of the whole population N. Subsequently, m random and different (but potentially identical in terms of genome) candidate solutions are randomly chosen. Thereafter, each chosen individuals fitness score is evaluated. The two most fit individuals of the tournament are then chosen for reproduction. The processes will continue until the requested amount of parents has been chosen. The requested amount of offspring is submitted as a fraction of the population.

Additionally we included a survival selection operation after parent selection, recombination, and mutation have been performed. In contrast to parent selection, for survival selection we chose another approach. We first evaluate the fitness of the combined set of parents and offspring. We exponentiate all fitness scores (all fitness scores are between 0 and 1) by a positive real number and then they are normalized through division by the sum of all exponentiated scores. The purpose behind this transformation is to magnify the relative differences between fitness scores. The fit individuals will have an even higher probability of survival. The higher the exponentiation factor, the more greedy our algorithm will be, and in turn, result in a less diversified population. Nonetheless, a new set of candidate solutions are chosen, in virtue of the exponentiated scores, equal in quantity to previous population.

Time Complexity: The time complexity of each tournament in parent selection is no more than $O(n)$. However, survival selection has a time complexity of $O(n + nlog(m))$. Here, n is how many values are chosen amongst and m, the amount of candidates to be drawn from n individuals. The time complexities are related to the precise implementation of NumPy's *random.choice* function. With m elements to be chosen among n possible choices, each with an associated probability p, the time complexity is increased.

3.3.2 Recombination

The **recombination operator** is one of the primary tools used to achieve population diversification. The parents that have been chosen through the use of the selection operator, are subsequently passed as input to the recombination operator.

The recombination transform can be represented as:

$$R : P \to O$$

where R is the recombination transform, P is the set of input parents, and O is the set of output offspring.

The transformation outputs the same of quantity of offspring as the quantity of parents that was inputted. Note that the set of offspring that can possibly be generated is dependent on the initial candidate solution, and more importantly, the recombination strategy itself. Thus, various strategies will entail a varying amount of introduced diversity within the offspring from the base population. There exists a plethora of different recombination strategies, but only six where evaluated within this project's scope. The table below highlights those chosen, together with their spacetime complexity 3.3.2.

Name	**Spacetime Complexity**
even_cut_and_crossfill	$O(n)$ space, $O(n)$ time
one_point_crossover	$O(n)$ space, $O(n)$ time
two_point_crossover	$O(n)$ space, $O(n)$ time
partially_mapped_crossover	$O(n)$ space, $O(n^2)$ time
pmx_dp_rm	$O(n)$ space, $O(n^2)$ time
ordered_crossover	$O(n)$ space, $O(n^2)$ time

Table 1: Recombination Strategies with Spacetime Complexity

Lastly, we'll make an observation of typical recombination rates. Typically, recombination rates vary between 70% and 90%. However, in the final solution, the recombination rate is set dynamically each generation. It is in fact regulated by a dynamic exploration factor which starts off large in the beginning of the evolutionary process and decreases until it reaches 10%.

The exploration factor is defined as:

$$\text{Exploration Factor} = \max\left(0.1, \frac{n}{n + g^k}\right),$$

where n is the genome size, g is the current generation, and $k \in \mathbb{R}$. Below follows a visual representation of the exploration factor.

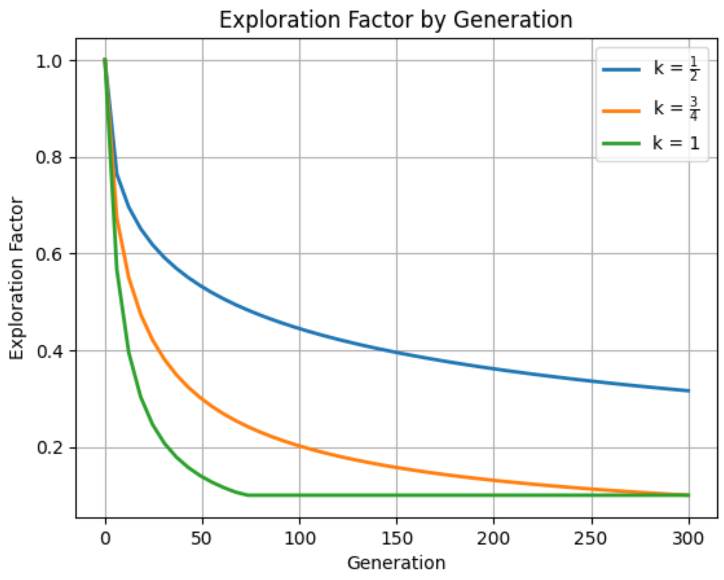

Figure 2: $n = 8$, $g \in [0, 300]$ and k = $[\frac{1}{2}, \frac{3}{4}, 1]$.

3.3.3 Mutation

Lastly, the **mutation operator** takes offspring from the recombination step as input, and mutates one or several genes inside of each candidate solution genome. More specifically,

$$M : O_{original} \rightarrow O_{mutated}$$

where M is the mutation transform, $O_{original}$ is the set of unchanged input offspring, and $O_{original}$ is the set of mutated output offspring.

The mutation rate is a percentage probability that a mutation occurs. Note that inside the final solution, the mutation rate is set to be dynamic, starting off low at 10%, and increasing for each subsequent generation until it reaches a maximum, user set mutation rate. Meaning, the algorithm will become more and more exploitative for each subsequent generation.

The exploitation factor is defined as:

$$\text{Exploitation Factor} = \max\left(0.1, 1 - \frac{n}{n + g^k}\right),$$

where n is the genome size, g is the current generation, and $k \in \mathbb{R}$.

Below follows a visual representation of how the exploitation factor changes over generations.

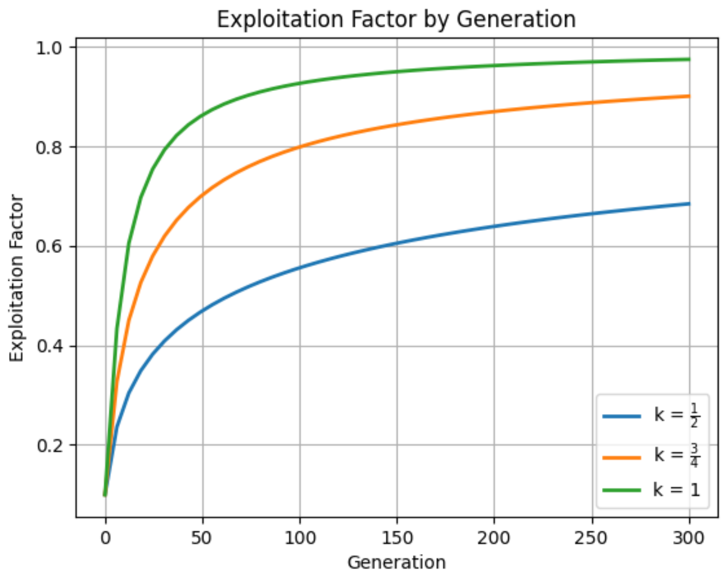

Figure 3: Here $n = 8$, $g \in [0, 300]$ and k $= [\frac{1}{2}, \frac{3}{4}, 1]$.

The mutation strategies that were investigated withing this project are listed in the table below, along with their spacetime complexity.

Name	**Spacetime Complexity**
swap_mutation	$O(n)$ space, $O(n)$ time
inversion_mutation	$O(n)$ space, $O(n)$ time
duplicate_replacement	$O(n)$ space, $O(n)$ time
creep_mutation	$O(n)$ space, $O(n)$ time
scramble_mutation	$O(n)$ space, $O(n)$ time

Table 2: Mutation Strategies with Spacetime Complexity

3.4 Literate Code

There are several evolutionary operators whose efficiency were experimented with. The general structure of the algorithm was iteratively improved upon throughuut the course. Initially it only contained static numeric parameters. As of the final version of the evolutionary algorithm, three of the parameters have been made dynamic. Those are: the recombination rate, mutation rate and genocide rate. The dynamcity introduced aids in the fine-tuning of the algorithm to specific choices of the genome size n. Our final evolutionary algorithm is presented below in *literate code*, see 1.

Algorithm 1 Literate Code: Final Evolutionary Algorithm

```
Require: $N \geq 0 \wedge (N \neq 2 \vee 3)$, where N represents the problem dimensionality.
  $evaluation\_counter = 0$
  while $(is\_max\_evals\_reached = False) \wedge (is\_solution = False)$ do
      calculate dynamic recombination and mutation rates
      select a subset individuals from the population as parents
      recombine the parents and their generate offspring
      mutate the offspring
      select individuals for survival from the combined set of parents and offspring
      update evaluation_counter += [size(population) + size(offspring)]
      if solutions exists in population then
          $is\_solution = True$
      end if
      calculate dynamic genocide rate
      if population has stagnated then
          apply dynamic genocide
      end if
  end while
```

The worst case spacetime complexity of the final algorithm is highlighted in the table below.

Component	**Time Complexity**	**Space Complexity**
Recombination	$O(n^2)$	$O(n)$
Selection of Individuals	$O(n + nlog(m))$	$O(n)$
Mutation of Offspring	$O(n)$	$O(n)$
Survival Selection	$O(n)$	$O(n)$
Solution Check	$O(n)$	$O(1)$
Stagnation Check and Genocide	$O(n)$	$O(1)$
Total	$O(n^2)$	$O(n)$

Table 3: Time and Space Complexity Analysis of the Algorithm

4 Experimental

4.1 Hardware & Software

The experiments were carried out by the use of a wide range of hardware setups. That is, both personal laptops and desktop PCs. The hardware requirements for this project are modest, and most modern computers will handle the tasks without issue.

With regards to software, all experiments were conducted using Python3. Several external libraries within Python were used, such as Numpy, Pandas, Plotly, Matplotlib, Seaborn, and SciPy. In addition to external libraries, standard Python libraries like ast, time, and logging were also used. Specific version details on these libraries are listed in the accompanying *requirements.txt* file.

4.2 Methodology

Our initial solution was heavily inspired by one outlined in *Introduction to evolutionary computing (2003)* [3]. That solution, for example, included "Cut-and-crossfill' (even_cut_and_crossfill by our notation), swap mutation and the rate of recombination. We subsequently incrementally made adjustments. Included in the adjustments is the implementation of new strategies for some of the basic evolutionary operators. Moreover, we evaluated the performance of novel techniques to avoid local minima. One such is the application of a genocide operator. It is triggered when the population stagnates. What is optimal is heavily context dependent. We narrowed our analysis to fewer dimensions than 15. This is in part due to that we think it suffices for generalization. Meaning, we can observe behavioural changes as we increase the dimensionality from say 4 to 15. Those may subsequently be further generalized for even higher dimensionality. The experimental results and analysis to substantiate that claim will be presented in the results section 5. Nonetheless, hardware limitations do in any case limit our explorations of higher genome sizes.

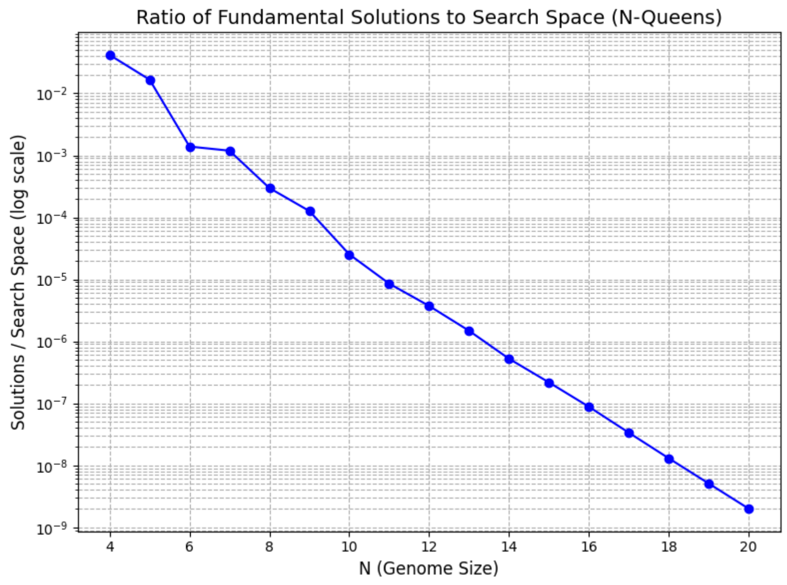

Figure 4: The decline of available solutions with an increasing genome size

Figure 4 highlights the exponential decrease of the available amounts of solutions as a fraction of the state space for genome sizes between 4 and 25. The y-axis is logarithmically scaled. One observes that the fraction of solutions at-least decrease exponentially. The effective search space can be reduced down to $n!$, which for large n is not searchable in a reasonable about of time, based on the teams available hardware.

Nonetheless, with this limitation in genome size, we did make an attempt in fine-tuning our parameters and operators. New versions were without exception bench-marked against the current version.

To begin with, our evaluation metric was the sum of all fitness evaluations made on candidate solutions. It is an effective metric that scale both with the population size, amount of offspring, in addition to genome size. In some instances, we additionally counted the amount of generations it took for algorithmic convergence. However, this was mainly done for visualization purposes.

As there is a strong stochastic nature to evolutionary algorithms, it is impossible to find the set perfect set of configuration parameters. The experiments that were conducted primarily focused on the eight dimension. However, some experiments were conducted in higher dimensions, up to 15.

4.3 Sampling

We desire to tune our parameters in such a manner as to decrease the expected evaluation count in our given context. Since there is a multitude of parameter combinations (much more than we have computation power to test), it is of vital importance that we strategically configure our parameters for each run. They should be configured in such a manner to cover a wast but different parts of the topological landscape (search space). Introducing multiple agent to cover the same subset of the topological landscape is redundant.

All sampling techniques are based on generating sample points between 0 and 1. These sample points are subsequently used to scale the parameter space, that is, pick samples from the parameter space.

There were four sampling techniques we employed, Uniform (grid) sequences, Halton sequences, Sobol sequences and Latin hypercube sampling sequence. Unfortunately, all suffer from the curse of dimensionality. Meaning, there is a exponential growth in the number of required sample points as the dimensionality (genome size) increases. However, the way in which each of them distribute their points across the search space differs. It leads to that some sampling methods better handle the increase in dimensionality better than others.

A uniform sequence simply splits each dimension in uniform slices and thereafter places point at each intersection. It is rather simple to implement however lacks efficiency in higher dimensions. The space diagonal of a hypercube increases in length for each dimension that is added to it. It entails that there for each added dimension is a larger unexplored hyperspace, i.e. no samples. Thus, uniformly sequencing each dimension works well in small dimensions but tends to scale poorly.

The Latin Hypercube Sampling method derives its name from a "Latin Square", which means that each element just occurs ones in a row and column. In 2D, for each dimension, it randomly picks a grid point for the other dimension. Latin Hypercube Sampling performs well in small and large dimensions dimensions because it balances between exploration and exploitation, avoiding unnecessary evaluations whilst it ensures that each subset of the search space gets covered.

Halton sequences use a unique prime for each dimension as a base. Thereafter, it uses these fractional increments in these primes to generate points. Each element in a Halton fraction is unique, so the space is uniquely sampled. They are also uniformly distributed, they are designed to minimize discrepancy, which tell us about how well points cover the search space. They fill space better than the uniform distribution, however, does face challenges in higher dimensions, that is, it becomes less uniform.

A Sobol sequences are more complex than Haltons sequence but tend to be less cluttered, especially in higher dimensions. Is more effective than Halton sequences [9].

The LHS was the sampling method of our choice. It is utilized in generating the results in section 5.

4.4 Strategies and Numeric Parameters

Using our sampling method of choice, thousands of setups were instantiated in the eight dimension (eight-queens). Each setup was run 10 times. The evaluation count of each setup for all of the ten runs was summed and thereafter divided by 10, that is, the average evaluation count per setup was calculated. We logged all important and relevant data associated with each setup. The logs were later read by several visualization function and insights were concluded through visual inspections. The visual inspections of the results included a bar plot highlighting the performance of the generated configurations. The best performing setup, based on our most comprehensive configuration setup in eight dimensions also became the basis for a choice of the numeric parameters. The operator strategies were chosen on the basis of visual inspection from several plots. The recombination strategy was selected based the heatmap in figure 7. Thereafter, the mutation strategies were evaluated by stacking the total evaluation count for each mutation strategy, the one with the lowest total is said to be the best performing. Additionally, a boxplot highlights the variance in performance and further supports the superiority of our selected mutation strategy in the given context. Lastly, we plotted all recombination rates against the mutation rates and coloring each data point based on its evaluation count. It was done in hopes of finding a functional dependency between the two patterns. The graph was made whilst keeping all other parameters constant.

4.5 Survival weighting exponent

Ability to select survival weighting exponent was a late addition to the algorithm and has not been included in LHS experiments. In order to see the impact of the survival exponent two experiments were conducted. Both using the optimal algorithm found previously. For the first a range of exponents was tested a low number of runs to see where the algorithm started to perform noticeably worse. From this the range 0.5 to 57.5 was selected and used to test 1000 runs to find the best performing exponent. The second experiment was conducted by testing 3000 runs of an exponent and averaging fitness scores per generation. This was done for the average population fitness, fitness of the best individual and fitness of the the worst individual. This was done for 3 exponents to visualise the impact on population diversity.

4.6 Cross-dimensional Experiments

4.6.1 Analysis of Recombination Rates

Investigates how the recombination rate impacts evaluation count as dimensionality increases, all else is kept constant. The recombination operator inherently introduces diversity. The higher the recombination rate, the higher the diversity. The hypothesis is that the effectiveness of high recombination rates drop off as the dimensionality grows. The results highlight our performance in 7 and 9 dimensions.

4.6.2 Analysis of Dynamic Recombination and Mutation Rates

The dynamic mutation and recombination rates are run against an identical setup. The only difference is the dynamic mutation and recombination rates are adjusted by the exploration and exploitation parameters. Each of the methods, the static and dynamic one, get as input the same initialized population, both output a generation count. The generation count represents the amount of generations it took to reach the solution. The evaluation of the outcome for each setup is visualized in a two dimensional line plot. An average is calculate for both graphs. They are colored in two different colors and put each other for easier comparison. A label is outputted with the average generation count displayed with two decimals of precision. In the static scenario, the recombination and mutation rates are static and kept high throughout. However, in the dynamic setup, the recombination rate starts at its maximum and subsequently declines. The mutation rate does the opposite. The results are visually displayed in genome sizes of 8, 11 and 13.

4.6.3 Is Genocide beneficial?

Quite often the population gets stuck at certain bottlenecks, that is, local minima. To tackle this, and by brute force introduce diversity upon stagnation, we investigated the effectiveness of genocide. More precisely, given a scenario in which the maximally fit candidate solution doesn't increase in its fitness score for three consecutive generations, a fraction of the population is genocided. Stagnation is measured by a non-change of fitness score within a specified threshold of tolerance $\epsilon < 0.02$. The tolerance bounds the difference between the previously most fit individual and the current most fit individual of the population. Upon genocide, each individual from the initial population is sorted by their normalized fitness score. The fitness score is exponentiated by a constant. The constant makes it more likely to select the worst performing candidate solutions for genocide. The genocided portion of the population is replaced by a newly and randomly generated set of candidate solutions. These candidate solutions have unique gene values, that is, no horizontal collisions. The experiment is run for almost fifty starting populations (setups), and generation count of solution convergence is displayed on the y-axis. Otherwise, a similar visualization scheme is utilized as in the previous subsection. The dimensions that were investigated include 6, 8 and 10.

4.6.4 When to trigger Genocide?

Note that the genocide was previously triggered when the difference between the difference between the currently and previously most fit individual is lesser than ϵ for more than three consecutive generations (the stagnation counter is reset at each genocide). However, we were interested in investigating if it makes any notable difference if the genocide instead was triggered one a stagnation of the mean fitness value of the whole population. The graphs were similarly generated as before, however, this time, 100 populations were used in testing. The only difference between plots will be on what basis the evolutionary algorithm triggers genocide. The results were generated for genome sizes of 6, 8 and 10.

4.6.5 Rate of Genocide?

Lastly, we investigated the the effects of having a dynamic genocide percentage. The idea is that higher genocide percentages are more desired early on in the search, and lower rates in the later stages of the search. This is especially meaningful in higher dimensions. The probability of generating solutions with worse fitness scores than one already has increases as the dimensionality grows. Additionally, destroying a large portion of the current population in high genome sizes will destroying the progression that many of them made.

4.7 Comparing algorithms

We compared the performance of the original genetic algorithm with the latest improved version. To evaluate the improvements in the new genetic algorithm, we benchmark both algorithms on two different genome types: GENOME 8 and GENOME 12.

Our objective is to assess the efficiency gains in terms of time measured in seconds and the number of evaluations needed to achieve a solution. We ran both algorithms for 50 iterations, with a maximum of 10,000 evaluations per iteration, measuring the average number of evaluations and the average time taken for each iteration.

5 Results & Analysis

5.1 Strategies and Numeric Parameters

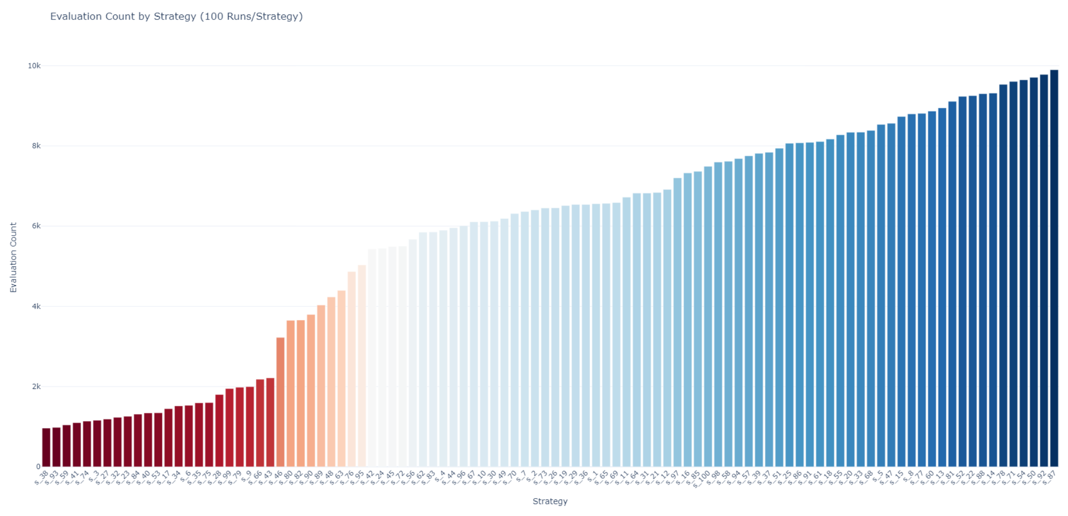

Figure 5: Bar plot of 500 setups. Ranked in accordance to their evaluation count, smaller is better. Sorted in ascending order.

Table 4: Combined Initialization, Recombination, and Mutation Strategies

Category	**Parameter / Strategy**	**Occurrence**
Initialization	Random	10
	Permutation	10
Recombination	Partially mapped crossover with duplicate removal	4
	Ordered crossover	5
	Partially mapped crossover	1
	Two point crossover	6
	One point crossover	2
	Even cut and crossfill	2
Mutation	Inversion Mutation	7
	Swap Mutation	4
	Duplicate Mutation	2
	Scramble Mutation	3
	Creep Mutation	4

The table above highlights of the specific operators found within the top 20 best performing configurations. One should keep in mind that there is a inherit stochasticity with respect to the rankings, thus, one should vary of extrapolating much information from the above top performers. If anything, their combination of strategies and numeric operators may be stated to work fairly well for the specified dimensionality and problem set. Nonetheless, we further scaled up the configuration set in hopes of finding general trends within the specified dimensionality.

Below follows three graphs that visually highlight some of our main results. Lower values are better, higher values are worse, that is, it took more operations to reach a solution. Clearly, partially mapped crossover with duplication removal, was the most fit recombination strategy in this context. It was followed by partially mapped crossover without mutation and ordered crossover with mutation. Partially mapped crossover and ordered crossover introduce a great level of population diversification in addition to removing duplicates (which ensures that a high fitness score is maintained).

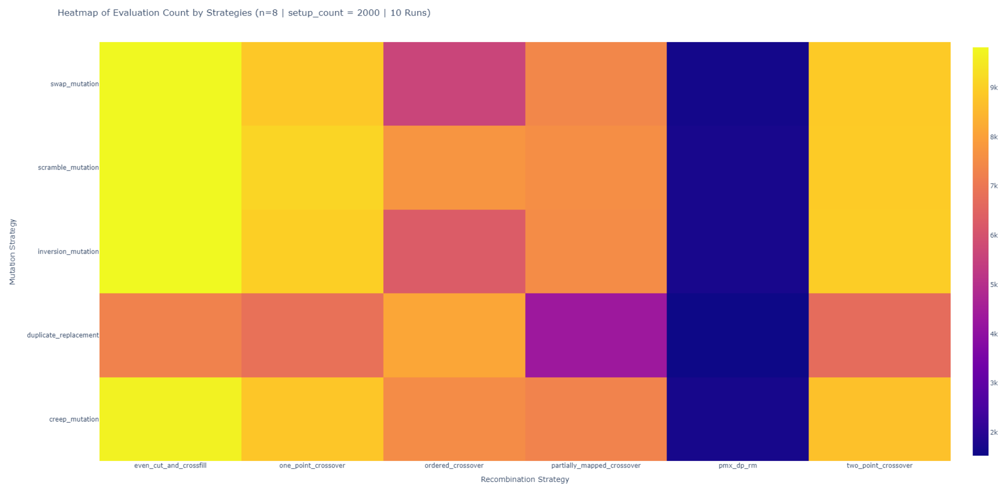

Figure 6: Each recombination operator is matched against a unique mutation operator. Each box is colored based on evaluation count. Partially mapped crossover (with and without) and Ordered Crossover performed best. The best performing mutation strategy is duplicate replacement.

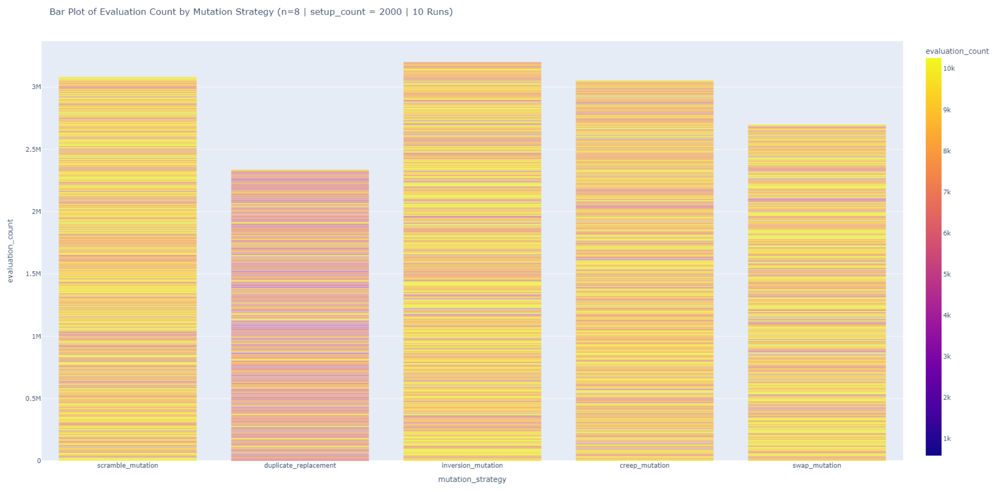

Figure 7: Stacks the total evaluation counter across all setups in which the mutation strategy appears. Duplicate remover performs the best, followed by swap mutation.

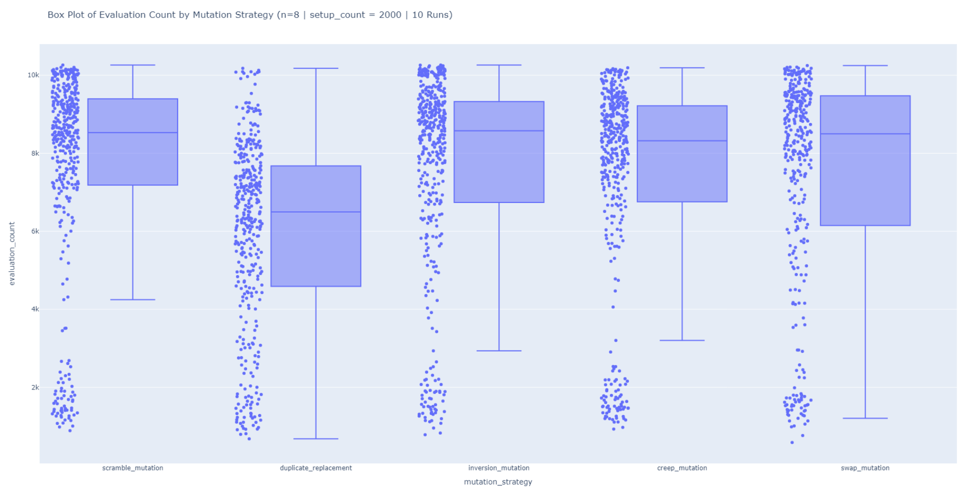

Figure 8: Box plot of mutation strategies. Duplicate replacement and swap mutation performs the best.

The best performing recombination operators are partially mapped crossover (PMX) and ordered crossover (OX). It is a excepted result since they both ensure that their offspring do not have row-wise collisions. This in combination with that they guarantee a substantial degree of exploration results in quick convergence rates. Whilst the others recombination operators do offer a decent degree of exploration, they do not remove duplicates. This makes each of their candidate solutions fitness scores suffer. This point is further highlighted with the use of duplicate replacement as an mutation strategy. It systematically and substantially lowers the evaluation count for the other less performing recombination strategies. However, we desired more clarity with respect to the relative performance of the mutation strategies. We made two plots in order to further investigate their performance. Figure 7 highlights the overall performance of each mutation strategy. One can easily through visual inspection extract that duplicate replacement not only had the lowest total evaluation count, but also best box-plot performance. Since the dimensionality is rather low, all will by random happen to find the solution quite often. However, one has also to keep in mind that these mutation strategies serve different purposes. Many are exploitation strategies, especially creep mutation and swap mutation. Thus, one may ideally want to combine between them. For example, only use the exploitative strategies to fine-tune solutions when they have managed to reach a threshold fitness score.

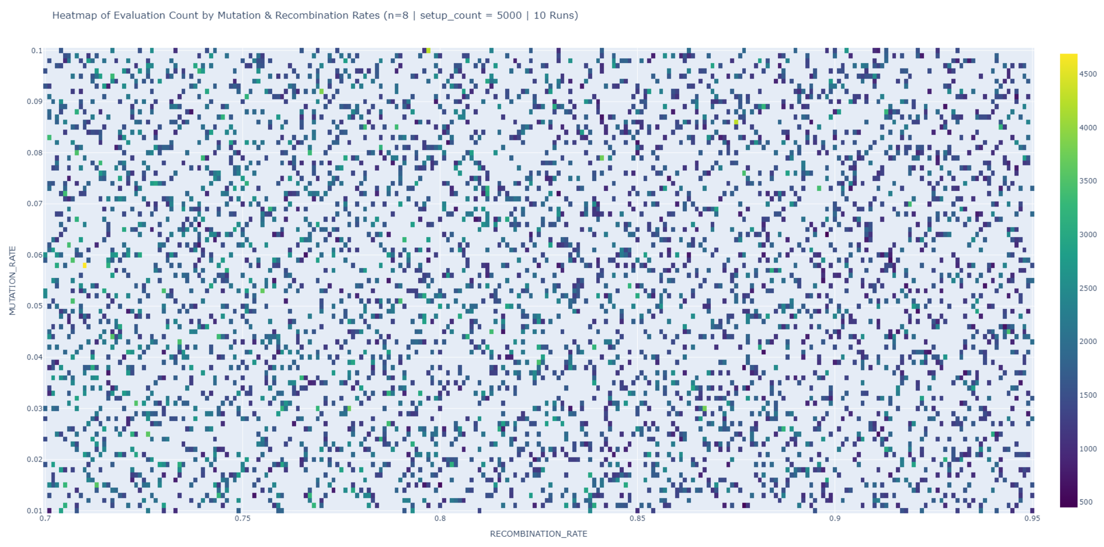

Figure 9: Each point represents a setup, colored by its average evaluation count. Stochastic pattern, apparent functional dependency.

The above diagram was an investigation into a functional dependency between the best performing combination of recombination and mutation rates. Unfortunately, no such functional dependency was observed. Instead a rather stochastic distribution of the configurations emerged. The reason for not being able to observe a functional dependency may be because of two reasons. One is that the functionally dependency may be too complex to be captured in a two dimensional plot. Another is that there is no pattern, that is, it is supposed to be stochastic. The recombination and mutation operators introduce stochastic variations into the population, and therefore the distribution of the best performing setups might be inherently stochastic. In addition, it might be that the solutions themselves are stocastically distributed in the space, under which one might expect to observe such a pattern. Nonetheless, further hypothesis testing may be needed.

5.2 Survival weighting exponent

Within the probabilistic survival selection operator, a weighting exponent has been included. This is in order to control the trade off between exploration and exploitation. Further explanation of its influence found in section 3.3.1. To select a suitable exponent for our final algorithm exponents from 0.5 to 57.5 have been explored. An optimal exponent in the case of our final seems to exist in the span 10 to 18 where a maximum efficiency is observed.

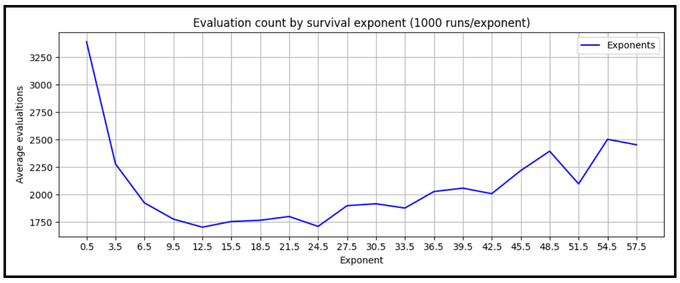

Figure 10: Evaluation of exponents

The impact of the exponent can be clearly seen in the figure bellow testing exponents 0, 15 and 50. An exponent of 0 results in a random selection of survivors and in turn a large population diversity with fitness scores ranging from 0.6 to 1 when a solution is found. In this case there is not enough incentive to achieve a high fitness and as a result the algorithm performs poorly. At an exponent of 15 within our optimal range, the population is diverse enough to not disregard potential parts of the solution. It has a much stronger incentive to achieve a high fitness score. This combination results in a balance between exploration and exploitation and has given best results. With an exponent of 50 a very high evolutionary pressure is applied to the population. The result of this is a low population diversity with the entire population settling between 0.9 and 1. The low diversity prevents optimal functioning by eliminating individuals which could have been needed to find a solution.

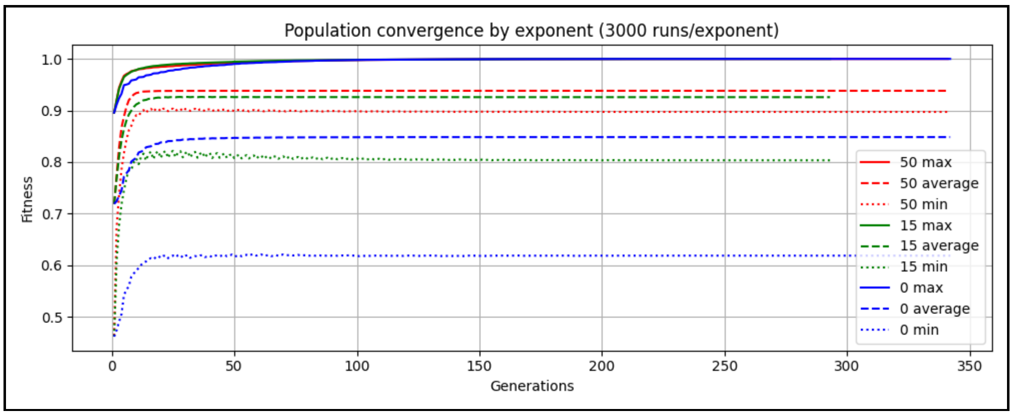

Figure 11: Exponent convergence

5.3 Cross-dimensional Experiments

5.3.1 Analysis of Recombination Rates

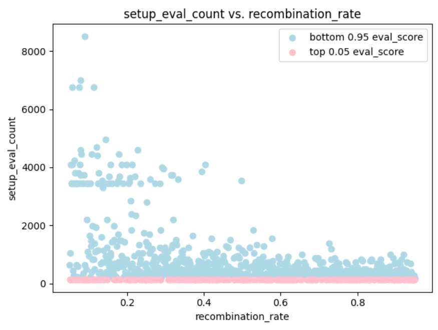

Figure 12: Performance Effects of Varying Recombination Rates at n=7

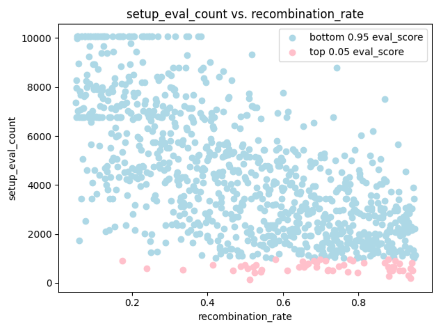

Figure 13: Performance Effects of Varying Recombination Rates at n=9

We observe that high recombination rates are less effective in higher dimensions than in lower. Effectiveness is measure by evaluation counts at convergence. In lower dimensions, high recombination rates entail low evaluation counts. At larger genome sizes, say 9, the share amount of high evaluation count solutions has increased. In general, the pattern is expected since there is a higher probability of producing offspring that have a lower fitness score than their more fit parents. Note that we also observe a slight shift amongst the best performing solutions as we increase the genome size. The lower recombination rate in low dimensions do not make a larger impact since the spaces are small. Even smaller recombination rates stand a great chance of randomly landing at a solution. This probability is greatly reduced at higher genome sizes.

5.3.2 Analysis of Dynamic Recombination and Mutation Rates

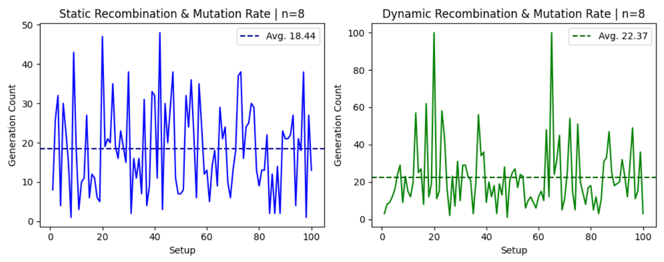

Figure 14: Static vs. Dynamic Recombination and Mutation Rates in n=8

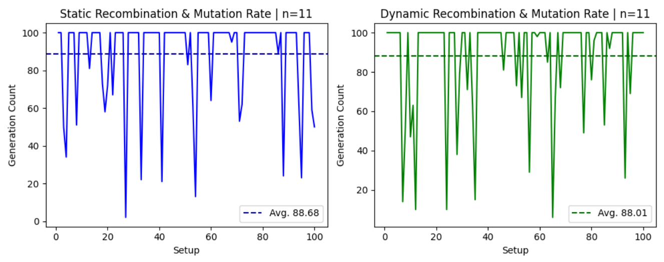

Figure 15: Static vs. Dynamic Recombination and Mutation Rates in n=11

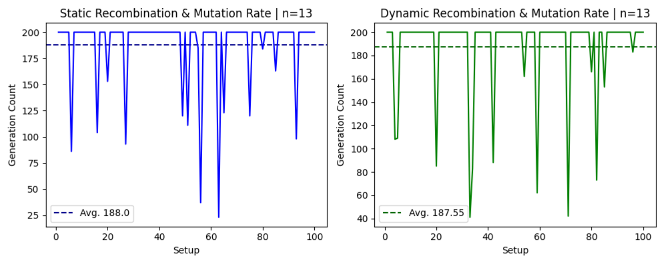

Figure 16: Static vs. Dynamic Recombination and Mutation Rates in n=13

The recombination and mutation rates are initially set high. The recombination rate subsequently decreases and the mutation rate increases. The main idea is to highlight the effectiveness of exploitation at higher genome sizes. And we surely observe as we increase n from 8 to 13 that the gap got tighter. Initially the static parameters performed better since exploitation is not as important. To increase our performance at lower dimension, we can simply decrease the rate at which recombination drops and mutation rate increases. This is accomplished by decreasing the k value in sections 3.3.2 and 3.3.3.

5.3.3 Is Genocide beneficial?

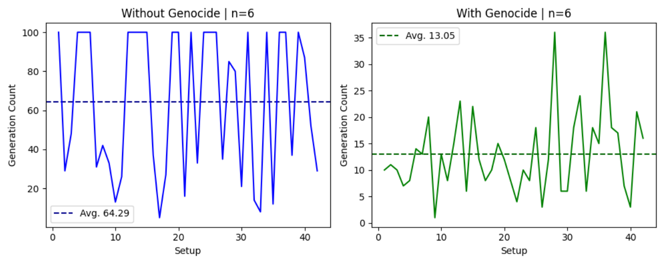

Figure 17: The effectiveness of Genocide in n=6

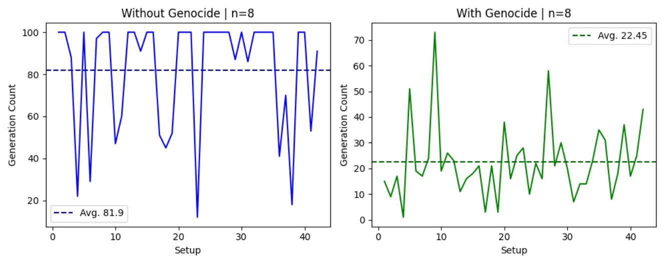

Figure 18: The effectiveness of Genocide in n=8

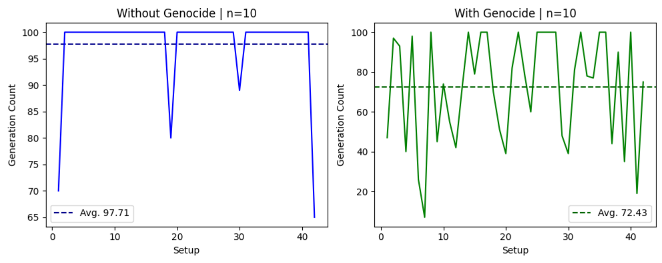

Figure 19: The effectiveness of Genocide in n=10

Genocide was clearly beneficial at all tested genome sizes. Especially at lower, where it enables us to quickly spawn individuals to explore huge subsets of the search space. Albeit, for similar reasons mentioned in previous sections, exploration loses its powers at higher dimensions. Nonetheless, it does make a difference. It is worthy of investigating whether one should not only take into account the fitness of individuals upon genocide but also their age. This would enable new and less fit candidate solutions to be genocided but older fit individuals to employ exploitation strategies, hence further increase the rate of convergence.

5.3.4 When to trigger Genocide?

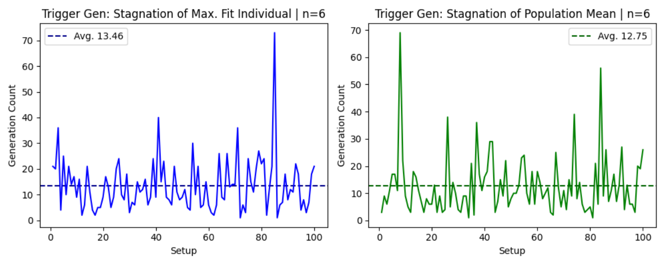

Figure 20: Trigger Genocide Analysis in n=6

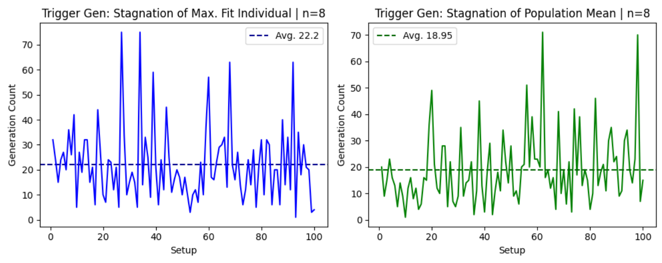

Figure 21: Trigger Genocide Analysis in n=8

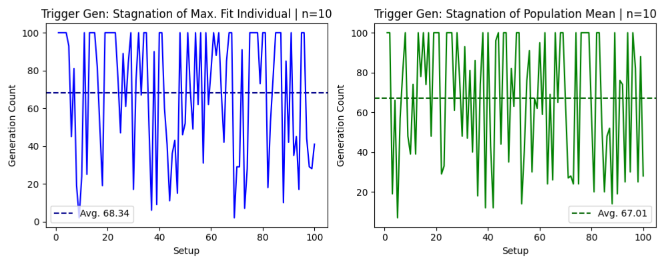

Figure 22: Trigger Genocide Analysis in n=10

It did not appear to make any notable difference if we triggered the genocide based on stagnation of the most fit individual in the population or the mean fitness of the population. At least, one can not meaningfully extrapolate that conclusion. We can note from figure 13 that as the mean tends to stagnate, so does the maximum value. However, in cases where that is not the case, one should trigger the genocide based on the mean in order to capture global population trends. It may not always be case that their behaviour is similar.

5.3.5 Rate of Genocide?

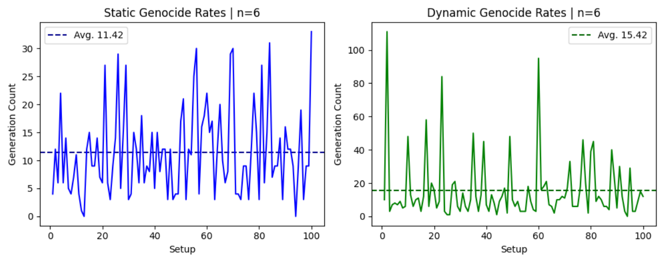

Figure 23: Rate of Genocide Analysis in n=6

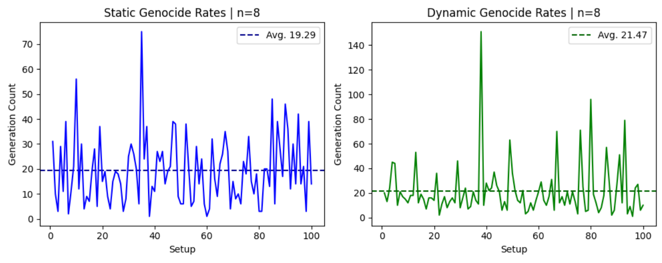

Figure 24: Rate of Genocide Analysis in n=8

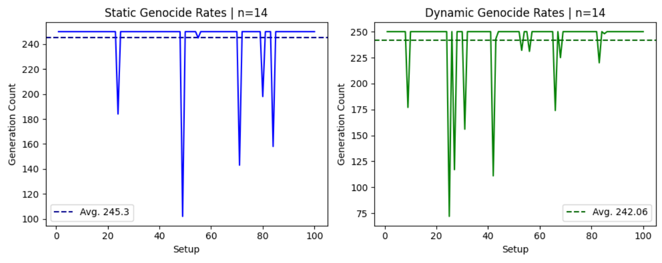

Figure 25: Rate of Genocide Analysis in n=14

We lastly experiment with varying genocide rates. The genocide rate decreases by generation count. It is regulated by the exploration factor. It is higher in the beginning and low at higher generation counts. Similar to past experiments, as the genome size kept increasing, the average convergence rate of the population with a dynamic genocide rate kept improving until it beat the static.

5.4 Results of algorithm comparisons

Algorithm Version	Average Evaluations	Average Time (s)
New and Improved GA	1224.0	0.0412
Old GA	1246.1	12.9645

Table 5: Performance Comparison for GENOME 8

Algorithm Version	Average Evaluations	Average Time (s)
New and Improved GA	6840.0	0.2372
Old GA	9800	208.1015

Table 6: Performance Comparison for GENOME 12

The new genetic algorithm performs better both in terms of time, as well as evaluations needed to find a solution. With the margins increasing with the genome size. Looking at the numbers, the new one managed to find solutions at **30,20%** less iterations whilst being **99.89%** faster.

$$\left(\frac{9800 - 6840}{9800}\right) \times 100 \approx 30.20\%$$

$$\left(\frac{208.10 - 0.237}{208.10}\right) \times 100 \approx 99.89\%$$

6 Final Remarks

6.1 Summary

Within this project we have explored the application of Genetic Algorithms specifically on the N-Queens problem, where the goal is to place find a valid placement of N chess queens on a $N \times N$ chess board, in such a way that no queen threatens each other. The project focused on how different combinations of genetic operators, parameter values, and additional strategies will effect the performance on this specific problem. We used parameter tuning techniques to evaluate 500 different setups against each other, which in turn produced interesting insights.

We saw that some recombination strategies significantly outperformed others. Even-cut-and-cross-fill performed worst overall, while partially mapped crossover performed best, partially given its ability to remove duplicates, which avoids a large amount of horizontally invalid solutions.

When testing different mutation strategies we didn't see quite the same results. Although there existed a clear outlier, they performed overall the same. The only strategy that sticks out among the rest is the duplicate replacement strategy, which mutates values by replacing them with those missing from the solution array. When this strategy was combined with partially mapped crossover it did not in fact show a statistically significant difference compared to other mutation strategies.

Interestingly we saw that adjusting mutation rate (between 0.1 and 1.0) and recombination rate (between 0.7 and 0.95) had less impact on performance compared to the choice of operators. The project also explored using dynamic parameter values that adjusts with every generation. This did not either produce better results.

Finally, the introduction of genocide to prevent stagnation improved performance markedly, showing promising improvements at the three different sizes of N-Queens that were test.

This project and report have demonstrated the effectiveness of evolutionary algorithms for N-Queens, which suggests its continued effectiveness on other problems.

6.2 The Future

We believe that we have produced some interesting results, but there are several more avenues that could be explored. Both in terms of improving what we have, as well as exploring other techniques. Listed below are things we believe could be interesting to explore.

Genetic operators: There are still many genetic operator strategies that we did not explore. A wider range of operators could yield even better results.

Operator selection: Implementing a system that selects operators based on the current generation could enhance adaptability.

Aging population: Introducing aging into the population, where older individuals are more exploitative and younger ones more explorative, could help maintain a balance between exploration and exploitation.

Problem size: Expanding the analysis to a wider range of N-Queens problem sizes could provide insights into the algorithm's scalability and effectiveness across for even larger problem sizes.

Comparison: A more in depth comparison with other studies, possibly including quantitative analysis, could give us broader insights while comparing our results to others.

References

[1] Jordan Bell and Brett Stevens. A survey of known results and research areas for n-queens. *Discrete Mathematics*, 309(1):1–31, 2009. doi: 10.1016/j.disc.2007.12.043.

[2] M. Boiiković, M. Golub, and L. Budin. Solving n-queen problem using global parallel genetic algorithm (gpga). In *Proceedings of the 25th International Conference on Information Technology Interfaces (ITI)*, 2003.

[3] Agoston E. Eiben. *Introduction to evolutionary computing*. Springer, 2003.

[4] S. S. Chauhan Gonder and P. Garg. Hybrid crossover operator in genetic algorithm for solving n-queens problem. *International Journal of Engineering and Advanced Technology*, 2022.

[5] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with NumPy. *Nature*, 585(7825): 357–362, 2020.

[6] V. Jain and J. S. Prasad. Solving n-queen problem using genetic algorithm by advanced mutation operator. *International Journal of Computer Applications*, 2018.

[7] U. Sarkar and S. Nag. An adaptive genetic algorithm for solving n-queens problem. *International Journal of Computational Intelligence*, 2017.

[8] Adam Slowik and Halina Kwasnicka. Evolutionary algorithms and their applications to engineering problems. *Neural Computing and Applications*, 32: 12363–12379, 2020. doi: 10.1007/s00521-020-04832-8.

[9] Ingrida Steponaviče, Mojdeh Shirazi-Manesh, Rob J. Hyndman, Kate Smith-Miles, and Laura Villanova. On sampling methods for costly multi-objective black-box optimization. *On Sampling Methods for Costly Multi-objective Black-box Optimization*, September 2015. Working paper.