Master's Thesis
Genetic Algorithms

DTU Compute
Department of Applied Mathematics and Computer Science

Implementation and Evaluation of Evolutionary Algorithms with Self-adjusting Operators

By: Benjamin Eriksen(s153724)
Supervisor: Carsten Witt

June 11, 2021

Preface

This is the master's thesis for the MSc programme in Computer Science and Engineering at DTU. It is created by Benjamin Eriksen and supervised by Carsten Witt.

I want to thank everyone who assisted me throughout my studies. Firstly Carsten Witt for supervising this thesis and putting a great effort into helping me develop and execute this project. I am also grateful for all of those who through listening to my ramblings about this project, helped me form it. A special thanks goes to my family, whose endless support prepared me for and helped me throughout my studies.

The entire project is located at
https://github.com/BenjaminEriksen95/GeneticAlgorithmProject.

Contents

Abstract

This thesis will focus on meta heuristic algorithms that use evolutionary operators to solve optimization problems especially those that have mechanisms for self-adjusting behavior. The focus is to demonstrate how the challenges of theoretical performance evaluation of these can be overcome through empirical tests in a framework that allows the algorithms to be tested on a series of optimization problems that model several signature challenges of optimization. In a addition to developing the framework for these empirical tests, several state-of-the-art algorithms are implemented and tested on known optimization problems such as TravelingSalesman and 3-Sat. Considering the performance of the different algorithms a new method is proposed and tested to demonstrate how the framework can easily do what it was intended to.

1. Introduction

Evolutionary algorithms is a sub-field of metaheuristics that attempt to replicate the evolutionary process of nature to create optimal solutions to problems. This thesis will focus on the evolutionary algorithms that are used to solve optimization problems, which will be referred to as genetic algorithms.

This thesis will investigate the mechanisms of genetic algorithms, the problems they are used to solve and their challenges in regards to parameter control. It will further explore how these challenges can be mitigated through the use of self-adjusting parameters, resulting in algorithms that are more generic and have an improved performance.

To understand this process lets first explore the concept of evolution. All living beings are attempting to solve the problem of life, which is to survive the evolutionary pressure and continue their genetic linage. Darwinian evolutionary theory argues that the fittest will survive. The fittest being, not the strongest or the smartest but, those which was the best fit for the environment. This process happens due to the evolutionary pressure which happens due to obstacles in the environment, such as a limit on the amount of resources or a predator. With each new generation, new genes are randomly added to the gene-pool through mutation and successful genes are passed down from the parent(s) allowing the species to adapt to the evolutionary pressure.

The field of evolutionary algorithms attempt to solve simpler problems by simulating an environment where the best from each solution is forward propagated through so that the quality of the solution or solutions trend toward the optimum solution.

There are several different strategies for genetic algorithms, which makes sense as they are used to solve a wide range of optimization problems that have different characteristics, meaning that some algorithms are more suited for some problems. A common problem for all of the genetic algorithms is that they are subject to a set of parameters that influence the performance of the algorithm, just as the evolutionary pressure varies depending on the environment. The parameters however are not limited to the selection pressure, but are often also used to determine the rate of exploration.

Choosing good parameters has been shown to be a task that is tedious at best, and impossible at worst. This is due to the issue, as highlighted by a holistic view of recent research, that static parameters are often unable to maximize performance of the algorithms. This has resulted in research into the field of self-adaptive genetic algorithms, which attempt to solve the problem of parameter control by incorporating mechanisms for adaptive parameters into the algorithms. These can be exogenous parameter adjustments

based on the state of the algorithms such as the time passed, the success of the previous parameters or even endogenous, where the strategy is encoded in the genetics of a solution.

This thesis will explore these methods and investigate and implement examples of some of the promising strategies to mitigate the challenges of parameter control. For this a software framework for empirical analysis will be implemented and the different algorithms tested. The focus is to be able to use the framework to extract knowledge of how the different mechanisms can be applied and modified to create algorithms that outperform the current state-of-the-art algorithms.

2. Genetic Algorithms

This chapter will introduce the concepts of genetic algorithms(Ga's), the problems they are used to solve and the methods by which they solve them. But firstly the meaning of genetic algorithms has to be clarified. The literature has widely different classifications within Evolutionary Computation/Evolutionary Algorithms(EA's). Many of the algorithms considered in this thesis are labeled as being in the set of class of evolutionary algorithms, although they do not fit into several of the subsets within the class of evolutionary algorithms. This means that it is at best imprecise and due to cause confusion. My belief is that all of the algorithms fit best into the set of genetic algorithms which is a subset of evolutionary algorithms. GA's are in some literature reserved for the evolutionary algorithms that solve optimization problems primarily through the method of crossover. In by opinion this categorization should include evolutionary algorithms that solve optimization problems through the use of evolutionary operators, just as the term evolution also includes organisms that evolve solely through mutation. This means that all algorithms in this thesis paper is labeled as genetic algorithms, and the theory will focus only on what is applicable to evolutionary algorithms that are used to solve optimization problems(section 2.1).

A genetic algorithm can thus be defined as a iterative randomized black-box algorithm that is used find the best solutions in the search space of an optimization problem through evolutionary operators. As the field is based upon the method of the evolutionary process, the terminology is also largely inspired by the field of genetics and evolution.

Each solution in the search space is called an individual and is represented by an encoding of its features. The encoding is called a chromosome and each feature is called a gene. For the problems in this thesis any chromosome a list of genes of a certain length, describing the size of the problem which is commonly referred to as n. The algorithms function iteratively by generating a set of new individuals from the current generation, then applying evolutionary pressure to only allow those that are sufficiently fit into the next generation. This is done through three evolutionary operators. Selection(section 2.2), which is used to apply evolutionary pressure and thus removing the less fit genes from the population, based on the fitness of the individual. Along with mutation(section 2.3) and crossover(section 2.4) which are used to evolve the current population to generate new valid offspring.

How these mechanisms can be combined into algorithms is explored in section 2.5, where examples of genetic algorithms from the literature are explored. Following this the challenge of parameter control is explored in section 2.6 to motivate the need for

self-adjusting genetic algorithms.

2.1 Optimization Problems

Optimization problems are problems that have a well defined search space consisting of an often huge set of valid solutions to the problem. The issue is therefore not finding a solution, as those are already given by the definition of the search space, but finding an optimal solution. An optimization problem has a score function, that when applied to any solution in the search returns a value. As each solutions has a score we can imagine that adding a score dimension to the search space will give us a landscape. The goal of the optimization problem is to find the lowest or highest point in the landscape, depending on whether it is a minimization or maximization problem.

So to be able to solve optimization problems with genetic algorithms a solution has to be able to be translated into genes that can be collected into a chromosome. The score function can be used to determine the fitness of a solution and thus be used to apply evolutionary pressure. The final step is to be able to traverse the search space of the problem, which varies depending on the encoding of solutions into chromosomes. The problems considered in this thesis is divided into two encoding categories. Problems with a binary encoding represent chromosomes as a list n of bits(binary encoding), where each bit is a gene. Problems with permutation encoding represent chromosomes as a list of n items, where each item is a gene. As the encoding is different, so is the search space. The number of possible solutions to a binary problem is 2^n, whereas the number of possible solutions to a permutation problems is $n!$. Considering each solution to an optimization problem is thus computationally infeasible as the size of n grows. In the following subsections we will explore how this applies to actual optimization problems.

2.1.1 OneMax

OneMax is the problem of maximizing the number of ones in a bit string. It is a simple, yet commonly used benchmark problem. The encoding of any solution to OneMax is simple done by representing each bit as a gene resulting in a chromosome being a list of n genes. This kind of encoding is called a binary encoding and is used in several problems. The fitness score of any chromosome $X = \{x_1, x_2, ..., x_n\}$ is thus

$$fitness(X) = argmax_{X \in \{0,1\}^n}(f(X) = \sum_{i=1}^{n} x_i)$$

2.1.2 LeadingOnes

LeadingOnes is the problem of maximizing the number of consecutive one bits in a bit string, starting from the 0'th location. This makes it more difficult than OneMax as there at all times exactly one bit that will result in a higher fitness. The encoding of any solution to LeadingOnes is done with binary encoding.

The fitness value of any chromosome $X = \{x_1, x_2, ..., x_n\}$ is thus

$$fitness(X) = argmax_{X \in \{0,1\}^n}(f(X) = \sum_{i=1}^{n} \prod_{j=1}^{i} x_j)$$

2.1.3 JumpM

JUMP$_m$ is the problem of maximizing the number of bits in a bit string, with the caveat that there is a jump in the fitness score next to the maximum fitness, which makes it a bimodal problem. The problem is similar to ONEMAX but has a m-sized jump in the score. The encoding of any solution to JUMP$_m$ is done with binary encoding. The fitness value of any chromosome $X = \{x_1, x_2, ..., x_n\}$ is thus.

$$fitness(X) = \begin{cases} m + |X|_1 & if\ |X|_1 \leq n - m\ or |X|_1 = n \\ n - |X|_1 & otherwise \end{cases}$$

where $|X|_1$ is the number of ones($|X|_1 = argmax_{X \in \{0,1\}^n}(f(X) = \sum_{i=1}^{n} x_i)$) in X.

2.1.4 3-Sat

3-SAT is the problem of satisfying a number of clauses with 3 literals and is a subset of the SATISFIABILITY problem and is NP-complete. 3-SAT is a multimodal problem which makes it more complex than previous problems, that were either unimodal or bimodal. The goal is to find a setting of m variables so that all of the n clauses are satisfied. Like the previous problem solutions are encoded as a binary list which represent a set of literals that are either positive or negative, meaning that if x_1 is possible the 1'rst entry in the list is a one and so on. This means that solutions are encoded as $X = [x_1, x_2, ...x_m]$. A clause is satisfied if least one of its 3 literals hold true. The set of clauses can be written as $C = [[a_1, a_2, a_3], [b_1, b_2, b_3], ..., [m_1, m_2, m_3]]$. The set clauses can be written as resulting in the following fitness score calculation.

$$f_i(X, C) = \begin{cases} 1 & if\ least\_one\_true(X, C[i]) \\ 0 & otherwise \end{cases}$$

$$fitness(X) = \sum_{i=1}^{m} (f_i(X, C))$$

2.1.5 TSP

TRAVELINGSALESMAN is the problem of finding the order of locations that result in the shortest route between a number of locations, where there it is possible to determine the distance between any pair of locations. Unlike the previous problems it is encoded as the order in which the locations are visited, resulting in a permutation encoding, as each solution must have each location represented exactly once. The problem is

classical NP-complete benchmark for permutation problems. A solution is represented as $X = [x_1, x_2, ...x_n]$ with a fitness score of.

$$fitness(X) = \sum_{i=1}^{n-1}(distance(x_i, x_{i+1}))$$

2.1.6 Sorting

SORTING is the problem of sorting a list of items. This is typically not seen as an optimization problem, but it can be considered as such by considering the level of sortedness of a list the score to optimize. The goal of the problem is to find the permutation of the items in the list that makes it fully sorted. The measures of how sorted a list is is called presortedness and can be measured in a series of ways. The measure used for this thesis is called INV by [STW04] is used. INV uses the number of pairs that are in the correct order as the heuristic for how sorted the list is. As with TRAVELINGSALESMAN a solution is represented by some permutation, in this case of the items. $X = [x_1, x_2, ...x_n]$

$$f_i(X) = \begin{cases} 1 & if\ x_i \leq_{i+1} \\ 0 & otherwise \end{cases}$$

$$fitness(X) = \sum_{i=1}^{n-1}(f_i(X))$$

2.2 Selection

Selection is the mechanism that applies evolutionary pressure. This happens through the selective maintenance of the population size. The method of selection is essential to the performance of the algorithm. If the selection pressure is very high the algorithm will rapidly converge towards the nearest optimum, but will be more likely to kill off solutions with high potential because they are in the early stages of their evolution. This is optimal for some problems where the algorithm are not likely to get stuck and very detrimental to others as it will never reach the optimum solution.

Some algorithms deploy less aggressive selection for the population, but use the mechanism of elitism to make sure that the best solutions are certain to reach the next generation.

Selection by itself is not very useful, as it will not generate any new solutions, only exploit those that deemed most beneficial from the current population.

2.2.1 Population

The initial population is sampled randomly from the search space. As selection maintains the population of individuals, the size of the population is essential to the functioning of the algorithm. The size of the population represents the number of possible solutions the algorithm can explore simultaneously. The more rugged the fitness landscape of the problem is, the more beneficial a larger population size tend to be. The issue is that each iteration of the algorithm has to evolve each individual in the population, causing

the computational complexity to grow rapidly, compared to smaller populations. The population size is often static and algorithms can therefore be split into two categories in regard to population size, multi and single population. Many algorithms use only a single individual in the population, allowing for a rapid evolution, especially on unimodal problems. To maintain the size of the population new individuals must be added as others are removed. This happens through the process of exploration which is done through mutation(section 2.3) and/or crossover(section 2.4).

2.2.2 Roulette Wheel Selection

Roulette wheel selection maintains the population size by selecting individuals with a probability proportional to their fitness. If the probability is directly proportional, the probability of picking c_i is:

$$p(c_i) = \frac{f(c_i)}{\sum_{j=1}^{n} f(c_j)}$$

This method allows exploring individuals based on their fitness, but does not guarantee that the best individuals make it through the selection, which can cause the evolution to revert. It thus works best with multi population algorithms and some degree of elitism, which will be explored later.

2.2.3 Tournament Selection

Tournament selection maintains the population size by creating n-sized winner takes it all tournaments. This selection method allows less fit genes to survive if they compete in a easy tournament, yet guarantees that the best individuals will be passed to the next generation.

2.2.4 Elitism

Elitism secures that the most fit individuals are not removed from the population by any of the evolutionary mechanisms. An algorithm is elitist if the fittest of a generation is always passed on to the next generation without mutation. This is done to secure that iterations can not cause a worsening of fitness. For single population algorithms this is often not necessary, as they are inherently elitist. For multi population algorithms this can be done adding a selection of the most fit individuals to the next generation without any changes from mutation and crossover. This is especially useful for selection methods such as roulette wheel, which doesn't guarantee that the best individual is selected for the next generation.

2.3 Mutation

Mutation is a mechanism for exploring the search space. Mutation must therefore create valid solutions to the problem, and thus have to take into consideration the encoding of the problem. A mutation operator takes a chromosome and creates a copy with a small randomly generated variation in the genes. It thus explores solutions that are similar to

the current population. Mutation in combination with selection allows the algorithm to evolve the population and thus traverse the search space towards the optimum solution.

Depending on the encoding of individuals different mutation operators are used, as they have to generate offspring within the search space of the problem. A useful mental image of mutation is, that it takes a step of some size in a direction from the parent individual. This step has some degree of locality, meaning that it prefers solutions that are close to the one being mutated. Some operators are considered global, as they can reach any solution in the search space, within a single mutation. Others can only reach within a certain bound of the current solution, and are thus considered local, as they can only reach what is within the locality of the current solution.

As the encoding of problems varies there is no generic mutation operators as they need to maintain the structure of the solution, which is why they are split into the encoding they are applicable to. Within each operator is a level of customizability, which is described by a variable k, which can affect the locality of the search through the mutation step size of the mutation operator.

2.3.1 Binary Mutation

The two main operators for binary mutation is *k-flip mutation*(k-flip) and *probability flip mutation*(p-flip).

- *k-flip mutation* flips k randomly selected bits in the chromosome. This is local mutation operator as only chromosomes that are k bits different from the mutated can be found. This means that algorithms utilizing this can get stuck if the mutation step size k is not sufficiently large. This operator is computationally effective as it can be handled by sampling k indexes and flipping those, resulting in a complexity of $O(k)$.
- *probability flip mutation* flips each bit with a probability $p = k/n$. This allows a single mutation step to reach any solution in a single step and the mutation operator is thus global. As this method has to consider every bit, the computational complexity is $O(n)$.

2.3.2 Permutation Mutation

The two main operators for permutation mutation is the *k-swap mutation* and *k-reverse mutation* operator. For permutation encodings there is no such thing as a global mutation operator. The operators do however differ in their step size, and we can thus consider some operators more aggressive than others.

- *k-swap mutation* moves an item to a random location, k times. It is, especially for low k values a not very aggressive mutation. The computational complexity is $O(n)$.
- *k-reverse mutation* picks 2 indexes and reverses the order of genes between the indexes. This is repeated k times. This mutation is more aggressive than *k-swap mutation*, and for higher values of k it is comparable to picking a random

chromosome. This mutation is usually quite good as it maintains most of the structure of an individual. The computational complexity is $O(k)$

2.4 Crossover

Crossover is a mechanism for exploring the search space, like mutation. Instead of evolving from a single parent, crossover combines the chromosome from two individuals. By itself, it is not able to reach any solution in the search space as it can only reach genes that are currently in the population. Instead it is used in combination with mutation. The idea behind it is that two individuals might have discovered good genes in different locations of the chromosome and they can combine these into an individual that surpassed both parents. The method used for crossover depends on the encoding of a chromosome, as the offspring individuals have to be valid solutions to the problem. Some crossover operators support a bias c, that makes the operator prefer one of the parents to the other.

2.4.1 Binary Crossover

The two main operators for binary crossover is *single-point crossover* and *biased crossover*.

- *single-point crossover* picks a cutoff point and uses the genes before $k \in [1, n-1]$ from the first parents and the rest from the second parent. This maintains the ordering and thus maintains structure, which can be an advantage for some problems.
- *biased crossover* uses each the gene from the first parent with probability c and the second with probability $1 - c$. By setting $c = .5$ the crossover is uniform. As each gene is independently picked the structure is not maintained.

2.4.2 Permutation Crossover

The operator used for permutation crossover is *ordered crossover*.

- *ordered crossover* copies a sub-string from the first parent to the exact location in the offspring, then fills in the remaining from the second parent in the order they occur in that parent.

2.5 Algorithms

This section will explore genetic algorithms without any self-adjusting mechanisms. In the field of genetic algorithms comma and plus notation is used to inform the reader about the nature of the algorithms. The meaning of comma(μ, λ) and plus$(\mu + \lambda)$ notation, refers to the population size and selection method. Here μ is the size of the parent generation and λ is the size of the offspring generation. With plus notation, the next generation is selected from both the parent and offspring generation, making the algorithm elitist by nature. With comma notation the next generation is selected only from the offspring generation. It is possible for plus and comma notation to be combined, as the new

generation can have both a mutation and crossover stage and the selection can happen from either of the stages.

When discussing these algorithms the iteration count is excluded as a parameter as the algorithms are run for a limited amount of time, and we do not care about how many iterations happen in that time and the iteration count does not affect the internal behavior of the algorithm.

Most of the algorithm have operators that are only applicable to a binary encoding. To handle this the pseudo code has been changed so that it is ambivalent to the mutation and crossover method, and the standard for each encoding is instead noted outside the pseudo code. The standard operators are based on by interpretation of the idea behind the algorithm.

2.5.1 Standard Single Population Genetic Algorithm $(1 + 1)$

The simplest genetic algorithm is the (1+1) GA. The algorithm is a variation of a hill climber algorithm, that uses global mutation step to avoid getting stuck in local optimums. Global mutation is however only possible for binary encodings(*probability flip mutation*), but a similar effect can be had by using a large mutation step(*k-reverse mutation*) for permutation encodings. This means that the population consist of single individual which is mutated randomly and if the offspring is better it replaces the population for the next iteration. This algorithm is not particularly interesting, but it is relevant to know as several more advanced algorithms build upon it in different ways. The pseudo code for this has been excluded as it is just used as a reference algorithm.

2.5.2 Standard Multi Population Genetic Algorithm $(\mu + (\lambda, \lambda))$

This algorithm is described in [Deb99] and functions by maintaining a population of size μ and both mutation and crossover to evolve the population. It also uses elitism to copy a percentage of the best in the population directly to the next generation. The algorithm takes 3 parameters;

- population size λ, which is the size of the population that is maintained.
- locality of mutation pm, which is the factor of aggression of the mutation.
- probability of crossover pc, which is the fraction of the population that is not copied directly to the next generation.

The pseudo code for the algorithm is seen in Algorithm 1 and the interpretation of the mutation and crossover operators are:
Binary:

- mutation: *probability flip mutation*
- crossover: *uniform crossover*

Permutation:

- mutation: *k-reverse mutation*

- crossover: *ordered crossover*

Algorithm 1 $\mu + (\lambda, \lambda)$ GA [Deb99]

for $i = 1, ..., \lambda \cdot 2$ **do**
  $elite\_size \leftarrow int((1 - pc) * \lambda$
  $Choose\ x_i \in 0, 1^n\ u.a.r.$
  $population_0.add(x_i)$
end for
for $t = 1, 2, 3...$ **do**
  Selection phase:
  $competitors \leftarrow sample(population_{t-1}, \lambda * 2)$
  for $i = 1, 2, ..., len(lambda * 2)$ **do**
    $survivors \leftarrow max(population_{t-1}[i], competitors[i])$
  end for
  Elitism:
  $elite \leftarrow$ pick best $elite\_size$ individuals from $survivors$
  Crossover phase:
  $shuffled1 \leftarrow sample(len(survivors), lambda - elite\_size$
  $shuffled2 \leftarrow sample(len(survivors), lambda - elite\_size$
  for $i = 1, 2, ..., lambda - elite\_size$ **do**
    $protos.add(crossover(shuffled1[i], shuffled2[i])$
    $protos.add(crossover(shuffled2[i], shuffled1[i])$
  end for
  Mutation phase:
  for $s : elite$ **do**
    $population_t.add(mutate_{pm}(s))$
  end for
  for $proto : protos$ **do**
    $population_t.add(mutate_{pm}(proto))$
  end for
end for

2.5.3 $(1 + (\lambda, \lambda))$ Genetic Algorithm

The algorithm is a single population algorithm that uses both mutation and crossover to traverse the search space. The algorithm is designed for binary problems and uses a single population from which λ_1 mutated offspring are explored through a mutation with mutation option k, which is sampled from a binomial distribution. The best offspring is chosen. Then the algorithm performs λ_2 crossovers of the population and the best offspring using a biased crossover. If a more fit individual is found the population is replaced with it. This is a clever method, as it allows using both mutation and crossover with a single population algorithm.

The algorithm was first introduced in [DDE15] which used 3 static parameters

- population size λ, which describes the number of mutations and crossovers to consider in each iteration.

- mutation probability $p = k/n$, which is used to sample the amount of mutation steps z from a binomial distribution $z \sim \mathcal{B}(n, p)$, where n is the size of the chromosome.
- crossover bias c, which decided how much the algorithm should be biased towards the current solution as opposed to it's mutated offspring.

The algorithm was later refined to carry individual variables for number of mutations(λ_1) and crossovers(λ_2).

The pseudo code for the algorithm is seen in Algorithm 2 is designed for binary encodings. The interpretation of the mutation and crossover operators are:
Binary:

- mutation: *probability flip mutation*
- crossover: *biased crossover*

Permutation:

- mutation: *k-reverse mutation*
- crossover: *ordered crossover*

Algorithm 2 $(1 + (\lambda, \lambda))$ GA [DD19a]

Choose $x \in 0, 1^n$ *u.a.r.*
for $t = 1, 2, 3...$ **do**
  Mutation phase:
  $z \sim \mathcal{B}(n, p)$
  for $i = 1, ..., \lambda_1$ **do**
    $x^{(i)} \leftarrow mutate_z(x)$
  end for
  Choose $x' \in \{x^{(1)}, ..., x^{(\lambda_1)}\}$ *with* $f(x') = max\{f(x^{(1)}), ..., f(x^{(\lambda_1)})\}$ *u.a.r.*
  Crossover phase:
  for $i = 1, ..., \lambda_2$ **do**
    $y^{(i)} \leftarrow cross_c(x, x')$
  end for
  Choose $y \in \{y^{(1)}, ..., y^{(\lambda_2)}\}$ *with* $f(y) = max\{f(y^{(1)}), ..., f(y^{(\lambda_2)})\}$ *u.a.r.*
  Selection phase:
  if $f(y) \geq f(x)$ **then**
    $x \leftarrow y$
  end if
end for

2.6 Parameters

All evolutionary algorithms are parameterized algorithms, meaning that the performance is subject to a set of parameters. This is because the different problems (and problem instances) can have significantly different fitness landscapes that requires different approaches to solve effectively. A single configuration is therefore not able to efficiently

solve all problems, or even all problem instances of a problem. As the algorithms that have been explored so far have shown, algorithms have widely different parameters that are used differently making optimal parameter setting tedious. A bigger problem is that different problems often require different parameter setting as the optimal way for navigating the fitness landscape varies.

In some cases there is no set of parameters that optimizes well throughout the entire algorithm. This is because the optimal strategy can vary depending on the state of the search. Lets consider the problem of golf for a second. When far away from the hole a long shot in a somewhat good direction beats a number of short shots in the exact right direction. As with golf the early stages of the algorithm might benefit from a smaller number of aggressive steps in a good direction, where in the later stages such a step would overshoot the optimum solution. A solution to the challenge of parameter control is self-adjusting algorithms.

3. Self-Adjusting Genetic Algorithms

This chapter will introduce the concept of self-adjusting genetic algorithms. It will look into the different approaches to self-adjusting mechanisms and how these are actualized in different self-adjusting genetic algorithms.

3.1 Self-Adjusting Parameter Control

Self-adjusting mechanisms allow the algorithm to dynamically adjust the parameters throughout the optimization and thus adapt to the actual problem. This method is used to significantly reduce the issue of parameter control, but results in a set of hyper parameters that again can be optimized. In some cases self-adjusting is done through the updating of internal parameters and in other cases it adds a mode to the algorithm that allow it to better handle certain issues. This section will go into the different approaches to self-adjusting mechanisms as they are classified by [DD19b].

3.1.1 State-Dependent Parameter Control

State-dependent parameter control uses the individual state of the search to update the parameters. It can be done by several parameters such as time, rank and state.

Time-Dependent

Time-dependent parameter control changes a parameter based on the time or number of iterations that have passed. In the beginning an algorithm is very likely to find a better solution and it is thus beneficial to use more aggressive searches. This is especially the case with local mutation, which can get stuck in local optimums. As the algorithm converges on the global optimum, the adjustments required can be very small, so that taking large evolutionary steps is no longer a valid strategy. This strategy introduces hyper parameter choices that decide how the time or iterations should affect the search. The challenge with this method is that it requires an understanding of the relation between time and search parameters. These will depend on the problem and problem instance.

Rank-Dependent

Rank-dependent parameter control sorts individuals into ranks and evolves them differently based on their rank, meaning how a solution score compared to the population.

When having multiple individuals in the population it can be useful to mutate less fit individuals more aggressively as a last resort to make them compete with the more fit individuals. This is because even if a minor mutation does improve it, it is still unlikely to be able to compete with the best. This can also be that a crossover shall be biased toward better ranked individuals. This strategy introduces hyper parameter choices that decide what ranks and how the rank of an individual should affect the evolutionary behavior. The challenge with this method is that it requires a multi population algorithm and an understanding of how the difference in fitness should be translated into ranks that influence the mutation or crossover.

Fitness-Dependent

Fitness-dependent parameter control is a less general variation of rank-dependent parameter control, where instead of splitting the individuals in different rankings, their individual score is used to decide their evolution. The challenge with this method is that is similar to the previous, but as there is no ranking, it can be utilized with single population algorithms.

3.1.2 Success-Based Parameter Control

Success-based parameter control updates the search parameters based on the progress of the algorithm, meaning that it takes into consideration the success of the previous iteration. Based on the success or lack thereof, the parameters are updated to improve the likeliness of success.

Multiplicative-Success

Multiplicative-success based parameter control changes some parameter based on whether the evolutionary generation was successful. An example is the popular 1/5'th rule([DD15]), which is used to estimate when the parameters are in their ideal state, which is when 1 in 5 iterations or mutations cause an improvement. This method balances the speed of optimization possible with taking larger steps, with the success rate of each iteration. The mechanism forces early exploration to be very optimistic and thus forces rapid convergence, while reducing the step size at later stages, so that the algorithm can pinpoint the optimum solution. It can be considered a more clever and general purpose variation of time-dependent parameter control(3.1.1), as the success rate can be used to more accurately estimate the stage of the algorithm. A similar mechanism can be achieved by dynamically updating the population generated in each iteration, this allows the algorithm to consider fewer options when improvements are likely, and increase the number of children considered as improvements become more rare allowing for better optimization rates.[1]

Stagnation Detection

Stagnation detection takes the opposite approach as the multiplicative-success, as it focuses on the number of non-successes that occur in a row. By comparing this number to

[1] [BB19]

the probability that a success should have occurred, the mechanism can alter its behavior to exit the local maximum that caused the stagnation. The JUMP_m optimization problem was created to test this kind of behavior. An example of this can be seen in [RW21]. This method can be used together with other mechanisms, but it might make calculating the occurrence of stagnation increasingly difficult.

3.1.3 Self-Adaptive Parameter Control

Self-adaptive parameter control uses endogenous parameter mechanisms as opposed to the previously introduced mechanisms, which uses exogenous parameter mechanism. Endogenous parameter control integrates the parameters into the individuals, so that an individual also carries information about the parameters by which it was conceived. As the better individuals are selected, the better parameters will also be forward propagated. Self-adaptive parameter control has some major advantages:

- No additional mechanism for parameter control has to be provided, besides from a representation of the parameters and the evolutionary operators for the parameters.
- It can be integrated into existing algorithms without any major changes.

This method was shown to have potential in [DWY18] and was further improved in [DWY21]. This field however, has not been widely explored in the literature.

3.2 Self-Adjusting Genetic Algorithms

This section will explore state of the art genetic algorithms that exploit some of the techniques mentioned in section 3.1.

3.2.1 $(1 + (\lambda, \lambda))$ $dyn(\alpha, \beta, \gamma, a, b)$ GA

The algorithm is a multiplicative-success based self-adjusting variation of the algorithm from subsection 2.5.3, introduced in [DDE15], and later refined in [DD19a]. The algorithm works the same, but instead of the parameters being set initially, they are set determined through a set of 5 hyper parameters and an internal value λ_0.

- a is the update strength of λ_0 when an improvement is found in the iteration.
- b is the update strength of λ_0 when no improvement is found in the iteration.
- α is used to increase the aggressiveness of the mutation.
- β is the rate of crossover to mutation.
- γ is used to determine the crossover bias.

The internal parameters are updated through an internal variable λ_0, which is adapted by multiplying it with growth rate a if the iteration did not find a better individual, or growth rate b if the iteration did find a better individual. The remaining parameters are used for the update rules for the internal parameters in relation to λ_0. The pseudo-code for the algorithm is seen below and the interpretation of the mutation and crossover

operators are :
Binary:

- mutation: *probability flip mutation*
- crossover: *biased crossover*

Permutation:

- mutation: *k-reverse mutation*
- crossover: *ordered crossover*

Algorithm 3 $(1 + (\lambda, \lambda))$ *dynamic* GA [DD19a]

Choose $x \in 0, 1^n$ *u.a.r.*
$\lambda_0 \leftarrow 1$.
for $t = 1, 2, 3...$ **do**
  $\lambda_1 \leftarrow \lfloor \lambda_0 \rceil$
  $\lambda_2 \leftarrow \lfloor \beta \cdot \lambda_0 \rceil$
  $p \leftarrow \frac{\alpha \cdot \lambda_0}{n}$
  $c \leftarrow \gamma / \lambda_0$
  Mutation phase:
  $z \sim \mathcal{B}(n, p)$
  for $i = 1, ..., \lambda_1$ **do**
    $x^{(i)} \leftarrow mutate_z(x)$
  end for
  Choose $x' \in \{x^{(1)}, ..., x^{(\lambda_1)}\}$ *with* $f(x') = max\{f(x^{(1)}), ..., f(x^{(\lambda_1)})\}$ *u.a.r.*
  Crossover phase:
  for $i = 1, ..., \lambda_2$ **do**
    $y^{(i)} \leftarrow cross_c(x, x')$
  end for
  Choose $y \in \{y^{(1)}, ..., y^{(\lambda_2)}\}$ *with* $f(y) = max\{f(y^{(1)}), ..., f(y^{(\lambda_2)})\}$ *u.a.r.*
  Selection and Parameter update phase:
  if $f(y) > f(x)$ **then**
    $x \leftarrow y$
    $\lambda_0 \leftarrow max(b \cdot \lambda_0, 1)$
  end if
  if $f(y) < f(x)$ **then**
    $\lambda_0 \leftarrow min(a \cdot \lambda_0, n - 1)$
  end if
end for

3.2.2 $(1 + 1)$ GA with Stagnation Detection

The algorithm is a version of the simple (1+1) GA(subsection 2.5.1), but instead of using global mutation, it uses local mutation, as converges a lot faster, but can get stuck in local optimums. To handle the issue of getting stuck the mechanism of stagnation detection is used. The algorithm is named $SD - RLS*$ by the authors, as it is based

on Randomized Local Search(RLS). The algorithm was introduced in [RW21], and was made to demonstrate how the mechanism of stagnation detection could be applied to most algorithms. The $*$ in the name is because it is a more robust version of their first iteration as it features a radius variable. The idea behind stagnation detection is to track the progress of the algorithm and intervene when the progress stagnates. With a local mutation operator such a *k-flip mutation* with $k = 1$, the algorithm stagnates when the changes from the current fitness to a better one requires a mutation step of size $k > 1$. The mechanism of stagnation detection tracks the progress and changes the strategy of the search by strategically trying larger and larger mutation step sizes until the fitness starts improving again. The algorithm has a parameter R which is used to determine when the mechanism shall activate and how many attempts at each value of k should be attempted based on the probability that an improvement should have happened if it was possible at the current k-value. The algorithm proceeds to $k+ = 1$ after $\binom{n}{s_t} \cdot \ln(R)$ iterations with no progress and thus systematically tests each value of k.

The pseudo-code for the algorithm is seen in Algorithm 4 and the interpretation of the mutation and crossover operators are :
Binary:

- mutation: *k-flip mutation*

Permutation:

- mutation: *k-reverse mutation*

Algorithm 4 $(1+1)$ GA with Stagnation Detection [RW21]

Choose $x \in \{0,1\}^n$ *u.a.r.*
$r_1 \leftarrow 1$
$s_1 \leftarrow 1$
$u \leftarrow 0$
for $t \leftarrow 1, 2, 3...$ **do**
  $y \leftarrow mutate_{s_t}(x)$
  $u \leftarrow u + 1$
  if $f(y) > f(x)$ **then**
    $x \leftarrow y$
    $s_{t+1} \leftarrow 1$
    $r_{t+1} \leftarrow 1$
    $u \leftarrow 0$
  else if $f(y) == f(x)$ **and** $r_t == 1$ **then**
    $x \leftarrow y$
  end if
  if $\binom{n}{s_t} \cdot \ln(R)$ **then**
    if $s_t == 1$ **then**
      if $r_t < n/2$ **then**
        $r_{t+1} \leftarrow r_t + 1$
      else
        $r_{t+1} \leftarrow n$
      end if
      $s_{t+1} \leftarrow r_{t+1}$
    else
      $r_{t+1} \leftarrow r_t$
      $s_{t+1} \leftarrow s_t - 1$
    end if
    $u \leftarrow 0$
  else
    $s_{t+1} \leftarrow s_t$
    $r_{t+1} \leftarrow r_t$
  end if
end for

3.2.3 $(1, \lambda)$ Self-Adaptive GA

The algorithm from [DWY18] and [DWY21]) builds upon the principles of the simple (1+1) GA(subsection 2.5.1), but instead of having a constant probability of mutation, it uses an endogenous mechanism for self-adaptive parameter control. It was build for binary encoding and uses the global *probability flip mutation* with an update strength of F. The algorithm either divides or multiplies the previous mutation step size with F, this allows the algorithm to quickly increase or decrease the aggressiveness of the mutation as it is necessary. The algorithm is designed to prefer lower mutation rates, as a high mutation rate becomes more and more likely to not result in any progress as it gets closer to the global optimum.

The algorithm takes 3 parameters:

- mutation rate λ, which is the number of offspring generated in each iteration.
- initial mutation probability r_0.
- update strength F, which is the evolution factor of the mutation probability.

The pseudo-code for the algorithm is seen in Algorithm 5 and the interpretation of the mutation and crossover operators are :
Binary:

- mutation: *probability flip mutation*
- crossover: *biased crossover*

Permutation:

- mutation: *k-reverse mutation*
- crossover: *ordered crossover*

Algorithm 5 $(1, \lambda)$ GA with adaptive mutation rate[DWY21]

Choose $x_0 \in \{0,1\}^n$ *u.a.r.*
for $t \leftarrow 1, 2, 3...$ **do**
  for $i \leftarrow 1, 2, 3, ..., \lambda$ **do**
    Choose $r_{t,i} \in \{r_{t-1}/F, F \cdot r_{r-1}\}$
    $x_{t,i} \leftarrow mutate x_{t-1}(x_{t-1})$
  end for
  Choose $i \in [1..\lambda]$ such that $f(x_t, i) = min_{j \in [1..\lambda]} f(x_t, j)$; in case of a tie, prefer an i with $r_{t,i} = r_{t-1/F}$; break remaining ties
  $(x_t, r_r) \leftarrow (x_{t,i}, r_{t,i})$
  Replace r_t with $min\{max\{F, r_t\}, F^{\lfloor \log_F(n/(2F)) \rfloor}\}$ randomly.
end for

4. Algorithm Design

This chapter explores the design of a genetic algorithm I have developed based upon the concept of Algorithm 5 resulting in a new algorithm with 2 variations. First the experiences leading to the algorithms will be introduced, then the actual algorithm idea will be explored.

4.1 Experiences

The modifications are a result of a series of observations I have made during the investigation of the performance, as documented in chapter 8. From these I have inferred some behavioral traits of different aspects of the algorithms.

4.1.1 Population

Population based algorithms are generally slow, but are very useful with some problems. The first advantage is that of exploring different areas instead of rapidly getting stuck in one. The other is being considering several sub-optimal solutions, which makes bridging jumps that require passing through an area of sub-optimal solutions easier, depending on the selection operator. This results in good performance on $\textsc{Jump}_m$ and 3-Sat. While being advantageous in some problems, in general multi population algorithms are a detriment to performance.

4.1.2 Crossover

Crossover based algorithms require several individuals and are thus often population based, which as mentioned is a detriment in regards to performance. The design by [DD19a] however lowered the population size to one, removing one of the harms of crossover based algorithms, but also removed the benefits of the population based algorithms. Crossover has thus not managed to add beneficial behavior to any of the problems considered in this thesis.

4.1.3 Mutation step size

For binary problems the use of global mutation is generally a lot slower than that of local mutation as the *k-flip mutation* is easier to control than the *probability flip mutation*, because the later can only be controlled by the probability, meaning that it can not be restricted to flip only a fixed number of bits. This makes it difficult for the *probability*

flip mutation-operator to optimize the individuals in the final stages of the algorithm. For permutation encodings there are not any viable way of generating global mutations that are not pseudo random, meaning that they are so far from the parent that they might as well have no relation. Instead the global mutation can be imitated through a larger mutation step size such as *k-reverse mutation* and the local mutation through a small mutation step size such as *k-swap mutation*. The idea behind global mutation is that it is impossible for the algorithm to become stuck at a sub-optimal peak, as it can always progress with some probability. Binary encodings local mutation as seen in [RW21] turned out to be extremely fast in solving most problems as mutational success was more probable. It was however unfit for solving problems where the mutation step became too small to bridge the gap, which is often the case with 3-SAT. Similar issues can occur with permutation problems, where the mutation is not able to make sufficient changes so that the algorithm exits the local optimum.

4.1.4 Self-adaptation

Self-adaptation proved to be a great method for improving the mutation optimization as it allowed the algorithm to converge faster as seen in Algorithm 5. The concept of having the self-adjusting mechanism be endogenous also made a lot of sense as it required less manual adjusting to the actual problem. Modifying the self-adaptive algorithm by [DWY21] to use local mutation instead of global mutation achieved some of the quick convergence benefits from Algorithm 4. It positively affected the performance on most problems significantly increased, at the cost performance on 3-SAT and JUMP_m.

4.2 The algorithm idea

These observations resulted in an idea for a new concept. An algorithm using only mutation for evolution, but instead of using only a single mutation operators, two is used. One local that allows rapid optimization, and one global/pseudo global to escape local optimums. By allowing the balancing of these through an extra hyper parameter λ_2, the algorithm could be set to a more aggressive search for a local optimum or a search more fit for a wicked landscape with many local optimums. To maximize performance on most problems the algorithm uses a single individual population. Utilizing the framework from Algorithm 5 with a local mutation; *k-flip mutation* for binary encodings and *k-reverse mutation* for permutation encodings. For the global mutation two different options was considered:

- Generating random solutions.
- Applying a global or more aggressive mutation operator that could reach further than the standard local mutation.

The first idea was to modify the algorithm to generates random solutions as a global evolutionary operator. The local and global mutation operators are thus:
Binary:

- local mutation: *k-flip mutation*

- global mutation: generate random solution

Permutation:

- local mutation: *k-reverse mutation*
- global mutation: generate random solution

Algorithm 6 $(1, (\lambda_1 + \lambda_2)$ GA with adaptive mutation rate and random additions(GABE1)

Choose $x_0 \in \{0,1\}^n$ *u.a.r.*
$r_0 \leftarrow r^{init}$
for $t \leftarrow 1, 2, 3...$ **do**
  for $i \leftarrow 1, 2, 3, ..., \lambda_1$ **do**
    Choose $r_{t,i} \in \{r_{t-1}/F, F \cdot r_{r-1}\}$
    $x_{t,i} \leftarrow mutatex_{t-1}(x_{t-1})$
  end for
  for $i \leftarrow \lambda_1 + 1, \lambda_1 + 2, ..., \lambda_1 + \lambda_2$, **do**
    $x_{t,i} \leftarrow generate\_random\_solution()$
  end for
  Choose $i \in [1..\lambda]$ such that $f(x_t, i) = min_{j \in [1..\lambda]} f(x_t, j)$; in case of a tie, prefer an i with $r_{t,i} = r_{t-1/F}$; break remaining ties
  $(x_t, r_r) \leftarrow (x_{t,i}, r_{t,i})$
  Replace r_t with $min\{max\{F, r_t\}, F^{\lfloor \log_F(n/(2F)) \rfloor}\}$ randomly.
end for

The algorithm takes 4 parameters:

- local mutation rate λ_1, which is the number of offspring generated through local mutation in each iteration.
- global mutation rate λ_2, which is the number of randomly generated solutions in each iteration.
- initial mutation probability r_0.
- update strength F, which is the evolution factor of the mutation probability.

To improve `GABE 1` the global mutation had to generate solutions that took advantage of the previous progress. The idea of sampling the mutation step size that was used in Algorithm 2 and Algorithm 3, seemed to allow for this mechanism by using the *probability flip mutation*(binary) and *k-reverse mutation*(permutation) operators. By sampling a binomial distribution with probability p(set by the user as a parameter) and the problem size n, the user could control the aggressiveness of the algorithm allowing anything from almost local mutation, to pseudo randomly generated samples from the population.

This could be done through the global binary operator *probability flip mutation* and The local and global mutation operators are thus:
Binary:

- local mutation: *k-flip mutation* with adaptive k.
- global mutation: *probability flip mutation* with sampled $prob = Bin(n, p)/n$.

Permutation:

- local mutation: *k-reverse mutation* with adaptive k.
- global mutation: *k-reverse mutation* with sampled $k = Bin(n, p)$.

Algorithm 7 $(1, (\lambda_1 + \lambda_2))$ GA with adaptive mutation rate and aggressive mutation(GABE2)

Choose $x_0 \in \{0,1\}^n$ *u.a.r.*
$r_0 \leftarrow r^{init}$
for $t \leftarrow 1, 2, 3...$ **do**
  for $i \leftarrow 1, 2, 3, ..., \lambda_1$ **do**
    Choose $r_{t,i} \in \{r_{t-1}/F, F \cdot r_{r-1}\}$
    $x_{t,i} \leftarrow local\_mutate x_{t-1}(x_{t-1})$
  end for
  for $i \leftarrow \lambda_1 + 1, \lambda_1 + 2, ..., \lambda_1 + \lambda_2$, **do**
    $z \sim \mathcal{B}(n, p)$
    $x_{t,i} \leftarrow global\_mutate_z(x_{t-1})$
  end for
  Choose $i \in [1..\lambda]$ such that $f(x_t, i) = min_{j \in [1..\lambda]} f(x_t, j)$; in case of a tie, prefer an i with $r_{t,i} = r_{t-1/F}$; break remaining ties
  $(x_t, r_r) \leftarrow (x_{t,i}, r_{t,i})$
  Replace r_t with $min\{max\{F, r_t\}, F^{\lfloor \log_F(n/(2F)) \rfloor}\}$ randomly.
end for

This version takes an additional parameter p which is used for the binomial sample, thus having 5 parameters.

- local mutation rate λ_1, which is the number of offspring generated through local mutation in each iteration.
- global mutation rate λ_2, which is the number of offspring generated through global mutation in each iteration.
- success probability p used to sample mutation step size from binomial distribution.
- initial mutation probability r_0.
- update strength F, which is the evolution factor of the mutation probability.

5. Framework Design

The purpose of this thesis is to implement and compare the empirical performance of genetic algorithms on different optimization problems. There are three levels of empirical investigations, of which the highest is the focus of this thesis:

- The lowest level is the internal behavior of an algorithm. This level of investigation is mainly the debugging and understanding of how internal values change over the iterations of a search.
- The next level is the performance of an algorithm depending on its parameter setting. This is used for investigating the optimal parameter settings for each algorithm, on each problem.
- The highest level is the comparison of different genetic algorithms with fixed parameters for each. For the comparison to be most useful the work in the previous level is quite significant, as it can alter the performance significantly and will thus impact the analysis on this level.

Although the highest level is the focus of this thesis, it depends on the other level and thus should contain functionality to support these. Thus we need a software framework that support running different genetic algorithms with different parameter settings on a set of different problems. As these should not be limited to the ones mentioned in this thesis, the the software should be effortlessly extendable in regards to both algorithms and problems. The software is modelled after the Model-View-Controller software design pattern.

5.1 Model-View-Controller

The Model-View-Controller design pattern is about separation of program logic into 3 main purposes. Model is the central component of the program, the purpose of it being the definition of logic and rules of the program, meaning it has to be a model of the problem. The purpose of the View is to give the users information about the run of the program, and is thus used as a representation of the model. The purpose of the controller is for the user to be able to interact with the program, and thus the model. The model, view and controller is connected so that input through the controller affects the model, which changes the view.

The design challenge for this piece of software was to make a model that encapsulated

genetic algorithms, and to still have the software be simple to understand and use.

The potential users of the software can be split into two categories, based on how they intend to use the program:

- those who intend to use the software as a black-box and investigate currently implemented algorithms and problems.
- those who have interest in improving the codebase with further algorithms, problems or operators, or optimizing to current algorithms. It is assumed that these users also are competent within software development and are familiar with common design patterns that makes the code easy to understand.

The experience of the first type of user requires good implementations of the view and controller components, while the experience of the second type also requires a good and well-defined model that is easy to understand and extend.

5.1.1 Model

The idea of a model is to represent the elements and their interaction at the level of abstraction makes them useful and removes all unrelated information. This means that the important elements of genetic algorithms and their interaction has to be abstracted into code. To make the design easily comprehensible I will make use of a number of design principles of software design. The principles include the 4 principles of object-oriented programming[1] and the SOLID[2] principles by Robert C. Martin.

The interacting objects of genetic algorithms can be boiled down to algorithms, problems, solutions and evolutionary operators. The model defines the structure and relation among these entities and should follow the single-responsibility principle to maintain a proper encapsulation of entities.

Problems

The different problems can be considered to belong to the class of problems. A problem is defined by its search space and its fitness function. The search space is defined through the randomly generated solutions and the evolutionary operators that define which solutions can be explored through evolution of the randomly generated solutions. As the evolutionary operators are used to define the search space, an algorithm is limited to those operators that generate valid solutions, but as there are several of those the algorithm should be free to use whichever fits the algorithm strategy, and the operators can therefore not be defined in the scope of a problem. Instead the problem contains information about which encoding it uses and thus which operators are applicable. This allows algorithms to pick evolutionary operators within the same encoding schema. The fitness function defines the objective of the problem and is thus inherently important to the problem. This means we can consider the problem an object-class in itself, which can be generated through and constructor that takes a problem instance and an encoding

[1] Encapsulation, Abstraction, Inheritance and Polymorphism

[2] Single-Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface-Segregation Principle and Dependency Inversion Principle

type. The class must also define a callable fitness method that maps a solution to a fitness score, along with a method to generate random solutions. This is made into a interface that describes the specifications of a problem, so that users can add problems that can be integrated easily, as long as they follow the interface specifications.

The problem instances for problems where there is only one problem instance for every problem size $(n, (m))$(OneMax, LeadingOnes, Jump_m and Sorting) require no generation as it is inferred from the problem size. However both 3-Sat and TravelingSalesman have different instances for each problem size, which requires some form of problem instance generation. This means that problem instances have to be either imported or generated. These can be imported from SATLIB[SAT00] and TSPLIB[TSP95]. The problems from both of these resources have significant gaps in their problem sizes and the problem difficulty varies a lot. This makes the results hard to intepret, as some smaller problems are significantly hard than the larger ones. An alternative is to implement functionality to create random problem instances of a certain problem size. This, as will be explored, however also has issues for some problems. One is the issue of replicability, which is due to the problem instances being generated internally, which makes them different across every run.

3-Sat For 3-Sat it is quite simple to generate fully solvable cases of 3-Sat a random setting of m variables is generated. Then n clauses are generated by sampling 3 literals from the random solution and negating 2 of them. This secures that there is at least one valid optimal solution to the problem, but also results in there being a significant number of optimal solutions, which makes the problems quite a lot easy to solve. This is easily solved by using large problem sizes to test algorithms, and is thus preferred to the problems from TSPLIB.

TSP For TravelingSalesman it is also quite simple to create solutions that are fully solvable. The problem instances can simply be generated by creating n random locations in 2 dimension. The issue is determining the optimum solution. This can either be solved by having a preparation run that determines the goal, or to have the goal be high enough that the algorithm wont stop until the time runs out, but will certainly not stop before the optimal is reached.

Solutions

The solutions to problems can be represented with a solution-object, that consists of a chromosome describing the solution and a fitness score. The chromosome size and structure can vary depending on the problem. The solution is interpreted through the problem class and does not have information about what problem it belongs to.

Algorithms

The class of genetic algorithm consist of an algorithm which needs two inputs to a constructor method; a problem-object and a set of parameters. The problem is given as an object from the problem class and the parameters can be packed into a dictionary so that the input is universal to all algorithm implementations. We thus consider a genetic

algorithm an object-class in itself, which can be initiated through a constructor call with two inputs; a problem and a set of parameters. Besides from this a callable method to start the run of the algorithm is required.

This is made into an interface that describes the specifications of a genetic algorithm, so that users can add new algorithms that can be integrated easily, as long as they follow the interface specifications.

As the encoding method of the problem defines the solution space, the algorithms must have a set of operators for each problem encoding, so that upon reading the encoding type, the algorithm can determine which set of evolutionary operators should be used to evolve the solutions. This allows algorithm specific operators, that can still be overwritten if necessary.

The parameter input consists of two groups. They are either global or algorithm specific. Global parameters determine settings that are universal to all algorithms, such as specify the stop criterion's. The local parameters specify settings that are unique to the algorithm configurations. The stop criterion's consist of a goal and time limit, and the algorithm will stop whenever one of these are reached. Sometimes the optimum is not known and goal setting is thus not possible. This is handled by setting an unreachable goal and relying on the time limit to terminate the algorithm.

The return value from the algorithm should contain the results from the run, consisting of several useful pieces of information, such as the time spent, the optimal solution found and the number of solutions explored. This information is collected into a result-class.

Evolutionary Operators

Evolutionary operators have already been discussed since they are related to both problems and algorithms. Both mutation and crossover operators are largely replaceable in the algorithms. This means that the program should allow overriding mutation or crossover operators of algorithms, allowing users to test out different configurations within the encoding schema. It is however worth to mention that using operators for a different encoding will break the algorithms. A smaller issue is that some algorithms use operators that have a variable option that affects the behavior, such as the number of bits to flip i *k-flip mutation* or the bias in *biased crossover*. This means that all operators have an input variable *option* that allows this value. The *option* input has a default, so that algorithms the don't take advantage of this can still use the operators. However some operators do not support this behavior and thus the option does nothing. This means that algorithms where this *option*-value is used, for the behavior of the self-adjusting, will work, but without the effect that it was intended to have.

5.1.2 Controller

The main purpose of the software is performance investigation and evaluation of algorithms. The interface should therefore be optimized towards this, meaning that a graphical user interface is not ideal as it would require a huge and complicated interface achieve the same level of seamless automatization that could be achieved with a terminal interface. Additionally it would be incompatible with any computational system that

does not have a graphical interface, such as the system used to run the tests for this thesis. Therefore the choice of interface is to have the software run through terminal interface, which run on anything and allows test scripts for running a battery of tests with ease.

```
python3 framework/src/main.py --help
framework/src/main.py --help
usage: main.py [-h] [--problem PROBLEM] [--size SIZE] [--m M]
               [--problem_file PROBLEM_FILE] [--goal GOAL]
               [--algorithm ALGORITHM] [--sample_size SAMPLE_SIZE]
               [--time_limit TIME_LIMIT] [--internal_log INTERNAL_LOG]
               [--random_seed RANDOM_SEED]
               [--mutation_operator MUTATION_OPERATOR]
               [--mutation_default_value MUTATION_DEFAULT_VALUE]
               [--crossover_operator CROSSOVER_OPERATOR]
               [--GA_standard_lambda GA_STANDARD_LAMBDA]
               [--GA_standard_pc GA_STANDARD_PC]
               [--GA_standard_pm GA_STANDARD_PM]
               [--GA_static_lambda1 GA_STATIC_LAMBDA1]
               [--GA_static_lambda2 GA_STATIC_LAMBDA2]
               [--GA_static_k GA_STATIC_K] [--GA_static_c GA_STATIC_C]
               [--GA_dynamic_alpha GA_DYNAMIC_ALPHA]
               [--GA_dynamic_beta GA_DYNAMIC_BETA]
               [--GA_dynamic_gamma GA_DYNAMIC_GAMMA]
               [--GA_dynamic_a GA_DYNAMIC_A] [--GA_dynamic_b GA_DYNAMIC_B]
               [--SD_RLS_R SD_RLS_R]
               [--GA_Adaptive_Mutation_lambda GA_ADAPTIVE_MUTATION_LAMBDA]
               [--GA_Adaptive_Mutation_F GA_ADAPTIVE_MUTATION_F]
               [--GA_Adaptive_Mutation_r0 GA_ADAPTIVE_MUTATION_R0]
               [--GABE1_lambda GABE1_LAMBDA] [--GABE1_lambda2 GABE1_LAMBDA2]
               [--GABE1_F GABE1_F] [--GABE1_r0 GABE1_R0]
               [--GABE2_lambda GABE2_LAMBDA] [--GABE2_lambda2 GABE2_LAMBDA2]
               [--GABE2_F GABE2_F] [--GABE2_p GABE2_P] [--GABE2_r0 GABE2_R0]
```

Figure 5.1: Run configurations

All of the algorithms have default parameter values, but they can all be changed through the interface with the commands seen in Figure 5.1

The program is designed so that only one problem at a certain size can be run at a time, but with several different algorithm for several samples. This is due to the complication of having different algorithm parameters for different problems simultaneously. In section 7.1 we will look into how `bash`-scripts can be used to automate this, so that several problem sizes can easily be run.

Below is example of how to compare the algorithms from subsection 3.2.3 and subsection 3.2.2 on the OneMax problem for 15 samples. The size of the problem is 500 and with the goal set to the optimum for the problem size(500) and limited by a running time of 15 seconds.

```
$ python3 main.py --problem OneMax --size 500 --goal 500 --time_limit
    15 --algorithm ["GAAdaptiveMut,SD_RLS"] --sample_size 15
```

It is not possible to run the same algorithm with different parameters for only some of the samples of a run, as it is overwritten in all locations.

5.1.3 View

The previous limitation in regards to a graphical combination along with the fact that the output of the program is a collection of data, means that the view is instead handled externally. As the algorithm runs the view, through the terminal, shows the progress of

the computations and the results so that one can follow and verify that everything is running as it is supposed to.

Besides from this the program exports the result of each algorithm into an external csv-file containing the runs for all algorithms run on that problem class. The main log method is intended for the middle and highest level of comparison. It is split into a different file for each problem and contains information such as the problem size, algorithm, the best fitness found, the time spent and the parameters. This means that the same algorithm with different parameters on different problem sizes, or that different algorithms can be compared, as we will look at in chapter 8.

The lowest level can be investigated through an optional internal log, which saves the internal variables of each iteration of the algorithm, along with the input parameters. This allows a view into the workings of the algorithm, and how different parameter settings affect the internal variables and affect the search.

6. Implementation

The implementation of the framework for comparison of genetic algorithms is implemented in **Python3**. The reason for the use of python is that it is simple to download and run on any system, and more importantly it is a simple language that is easy to understand, which makes further development easier. Another major reason is that python applications tend to be significantly shorter. Python does have some downsides in regards to performance due to high-level data types and dynamic typing, but this is not considered of importance as the performance is intended for comparison with other algorithms on the same framework, and not for comparison with implementations in other languages. These advantages make it a good pick for a piece of software that is intended for further development by other users.

This chapter will delve into the technical aspects of the code such as the program structure (section 6.1), which goes into the packages and how they relate to each other and are combined into the framework.

6.1 Framework Implementation

The codebase consists of 3 packages; a **problems**-package, a **algorithms**-package and a **common**-package. Besides from these is the `main.py` file that contains the interface that connects everything into a usable piece of software.

6.1.1 common

The **common**-package contains a series of objects and functions that are used in several places and can not be limited to one of the other packages and do not justify their own package.

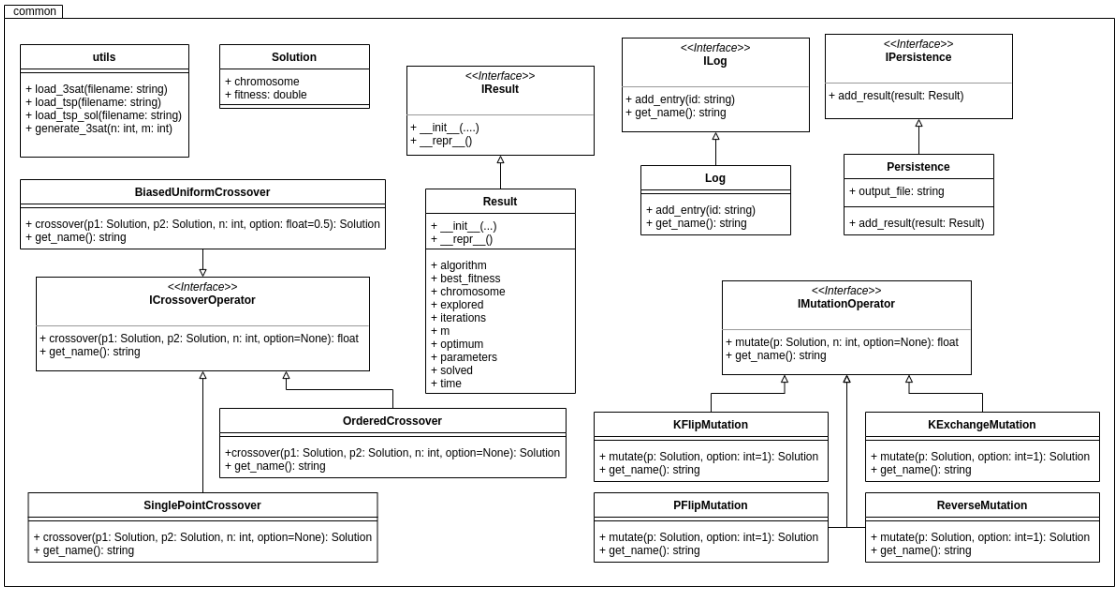


Figure 6.1: **common**-package

- `__init__.py` contains the interfaces for the other classes in the package.
- `solution.py` contains the structure and functionality of a solution.
- `mutation_operators.py` contains the different mutation operators.
- `crossover_operators.py` contains the different crossover operators.
- `log.py` provides an object for internal investigation of algorithms.
- `results.py` contain an result object which is used to collect the data of completed algorithm runs.
- `persistence.py` is used to export results to a `csv`-file.
- `utils.py` contains methods for loading 3-SAT problems and TRAVELINGSALESMAN problems and solutions.

Evolutionary operators

Evolutionary operators can be added in their respective classes by implementing the interface of the respective operator. Additionally the operator should be added to either the `return_mutation_operator` or `return_crossover_operator`-method, so that it can be used through the terminal by its operator name.

The operators support an additional argument through the *option* parameter which is defaulted to some standard parameter. The mutation operators all use some integer as the additional parameter as this prevents any changes in algorithms based on operator choice. The crossover operators do support having an *option*, which can be used to determine a bias by having some float from 0 to 1 determine how much each parent should influence the crossover. This however is rarely supported, so for most of the crossover

operators this *option* does not affect the crossover. However the support for adding bias to crossover operators is supported and would most likely improve the algorithms that make use of this feature.

6.1.2 problems

The **problems**-package contains an `__init__.py`-file with the interface for a problem and 6 problem definitions. The currently implemented problems are: OneMax, LeadingOnes, Jump$_m$, 3-Sat, Sorting and TravelingSalesman. Additional problems can be added to this file and will then be usable through the interface. The exception to this is problems that are loaded externally, which will require changes to the *main.py* file and a method for loading the problem instance.

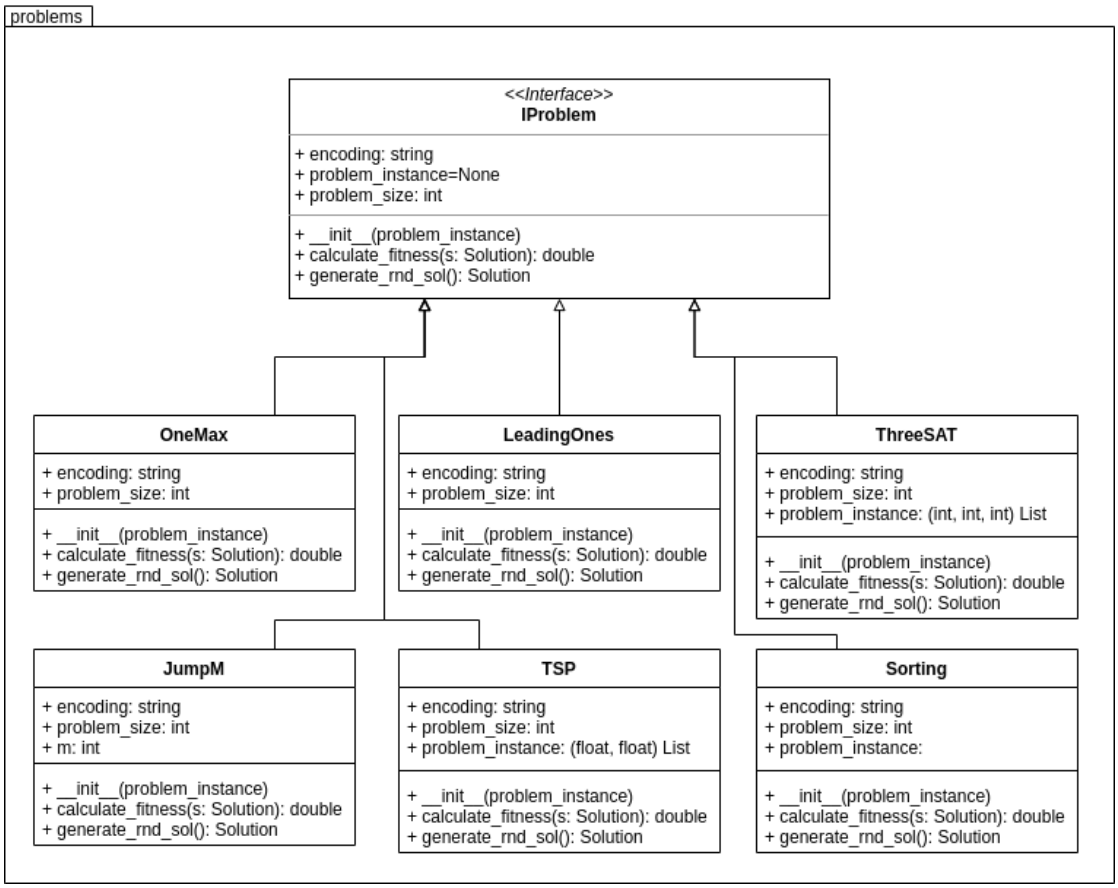


Figure 6.2: **problems**-package

6.1.3 algorithms

The **algorithms**-package contains an `__init__.py`-file with the interface for genetic algorithms and two files for implementations of genetic algorithms, one for non self-adjusting and one for self-adjusting algorithms. The algorithms implemented currently are:

- GAStandard - Algorithm 1([Deb99])
- GAStatic - Algorithm 2([DD19a])
- GADynamic - Algorithm 3([DD19a])
- SD_RLS - Algorithm 4([RW21])
- GAAdaptiveMut - Algorithm 5([DWY21])
- GABE1 - Algorithm 6
- GABE2 - Algorithm 7

The implementation of each algorithm can be seen in the attached code.

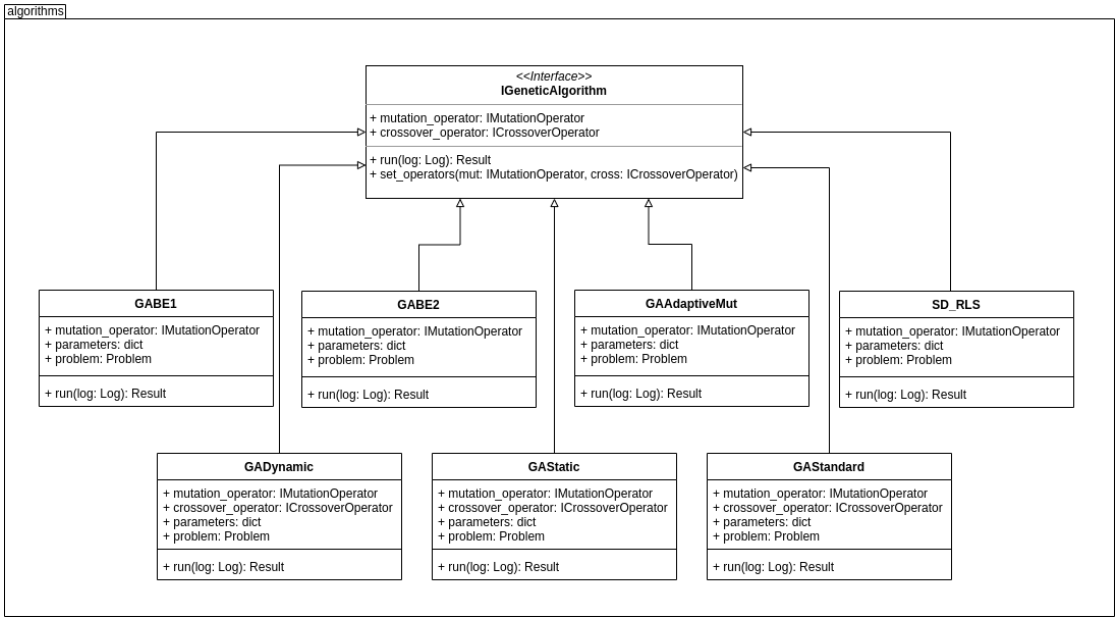


Figure 6.3: **algorithms**-package

7. Testing

To test the usefulness of the framework the algorithms are run on each of the problems with increasingly large problem instances. This allows investigation into the efficiency of each algorithm. This chapter focuses on the comparative performance of algorithms and not the optimal performance of each algorithm on a problem. This means that the parameters of each algorithm is not optimized for the problem in any way, as this is considered out of scope. As this is an essential part of investigating genetic algorithms the framework does contain functionality, which is not explored in this thesis, but can be used to optimize the parameters for each problem. As comparative testing is the focus of the thesis, this chapter will look into the test scripts (7.1), the computational specifications of the test system and the measure of performance (7.3).

7.1 Test scripts

As there is no functionality to run several problem sizes simultaneously it is useful to be able to create a small script that runs the program repeatedly for several, often growing, problem sizes to see how the algorithm performs in comparison to the problem size. An effective way is to do this with a bash script, which allows serialization of terminal commands. The script obviously have to be modified depending on whether the problems are loaded from a set of files or just generated from a problem size n. For LeadingOnes we just need to iterate over the size of n as seen below.

```
#!/bin/bash

for n in 10 30 50 70 90 100 120 140 160 180 200 220 240 250
do
  python3 framework/src/main.py --problem LeadingOnes --size $n --goal
      $n --time_limit 15 --sample_size 15 --algorithm ["GABE1,GABE2,
      GAAdaptiveMut,SD_RLS,GAStatic,GADynamic,GAStandard"] --SD_RLS_R $
      ((n+1))
done
```

We also need to include the value R for `SD_RLS* GA`, as it depends on the size of the problem n.

For 3-Sat and TravelingSalesman problems the program can both be run with generated problem instances or they can be loaded from already existing problems. The

pregenerated problems come from from SATLIB[1] and TSPLIB [2]. The script for running a set of those problems is seen below.

```
#!/bin/bash

for problem_instance in $(ls problems/3sat)
do
    python3 framework/src/main.py --problem ThreeSAT --problem_file
        $problem_instance --time_limit 60  --sample_size 2 --algorithm ["
        GABE1,GABE2,GAAdaptiveMut,SD_RLS,GAStatic,GADynamic,GAStandard"]
        --SD_RLS_R $(wc -l problems/3sat/$problem_instance | head -n1 |
        cut -d " " -f1)
done
```

For the 3-Sat problem the goal is inferred from the number of clauses, but for TravelingSalesman the goal must be loaded through the optimal solution to the problem, also found on TSPLIB. For generated TravelingSalesman instances the goal is set to 0 as the actual optimum is not known. This results in larger computation times as the goal cannot be reached. There is some theory into average optimal distance for randomly generated TravelingSalesman instances, but that has been considered out of scope for this thesis.

7.2 Computational specifications

All tests were run on DTU Compute's HPC servers with a Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz CPU.

7.3 Measuring performance

Genetic algorithms are driven by randomization. This means that theoretical performance estimations are very complicated and has to be repeated for different problems as it is based on the probability of an improvement. This obviously varies depending on the problem and thus have to be changed depending on the problem. This probability also changes based on the operators used and a change in operators will therefore invalidate the theoretical estimates. For self-adjusting algorithm this is even more complex, as the current operators behavior can depend on the previous iterations, making theoretical inferences about behavior very difficult.

Instead of relying on theoretical performance estimations the performance can be inferred from benchmark problems to are similar to the actual problem or on the actual problem.

To evaluate the algorithms some measures of performance that demonstrates how the algorithm performs as the problem size increases. The two relevant measures are time complexity, which describes the number of solutions in the search space that was investigated and running time, which describes the time it takes the algorithm to terminate either by finding the optimum solution or hitting the stop criterion. Time complexity

[1] https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html
[2] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

is useful when comparing algorithms across different systems and implementations, as the time complexity provides a useful heuristic for the actual running time, which differs across systems and implementations. However as the algorithms are all implemented and run in the same framework it is also possible to compare the actual running time. This is useful as the time complexity does not take into consideration the time spent on finding each solution, which can result in an algorithm with a low time complexity to actually perform significantly worse in regards to running time.

7.4 Data Processing

The data processing is done separately from the framework in a small independent **Python3** program. The processing is done using **pandas**[3] for data management and both **matplotlib**[4] and **seaborn**[5] for visualizations and plots.

The data processing package consists of two files. A `plots.py` and a `make_plots.py`. The `plots.py`-file contains several functions for plotting results;

- `col_vs_problem_size` for plotting some variable in the **pandas** DataFrame on the y-axis against the problem size on the x-axis. This allows performance comparison for both time complexity and running time, but only for runs that reach the optimum score, thus providing an useful, but incomplete picture of the results. The plots include a 95% confidence interval for each algorithm.
- `mean_score` for plotting the average difference from the optimum score for each algorithm on each problem size, which allows the interpretation on how the algorithms the algorithms perform on different problem sizes, even when they do not reach the optimum.
- `mean_score_tsp` for plotting the average difference from the optimum score found for each algorithm on each problem size, which allows the interpretation on how the algorithms the algorithms perform on different problem sizes, even when an actual optimum value is not known.
- `plot_problem` takes a problem and generates the relevant plots. It is used to create all the plots in a single function call.

These plots give a detailed view of the performance of each algorithm on each problem, as we will see in the following chapter.

[3] https://pandas.pydata.org/
[4] https://matplotlib.org/
[5] https://seaborn.pydata.org/

8. Results

This chapter contains the results of the algorithms on each different optimization problem. The parameter configuration for each algorithm on each problem is found at A.1. The results of all the tests are included in the accompanying zip-file along with all the plots in this thesis.

8.1 OneMax

OneMax is a unimodal problem where all genes independently affect the fitness score. This results in a problem that is easily solved and thus provides a neat entry point benchmark that tests how well each algorithm adapts to simple environment.

To compare the set of algorithms, the running time and computational complexity is investigated in Figure 8.1 and Figure 8.2. The algorithms were run with a time limit of 30 seconds with a sample size of 15, regardless of the problem size.

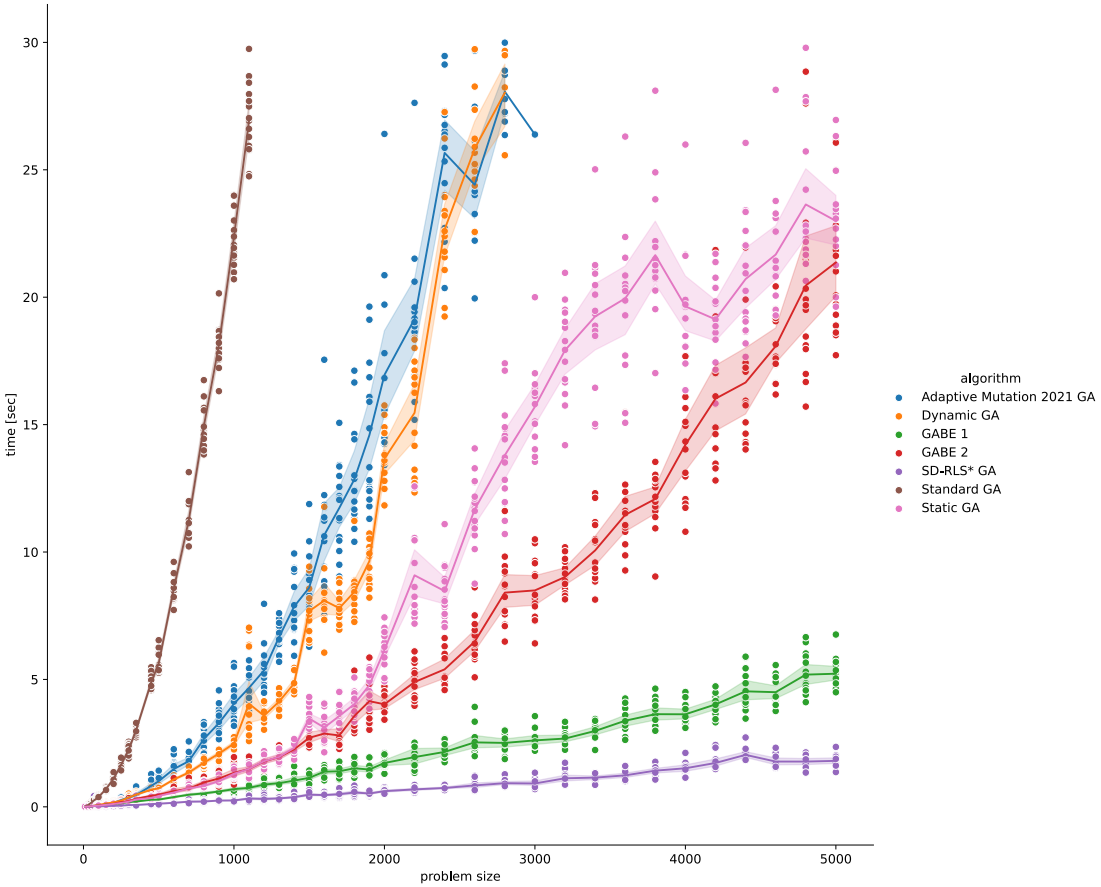


Figure 8.1: OneMax Running Time

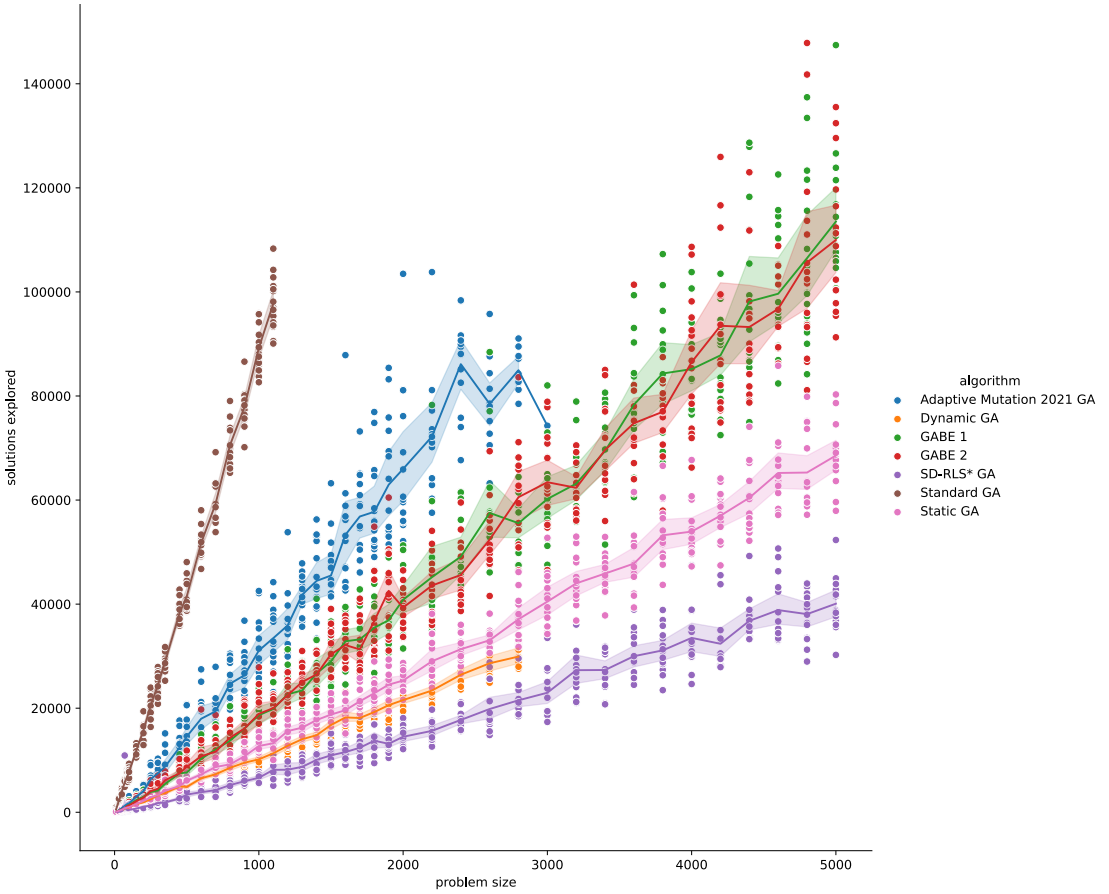


Figure 8.2: OneMax Computational Complexity

From these plots it is seen how mechanisms, such as multi-populations and global mutation, that are intended to navigate more complex environments are a big detriment on the OneMax problem. An example of this is `Static GA`, which regardless of having a low computational complexity has a very high running time. This hints toward a too costly method of determining the search direction.

The algorithm that performs the best `SD_RLS* GA`, which benefits from utilizing the speed of *k-flip mutation* local mutation with no overhead from other strategies as stagnation is very unlikely to be registered.

8.2 LeadingOnes

LeadingOnes is a unimodal problem where all the genes interdependently affect the fitness score. This results in a problem that is slightly harder, as the algorithm always have to hit the exact bit that results in an improvement.

To compare the algorithms the running time and computational complexity is investigated in Figure 8.3 and Figure 8.4. The algorithms were run with a time limit of 15 seconds

with a sample size of 15, regardless of the problem size.

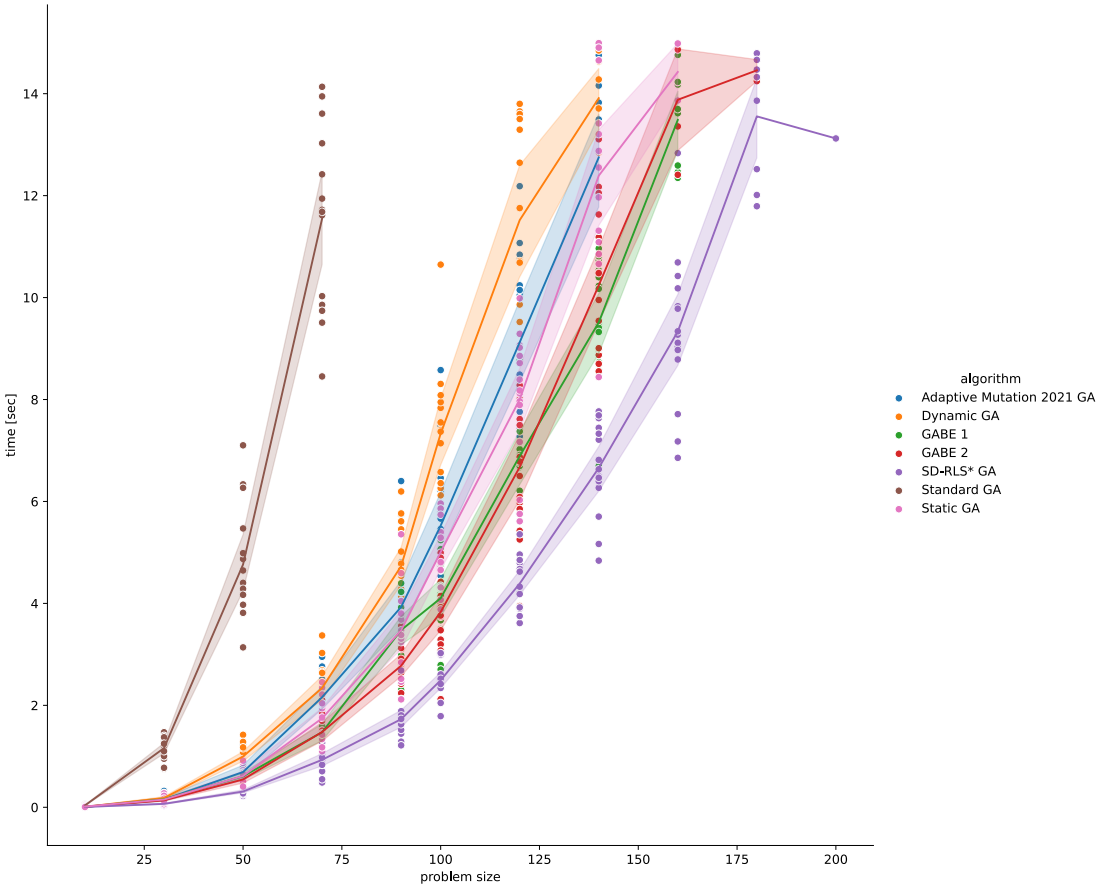


Figure 8.3: LeadingOnes Running Time

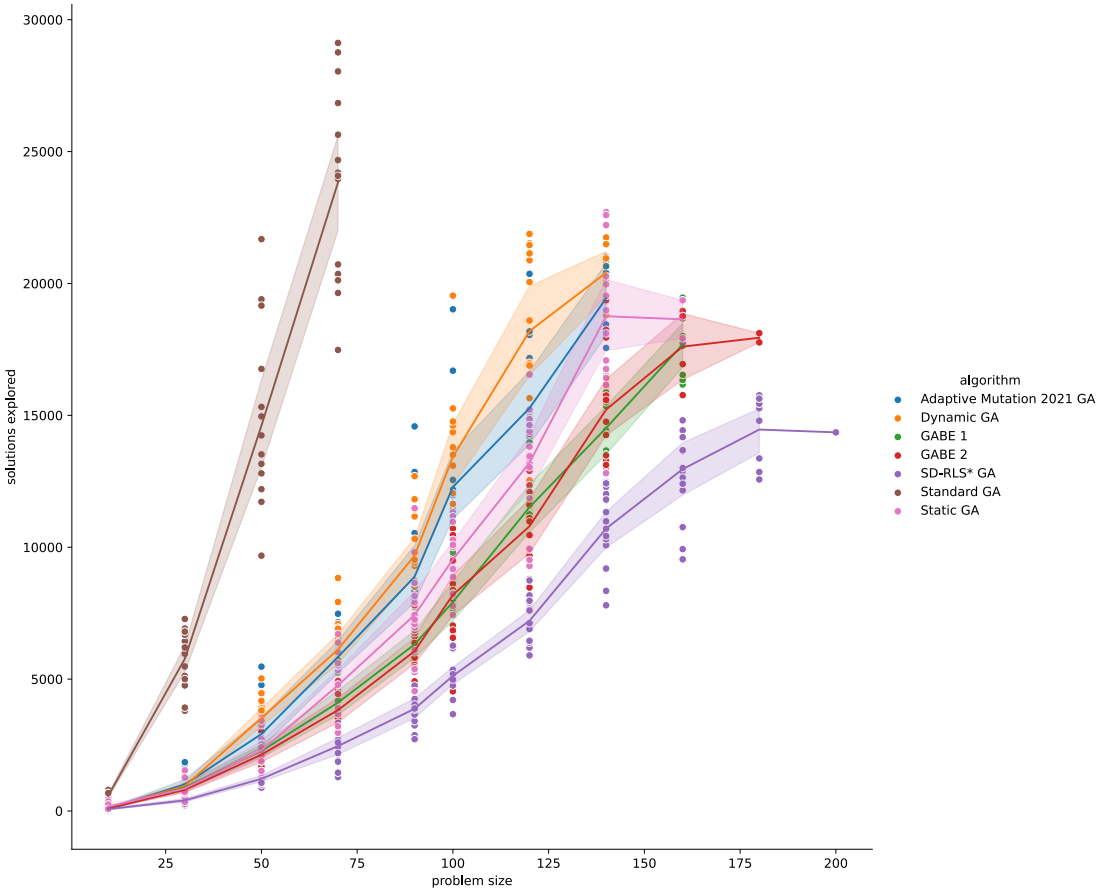


Figure 8.4: LeadingOnes Computational Complexity

The results are quite similar to OneMax. It is interesting to notice that as the problem is a bit more complicated the differences in performance between algorithms become smaller. The best algorithms are still those that utilize local mutation with a single individual in the population.

While OneMax and LeadingOnes problems are fine for benchmarking hill climbing behavior, they do not test if the algorithms are able to find a global optimum when there are local optimums. They therefore model very simple problems and the performance on these problems are not necessarily indicative of other problems.

8.3 JumpM

Jump$_m$ is a bimodal problem that resembles OneMax but where a gap of size m exists between the local and global maximum. This tests if algorithms are capable of exiting local maximums of size m, when the fitness of the gap is the lowest scores possible. Unlike the previous problems, where an improvement in the individual either didn't change or caused an improvement in the fitness score, Jump$_m$ has a negative fitness response to an improvement in the late stage of the optimisation. This means that for algorithms with

only one individual in the population all of the missing 0 bits have to be flipped at the same time, which is very unlikely especially for large m and n values.

To compare the algorithms the running time and computational complexity is investigated in Figure 8.5 and Figure 8.6. The algorithms were run with a time limit of 30 seconds with a sample size of 15, regardless of the problem size.

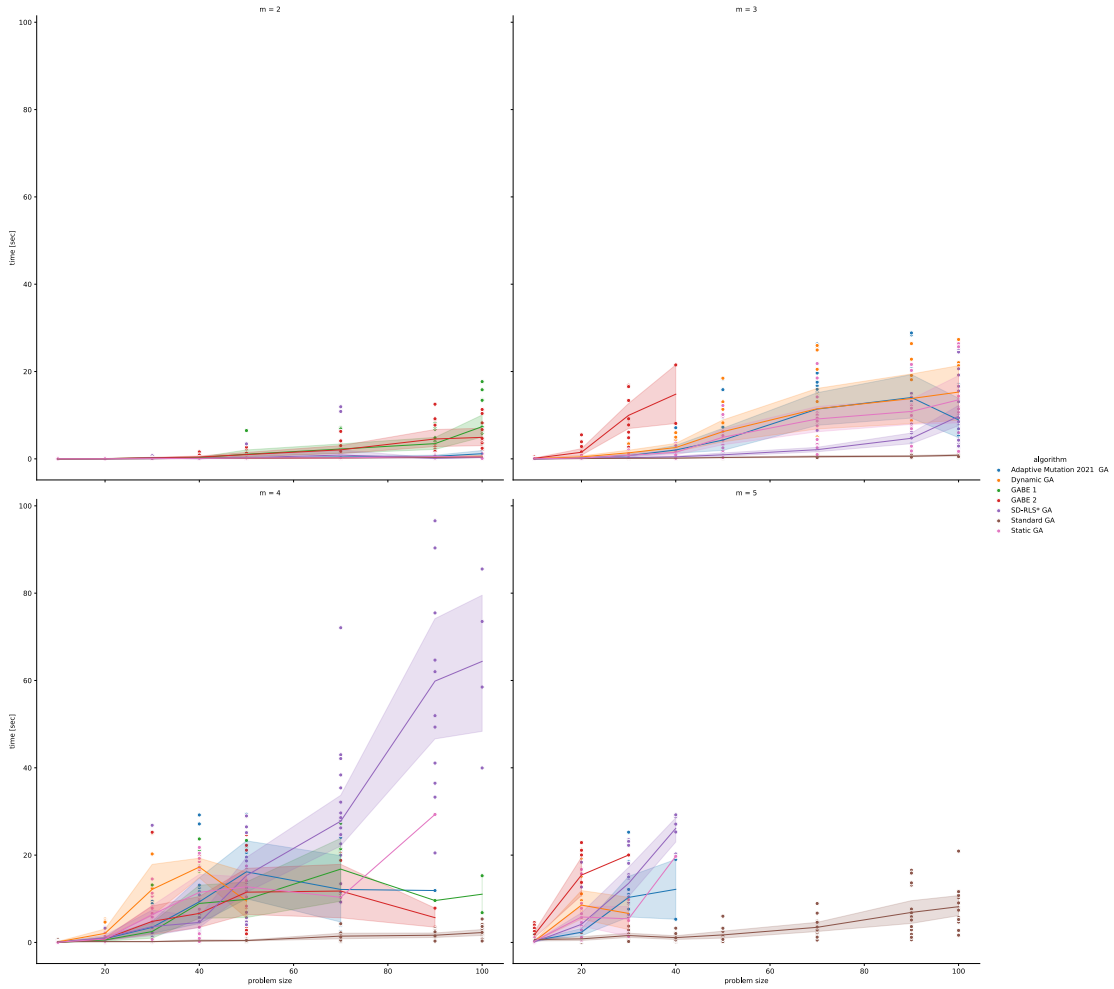


Figure 8.5: $\textsc{Jump}_m$ Running Time

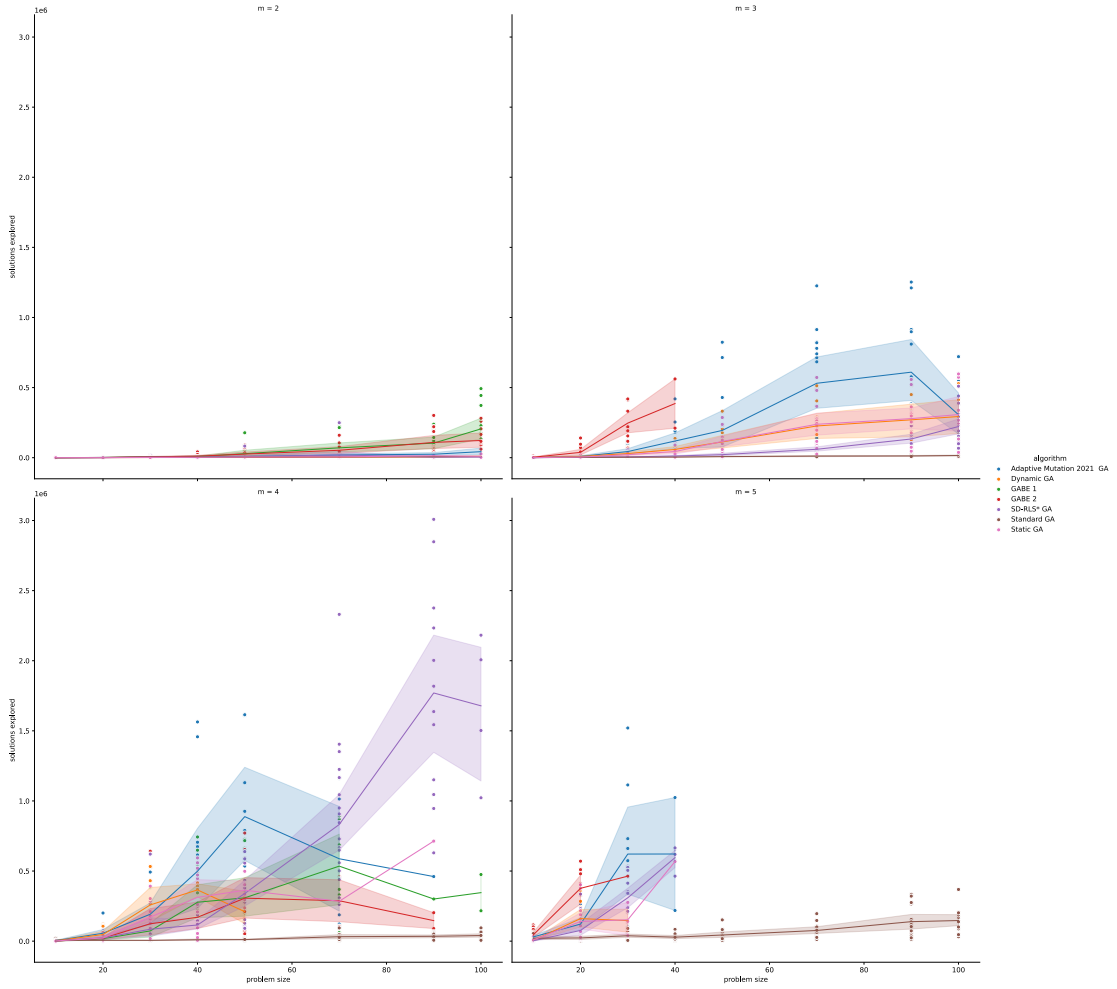


Figure 8.6: JUMP$_m$ Computational Complexity

The results show how `SD_RLS* GA` still performs great, but that `Standard GA` which previously performed poorly, has the best performance on JUMP$_m$ problems. This demonstrates the importance of choosing the right type of algorithm for the problem. It is worth mentioning `SD_RLS* GA` is designed for this problem and is likely to be the only algorithm that can solve problems with a growing m. It is however quite time consuming as m grows and it is therefore not able to solve the largest instances within the time limit. This is unlike the other algorithms, which would likely never solve the problem if the m was just a bit larger. Both `GABE 1` and `GABE 2` performs quite well on $m = 2$ and $m = 4$, but way worse on $m = 3$ and $m = 5$. This is due to the adaption of the k in its *k-flip mutation* which grows with a factor $K = 2$, which means that the k will never be 3 or 5 and it is thus impossible for it to jump the gap with the main mutation operator.

The first figures consider only successful runs, and as many runs fail it is valuable to look at the average score of each algorithm for each problem size.

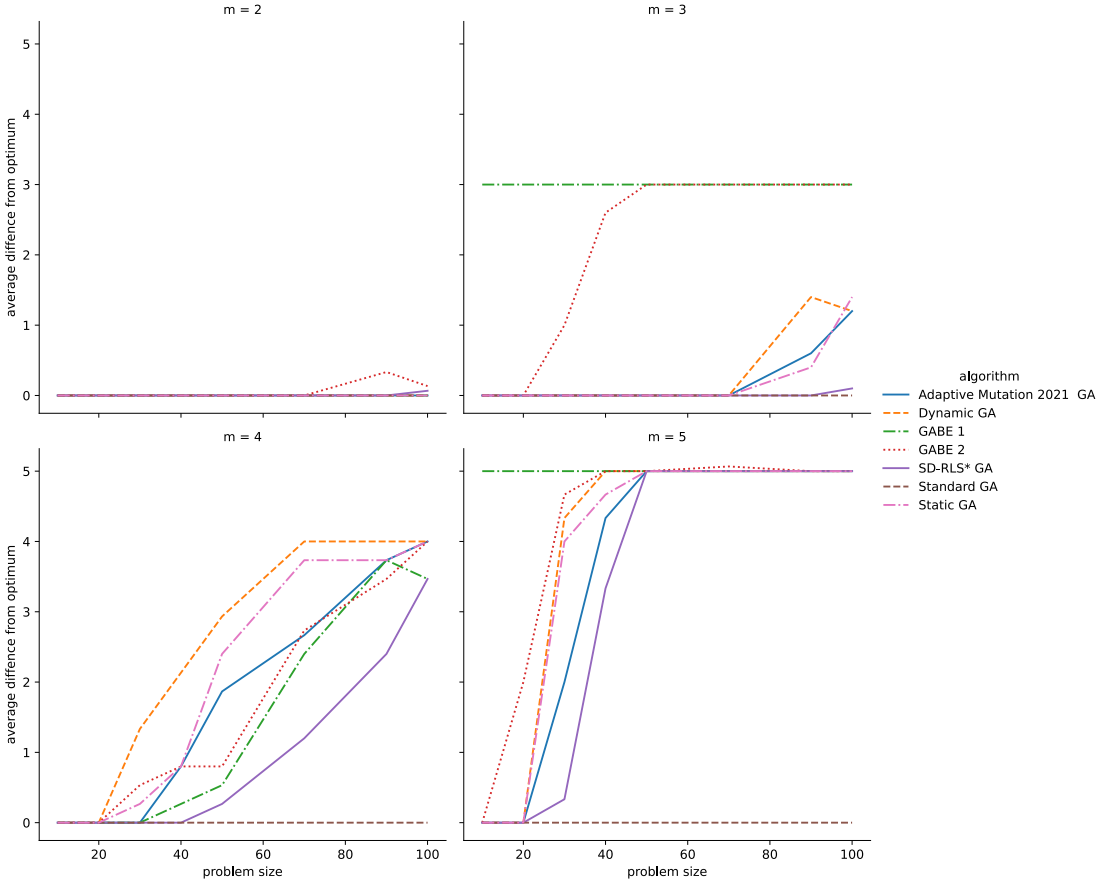


Figure 8.7: Average distance from optimum solution

From Figure 8.7 it is seen how the growth in problem size causes more samples to be unsuccessful. The success of `SD_RLS* GA` demonstrates the effectiveness of local mutation and stagnation detection of problems, but as will later be demonstrated real problems are rarely have neat small gaps between two peaks as the ones that were tested with JUMP_m. The success of `Standard GA` demonstrates the usefulness of multi-population algorithms on problems where the algorithm can get stuck in local optimums.

8.4 3-Sat

3-SAT is a multimodal problem with unknown gap sizes between local and global optimums. Unlike JUMP_m the starting location of an individual can result in benefits in regards to not getting stuck, however the sample size should make this difference negligible. This problem benchmarks for the similar behavior as JUMP_m, but is much more chaotic as there are more local optimums with no predefined structure.

To compare the algorithms the running time and computational complexity is investigated in Figure 8.8 and Figure 8.9. The algorithms were run with a time limit of 60 seconds with a sample size of 15, regardless of the problem size.

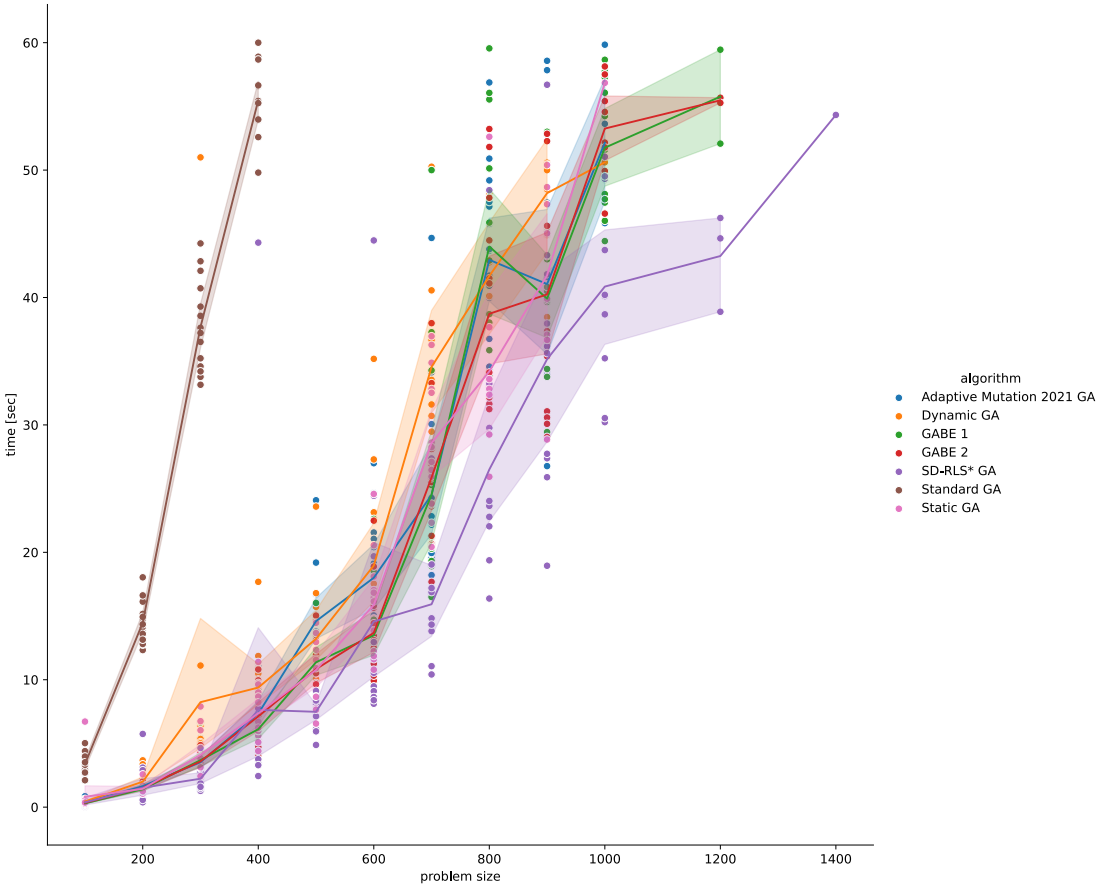


Figure 8.8: 3-Sat Running Time

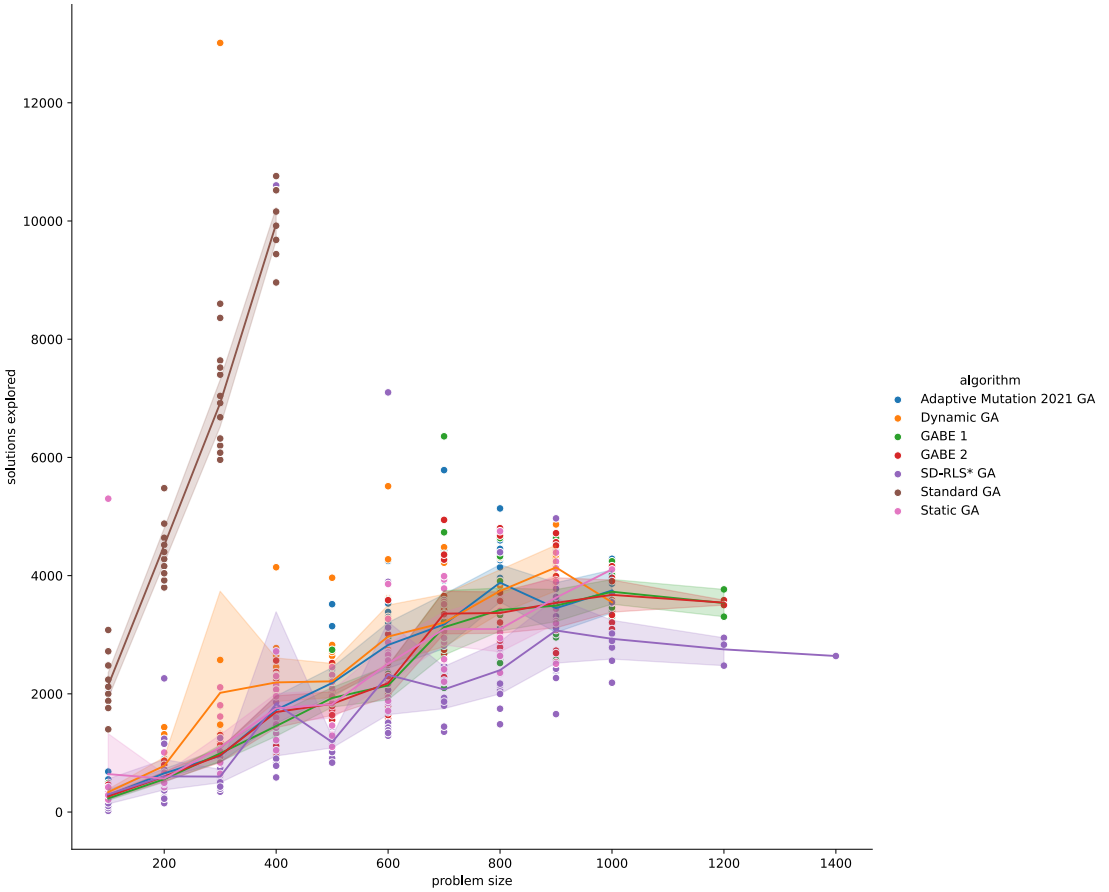


Figure 8.9: 3-SAT Computational Complexity

Most of the algorithms perform very similar, with the exception of `Standard GA` that performs very poorly. This is interesting as it performed very well on JUMP$_m$ that is quite similar to 3-SAT. The reason for this difference is likely that the population size was almost as large as the problem size for JUMP$_m$ and the jumps in the fitness landscape were considerably smaller than those for 3-SAT. As many instances remain unsolved, the average distance from the optimum of all samples are shown in Figure 8.10.

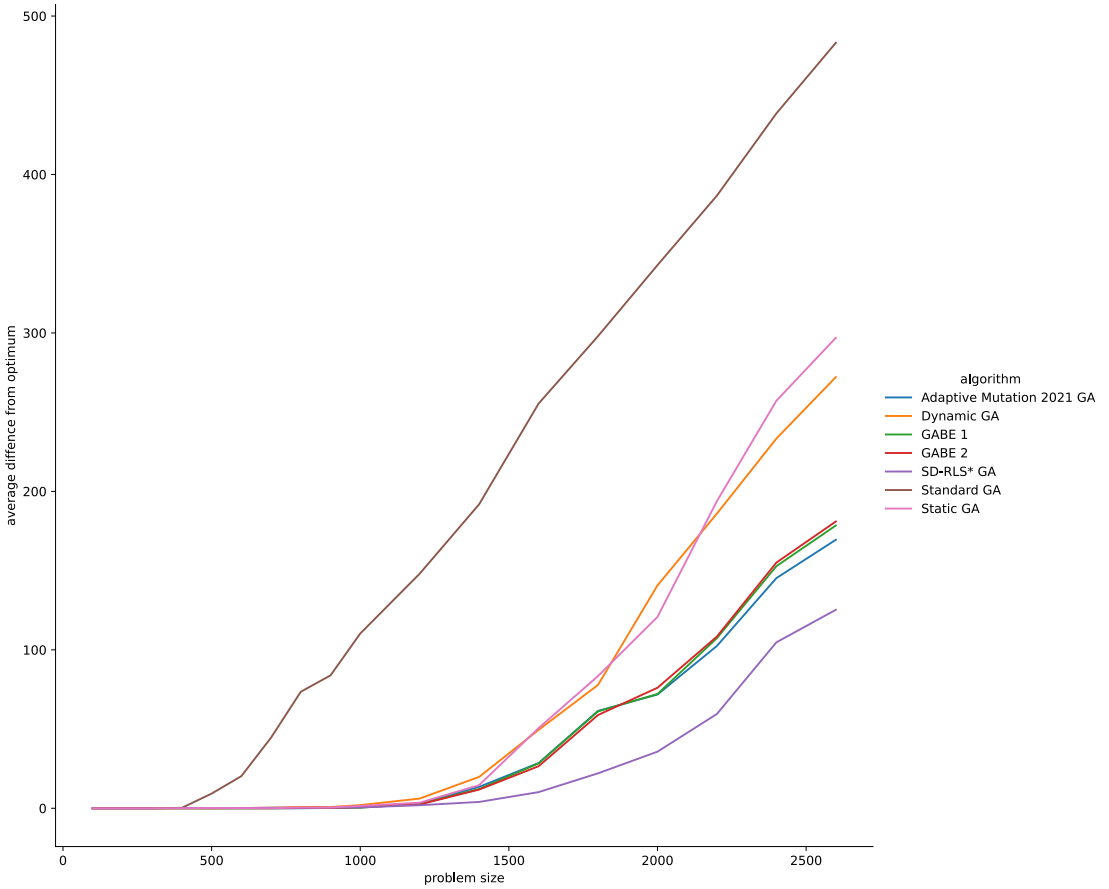


Figure 8.10: 3-Sat. Average distance from optimum

Here the benefits of global mutation, as used by `Adaptive Mutation 2021 GA` is seen more clearly. Where the algorithm had a rough time on many of the previous problems, it shines on 3-Sat where the mutation method results in an advantage in crossing the gabs between local optimums. The other effective strategies is the local mutation with stagnation detection and the adaptive local mutation as deployed by `GABE 1` and `GABE 2`.

The issue with the generated instances of 3-Sat is that they are likely to have more global optimums, making it less punishing for algorithms with poor global traversal of the search space. This is seen from the results of the benchmark instances from [SAT00] in Figure 8.11 below.

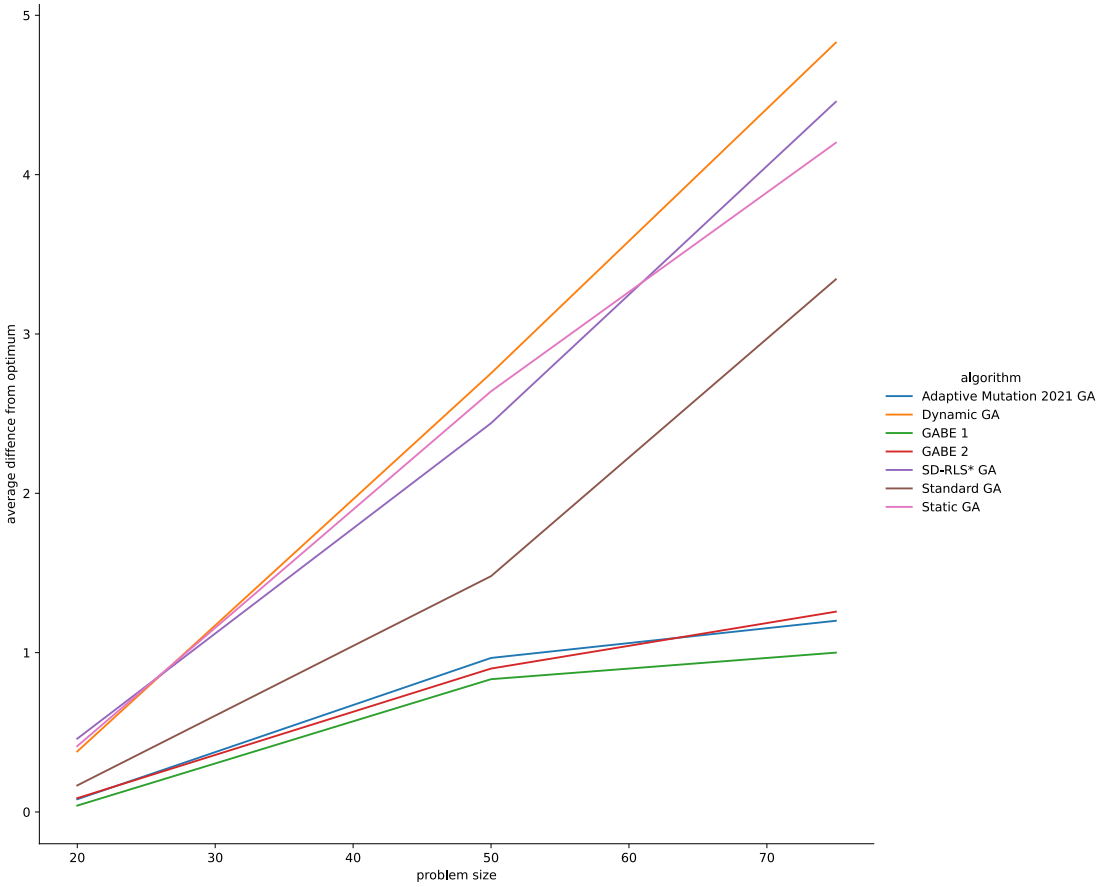


Figure 8.11: 3-Sat. Average distance from optimum

It is seen how `SD_RLS* GA` which performed the best previously is unable to solve instances with fewer and possibly only one global optimum. While the more globally adept `Adaptive Mutation 2021 GA`,`GABE 1` and `GABE 2` algorithms outperform it on these problem instances.

8.5 TSP

TravelingSalesman is a multimodal problem with permutation encoding, meaning that the search space grows huge as n increases. There is no fixed structure in the gaps between the local and global optimums. For TravelingSalesman there is two sets of problems, those from [TSP95] and the generated. For the former the optimal solutions are known, so they can be used to demonstrate that the algorithms are in general able to solve the instances. The plots for these are found in the appendix(section A.2). For the latter set of problems there is no known optimum solutions, which makes the plots for running time and computational complexity impossible to create. It is also impossible to plot the distance from the optimum solution. Instead the best found solution is used as the baseline that the algorithms averages are plotted against. The algorithms were run

with a time limit of 60 seconds with a sample size of 5, regardless of the problem size and the results are seen in Figure 8.12.

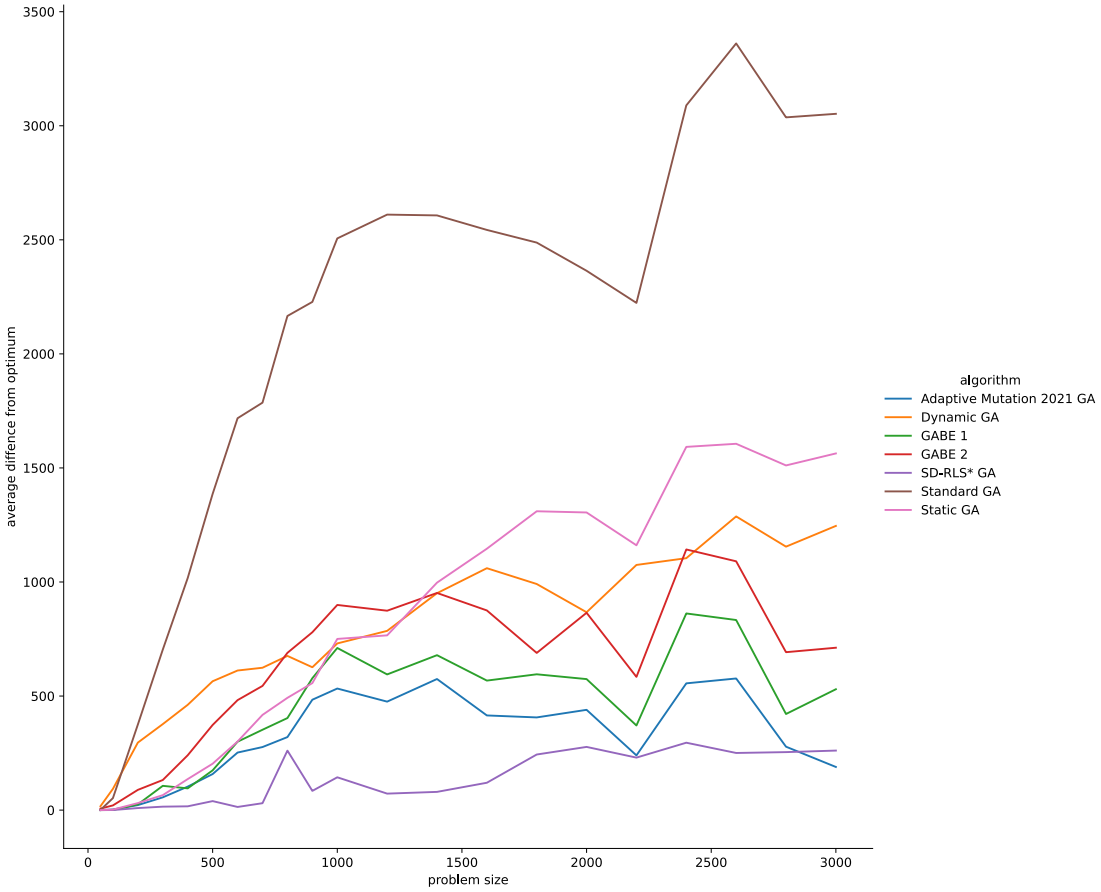


Figure 8.12: TravelingSalesman. Time limit: 60 sec, sample size: 5. Average distance from optimum solution

The results show that `SD_RLS* GA` is again the best algorithm, closely followed by `Adaptive Mutation 2021 GA`, `GABE 1` and `GABE 2`.

8.6 Sorting

Sorting is a multimodal problem with permutation encoding like TravelingSalesman. As with TravelingSalesman the search space is rapidly growing, but where the algorithms could solve quite large instances of TravelingSalesman, Sorting turns out to be significantly harder for the algorithms to solve.

To compare the algorithms the running time and computational complexity is investigated in Figure 8.8 and Figure 8.9. The algorithms were run with a time limit of 15 seconds with a sample size of 15, regardless of the problem size.

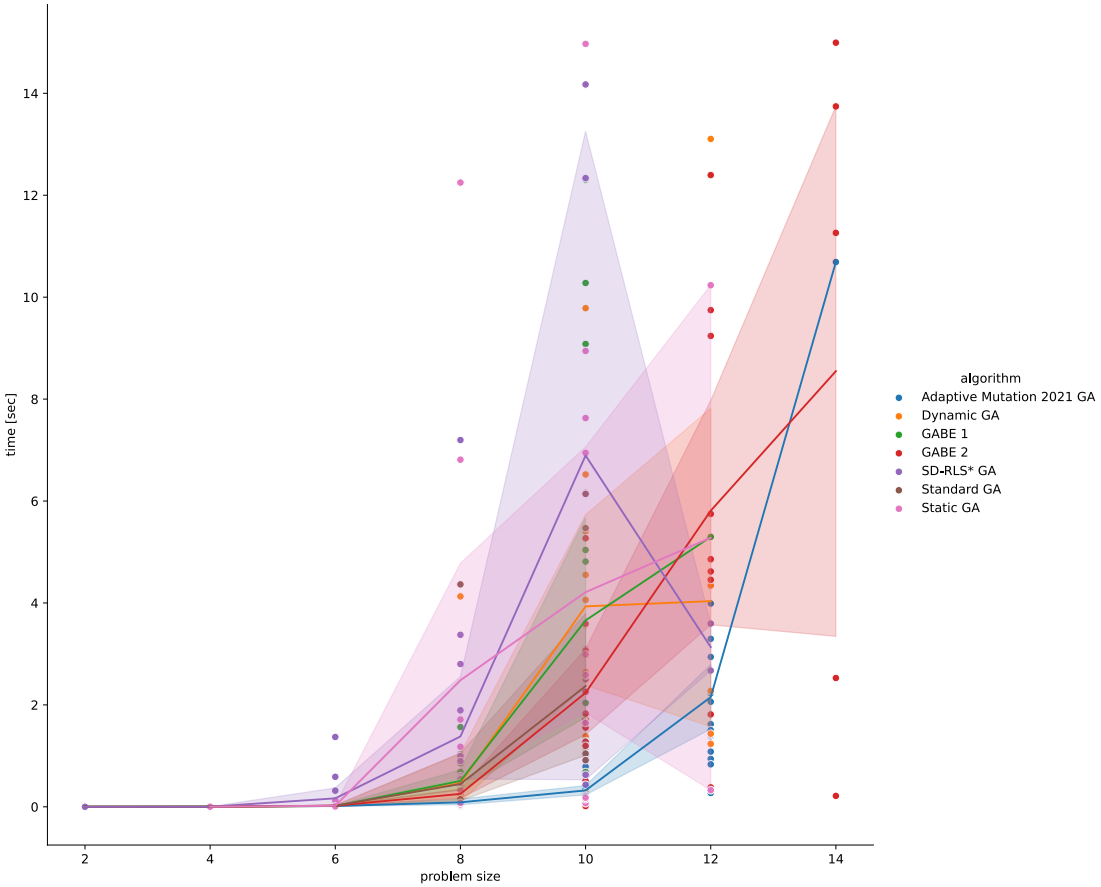


Figure 8.13: Sorting Running Time

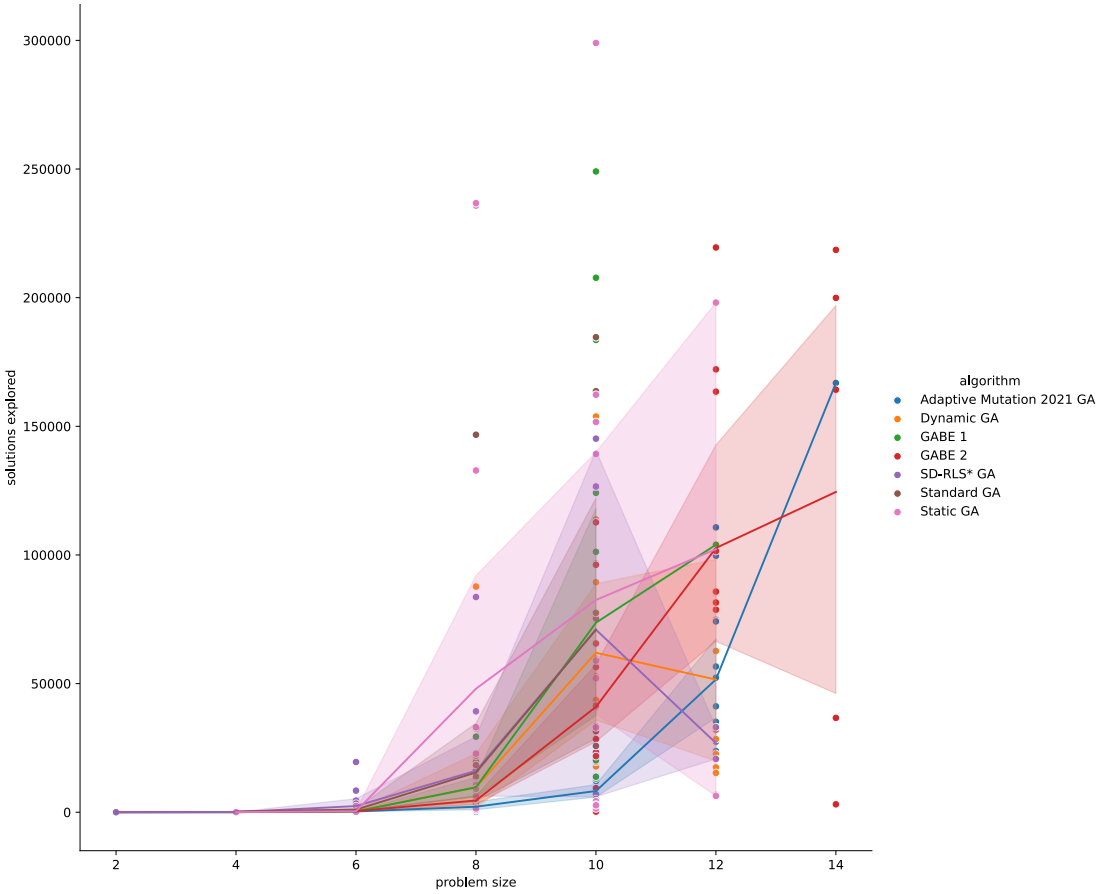


Figure 8.14: Sorting Computational Complexity

It is seen how the algorithms struggle even for small problem sizes. This makes it hard to interpret much from the plots, so the average deviation from the optimum solution is investigated.

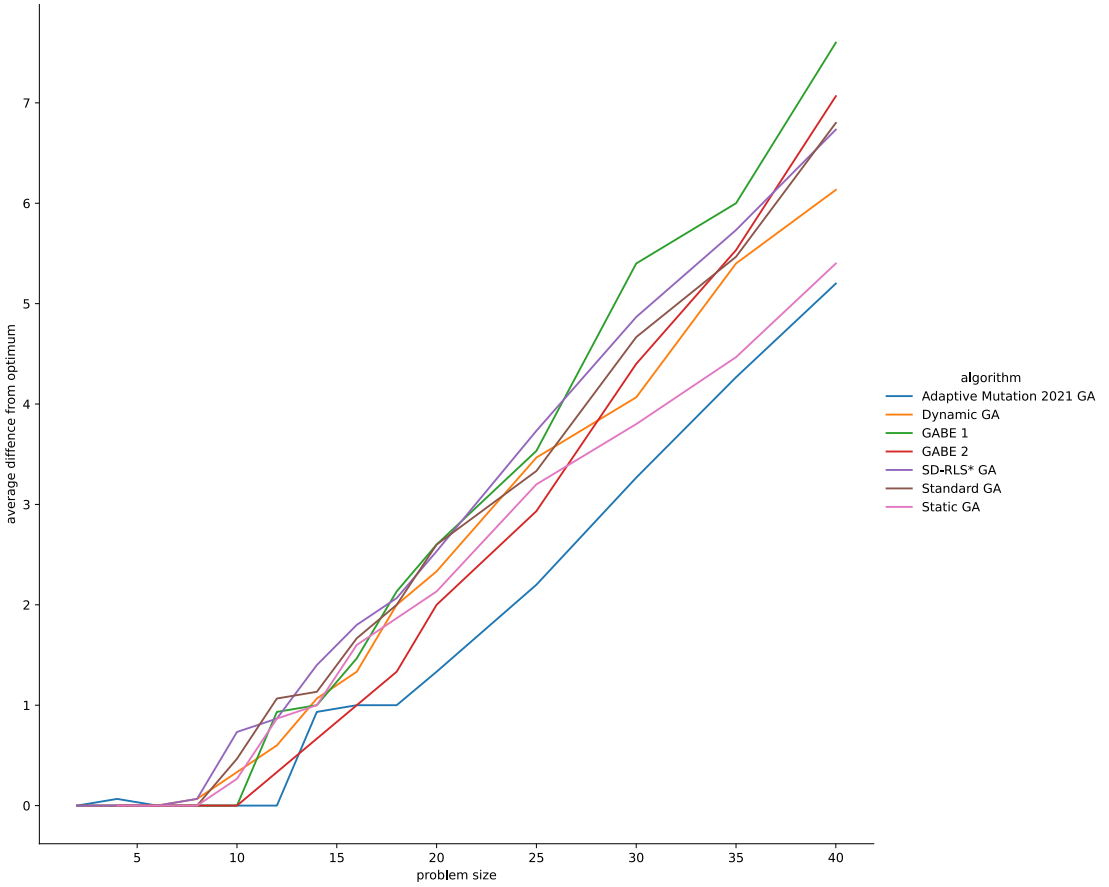


Figure 8.15: Sorting. Average distance from optimum solution

Figure 8.15 shows a cutoff at around $n = 10$, where the algorithms are unable to solve the problems within the time frame. We can gauge the performance by the cutoff point where it is no longer able to solve the problems consistently.

The reason for the cutoff is that the algorithms reach a solution with several locally sorted subsets, which is rewarded by the heuristic. This means that the mutation operators are not fit for rearranging into the globally sorted solution. This problem can be solved in 2 ways. Either with a heuristic, that rewards absolute positioning instead of the current or a mutation operator that is better suited for the problem signature. The *k-swap mutation* mutation operator moves elements to a new location and therefore seem like a good fit for the issue. This is tested through the program by overwriting the mutation operators with *k-swap mutation*. The average distance from the optimum with the new operators is run with the best algorithms and the results are shown below.

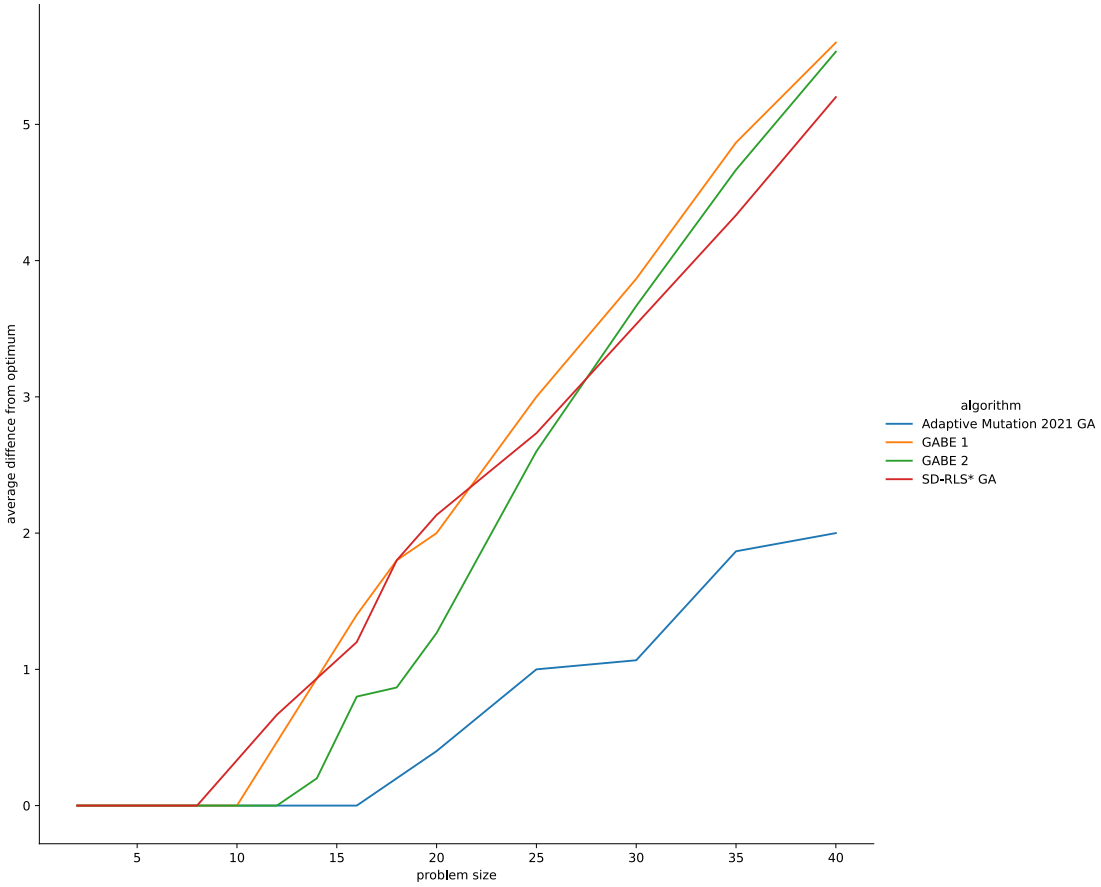


Figure 8.16: SORTING with *k-swap mutation*. Average distance from optimum solution

The results demonstrate how each algorithm performs significantly better as the mutation operators is replaced. This is especially the case for `Adaptive Mutation 2021 GA`. The reason for this is probably that *k-reverse mutation* is inherently slower than *k-swap mutation*, but that `Adaptive Mutation 2021 GA` can make up for this by increasing the mutation step size to reduce the inefficiency of the operator. This demonstrates how the pick of evolutionary operators are not only highly problem dependent, but also can have different effect based on the heuristic of the problem. This shows why the functionality of easily overwriting operators for empirical testing of different methods are a highly efficient way to test and improve the performance of algorithms.

9. Discussion

Before concluding this thesis, there is certain limitations to be mentioned.

The two different encodings required some personal interpretation of the algorithms, as they were all designed for binary encodings. I attempted to maintain the general idea in my conversion of the algorithms, but the concept of bias as was used in the algorithms from [DD19a] was not explored for many crossover operators, making the conversion imperfect.

The number of the problems with permutation encodings investigated in this thesis, leaving further work to be done in investigating the different types of problem, which could reveal an area of competence for some of the algorithms that weren't suited for the permutation problems investigated in this thesis.

This thesis focused on comparing the performance of different algorithms and not on the different parameter settings for each algorithm. This means that I chose parameters that were deemed good enough, without investigating optimal parameter settings. This means that the results of the comparison is subject to change if the parameters of each algorithm is optimized for the problem and problem sizes.

The implementation and performance of the implemented algorithms were as expected, with the exception of my implementation of the dynamic algorithm from [DD19a], which had a worse performance than the static version unlike the results from the team. I was not able to identify the reason for this.

The program does not utilize any form of parallelization. Implementing this will significantly reduce the computation time, but was deemed to not be a priority in regards to the focus of the thesis. This is justified by it not having any impact of the quality of the results from the program, which was the prime factor.

For my own algorithms I initially wanted to add an additional mechanism of self-adaptation, but did not have time to go through the design and test process, and thus decided to leave it undone.

10. Conclusion

The aim of this thesis project was to explore the field of evolutionary algorithms for solving optimization problems. To validate an empirical approach to testing and developing genetic algorithm by designing, implementing and testing a framework for testing algorithms. To evaluate current state-of-the-art self-adjusting genetic algorithm on set of benchmark optimization problems, to gain an understanding of the pros and cons of the different designs and strategies. And finally to develop an algorithm that based on the investigation was able to outperform the explored strategies on some optimization problems.

In chapter 2 and chapter 3 the theory behind the field of evolutionary algorithms focused on solving optimization problems was explored, along the mechanisms for self-adjusting genetic algorithms. Several state-of-the-art algorithms were investigated and documented.

In chapter 5 and chapter 6 the design and implementation for the testing framework was documented.

My own algorithm contributions were documented in chapter 4 with an algorithm that was shown to be able to compete with and even outperform most of the other algorithms on all benchmark problems, as demonstrated in chapter 8. While both algorithms performs quite well, the concept still feels incomplete. The reason for this is that a mechanism for adjusting the size of the global mutation step is not included. However the second version has demonstrated how the idea of mixing several mutation operators can be a valid approach to genetic algorithms. As `SD_RLS* GA` performed incredibly well with local mutation I do believe that optimizations to the adaptive mechanism in both algorithms, either in parameters settings or in adjusting the update rules, could result in significant improvements to optimization times.

The implementation of several genetic algorithms, evolutionary operators and optimization problems demonstrate how simple the framework is to extend in regards to all of these.

The testing of the more complicated problem Sorting demonstrated how the performance of an algorithm can be improved significantly by choosing evolutionary operators that fit signature of the problem and how the framework allowed that to be done seamlessly.

The framework allows a quick path from idea to feedback on the performance of an algorithm, and allows more continuity in building algorithms, as opposed to a more theoretical approach. I can thus conclude that empirical testing through a framework, such as the one developed in this thesis is a great tool for developing and testing genetic

algorithms. It is especially useful in regards to self-adjusting genetic algorithms, for which it can be almost impossible to predict performance theoretically.

Glossary

NP Complexity class for optimization problems.

***k*-flip mutation** Mutation operator for binary encodings. Flips k random bits in parent chromosome.

***k*-reverse mutation** Mutation operator for permutations encodings. Removes 2 random edges and connects endings in a different way, k times.

***k*-swap mutation** Mutation operator for permutations encodings. Moves k entries in parent chromosome.

biased crossover Picks each gene independently from the first parent with probability c or the second parent otherwise. c can be changed for a biased crossover..

ordered crossover Ordered crossover(OX).

probability flip mutation Mutation operator for binary encodings. Flips each bit in parent chromosome with probability p.

single-point crossover Picks a point in $k \in [1, n-1]$ and takes the k first genes from the first parent and the remaining from the second..

uniform crossover *biased crossover* with $c = .5$.

3-Sat A subset of the Satisfiability problem with n clauses with exactly 3 literals. Encoding:Binary.

Jump$_m$ The problem of maximizing the number of bits in a bit string, with a m sized jump in the fitness scores. Encoding:Binary.

LeadingOnes The problem of maximizing the number of consecutive one bits in a bit string, starting from the 0'th location. Encoding:Binary.

OneMax The problem of maximizing the number of ones in a bit string. Encoding:Binary.

Sorting The problem of sorting a list of items. Encoding:Permutations.

TravelingSalesman A NP-complete minimization problem containing n locations for a 'salesman' to visit with the shortest total path. Encoding:Permutation.

chromosome A collection(often list) of genes.

evolutionary algorithm Class of metaheuristics that solve problems based on the strategies of evolution.

gene The smallest changeable part of a chromosome.

genetic algorithm Type of evolutionary algorithm used for solving optimization problems.

search space The set of all valid solutions to an optimization problem.

Bibliography

[BB19] Anton Bassin and Maxim Buzdalov. The 1/5-th rule with rollbacks. *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2019.

[DD15] Benjamin Doerr and Carola Doerr. Optimal parameter choices through self-adjustment. *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, 2015.

[DD19a] Nguyen Dang and Carola Doerr. Hyper-parameter tuning for the $(1 + \lambda, \lambda))$ ga. *Proceedings of the Genetic and Evolutionary Computation Conference*, page 889–897, 2019.

[DD19b] Benjamin Doerr and Carola Doerr. Theory of parameter control for discrete black-box optimization: Provable performance gains through dynamic parameter choices. *Natural Computing Series*, page 271–321, 2019.

[DDE15] Benjamin Doerr, Carola Doerr, and Franziska Ebel. From black-box complexity to designing new genetic algorithms. *Theoretical Computer Science*, 567:87–104, 2015.

[Deb99] Kalyanmoy Deb. An introduction to genetic algorithms. *Sadhana*, 24(4-5):293–315, 1999.

[DWY18] Benjamin Doerr, Carsten Witt, and Jing Yang. Runtime analysis for self-adaptive mutation rates. *Proceedings of the Genetic and Evolutionary Computation Conference*, 2018.

[DWY21] Benjamin Doerr, Carsten Witt, and Jing Yang. Runtime analysis for self-adaptive mutation rates. *Algorithmica*, 2021.

[RW21] Amirhossein Rajabi and Carsten Witt. Stagnation detection with randomized local search. *Lecture Notes in Computer Science*, page 152–168, 2021.

[SAT00] SATLIB, 2000. A subset of the Uniform Random-3-SAT collection is used.

[STW04] Jens Scharnow, Karsten Tinnefeld, and Ingo Wegener. The analysis of evolutionary algorithms on sorting and shortest paths problems. *Journal of Mathematical Modelling and Algorithms*, 3(4):349–366, 2004.

[TSP95] TSPLIB. Tsp benchmark problems, 1995. A subset of the TSP problem collection is used.

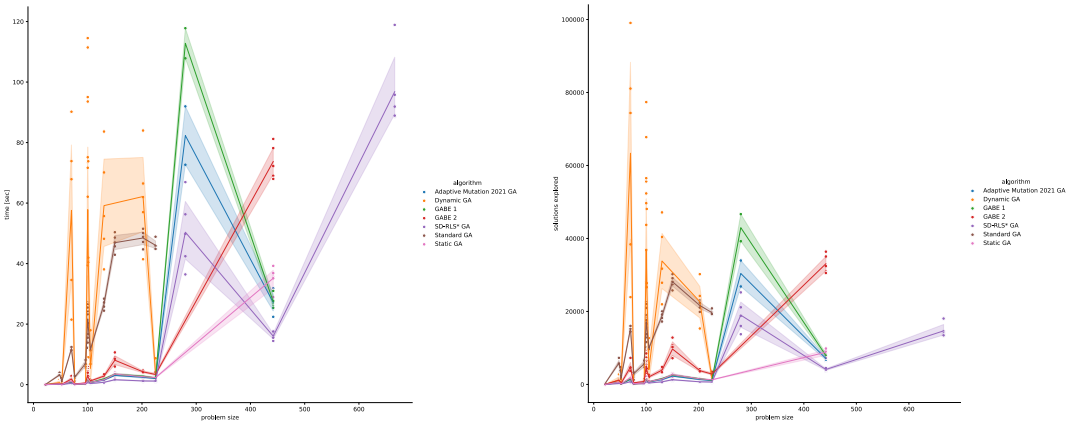
A. Appendices

A.1 Test parameters

GAStandard	**OneMax**	**LeadingOnes**	**JumpM**
lambda	100	100	100
pc	0.2	0.2	0.2
mutation	KFlipMutation	KFlipMutation	KFlipMutation
crossover	BiasedUniformCrossover	BiasedUniformCrossover	BiasedUniformCrossover
GAAdaptiveMut	**OneMax**	**LeadingOnes**	**JumpM**
lambda	12	12	12
F	2.5	2.5	2.5
r0	10	10	10
mutation	PFlipMutation	PFlipMutation	PFlipMutation
SD_RLS	**OneMax**	**LeadingOnes**	**JumpM**
R	n+1	n+1?	n+1
mutation	KFlipMutation	KFlipMutation	KFlipMutation
GAStatic	**OneMax**	**LeadingOnes**	**JumpM**
lambda1	15	15	15
lambda2	4	4	4
k	3	3	3
c	0.3	0.3	0.3
mutation	KFlipMutation	KFlipMutation	KFlipMutation
crossover	BiasedUniformCrossover	BiasedUniformCrossover	BiasedUniformCrossover
GADynamic	**OneMax**	**LeadingOnes**	**JumpM**
alpha	0.7	0.7	0.7
beta	1.4	1.4	1.4
gamma	1.24	1.24	1.24
a	1.67	1.67	1.67
b	0.69	0.69	0.69
mutation	KFlipMutation	KFlipMutation	KFlipMutation
crossover	BiasedUniformCrossover	BiasedUniformCrossover	BiasedUniformCrossover
GABE1	**OneMax**	**LeadingOnes**	**JumpM**
lambda	12	12	12
lambda2	2	2	2
F	2.5	2.5	2.5
r0	10	10	10
mutation	KFlipMutation	KFlipMutation	KFlipMutation
GABE2	**OneMax**	**LeadingOnes**	**JumpM**
lambda	12	12	12
lambda2	2	2	2
F	2.5	2.5	2.5
r0	10	10	10
local_mutation	KFlipMutation	KFlipMutation	KFlipMutation
global_mutation	PFlipMutation	PFlipMutation	PFlipMutation

GAStandard	**3Sat**	**TSP**	**Sorting**
lambda	100	100	100
pc	0.2	0.2	0.2
mutation	KFlipMutation	ReverseMutation	ReverseMutation
crossover	BiasedUniformCrossover	OrderedCrossover	OrderedCrossover
GAAdaptiveMut	**3Sat**	**TSP**	**Sorting**
lambda	12	12	12
F	2.5	2.5	2.5
r0	10	10	10
mutation	PFlipMutation	ReverseMutation	ReverseMutation
SD_RLS	**3Sat**	**TSP**	**Sorting**
R	n+1	n+1	n+1
mutation	KFlipMutation	ReverseMutation	ReverseMutation
GAStatic	**3Sat**	**TSP**	**Sorting**
lambda1	15	15	15
lambda2	4	4	4
k	3	3	3
c	0.3	0.3	0.3
mutation	KFlipMutation	ReverseMutation	ReverseMutation
crossover	BiasedUniformCrossover	OrderedCrossover	OrderedCrossover
GADynamic	**3Sat**	**TSP**	**Sorting**
alpha	0.7	0.7	0.7
beta	1.4	1.4	1.4
gamma	1.24	1.24	1.24
a	1.67	1.67	1.67
b	0.69	0.69	0.69
mutation	KFlipMutation	ReverseMutation	ReverseMutation
crossover	BiasedUniformCrossover	OrderedCrossover	OrderedCrossover
GABE1	**3Sat**	**TSP**	**Sorting**
lambda	12	12	12
lambda2	2	2	2
F	2.5	2.5	2.5
r0	10	10	10
mutation	KFlipMutation	KExchangeMutation	KExchangeMutation
GABE2	**3Sat**	**TSP**	**Sorting**
lambda	12	12	12
lambda2	2	2	2
F	2.5	2.5	2.5
r0	10	10	10
p	0.3	0.3	0.3
local_mutation	KFlipMutation	KexchangeMutation	KExchangeMutation
global_mutation	PFlipMutation	ReverseMutation	ReverseMutation

A.2 TSPLIB Results



(a) Running Time (b) Computational Complexity

Figure A.1: TravelingSalesman[TSP95]. Time limit: 120 sec, sample size: 5

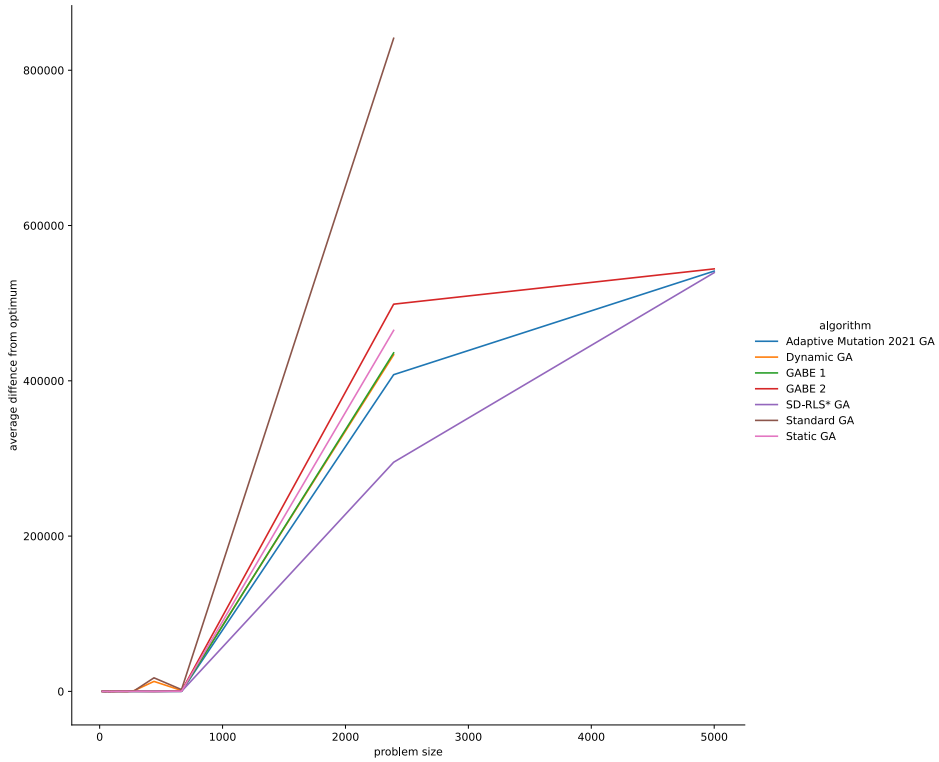


Figure A.2: A