Improving Usability of Genetic Algorithms through Self Adaptation on Static and Dynamic Environments

Reamonn Norat
University of Central Florida

IMPROVING USABILITY OF GENETIC ALGORITHMS THROUGH SELF ADAPTATION ON STATIC AND DYNAMIC ENVIRONMENTS

by

REAMONN NORAT
B.S. Embry-Riddle Aeronautical University, 2013

A thesis submitted in partial fulfilment of the requirements
for the degree of Master of Science
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2020

ABSTRACT

We propose a self-adaptive genetic algorithm, called SAGA, for the purposes of improving the usability of genetic algorithms on both static and dynamic problems. Self-adaption can improve usability by automating some of the parameter tuning for the algorithm, a difficult and time-consuming process on canonical genetic algorithms. Reducing or simplifying the need for parameter tuning will help towards making genetic algorithms a more attractive tool for those who are not experts in the field of evolutionary algorithms, allowing more people to take advantage of the problem solving capabilities of a genetic algorithm on real-world problems.

We test SAGA and analyze its the behavior on a variety of problems. First we test on static test problems, where our focus is on usability improvements as measured by the number of parameter configurations to tune and the number of fitness evaluations conducted. On the static problems, SAGA is compared to a canonical genetic algorithm. Next, we test on dynamic test problems, where the fitness landscape varies over the course of the problem's execution. The dynamic problems allows us to examine whether self-adaptation can effectively react to ever-changing and unpredictable problems. On the dynamic problems, we compare to a canonical genetic algorithm as well as other genetic algorithm methods that are designed or utilized specifically for dynamic problems. Finally, we test on a real-world problem pertaining to Medicare Fee-For-Service payments in order to validate the real-world usefulness of SAGA. For this real-world problem, we compare SAGA to both a canonical genetic algorithm and logistic regression, the standard method for this problem in the field of healthcare informatics.

We find that this self-adaptive genetic algorithm is successful at improving usability through a large reduction of parameter tuning while maintaining equal or superior results on a majority of the problems tested. The large reduction of parameter tuning translates to large time savings for users of SAGA. Furthermore, self-adaptation proves to be a very capable mechanisms for dealing with the difficulties of dynamic environment problems as observed by the changes to parameters in response to changes in the fitness landscape of the problem.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

We investigate the use of self-adaptive parameters in a Genetic Algorithm (GA) Holland 1975, a type of Evolutionary Algorithm (EA), as a way to both simplify usability and improve performance on dynamic problems. GAs have been successfully applied to a variety of difficult problems; however, outside of academia, GAs are not used as often as their successes show they could be used. One of the reasons for this lack of use is the difficulty in setting up and tuning the parameters for an effective GA A. E. Eiben and Smit 2011a. A GA has many parameters that must be optimized for each problem to which it is applied; performance is highly sensitive to these parameters Grefenstette 1986; Schaffer et al. 1989 and sub-optimal parameters can beget unacceptable results. Furthermore, these parameters must be individually tuned for each and every unique problem; effective parameters on one problem may be ineffective on another. Thus, this parameter tuning process requires a substantial investment of time and effort. The difficulty of accurate parameter tuning is further exacerbated for those who do not have extensive apriori knowledge of either the problem at hand or the inner mechanisms of a GA. The process of tuning involves a direct trade-off: either take the time to tune a large number of parameter values to be certain sufficiently good parameter values are found, or tune fewer parameters values to decrease tuning time at the risk of missing superior parameter values and therefore obtaining inferior results. All of this means that the GAs can be an unattractive option for those who need a simple yet useful tool to solve a specific real-world problem but are not already experts in the field of EAs. As a result, the problem-solving power of the GA is not being used to its full potential.

Another downfall of conventionally tuned GA parameters is that the parameters are static. This static nature can be detrimental for many problems, but is of particular concern for problems of a dynamic nature; wherein the problem changes over the course of the algorithm's execution. This dynamic change can take multiple forms, from a slow and steady shifting of the problem's landscape to a landscape that is static for periods of time, but occasionally undergo sudden and drastic changes. In dynamic problems, the amount of exploration or exploitation required to find good results can change from moment to moment depending on the problem's dynamics. Static parameters are unable to adapt to these changing requirements due to their fixed nature. The complexity of dynamic problems has led to the development of multiple mechanisms and algorithms specifically designed for overcoming the difficulty of dynamic problems Hughes 2016. The mechanisms for dynamic problems, however, can make these algorithms less effective on conventional static problems. Furthermore, these extra mechanisms often increase the tuning burden through the introduction of new parameters.

A solution to both of these issues can be found in self-adaptive parameters, wherein parameters are dynamic and the algorithm itself changes the parameters over the course of the algorithm's execution. Self-adaptive parameters both reduces the need for parameter tuning and allows for better adaptation to the changing landscapes of dynamic problems. The use of dynamic parameters, known as parameter control, consists of three categories: deterministic, adaptive, and self-adaptive A. E. Eiben, Hinterding, and Michalewicz 1999. Deterministic parameters change throughout the algorithm's execution in a set, pre-determined manner, such as having a parameter value linearly increase or decrease over the course of the algorithm's execution. Adaptive parameters use some sort of feedback loop, e.g. basing a parameter's value on the recent efficacy of said parameter. Self-adaptive parameters are encoded within each individual of the population and are evolved using the same evolutionary process through which the problem's solution is evolved.

Our work examines the self-adaptive method because it is the only method that does not intrinsically require the introduction of one or more new parameters to determine how another parameter is to be changed. Regarding algorithm usability, deterministic and adaptive methods intrinsically add new parameters in the form of planned schedules, equations, or feedback relationships between values; each of which must be tuned or designed by the user. We want to avoid introducing new parameters as much as possible since our goal is to simplify GA usability by reducing the total number of parameters that must be tuned. Regarding dynamic problems, if the algorithm's parameters are also dynamic through self-adaptation, then the parameters can update as necessary to track the changing requirements of a dynamic problem. Hence, a self-adaptive GA that does not use any specialized mechanisms for dynamic problems may be competitively effective at dynamic problems without sacrificing the algorithm's usability nor viability on static problems.

For our implementation of a Self-Adaptive Genetic Algorithm (SAGA), we apply self-adaptation to the parameters of mutation rate, crossover rate, the selection of which mutation operator to use, the selection of which crossover operator to use, and the tournament size for parent selection. Mutation and crossover are GA parameters that are each controlled by their respective rates and for each, there exists many different implementations, or operators, from which to choose. The tournament size for parent selection controls the balance of exploration and exploitation of the parent selection process. These parameters are chosen because they are highly influential on a GA's performance, are relatively simple to encode within each individual, and have a wide range of optimal values depending on the problem being solved.

The usability ideal is to have a totally "parameter-less" GA wherein every single parameter is self-adaptive and requires no tuning whatsoever. Such a parameter-less GA, however, is out of our scope. Attempts at such a system have been made, but these inevitably end up using one or more "default" values for some of the parameters Lobo and Goldberg 2004, which we argue is not the same as removing the need to tune the parameter. Furthermore, such default parameter values are static and therefore, do not aid us within the domain of dynamic problems. Some parameters inherently lend themselves much more easily to self-adaptation, while others do not, due to the manner of the parameters' implementations. For instance, population size is a parameter that acts on a global scale for the algorithm which means that there is no intuitive way to encode population size within each individual. Another consideration is that increasing the number of self-adaptive parameters increases the size of the encoded data within an individual and therefore increases the size of the search space. So while self-adaptation may decrease the tuning difficulty, it comes at the cost of increasing computational difficulty. Care must therefore be given to not increase the computational difficulty through ineffectual implementations of self-adaptation to a degree which harms the quality of the algorithm's results.

Parameter control for GAs is generally considered atypical. In another category of EA, Evolutionary Strategies (ES), parameter control is much more common, to the point that using an adaptive mutation rate is canonical Beyer 1996. We believe that if the idea of parameter control is effective at finding effective mutation rates in other, similar EA methods, then parameter control should work for GA parameters as well if implemented properly.

It may at first seem counter-intuitive that we are claiming to make GAs simpler to use when we are actually increasing the complexity of the algorithm through the addition of self-adaptive parameters. The internals of the SAGA we introduce here is indeed more complex than a canonical GA (CGA). There is a distinction, however, between decreasing the complexity within the algorithm and decreasing the complexity of using the algorithm. Our goal is to help push GAs towards being a usable black-box style system in which a user can be given a GA and effectively use it without needing to know all the details of what is going on inside the box. Complexity within the black box is therefore irrelevant to the user as long as the external complexity is sufficiently low and the algorithm returns a satisfying result in a timely manner.

It must be emphasized that the focus of our work is not on finding fitness performance improvements over static parameters. There already exists many studies focusing on the potential improvements to results from utilizing various forms of parameter control over static parameters Bäck 1992a; Contreras-Bolton and Parada 2015; Yoon and Moon 2002. The focus of this study is rather to see if SAGA can effectively adapt parameter values such that the final results are at least equivalent to a comparable manually-tuned CGA while obtaining the desired usability improvements. We are particularly interested in the usability improvements for static problems, where we expect that a set of optimally tuned static parameters are likely to perform quite well. On the dynamic problems, we expect SAGA to have better performance over a CGA, but we are not certain if SAGA will have better performance over the other GA mechanisms that are tailored specifically towards dynamic problems. Regardless of whether there are fitness improvements, we expect SAGA to have usability improvements over these dynamic problem GA mechanisms as well.

This work begins with a look into previous research of parameter control for GAs. Next, we give a description of the CGA, as it is the foundation from which SAGA is built and is the primary benchmark to which we compare SAGA. We then describe SAGA and how exactly it extends the CGA. We perform three sets of experiments; the first compares SAGA to CGA on a set of benchmarking static problems, with an eye towards comparing both fitness results and runtime results. Next, we test SAGA on a set of benchmarking dynamic problems; comparing to CGA and other GA mechanisms specifically designed for dynamic problems. Finally, we test SAGA on a real-world static classification problem in the field of health informatics, comparing to CGA and the standard method in the field. Our goal is to make GAs more easily usable for real-world applications outside the field of EA; thus, we include this real-world problem to test our hypothesis.

CHAPTER 2: LITERATURE REVIEW

EA researchers have invested extensive study into parameter tuning and parameter control due to the great importance and difficulty of finding effective parameters. We investigate the research in this field, focusing on three areas: The difficulty of parameter tuning, which in turn led into the development of parameter control, and a look into the three categories of parameter control. In addition, we look into the research of GAs for dynamic problems.

Effective and efficient parameter tuning is a problem that researchers are still contending with today. Early research focused on finding parameters that are globally optimal across all problems Kenneth Alan De Jong 1975; Schaffer et al. 1989. The in-feasibility of exhaustively testing all parameter value combinations led to the use of a meta-level GA to aid in the search of optimal GA parameters Grefenstette 1986. The search for global optimal parameter values ultimately found contradicting results; leading to the understanding that optimal parameter values that are global across all problems do not exist Wolpert and Macready 1997. A more recent study focuses instead on quantifying and comparing the importance of each of the parameters to help users decide which parameters to devote extra tuning effort towards Mills, Filliben, and Haines 2015. The difficulty of parameter tuning has even led to attempts to make a GA completely "parameter-less", removing the need to tune anything at all Harik and Lobo 1999; Lobo and Goldberg 2004. We, however, disagree with the assertion that this method is truly parameter-less; in reality it is a GA that uses pre-determined parameters that are recommended by the authors. Another parameter-less attempt is made on GAMBIT, a GA specialized for mixed-integer problems Sadowski, Thierens, and Bosman 2018; in this attempt, multiple parameter values are tested according to a pre-determined schedule as recommended by the authors. The continuing difficulty of parameter tuning through the years has resulted in enough research to warrant a survey on parameter tuning methods A. E. Eiben and Smit 2011b.

Parameter control, which refers to the the use of dynamic parameters, offers an alternative to parameter tuning, though it brings its own set of difficulties. Each of the three categories of parameter control: deterministic, adaptive, and self-adaptive must be able to change the parameters in a beneficial manner. Many parameter control methods, particularly deterministic and adaptive methods, have the unfortunate side effect of adding in new parameters which themselves must be tuned. Researchers have proposed a myriad of diverse parameter control methods over the years; many of which are discussed within the multiple parameter control survey papers that have been published over the years A. E. Eiben, Hinterding, and Michalewicz 1999; A. E. Eiben and Smit 2011a; Hinterding, Michalewicz, and A. E. Eiben 1997; Karafotias, Hoogendoorn, and A. E. Eiben 2015. Beyond our primary concern of improving usability, research on parameter control of multiple operators has also oftentimes found that there is an inherent synergy from including a set of multiple crossover or mutation operators in an EA; leading to superior performance over an EA that only uses a single operator from this set Murata and Ishibuchi 1996; Contreras-Bolton and Parada 2015; Hong, Wang, Lin, et al. 2002; Yoon and Moon 2002.

Deterministic control is the simplest of the three parameter control methods. The earliest research on parameter control looks into deterministic control. This research on deterministic control has most commonly been applied to mutation rate, due to it being the simplest parameter to control Terence C. Fogarty 1989; Hesser and Männer 1990. These methods find pre-determined schedules for the mutation rate to follow, decreasing over the course of the algorithm's execution. A steady decrease of mutation is desired because a higher mutation rate favors exploration over exploitation, and a larger focus on exploitation is desired in the later stages of the algorithm's execution. For other parameters, such as crossover and the selection of an operator to use, deterministic control doesn't make much sense, as it is much more difficult to create a good pre-determined schedule for these more complex parameters.

Since the early insights into deterministic mutation rate control, the field has moved on to the other methods of adaptive and self-adaptive control and hasn't looked back. Deterministic control does not fit our goals, as creating a useful deterministic parameter schedule requires extensive knowledge of both the algorithm and the problem at hand. Furthermore, deterministic control doesn't makes sense for dynamic problems, where the dynamics may be unknown, and therefore an effective deterministic schedule cannot be created.

In our research, we found adaptive control to be the most popular method of parameter control, particularly adaptive control of operator rates. In adaptive control, some sort of feedback mechanism is used to control the dynamic parameter, with fitness being the most common choice for this mechanism. Adaptive control of mutation rate can be seen as a logical evolution from deterministic control. Mutation rate can be controlled through the average fitness of the population Thierens 2002, such that the better the average fitness, the lower the mutation rate. An alternative to looking at the current fitness of the population is to use the recent fitness contributions of an operator to control operator rates Blum et al. 2005; Hansen and Ostermeier 2001; Lin, W.-Y. Lee, and Hong 2003. The implementation details differ in these examples but they all follow the general guidelines that if an operator makes a positive fitness contribution, the operator's associated rate is increased and vice-versa. Law and Szeto (2007) take the adaptive mutation rate idea and extends it to also adapt crossover rate such that higher fitness individuals have both a lower mutation rate and a higher crossover rate, while also investigating controlling the crossover to apply to only very similar or only very different individuals. A more advanced option is the use of reinforcement learning, which has been successfully used to for an adaptive mutation rate Karafotias, A. E. Eiben, and Hoogendoorn 2014.

Adaptive control of the selection of operators is also a common adaptive strategy. This adaptive control of operator selection works very similarly to the control of operator rates, wherein the probability of using a particular operator is related to how beneficial to fitness that operator has recently been Acan et al. 2003; Hadka and Reed 2013; Hong, Wang, and Chen 2000; Hong, Wang, Lin, et al. 2002; Julstrom 1997. Another adaptive strategy is to use a meta level EA to determine operator selection on subsequent GAs Contreras-Bolton and Parada 2015, which can be seen as an adaptive extension of the original meta-level GA study Grefenstette 1986. One alternative to utilizing recent fitness contributions as the control mechanism for the adaptation, is to instead use population diversity Mc Ginley et al. 2011; in this EA, called ACROMUSE, the objective of the adaptive behavior is to maintain a diverse population over the course of the algorithm's execution.

While adaptive control does remove the need to tune some parameters, it tends to replace these parameters with other, new parameters. The new parameters are used to determine the exact behavior of the adaptive mechanisms in the form of adaptive equations and user-defined constants within the equations. For instance, when looking at recent fitness contributions, as many of these methods do, what exactly defines "recent"? Or how much more influential is a result that is one generation more recent that another result? These are factors of the adaptive behavior that must be defined by the user through new parameters. Thus, despite the clear improvements adaptive control has over deterministic control, the introduction of new parameters makes adaptive control an unfit choice for our particular goals.

Self-adaptive control is the final form of parameter control and is the method in which we are interested. The basic premise of self-adaptive control is to encode a parameter within the individuals of the EA's population and allow the pre-existing evolutionary mechanisms to intrinsically adapt the encoded parameters. As was the case on the other parameter control methods, the simplest parameters to make self-adaptive are the rate parameters; the rates are encoded on each individual and then each individual uses their own unique rate for that parameter Bäck 1992a; Bäck 1992b; Hinterding 1995; Serpell and J. E. Smith 2010; J. Smith and T. C. Fogarty 1996. Certain parameters that require more than one individual, such as crossover, necessitate some sort of aggregation mechanism of the parameters of the involved individuals. For GAs that have more than one operator for mutation and/or crossover, operator selection can be self-adaptively controlled through the encoding of one or more new values onto the individuals to represent each operator. These encoded values can either be used locally to determine the individual's own preferred operator Hinterding 1997; Spears 1995; Serpell and J. E. Smith 2010, or these values can be aggregated across the population to determine global operator selection probabilities Spears 1995. In addition to these empirical studies, theoretical analysis concludes that a $(1,\lambda)$ EA with a self-adaptive mutation rate has a faster expected runtime than an EA with a static mutation rate Doerr, Witt, and Yang 2018.

Self-adaptive parameter control can successfully control various parameters without the cost of introducing new parameters. Since the parameters are encoded within the individuals and can be modified via the evolutionary mechanisms, no new adaptive mechanisms are intrinsically required to be developed by the user. Thus, self-adaptive control is the best fit for our goals and is the method on which we focus our efforts.

In addition to our focus on algorithm usability, we are also interested in utilizing parameter control for dynamic problems. The fitness landscape of a dynamic problem changes over the course of the algorithm's execution; thus, we hypothesis that parameters that can change during execution may be superior than static parameters at tracking a changing environment. Multiple tools exist for EAs for handling dynamic environments; most of these tools use static parameters and introduce one or more new parameters or mechanisms tailored for dynamic problems Hughes 2016. These mechanisms encourage diversity in the population, as encouraging diversity helps EAs find moving optimaFriedrich et al. 2009. A more diverse population is better able to find moving optima because a converged population's exploration capabilities are greatly limited to exploring in a small area around the point of convergence. One early method, called *hypermutation* Cobb 1990; Cobb and Grefenstette 1993; Morrison and Kenneth A. De Jong 2000 uses two mutation rates; the standard mutation rate which is used for most time steps, and the hypermutation rate, which is a much higher rate than the standard rate and is used when the algorithm detects a change in the fitness environment through a drop in fitness. When a change is detected, the hypermutation rate is used for a few iterations to increase population diversity, before switching back to the standard rate. Another method that does not require the detection of a fitness landscape change is *random immigrants* Grefenstette 1992; Cobb and Grefenstette 1993, where every generation, a (typically small) percentage of the population is replaced by new randomly generated individuals. A similar concept is used for *mass extinction* Mathias and Ragusa 2016, where a large portion of the population is replace by new randomly generated individuals; unlike random immigrants, this large extinction occurs rarely rather than every generation, when a pre-defined trigger occurs. Mass extinction was not actually developed with dynamic problems in mind, but is a clear fit. The *Clearing* mechanism Pétrowski 1996 encourages population diversity by penalizing the clustering of individuals within a solution space. When such a cluster of individuals occur, only the best fit of the cluster is given a fitness score. ACROMUSE, Mc Ginley et al. 2011, the EA described earlier that uses adaptive mechanisms controlled through population diversity, is designed specifically with dynamic problems in mind, showing the efficacy of parameter control for dynamic problems. Our approach will use self-adaptive parameter control, and so will be unique from ACROMUSE or any of the other common dynamic problem tools.

CHAPTER 3: METHODOLOGY

In this chapter, we describe the mechanics of SAGA. First, a canonical GA (CGA) is described; SAGA is an extension of the CGA, so understanding the CGA is a pre-requisite to understanding SAGA. Afterwards, we explain the implementation details of SAGA and how it differs from a CGA.

3.1 Canonical Genetic Algorithm

A GA is a form of machine learning inspired by the workings of genetic evolution. Similar to many search algorithms, a GA combines a balance of exploration with exploitation in an attempt to find the optimal solution to a problem. Algorithm 1 shows the basic structure of a GA. In a GA, there exists a *population* of multiple candidate solutions; each one of these candidate solutions is called an *individual*. The individuals of this population evolve over many iterations, known as *generations*, using *genetic operators* and a "survival of the fittest" selection mechanism to improve solution quality.

Algorithm 1 gives the basic outline of a GA.

Algorithm 1 Genetic Algorithm

```
1: Initialize population of candidate solutions
2: while stop condition is false do
3:     Evaluate population (fitness)
4:     Probabilistically select parents for next generation (selection)
5:     Apply crossover to parents to generate children (crossover)
6:     Apply mutation to children (mutation)
7:     Replace population with children
8: end while
```

Every individual encodes a solution to the problem at hand; usually through a numerical format such as bits, integers, or floating-point numbers. The manner in which the solution is encoded, meaning the numerical type, ordering, and what each value represents, is known as the *representation*. Each individual has a *fitness*, which represents the solution quality of that individual. Every generation, a subset of the population is probabilistically chosen to be the parents of the next generation through a process called *selection*. The genetic operators are the mechanisms through which the individuals are changed; usually these consist of the operators of *crossover* and *mutation*. The fitness of an individual is recalculated whenever the encoded information of the individual is modified.

3.1.1 Representation and Initialization

We use a representation of a single array of floating-point numbers in the range [0.0, 1.0]. Initially, every individual of the population is randomly generated, meaning that all values in the individuals are assigned a random number within the allowable range. The length of the encoded data depends on the number of dimensions in the problem, as each dimension requires one or more encoded values.

3.1.2 Fitness

Fitness refers to a "score" of each individual in a population that indicates the quality of the individual's solution to the problem. Fitness is calculated by running the data encoded within an individual through a *fitness function*, where the fitness function is the equation that represents the problem. Fitness can be set up such that either a lower fitness is superior, or a larger fitness is superior, depending on which makes the most sense for the problem at hand. Fitness is calculated for every individual once at the start of the algorithm. After this initial fitness evaluation, an individual's fitness is recalculated whenever the individual is modified through the genetic operators.

If an individual is unchanged from one generation to the next, then there is no need to recalculate its fitness as the fitness function will return the same result. Note that for dynamic problems an individual may require a fitness recalculation even when the individual remains unchanged, as the problem itself may change. The overall goal of a GA is to see a steady improvement in the fitness of the individuals over time until the optimal solution is found or the algorithm terminates. At the conclusion of the algorithm's execution, the encoded data within the individual with the best fitness is taken as the solution.

3.1.3 Parent Selection

Selection is the process BY which the parents of the next generation of individuals are determined and is how the "survival of the fittest" aspect of evolution is applied to a GA. The probability for an individual to be selected is directly tied to their fitness. Individuals with superior fitness have a higher probability of being selected than those with worse fitness. There are multiple implementations for the specific way in which the superior fitness individuals are given higher selection probabilities. The method we use is called *tournament selection*. Tournament selection is a very commonly used selection method for GAs. In tournament selection, two or more individuals from the entire population are randomly chosen to participate in a tournament. the individual in the tournament with the best fitness is selected as a parent. There are variants of tournament selection in which the winner of a tournament is selected probabilistically; CGA uses the more common method where the tournament winner is selected deterministically. The tournaments repeat until enough parents have been selected. An individual may be selected as a parent multiple times, in which case it appears in the list of parents multiple times. As a result, there will be more offspring descended from that particular parent.

3.1.4 Genetic Operators: Crossover and Mutation

Crossover is a genetic operator that simulates reproduction. In crossover, two or more individuals are mixed together to form new individuals whose encoded information is a mixture of their parents' information. The basic idea of crossover is that different individuals will have different useful pieces of information. Through crossover; these disparate pieces of useful information can be combined into a single individual whose fitness is higher than the fitness of either parent. This combination of useful pieces of information is of course not guaranteed and often the opposite can happen where a child is worse than either parent. When combined with the survival of the fittest aspect of selection however, the GA can exploit the beneficial children often enough such that the average population fitness improves over time. Crossover is controlled by a parameter known as *crossover rate*, which is the probability that two parents will undergo crossover.

Mutation is a genetic operator that simulates random genetic mutations in nature. In mutation, each discrete piece of data in an individual has a small chance to change to a new value. The chance for mutation to occur is called the *mutation rate*. The manner in which mutation changes a value is usually stochastic in nature.

Crossover and mutation have many different implementations, called *operators*, from which to choose. The choice of which operators to use for crossover and mutation is an important design decision that can greatly affect a GA's performance on a given problem. Operators determine how a GA searches through a problem's search space. The various existing operators are designed for varying goals, search behaviors, representations, and problems. There are many available operator options to choose from and selecting which operator to use is a difficult problem in and of itself. In order to allow for fair comparisons, the operators we use for CGA are the same operators that are a part of SAGA.

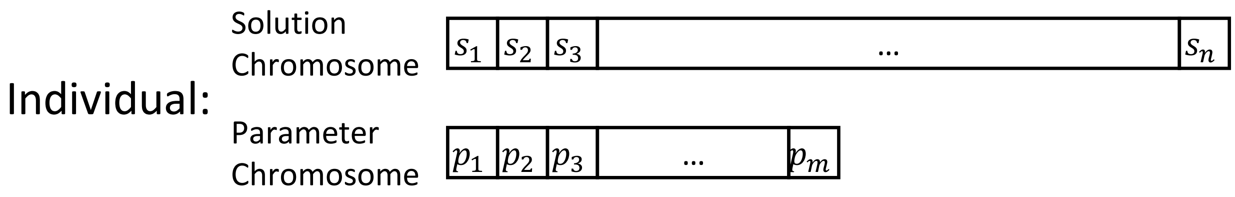

Figure 3.1: Illustration of the two chromosome individual of SAGA.

3.2 Self-Adaptive Genetic Algorithm

To create SAGA, we modify a CGA to add in self-adaptive parameters. The five self-adaptive parameters we examine are: parent selection pressure, mutation rate, crossover rate, mutation operator selection, and crossover operator selection. We first describe the representation details of SAGA, as now each individual must encode both the problem's solution as well as the self-adaptive parameters. We then describe the three components of a GA to which we apply self-adaptation. The first is self-adaptation of parent selection, followed by self-adaptation of the mutation rate and crossover rate, and last is the self-adaptation of mutation and crossover operator selection. The last component, operator selection is the most complex as it is the largest divergence from a CGA.

3.2.1 Encoding of Self-adaptive Parameters

Each individual in SAGA must encode both the solution to the problem as well as the self-adaptive parameters. Thus, the data structure of an individual in SAGA differs from and individual in a CGA. This extra encoded data is also a contributing factor in our decision on what representation to use for SAGA. Finally, we address the issue of encoding order bias, wherein the order in which data is encoded can effect the evolutionary process.

Figure 3.1 shows the data structure of an individual in SAGA. SAGA uses a two *chromosome* structure, where a chromosome is an array of data. The first chromosome is the *solution chromosome*, which, as the name implies, encodes the solution to the problem. The fitness of an individual is calculated using only the solution chromosome. The second chromosome is the *parameter chromosome*, which encodes all of the self-adaptive parameters. The contents of both chromosomes are modified by the genetic operators. In the CGA, an individual consisted of a single chromosome, thus the extra terminology of "chromosome" was essentially redundant with "individual". Now with two chromosomes, however, "chromosome" and "individual" have two distinct meanings. The solution chromosome of SAGA is identical to the single chromosome of CGA.

SAGA uses a floating-point representation for both chromosomes in the range of [0.0, 1.0]. We use this consistent representation for all values in both chromosomes so that the same set of operators can act equivalently on both chromosomes. Recall that many operators only work on certain data types. Allowing different data types on either chromosome would increase the complexity of the algorithm; either the operators would have to be modified to ensure appropriate behavior on all of the included data types or a completely unique set of operators would be needed for each chromosome. Either option is unattractive when our focus is on improving ease of use. We chose floating-point values in the range of [0.0, 1.0] as the consistent data type because it is a natural fit for how the self-adaptive operators are encoded and can be mapped to any other data type of any range.

A final question to consider is the encoding order within the parameter chromosome. Research in GAs has found that the ordering of information in a chromosome can cause inherent bias in the evolutionary progression Wu and Garibay 2002. In SAGA, the ordering of the parameters within the parameter chromosome is irrelevant to the actual usage of the parameters. It is possible, however, that position bias within the parameter chromosome could have a noticeable effect on the evolution of these parameters. To account for this possibility, we randomize the encoding order of the parameter chromosome every run.

3.2.2 Self-Adaption of Parent Selection Pressure

We apply self-adaptation to the *selection pressure* of parent selection. Selection pressure signifies the probability that the superior individuals in the population will be selected for crossover. A higher selection pressure means a higher probability of the superior individuals being selected. A high selection pressure, therefore, results in a larger emphasis on exploitation, while a lower selection pressure results in a larger emphasis on exploration.

SAGA uses tournament selection for parent selection, the same as CGA. Tournament selection has a simple way to adjust the selection pressure through changing the *tournament size*, where the tournament size is the number of individuals participating in the tournament. In tournament selection, a larger tournament size results in a higher selection pressure. Thus we apply self-adaptation to the tournament size of tournament selection. Each individual's parameter chromosome encodes the *tournament weight* which is a floating-point value between 0.0 and 1.0. Each tournament is given a maximum weight limit of 2.0. Individuals are sequentially added to the tournament until the summed weight of the individuals in the tournament surpasses the weight limit. The winner of the tournament is deterministically selected as the individual with the best fitness.

With this setup, individuals with lower weights lead to larger tournament sizes and therefore higher selection pressures. The weight limit of 2.0 ensures that there will always be at least 2 individuals in the tournament. On initialization of the algorithm, the weights are not distributed randomly in the allowed range of [0.0, 1.0]; rather, they are distributed in a Gaussian distribution centered at 1.0 with a standard deviation of 0.1. With this initialization, the population will start with a lower selection pressure, with most tournaments having a size of 2. The population is then free to evolve to higher selection pressures over the course of the algorithm's execution.

3.2.3 Self-Adaption of Crossover and Mutation Rate

Each individual's parameter chromosome encodes a single value for the mutation rate parameter and a single value for the the crossover rate. The mutation rate is encoded as a floating-point value in the range [0.0, 1.0]. The encoded mutation rate directly represents the probability that each value within the chromosomes of that individual will mutate. Crossover is also encoded as a single floating-point value in the range [0.0, 1.0] Two or more individuals are required for crossover, so the probability of crossover between a pair of potential parents is the average of both parents' encoded crossover rates.

3.2.4 Self-Adaption of Crossover and Mutation Operator Selection

Self-adaptive *operator selection* is our largest deviation from the CGA. Unlike in a CGA, where a single crossover operator and a single mutation operator are used throughout the GA's execution, in SAGA, multiple operators can be used for both crossover and mutation. Operator selection is the process to determine which specific operator to use whenever mutation or crossover is due to occur. Operator selection occurs independently on every crossover and mutation operation. We describe how operator selection is encoded for both mutation and crossover, which operators we use for mutation and crossover and why, and finally we discuss how we make the probabilistic selection for operator selection and why we introduce a new parameter for the probabilistic selection.

3.2.4.1 Encoding of Self-Adaptive Crossover and Mutation Operators

Each of the possible mutation and crossover operators have one or more encoded values in the parameter chromosome of each individual. This encoding is very similar for both mutation and crossover with only a slight difference for crossover. We also address how SAGA handles operator arguments, as it is common for genetic operators to have one or more arguments that control how the operator behaves.

Each potential mutation operator has an encoded floating-point *usage rate* value in the range [0.0, 1.0] in the parameter chromosome. Table 3.1 lists the mutation operators considered in this study. Each time a mutation operation is to occur, one of the mutation operators is selected. Operators with a higher usage rate have a higher probability of being selected than operators with lower usage rates.

Each potential crossover operator has an encoded usage rate value in the range [0.0, 1.0] in the parameter chromosome. Table 3.1 lists the crossover operators considered in this study. Crossover requires two or more parent individuals, thus, each time a crossover operation is to occur, a crossover operator is probabilistically selected using the parents' averaged crossover operator usage rates. Operators with a higher averaged usage rate have a higher probability of being selected than operators with lower averaged usage rates.

Table 3.1: Mutation and crossover operators with their required arguments.

	Operator	Number of Arguments
Mutation	Uniform	0
	Gaussian	1
	Polynomial K. Deb and D. Deb 2014	1
	Swap	0
Crossover	Uniform K. Deb and Agrawal 1995	0
	Arithmetic Michalewicz 1992	1
	Linear Wright 1991	0
	Simulated Binary (SBC) K. Deb and Agrawal 1995	1
	Blend Eshelman and Schaffer 1993	1
	Simplex Tsutsui, Yamamura, and Higuchi 1999	1
	Parent-Centric (PCX) K. Deb, Joshi, and Anand 2002	2

Some crossover and mutation operators require one or more extra arguments regardless of whether the GA is self-adaptive or canonical. Gaussian mutation, for example, requires an argument for standard deviation to determine the spread of the Gaussian distribution. In a CGA, these arguments are another value to be manually tuned. In SAGA, additional floating-point values in the range [0.0, 1.0] are encoded for each additional argument in the parameter chromosome. The encoded value for an argument represents the value of the argument. If the allowable range of an argument is different than the encoded value range of 0.0 to 1.0, then the encoded value is directly mapped to the required range upon usage of that argument.

3.2.4.2 Descriptions of Mutation and Crossover Operators

There exists a large number of operators for mutation and crossover and we can only include so many in a single study if we want to be able to effectively analyze their results. We select a set of mutation and crossover operators to cover a variety of operator behaviors. We give a brief description of each of the operators as well as the reasoning behind their inclusion.

As seen in table 3.1, we include four mutation operators in our system:

- Uniform random: A value is replace by a new value that is randomly chosen from within the range of allowable numbers. This operator has a relatively high chance of a large change in value.
- Gaussian: A value is replaced by a new value randomly chosen from a Gaussian distribution centered on the current value. This operator has a relatively middling probability of a large change in value.
- Polynomial: A value is replaced by a new value randomly chosen from a polynomial distribution centered on the current value. This operator has a relatively small probability of a large change in value. Very similar to Gaussian, but with a distribution that is more likely to create a small change in value.

- Swap: A value is swapped with another value selected randomly from within the same chromosome. This operator is included in order to have a mutation operator that is fundamentally differently from the other mutation operators; it does not create new values as the others do; instead it re-arranges the existing data within a chromosome.

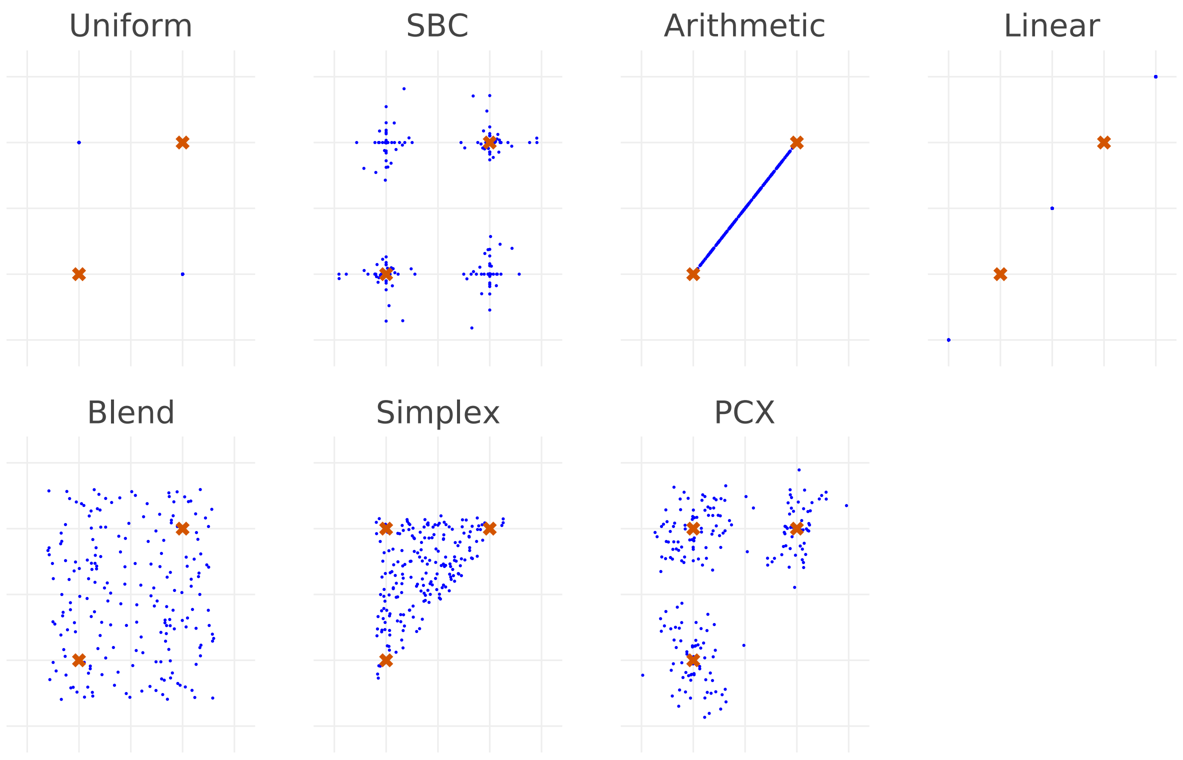

Figure 3.2: Plots of the children created from each crossover operator.

As seen in table 3.1, we include seven crossover operators in our system. Figure 3.2 provides a visualization of each crossover operator's behavior. These plots show, in a two-dimensional space, 100 children generated from the same set of parents for each crossover method. Parents are designated by an orange X and children are designated by blue dots. Each of these crossover operators generate two children and most use two parents though a couple use three parents. The seven crossover operators are:

- Uniform: A very simple crossover operator originally designed for binary representation. For each value in a child individual, one of the two parents is randomly selected with equal probability and the value is copied from that parent. Uniform crossover therefore does not create new values in floating-point representation, but instead mixes the existing data contained in the two parents.

- Simulated Binary Crossover (SBC): SBC is designed to have a similar search power on floating-point representation as uniform crossover has on binary representation. As such, unlike directly using uniform crossover, SBC can create new values in the chromosome while still retaining similar behavior to uniform crossover.

- Arithmetic: A simple floating-point operator that creates children whose encoded floating-point values are weighted averages of the parents' values. Only the two children with the highest fitness are kept and the third child is discarded.

- Blend: Blend crossover has a very large area in which offspring can be generated within a N-dimensional rectangular prism, where N is the number of dimensions to the problem. The parents define opposite corners of this rectangular prism, with the bounds going somewhat past the location of the parents so as to allow for more than only mean-focused children.

- Simplex: Simplex crossover is similar to blend but uses three parents to generates a 2-simplex, or triangle, with a parent at each each vertex of the triangle and from within which the children are uniformly distributed. Similar to Blend, the bounds of the simplex extend slightly past the location of the parents.

- Parent-Centric Crossover (PCX): As the name implies, this operator creates children that are near to the parents. Similar to simplex, PCX uses three parents. A mean point between the three parents is calculated and the relative distance and positioning of each parent from this central point controls how far from the parents an offspring can appear.

3.2.4.3 SAGA Operator Selection method

As stated earlier, every time crossover or mutation occurs in SAGA, a probabilistic selection is made to determine which operator will be used, which we call operator selection. There are many ways in which to make this probabilistic operator selection. We tested multiple probabilistic selection methods on multiple problems and found that which operator selection method performs best is problem specific. Because of these findings, we decided to treat the operator selection method as a new parameter introduced by SAGA that must be manually-tuned per problem. The decision to introducing a new parameter is not taken lightly since our goal is the reduction of parameter tuning. As will be elaborated on in the experiments, however, we find that the fitness payoffs are worth this tuning cost, and that the tuning cost of SAGA is still significantly lower than CGA. Future work could potentially turn this operator selection method into a self-adaptive process as well.

The probabilistic selection methods we use for operator selection are all derived from common parent selection methods, seeing as how both operator selection and parent selection are both similar probabilistic selections. The methods we use are:

- Best: the operator with the highest usage rate is deterministically selected.
- Random: the operator is selected randomly. We include random selection to act as a benchmark, as it has been shown that pure random selection alone can be beneficial for operator selection Karafotias, A. E. Eiben, and Hoogendoorn 2014.
- Proportional: Can be thought of as spinning a roulette wheel where each operator has a segment of the wheel. The size of each operator's segment, and therefore it's probability to be selected, is directly correlated to the usage rate. If operator A has a usage rate that is double that of operator B, then usage A is twice as likely to be selected as operator B.

- Linear Rank: Similar to proportional selection and can use the same roulette wheel analogy. The difference is in how the sizes of each operator's segment of the wheel is determined. Each operator is assigned a unique rank based on its usage rate. Then, the size of each segment on the roulette real is determined through the rank (1, 2, 3, etc...) of the operator, rather than its absolute usage rate, where rank 1 is the lowest rank.

- Tournament: A tournament of 2 or more operators is randomly selected. Here, the winner of a tournament is decided by the operators' usage rates. We test out multiple variants of tournament selection covering a range of tournament sizes and tournament winner probabilities:

  - Tournament Size: 2, 3, 4, all
  - Winner Probability: 1.0, 0.9, 0.8

  Where "all" designates a tournament size that includes every available operator (four for mutation and seven for crossover) and winner probability is the probability that the highest scoring choice will win. In the case of a winner probability < 1.0 and the highest scoring choice does not win, then the highest scoring choice is removed from the tournament tournament is repeated with the remaining participants. This process continues recursively until a winner is selected or there is only a single individual left in the tournament, in which case it is the winner by process of elimination. We do not run tournament with size = all and winner probability = 1.0, as this is equivalent to Best selection, which is already included. Thus, in total we have 11 tournament selection variants.

These plots each have generations on the x-axis and mean number of usages over 50 runs of an operator on the y-axis. Each crossover method includes a 95% confidence interval. We show these plots now, not to go into an in-depth analysis of results here, but simply to provide illustration for the reasoning behind our decision to introduce this new parameter. Within these plots, it can clearly be seen that there is a large variety in the behaviors that arise out of the different operator selection methods. Figure 3.3 is organized such that the plots towards the top have lower selection pressures than the plots below. In general, it can be seen that higher selection pressures lead to certain operators standing out much more than others, while low selection pressures lead to little or no distinction in the usage of the different operators.

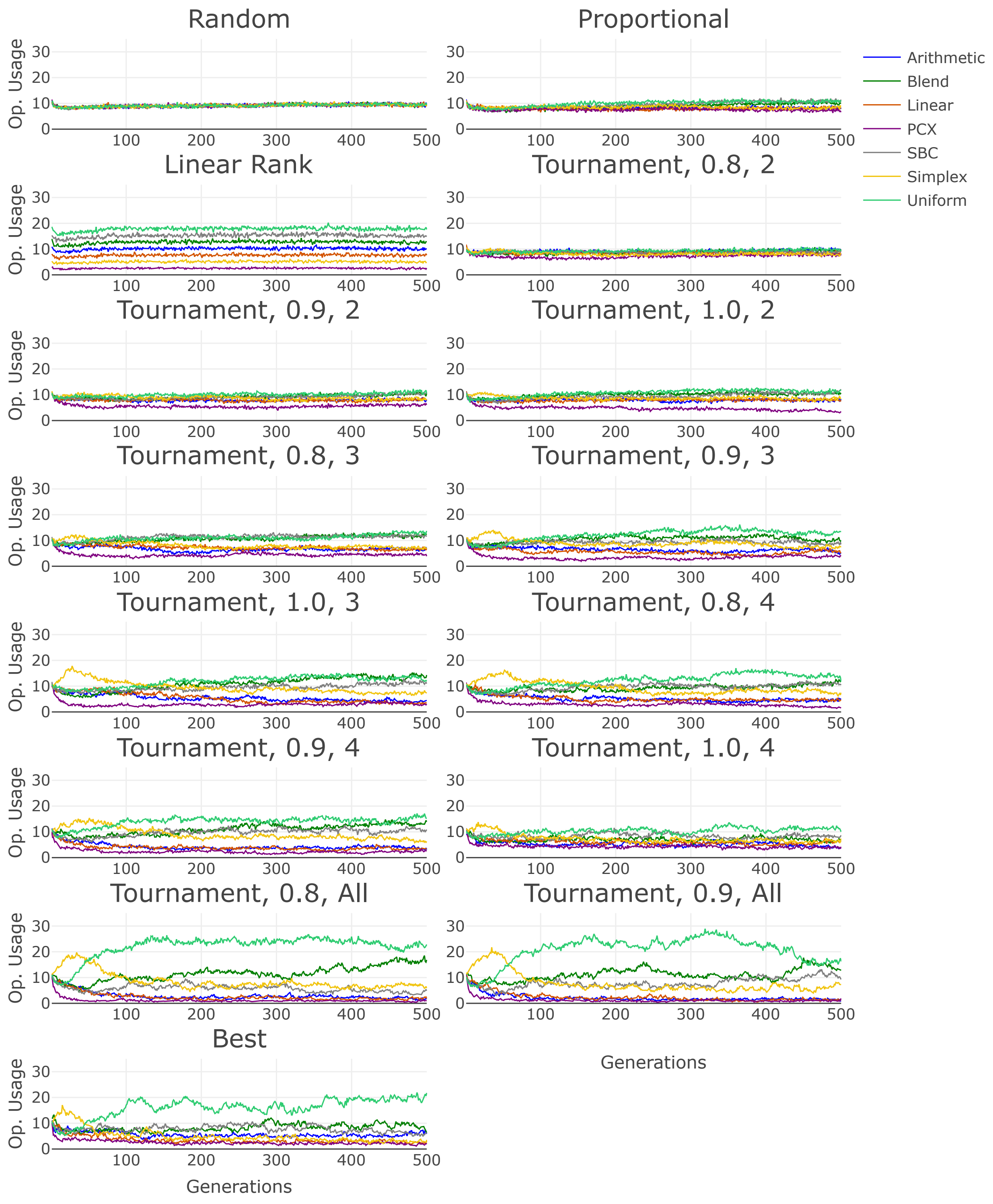

Figure 3.3: SAGA crossover operator usages from different operator selection methods.

CHAPTER 4: STATIC PROBLEMS

In this chapter, we test SAGA on a set of static problems. This chapter answers the most immediate question of how SAGA compares to a CGA on common benchmark problems. In the experiments of this chapter, we compare both the fitness results and the durations of the algorithms as measured by the number of fitness evaluations. We also want to understand how SAGA is adapting to these problems, so we conduct an analysis of the self-adaptive parameters of SAGA on each problem.

4.1 Motivation

In this experiment, we test the primary goal of SAGA; can it greatly reduce the amount of manual parameter tuning, and therefore time investment, required of the user whilst maintaining at least equivalent fitness results as a well-tuned CGA?

We compare the performance of SAGA and CGA on a set of static benchmark problems. We use a set of static benchmark problems to act as a comparison of SAGA and CGA in order to answer this question. The comparison is against a CGA because we wish to conduct an "apples-to-apples" comparison, as opposed to an "apples-to-oranges" comparison against other evolutionary methods, such as evolutionary strategies, that are known to perform well on problems in our test suite but are not directly comparable to SAGA. SAGA is built upon a CGA; therefore, comparing to the CGA shows the changes that are directly attributed to the self-adaptive mechanisms of SAGA. Furthermore, our problem suite includes multiple types of problems, both continuous and discrete, such that there is not one single EA that is known to be best on all of our problems. Thus, if we were to compare against the best known EA of each problem, the study would instead end up as a broad comparison of many different EAs, instead of a focused study on SAGA. Our focus for this study is on the direct improvements that the addition of self-adaptation can have upon a GA, particularly on usability and runtime as measured in the number of fitness evaluations executed by the algorithm.

4.2 Experimental Setup

We test SAGA and CGA on a suite of static problems. We begin by describing our test suite and how each problem is represented by the GAs. Next we give the parameter setups for both CGA and SAGA and detail the number of *configurations* that each will execute on each of the problems, where we define a configuration as a unique set of parameter values.

4.2.1 Problems

We are not designing SAGA with an eye towards any particular type of problem, so a suite of multiple benchmark problems is assembled with the goal of covering a variety of problem types. Our problems are selected with a focus on facilitating deep and useful analysis on SAGA's behaviors on a variety of problems. Thus, we do not do test against a massive set of problems that are all within the same domain, as is often the case in studies that focus on testing the performance of a particular algorithm on a particular category of problem. Our suite contains six problems and is comprised of both continuous domain problems and discrete domain problems. The discrete problems have an added complication of requiring a mapping from SAGA's floating-point representation, which is explained for each of the discrete problems.

Table 4.1 shows the problems in our static problem test suite. The table lists the problems' domain of either continuous or discrete, the number of dimensions, the optimal fitness value, and any noteworthy characteristics of the problem. All problems except Knapsack are minimization problems.

The continuous problems are relatively straight forward; each one corresponds to a mathematical function over n-dimensional space. We cover a variety of continuous problem landscapes.

Table 4.1: Static test problems.

Problem	Domain	Dimensions	Characteristics	Optimal
Sphere	Continuous	50	Unimodal, separable	0.001
Ackley	Continuous	50	Multimodal, non-separable	0.001
Rosenbrock	Continuous	50	Unimodal, non-separable	0.001
0-1 Knapsack	Discrete	100	Uncorrelated weights, values	9147
N-Queens	Discrete	100		0
TSP	Discrete	52	Berlin52	7542

- The Sphere problem is continuous, uni-modal, and separable and is expected to be a trivial problem. The Sphere problem is defined by the function:

$$f(x) = \sum_{i=1}^{n} x_i^2 \tag{4.1}$$

  Where $f(x)$ is the fitness of at a location, n is the number of dimensions, and x is in the range [-5.12, 5.12]

- Ackley is a continuous and non-separable problem and more difficult than Sphere in that it is multi-modal. Ackley is defined by the function:

$$f(x) = -20exp(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}) - exp(\frac{1}{n}\sum_{i=1}^{n} cos(cx_i)) + 20 + exp(1) \tag{4.2}$$

  Where $f(x)$ is the fitness of at a location, n is the number of dimensions, and x is in the range [-32.768, 32.768]

- Rosenbrock is a much more difficult continuous problem; it is unimodal and non-separable with the optimum located in a narrow parabolic valley. Rosenbrock is defined by the function:

$$f(x) = \sum_{i=1}^{n} [100(x_{i+1} - x_i^2) + (1 - x_i^2)] \tag{4.3}$$

Where $f(x)$ is the fitness of at a location, n is the number of dimensions, x is in the range [-5.0, 10.0] The Rosenbrock problem is designed such that finding the valley is easy, but finding the optimum within the valley is difficult.

In addition to the continuous domain problems, we experiment with three discrete domain problems to test if SAGA can still be effective on problems that are not directly represented through SAGA's floating-point representation. The discrete problems require mappings from SAGA's representation of floating point values in the range [0.0, 1.0] to the discrete representations required. Recall that this mapping occurs at the time of and for fitness evaluation; the encoded values remain as floating-point values during the genetic operations. A bit more explanation is required for each of these problems to define how we handle this mapping.

- The 0-1 Knapsack problem is a discrete problem where a list of items with associated weights and values are given; the algorithm must then fit as many items within a hypothetical Knapsack with the goal of maximizing the collected items' value while staying underneath the Knapsack's weight limit. In our instance of the Knapsack problem, which is Pisinger (2005), the weights and values of items are uncorrelated. Knapsack requires a representation that denotes whether each of the N items should be included in the Knapsack or not included; a binary decision for each item. Therefore, we map all encoded floating-point values below 0.5 to 0 and any value above 0.5 to 1 where 0 denotes an item that is not in the Knapsack and 1 denotes an item that is in the Knapsack.
- N-Queens is a discrete problem where there exists $\frac{n}{2}$ chess queen pieces on an $\frac{n}{2}x\frac{n}{2}$ chess-

board, where n is the number of dimensions. There are N dimensions for $\frac{n}{2}$ queens because each queen requires two values, an x coordinate and a y coordinate, to denote the queen's location. The goal is to arrange the queens on the board such that no queen can move into another queen's space according to the rules of chess where a queen can move any distance of spaces horizontally, vertically, or diagonally. In the chromosome, the first value represents the first queen's x coordinate and the second value represents the first queen's y coordinate; this repeats down the chromosome for all $\frac{n}{2}$ queens. For N-Queens, each queen is mapped by multiplying the encoded floating-point number by $\frac{n}{2}$ and flooring the result to obtain an integer representing a discrete position along a dimension of the board.

- The travelling salesman problem (TSP) is a very common benchmark problem in many fields, wherein the goal is to transverse all n nodes in a network once in the ordering that minimizes the cost. The specific TSP instance we test is Berlin52, obtained from TSPLIB R. 1997. In this TSP problem, the network is fully connected and each node has a x and y coordinate; the cost of travelling from one node to another is the euclidean distance between the two nodes. For TSP, we use Random-Keys Bean 1994 for the mapping of the encoded floating-point data to discrete data. In Random-Keys, each floating-point value represents an associated integer n_i where n_i is the position of the floating-point number within in the chromosome, i.e. the first floating-point number represents 0, the next represents 1, and so on until the final floating-point number in the chromosome which represents $n-1$. For TSP, these integers denote each of the n nodes within the network. The integers are the sorted according to ascending order of their associated floating-point numbers. This sorted array of integers then represents the ordering of the nodes to traverse in TSP.

4.2.2 CGA Setup

Table 4.2 lists the parameters necessary for running CGA. In the table, the parameters that are to be manually tuned are those that list multiple values. The values that are to be tuned are the same values that SAGA self-adapts. We also elaborate on how exactly the parameter tuning is to be conducted.

For the tuning of CGA, we run an exhaustive combination of all of the crossover and mutation parameters listed in Table 4.2. In total, this results in 1,228 CGA configurations. We conduct an exhaustive combination of all these parameter values in order to find the best possible CGA configuration of the parameter values available. This best configuration can then be compared to SAGA.

Table 4.2: CGA parameters for static problems.

Parameter	Values
Population	200
Generations	500
Runs	50
Mutation Rate	[0.001, 0.01, 0.05]
Mutation Operator	[Uniform, Gaussian, Polynomial, Swap]
Mutation Argument Value	[0.05, 0.25, 0.5]
Crossover Rate	[0.4, 0.6, 0.8]
Crossover Operator	[Uniform, SBC, Arithmetic, Linear, Blend, Simplex, PCX]
Crossover Argument Value	[0.25, 0.5, 0.75]
Parent Selection Method	Tournament
Parent Selection Tournament Size	[2, 3, 4, 5, 6]

We do not include parent selection tournament size in the exhaustive combination of parameter values for the simple reason that it would increase the number of configurations by a factor of 5, resulting in 6,120 configurations per problem, which is prohibitively large for our experiments. Rather, we tune the 1,228 configurations with a parent selection tournament size of 2; then we take the best configuration resulting from that round of tuning and tune this one configuration on the different parent selection tournament sizes. In total then, there are 1,228 CGA configurations. This decision we had to make for parent selection tournament size is a perfect example of the issues we are attempting to address in this study, where we as the user have to make a trade-off between full tuning and practical limits of computational resources and time.

Testing such a large number of CGA configurations may be considered excessive. SAGA, however, covers a larger range of potential parameter values than is covered by the 1,228 CGA configurations. Both have coverage of the same operators, but the rates and argument values used in the CGAs are limited to only 3 instances of their allowable ranges. SAGA can evolve these values over their entire continuous range. It can be argued that our set of operators and parameter values could be trimmed based on knowledge of what generally works well for GAs and for our test problems; however, we are focused on making GAs more accessible for those who would be lacking this prior knowledge or on novel problems where this prior knowledge does not yet exist. Therefore, we do not assume this prior knowledge to trim our set of parameter values.

4.2.3 SAGA Setup

Table 4.3 lists the parameters for necessary for SAGA. The only parameter of SAGA that is tuned per problem is the operator selection method. The parameter tuning for SAGA is very simple; there are fifteen operator selection methods, so we run a total of fifteen SAGA configurations on every problem.

Table 4.3: SAGA parameters for static problems.

Parameter	Values
Population	200
Generations	500
Runs	50
Operator Selection Method	[Random, Proportional, Linear Rank, Tournament x 11, Best]
Parent Selection Method	Tournament

4.3 Tuning For the Best Parameter Configurations

Before we can make comparisons in the results of CGA and SAGA, we must first determine the best parameter configuration for each algorithm on every problem. In other words, we must tune the parameters. In this section, we detail the outcomes of our tuning process of both CGA and SAGA and list the best CGA and SAGA configurations.

Note that our primary evaluation metric for the fitness of a configuration is *median fitness*, although we also list the *mean fitness* and *best fitness*. Where median fitness is the median of the fitnesses of the best individual across all 50 runs of a particular configuration, mean fitness is the mean of the fitnesses of the best individual across all 50 runs, and best fitness is the single best fitness result in all 50 runs. While mean is the standard fitness metric for GAs, we instead use median fitness as our primary fitness metric, because SAGA has many more outliers than CGA, leading to skewed means. Thus, we find median to be a more reliable metric in these experiments. We use median fitness for the tuning of CGA as well in order to ensure a fair comparison by using the same primary fitness metric on the tuning process of both algorithms.

4.3.1 CGA Parameter Tuning

The tuning of CGA is conducted in two phases, as described in Section 4.2.2. In the first phase we tune 1,228 configurations, where each configuration uses a parents election tournament size of 2. In the second phase, we take the best configuration from the first phase, and tune the parent selection tournament size.

Table 4.4 lists the best configuration found in the first phase of CGA tuning. The table lists the median fitness, mean fitness with 95% confidence interval, best fitness, and best crossover and mutation parameter configuration. The two parameter configuration columns list the name of the operator, the rate, and any operator argument values (when applicable), in that order. In this table, we can see that which configuration performs best is problem-dependent. Swap mutation with a high mutation rate of 0.05 is favored on the continuous problems, while Gaussian mutation with a low mutation rate of 0.001 is favored on the discrete problems. For crossover, there is a mix of SBC, Blend, and Uniform crossover. A high crossover rate of 0.8 is always preferred except on Rosenbrock, which uses a rate of 0.4.

Table 4.4: CGA Parameter Tuning: Best configuration for each problem with a parent selection tournament size of 2.

Problem	Fitness			Parameter Configuration	
	Median	Mean	Best	Mutation	Crossover
Sphere	4.93e-16	6.25e-16 $\pm$ 1.78e-16	1.06e-16	Swap, 0.05	SBC, 0.8, 0.75
Ackley	2.25e-8	2.30e-8 $\pm$ 2.17e-9	7.54e-9	Swap, 0.05	SBC, 0.8, 0.75
Rosenbrock	5.04	18.59 $\pm$ 9.06	3.74e-5	Swap, 0.05	Blend, 0.4, 0.25
Knapsack	9147	9046 $\pm$ 32	9147	Gaus, 0.001, 0.5	SBC, 0.8, 0.25
TSP	8476.94	8567.32 $\pm$ 97.37	8051.35	Gaus, 0.001, 0.25	Blend, 0.8, 0.5
N-Queens	4	4.32 $\pm$ 0.32	2	Gaus, 0.001, 0.5	Uniform, 0.8

Table 4.5 shows the second phase of CGA tuning, where parent selection tournament size is tuned. This phase of CGA tuning uses the best configuration of mutation and crossover parameters that was found in the first phase and is listed in Table 4.4. Table 4.5 lists the results of each parent selection tournament size in each generation for each problem. The best configuration for each problem, as determined by median fitness, is highlighted in bold. In this table, we see that the parent selection tournament size parameter is also problem specific. Rosenbrock, Knapsack, and TSP find best results with a tournament size of 2, while Sphere, Ackley, and N-Queens find the best results with a larger tournament size of 4.

For N-Queens we actually see a tie of median fitnesses on all parent selection tournament sizes above 2. While median fitness is our primary metric, we use the other fitness metrics as tie-breakers. Even when using the other fitness metrics, there is not a significant difference between parent selection tournament sizes of 4 and 6; both have equal median fitness and best fitness, and the mean fitnesses are not significantly different as shown by the overlapping 95% confidence intervals. Regardless, we only need a single configuration for the comparison to SAGA, so we select the parent selection tournament size of 4.

Table 4.5: CGA Parameter Tuning: Parent selection tournament size configurations.

Problem	Tourn Size	Fitness		
		Median	Mean	Best
Sphere	2	4.93e-16	6.25e-16 ± 1.78e-16	1.06e-16
	3	8.06e-24	1.43e-23 ± 4.09e-24	6.31e-25
	4	**4.35e-26**	**4.35e-26 ± 2.00e-27**	**2.89e-26**
	5	4.38e-26	4.39e-26 ± 1.87e-27	3.08e-26
	6	4.56e-26	4.48e-26 ± 1.57e-27	3.01e-26
Ackley	2	2.25e-8	2.30e-8 ± 2.17e-9	7.54e-9
	3	3.87e-12	4.28e-12 ± 7.54e-13	1.36e-12
	4	**3.70e-13**	**3.72e-13 ± 6.44e-15**	**3.17e-13**
	5	3.75e-13	3.73e-13 ± 8.10e-15	3.02e-13
	6	3.86e-13	3.87e-13 ± 5.26e-15	3.01e-26
Rosenbrock	**2**	**6.39**	**17.76 ± 9.03**	**1.08e-4**
	3	17.30	51.79 ± 21.41	0.02
	4	21.91	48.21 ± 18.18	0.02
	5	41.53	66.09 ± 19.10	0.40
	6	86.91	98.63 ± 23.88	0.01
Knapsack (max)	**2**	**9147**	**9046 ± 32**	**9147**
	3	8929	8906 ± 48.46	9147
	4	8929	8920 ± 54.12	9147
	5	8900	8874.60 ± 52.00	9147
	6	8905.50	8814.10 ± 99.98	9147
TSP	**2**	**8476.94**	**8567.32 ± 97.37**	**8051.35**
	3	8724.91	8750.13 ± 105.47	7846.48
	4	10556.99	9145.60 ± 393.63	8305.41
	5	11080.38	11408.87 ± 282.28	9130.34
	6	11028.23	11395.97 ± 214.74	8923.54
N-Queens	2	4	4.32 ± 0.32	2
	3	3	3.18 ± 0.23	2
	4	**3**	**2.62 ± 0.22**	**1**
	5	3	3.02 ± 0.24	1
	6	3	2.76 ± 0.23	1

4.3.2 SAGA Parameter Tuning

The tuning of SAGA is much simpler and quicker than CGA. SAGA has only fifteen parameter configurations to test on each problem. The only parameter to tune for SAGA is the operator selection method.

Table 4.6 lists the best operator selection method for each problem alongside the fitness results. In the case of tournament selection the tournament's winner probability and tournament size are also listed in that order. Tournament is the best operator selection method on all static problems, though different variations of tournament selection with different winner probabilities and tournament sizes. Random selection is outperformed on all problems, showing the probabilistic selection methods are indeed more beneficial than the simple random selection benchmark.

Table 4.6: SAGA Parameter Tuning: Best configuration for each problem.

Problem	Fitness			Operator
	Median	Mean	Best	Selection Method
Sphere	1.61e-15	4.12e-4 ± 8.19e-4	3.16e-21	Tournament, 0.8, all
Ackley	1.70e-8	4.78e-3 ± 9.28e-3	1.22e-10	Tournament, 0.9, all
Rosenbrock	9.79	29.43 ± 12.11	0.29	Tournament, 0.8, 4
Knapsack	8929	8917.38 ± 68.07	9147	Tournament, 0.9, 3
TSP	9735.72	10588.22 ± 788.28	8343.10	Tournament, 0.8, 2
N-Queens	5	5.73=2 ± 0.69	1	Tournament, 1.0, 2

4.4 Results

Now that the best parameter configurations have been found for CGA and SAGA, we can compare the results of the two algorithms. We conduct two experiments comparing CGA and SAGA. The first compares the two algorithms at equal generations and the second allows SAGA to run for an increased number of generations. We justify this increased number of generations through the time savings obtained by reducing parameter tuning. These results compare fitness and computation time through the metric of number of fitness evaluations.

4.4.1 Equal Generations

This experiment compares SAGA to CGA at an equal number of generations for both GAs. The purpose of this experiment is twofold. The first is to determine if the fitness results of SAGA can match the best-tuned CGA configuration when compared at an equal number of generations. The second is to compare the number of fitness evaluations of the two GAs when run at an equal number of 500 generations in order to compare the time investment and computational cost for the tuning of each algorithm. We find that the best CGA configuration finds better fitness results than the best SAGA configuration at 500 generations; however, we take a look at what percent of CGA configurations are capable of outperforming SAGA in order to see how necessary the extensive tuning of CGA is to its performance.

4.4.1.1 Fitness Comparison

Table 4.7 shows the fitness results of the comparison of the best SAGA configuration to the best CGA configuration when each is run at an equal number of 500 generations. The table shows, for both GA types, the median fitness, mean fitness with a 95% confidence interval, and best fitness. The data shown in this table comes from Tables 4.5 and 4.6 for the CGA and SAGA data, respectively. The bolded values denote the superior values.

In this experiment, the best CGA configuration finds superior fitness results than SAGA on nearly all metrics. There are only three instances where SAGA is not clearly outperformed by CGA in this table. On Knapsack, SAGA ties with CGA the best fitness value at 9147, which is the optimal for the problem, on Rosenbrock, There is overlap of the 95% confidence intervals of the mean fitnesses of the two algorithms, and on N-Queens, there is a tie of best fitness at 1, just shy of the optimal value of 0.

Figure 4.1 shows the data from Table 4.7 in box plot form. The figure contains one plot per problem. On each plot, the blue box is the best CGA configuration and the green box is the best SAGA configuration. Each box plot shows the median, interquartile range, inner and outer fences, and outliers. Showing this data in box plot illustrates our reasoning for using median fitness as the primary performance metric. SAGA finds a much larger number of outliers than CGA on the Sphere, Ackley, Knapsack, and TSP problems. CGA uses the same exact static parameters every run, leading to more consistency across runs. SAGA, on the other hand, does not evolve the same parameter settings every run, leading to more diversity in the fitness results from run to run. Median fitness, therefore, is a more reliable metric than mean fitness to compare the two algorithms, because mean fitness is skewed by the outliers. These results do not appear promising for SAGA; however, as further analysis will reveal, the results are not as CGA favored as they first appear.

Table 4.7: Comparison of the fitness results of SAGA and CGA at 500 generations.

Problem	Algorithm	Fitness		
		Median	Mean	Best
Sphere	CGA	**4.35e-26**	**4.35e-26 ± 2.00e-27**	**2.89e-26**
	SAGA	1.61e-15	4.12e-4 ± 8.19e-4	3.16e-21
Ackley	CGA	**3.70e-13**	**3.72e-13 ± 6.44e-15**	**3.17e-13**
	SAGA	1.70e-8	4.78e-3 ± 9.28e-3	1.22e-10
Rosenbrock	CGA	**6.39**	**17.76 ± 9.03**	**1.08e-4**
	SAGA	9.79	**29.43 ± 12.11**	0.29
Knapsack (max)	CGA	**9147**	**9046 ± 32**	**9147**
	SAGA	8929	8917.38 ± 68.07	**9147**
TSP	CGA	**8476.94**	**8567.32 ± 97.37**	**8051.35**
	SAGA	9735.72	10588.22 ± 788.28	8343.10
N-Queens	CGA	**3**	**2.62 ± 0.22**	**1**
	SAGA	5	5.73=2 ± 0.69	**1**

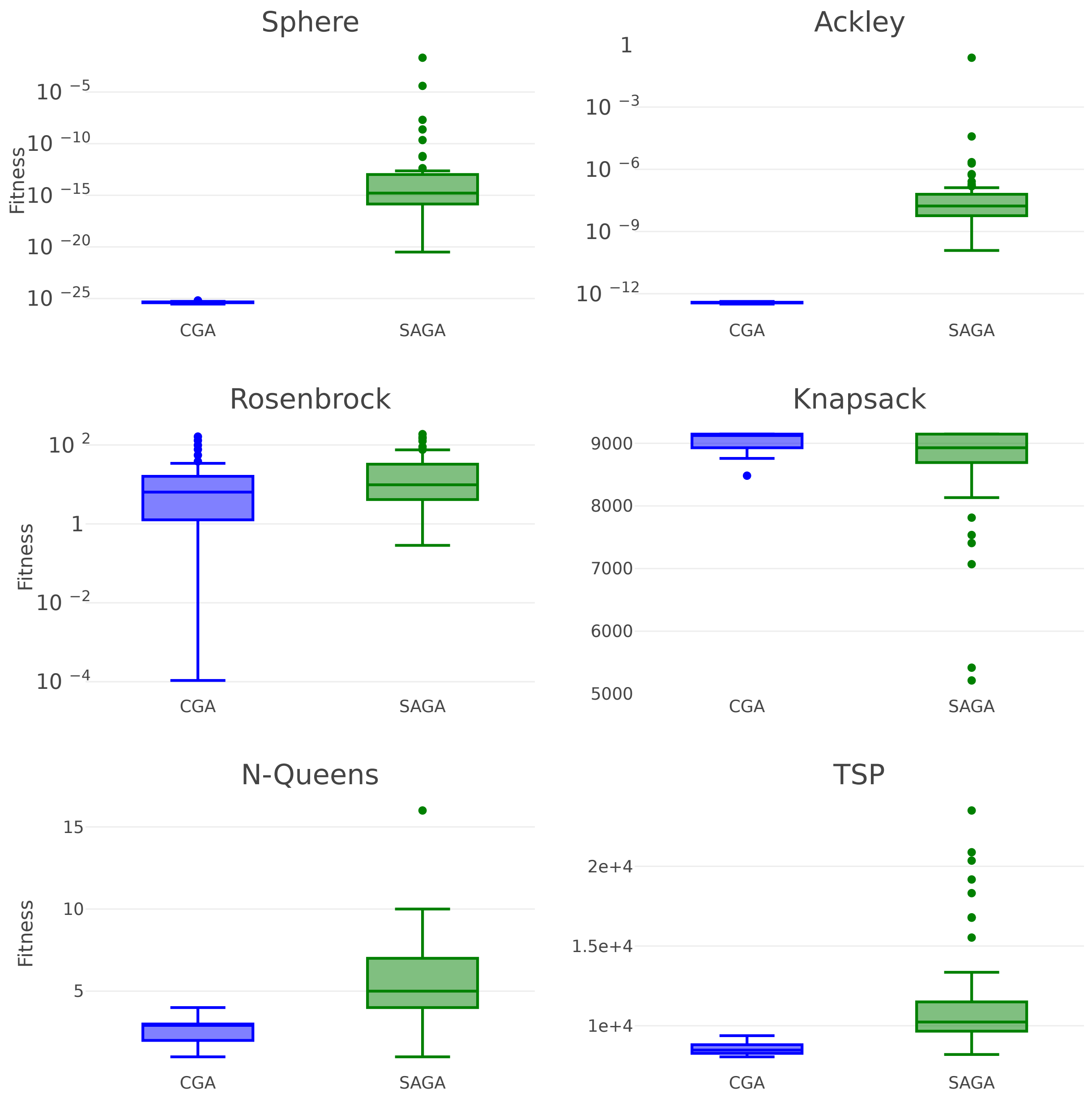

Figure 4.1: Box plots of fitness results from the best configurations of CGA and SAGA at 500 generations.

Table 4.8: Comparison of the number of fitness evaluations of SAGA and CGA at 500 generations.

Problem	Algorithm	Best Configuration Evaluation (BCE)	Total Evaluations (TE)
Sphere	CGA	98,656.66	4,774,746,486
	SAGA	**76,973.24**	**60,420,931**
Ackley	CGA	98,657.94	4,774,684,957
	SAGA	**80,328.58**	**60,859,591**
Rosenbrock	CGA	92,486.34	4,774,788,395
	SAGA	**68,878.98**	**56,455,247**
Knapsack	CGA	82,112.38	5,061,767,431
	SAGA	**50,670.86**	**44,731,466**
TSP	CGA	81,212.64	4,792,032,767
	SAGA	**57,043.12**	**47,702,211**
N-Queens	CGA	82,103.00	5,061,864,626
	SAGA	**56,460.12**	**41,542,730**

4.4.1.2 Number of Fitness Evaluations Comparison

In addition to the fitness results, we also need to look into the amount of time it takes to achieve said fitness results. Recall that one our primary goals is to improve GA usability through a reduction of parameter tuning. By comparing the time required for tuning CGA and SAGA, we can find how much time savings can be achieved with SAGA.

The number of fitness evaluations is our metric for time. Number of fitness evaluations is indicative of both tuning time and computational cost; the more fitness evaluations there are, the greater the computational cost, and the greater the computational cost, the longer the duration of the tuning runs. We use number of fitness evaluations instead of other potential time measurements such as CPU time, because fitness evaluations is agnostic to hardware specifications. We are interested in two measurements of fitness evaluations; the *Best Configuration Evaluations* (BCE), which is the average fitness evaluations per run on the best configuration, and the *Total Evaluations* (TE), which is the sum of all fitness evaluations of all runs on all configurations. The former metric facilitates a comparison of the best configurations of CGA and SAGA, while the latter metric facilities a comparison of the tuning effort invested into each algorithm. Table 4.8 shows the shows the BCE and TE of CGA and SAGA on all problems.

The significance of the BCE column in Table 4.8, is that it shows that although SAGA and CGA run for an equal number of generations, they do not execute an equal number of fitness evaluations. Across all problems, there are less fitness evaluations for SAGA. Recall, that a fitness evaluation is done only when an individual in the population is modified through the genetic operators of mutation and crossover. A lower number of fitness evaluations is the result of lower a mutation rate and/or crossover rate. SAGA evolves lower rates than CGA (which is further analysed in Section 4.5.2) resulting in lower fitness evaluations than CGA at an equal number of generations.

If SAGA is making less fitness evaluations, then SAGA is doing less work in improving the fitness of the population than CGA because the genetic operators are how the algorithm makes fitness improvements upon the population. Thus, the results shown in this experiment are not truly fair comparisons of SAGA and CGA. Running for an equal number of generations means that CGA is allowed to do more work towards fitness improvement than SAGA.

The significance of the TE column in Table 4.8, is that it shows the massive usability improvements of SAGA. This number, the sum of every run executed on every configuration, is indicative of the amount of computation time required for parameter tuning. There is a massive decrease in the total number of fitness evaluations of SAGA as compared to CGA. Ackley has the largest number of SAGA fitness evaluations at 60,859,591 which is a mere 1.23% of the number of fitness evaluations performed by CGA. While we already knew that SAGA should have less total fitness evaluations than CGA on the basis of SAGA having only 15 configurations to CGA's 1,228, these numbers help to truly illustrate just how big of a difference there is.

These fitness evaluation results are the reason why we have two sets of experiments in this chapter; the first with both algorithms running for an equal number of generations, and a second where

SAGA runs for a larger number of generations. First, SAGA is making less fitness evaluations per generation, so we should allow SAGA to run for at least an equal number of fitness evaluations as CGA. Second, because of the massive tuning time savings SAGA achieves, we can actually run SAGA for a much larger number generations whilst still staying well under the number of total fitness evaluations executed by CGA.

4.4.1.3 Likelihood of a CGA Configuration Finding Superior Fitness Than SAGA

Before we go into the second experiment, there is one last important comparison to be made between SAGA and CGA. We have shown that the best CGA configuration finds better fitness results at equal generations than the best SAGA configuration but at the cost of a great many more fitness evaluations and longer tuning time. We have not yet, however, determined if the amount of tuning we do on CGA is truly necessary. If we were to select a single CGA configuration at random, what is the probability that it returns superior fitness results than the best SAGA configuration? If this probability is sufficiently high, then it would mean our extensive CGA tuning was unnecessary, and our CGA results could be achieved with a much lower number of fitness evaluations. If the probability is low, then it means our CGA tuning was necessary, as it indicates that finding the correct set of parameters for the problem is difficult.

Table 4.9 shows the percent of the 1,228 total CGA configurations that return a more optimal median fitness than the best SAGA configuration. Figure 4.2 illustrates this same information. Figure 4.2 shows, for each problem, the median best fitness for all 1,228 CGA configurations compared to the median best fitness of the best SAGA configuration. There are 1,228 vertical bars, one for each of the CGA configurations. The CGA configurations are sorted according to the median best fitness. The median fitness of the best SAGA configuration is represented by the horizontal red line. A CGA configuration is colored green when it outperforms the best SAGA configuration and blue otherwise.

Table 4.9: Percentage of CGA configurations that find superior median fitness than the best SAGA configuration at 500 generations.

Problem	% CGA configurations superior to best SAGA configuration
Sphere	2.53%
Ackley	3.02%
Rosenbrock	0.25%
Knapsack	15.85%
TSP	9.23%
N-Queens	8.99%

Table 4.9 and Figure 4.2 reveal that, yes, the extensive tuning we conducted for CGA was indeed necessary, particularly for the continuous problems. A small minority of CGA configurations find superior median fitness than the best SAGA configurations. The most extreme case is Rosenbrock, where only 0.25% of CGA configurations find a superior median fitness than the best SAGA configuration. CGA's highest percentage is on Knapsack, where 15.85% of CGA configurations find a superior median fitness than the best SAGA configuration. It may be possible to make the argument that Knapsack, therefore, did not require such extensive parameter tuning. We argue that such extensive tuning is still required, as 15.85% is still a small minority of configurations and an inexperienced user would not know how best to trim down the number of parameters to ensure that the well-performing parameters are still included in their set of configurations.

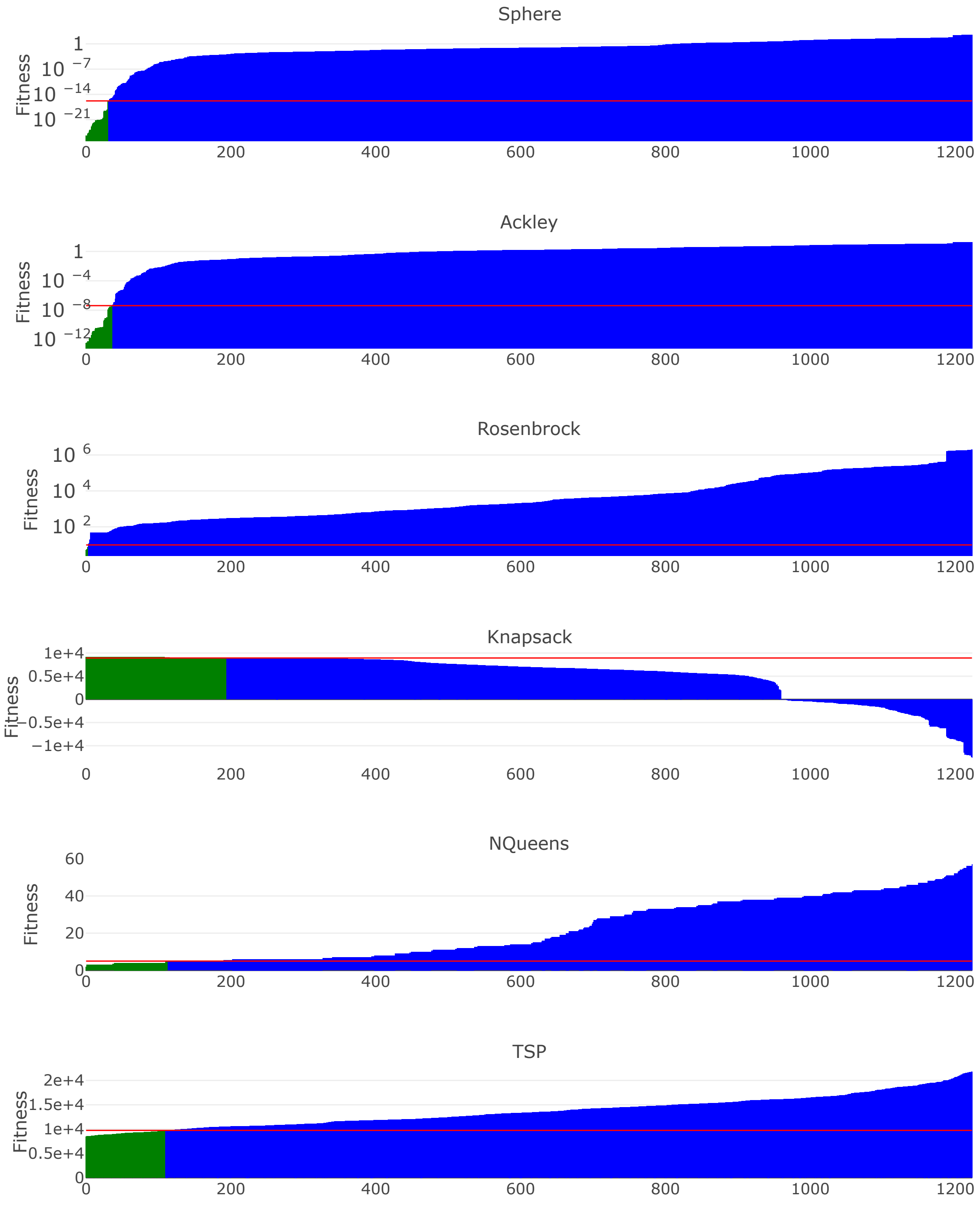

Figure 4.2: Comparison plot of all CGA configurations to the best SAGA configuration.

4.4.2 Extended SAGA Generations

Here, we see if the best SAGA configuration can match the median fitness of the best CGA configuration by running SAGA for an increased number of generations. Due to the massive time savings of SAGA in the previous experiment, and because the extensive CGA tuning is shown to be necessary, we run another experiment where the number of generations for SAGA is extended while staying well within the TE of CGA. For the sake of consistency, we take a look at the same type of results as in the previous experiment, with the first being the fitness results, followed by the number of fitness evaluations, and finally a look at the percent of CGA configurations that find a better median fitness than SAGA.

CGA is kept at the original number of 500 generations and SAGA runs for an increased number of generations, with a termination condition of either matching the median best fitness of the best CGA configuration, as listed in Table 4.7, or until 3,000 generations, whichever occurs first. All other parameters are unchanged on both GAs. If we chose to, we could run SAGA for many more than 3,000 generations while still keeping the total number of fitness evaluations across all configurations much lower than CGA. We chose to limit the generations to a maximum of 3,000, however, as executing a single run for an excessively long number of generations is a large time investment, even if the total number of fitness evaluations across all runs is lower. Multiple runs can execute in parallel if sufficient computing resources are available; the generations within a run, on the other hand, cannot execute in parallel because each generation is dependent on the previous generation occurring first. Thus we found that 3,000 generations is a sufficiently large number of generations to allow for improved fitness results without causing a SAGA run to have a prohibitively long runtime when compared to a CGA run.

The increased number of generations for SAGA is meant as a simple means of improving SAGA's fitness results while keeping SAGA's number of fitness evaluations lower than CGA. There are other potential ways of achieving this goal which may be either more efficient in terms of a smaller increase of the number of fitness evaluations or more effective on certain problems. Other methods include increasing the population size, or a more nuanced approach such as a restart mechanism that will restart a run with an increased population size if fitness stops improving on the current run. We did not test these other methods because detailed tuning or adaptation of the parameters of population size and generations is out of the scope of this study. It may be that a more complex method of handling these parameters of population size and generations will see improved results, but we decided to keep a limit on the number of variables at play in this study; hence why we went with the simplest available option of running SAGA for an increased number of generations.

4.4.2.1 Fitness Comparison

Table 4.10 shows the fitness results of the comparison of the best SAGA configuration to the best CGA configuration when SAGA runs for an extended number of generations. The table shows, for both GA types, the median fitness, mean fitness with a 95% confidence interval, and best fitness. The bolded values denote the superior values. A * denotes that the value became small enough such that floating-point accuracy was lost resulting in the value being rounded to 0. The CGA results shown here are the same as from Table 4.7. Figure 4.3 shows the data from Table 4.10 in box plot form. The figure has one plot per problem. On each plot, the blue box is the best CGA configuration, and the green box is the best SAGA configuration. Each box plot shows the median, interquartile range, inner and outer fences, and outliers. Here we again see that the median of the best SAGA configuration matches the median of the best CGA configuration on Sphere, Ackley, Rosenbrock, and Knapsack.

With additional generations, SAGA now finds equal or superior median fitness to the best CGA configuration on the Sphere, Ackley, Rosenbrock, and Knapsack problems. On the N-Queens and

TSP problems however, SAGA never finds equal median fitness with the best CGA configuration within the 3,000 generation limit, though SAGA does find an equal or superior best fitness on both problems. The mean fitness results of SAGA are still worse than CGA's on the majority of problems due to outliers. These results show that SAGA can indeed match CGA's median fitness results on 4 out of 6 of our problems through the simple means of allowing SAGA more fitness evaluations.

Table 4.10: Comparison of the fitness results of SAGA and CGA with SAGA running for extended generations.

Problem	Algorithm	Fitness		
		Median	Mean	Best
Sphere	CGA	4.35e-26	**4.35e-26 ± 2.00e-27**	2.89e-26
	SAGA	**4.28e-26**	3.33e-4 ± 6.47e-4	**0***
Ackley	CGA	3.70e-13	**3.72e-13 ± 6.44e-15**	3.17e-13
	SAGA	**3.54e-13**	4.78e-3 ± 9.28e-3	**4.00e-15**
Rosenbrock	CGA	6.39	**17.76 ± 9.03**	**1.08e-4**
	SAGA	**6.38**	**25.57 ± 11.43**	0.16
Knapsack (max)	CGA	**9147**	**9046 ± 32**	**9147**
	SAGA	**9147**	**8910 ± 102.08**	**9147**
TSP	CGA	**8476.94**	**8567.32 ± 97.37**	8051.35
	SAGA	9267.61	10421.81 ± 947.84	**8018.58**
N-Queens	CGA	**3**	**2.62 ± 0.22**	**1**
	SAGA	4	4.32 ± 0.47	**1**

As was the case in the previous experiment, SAGA still finds a larger spread of results with more outliers on all problems except Rosenbrock, showing that SAGA's larger variation in fitness results per run persists when SAGA runs for a larger number of generations. This continued variation is expected as the variation is caused by each run evolves different parameters, which increasing generations does nothing to change. One effect of the larger variation of runs of SAGA, is that when SAGA's median fitness is near CGA's median fitness, SAGA often has a superior best fitness. SAGA finds superior best fitness on Sphere, Ackley, and TSP. SAGA matches the best fitness of CGA on Knapsack and N-Queens; on Knapsack SAGA cannot find superior best fitness, as both

Table 4.11: Comparison of the number of fitness evaluations of SAGA and CGA with SAGA running for extended generations.

Problem	Algorithm	Best Configuration Evaluation (BCE)	Total Evaluations (TE)	
Sphere	CGA	**98,656.66**	4,774,746,486	500
	SAGA	123,670.40	**97,305,764**	820
Ackley	CGA	**98,657.94**	4,774,684,957	500
	SAGA	123,959.00	**93,739,356**	786
Rosenbrock	CGA	92,486.34	4,774,788,395	500
	SAGA	**76,001.44**	**62,403,248**	555
Knapsack	CGA	**82,112.38**	5,061,767,431	500
	SAGA	197,425.20	**119,887,894**	1,777
TSP	CGA	**81,212.64**	4,792,032,767	500
	SAGA	219,685.30	**154,650,394**	3,000*
N-Queens	CGA	**82,103.00**	5,061,864,626	500
	SAGA	196,870.13	**211,468,751**	3,000*

algorithms find the optimal fitness. Rosenbrock is the only problem where CGA finds a noticeably superior best fitness to SAGA.

4.4.2.2 Number of Fitness Evaluations Comparison

In addition to the fitness results, we also examine into the amount of time it takes to achieve said fitness results. SAGA's improved fitness results from the previous experiment come at the cost of increased fitness evaluations. Thus, we must ensure that the number of fitness evaluations conducted by SAGA is still significantly lower than CGA.

Table 4.11 shows the number of fitness evaluations made by CGA and SAGA on all problems. Three measurements are shown for each problem; BCE, TE, and the number of generations ran by the algorithm. A * in the generations column denotes that the maximum number of generations of 3,000 was reached.

The BCE column in Table 4.11, shows that allowing SAGA to match CGA results in the BCE now

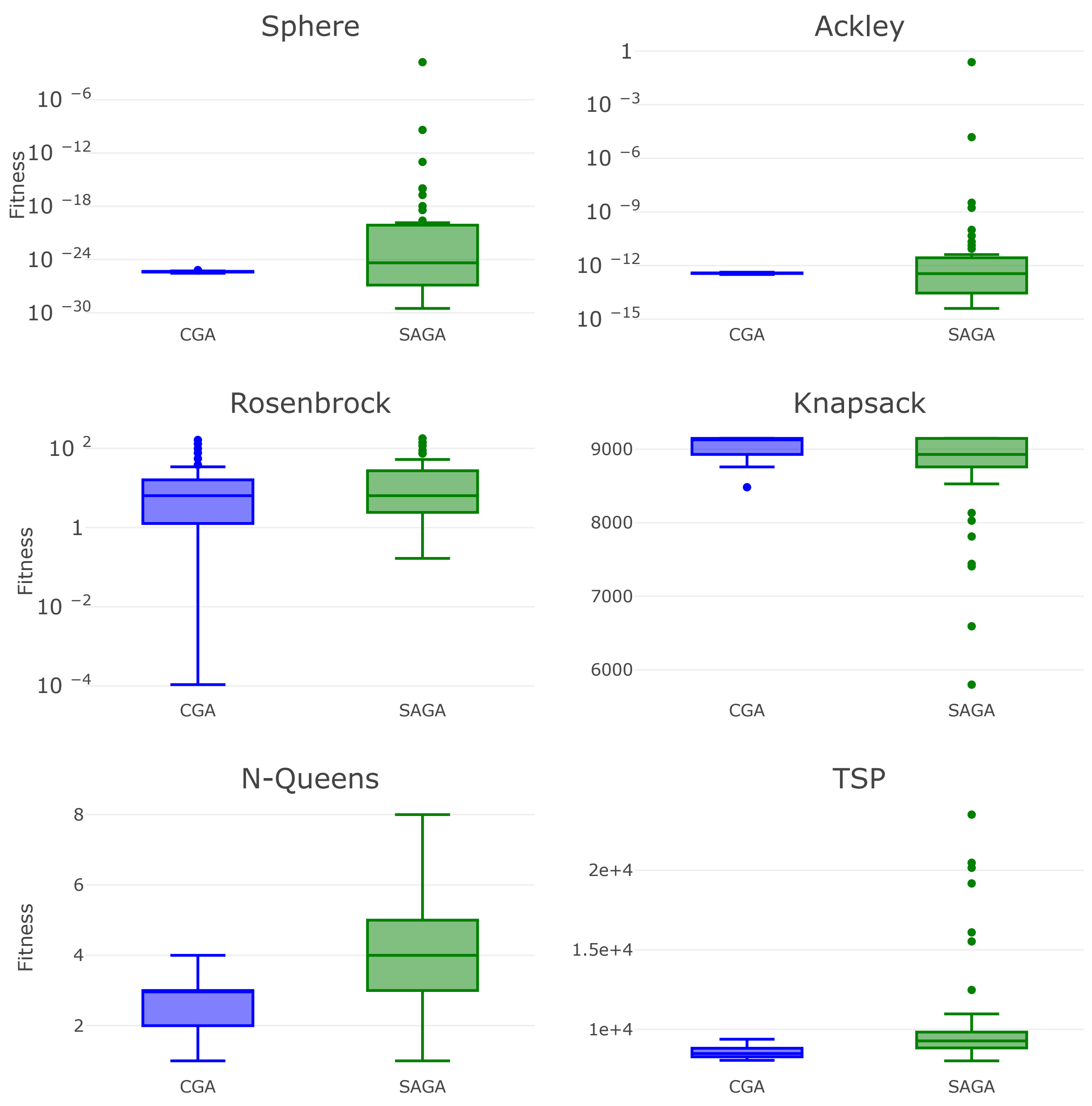

Figure 4.3: Box plots of fitness results from the best configurations of CGA and SAGA with SAGA running for extended generations.

being larger for SAGA than CGA on all problem except Rosenbrock. Therefore, on a per run basis, SAGA is more computationally expensive than CGA to find an equivalent fitness result on most of our problems. The TE column in Table 4.11, reveals that the total number of fitness evaluations for SAGA is still significantly less than that of CGA due to CGA tuning requirements. TSP has the largest number of fitness evaluations on SAGA at 211,468,751 which is only 4.28% of the fitness evaluations performed by CGA. The last column, generations, shows that SAGA runs for at least 50% more generations on all problems except Rosenbrock, which only runs for an extra 55 generations. The continuous problems required less extra generations than the discrete problems, particularly since SAGA on N-Queens and TSP never finds equivalent median fitness to CGA in our setup.

These total fitness evaluation results reinforce our justification for running SAGA an extended number of generations. Whilst the per-run number of fitness evaluations of SAGA is now higher on most problems due to the increased number of generations, the total number of fitness evaluations is still always significantly lower than CGA. We also see that on it is possible for SAGA to fully outperform CGA, as is the case on Rosenbrock, where SAGA finds better fitness results with less fitness evaluations per run than CGA.

4.4.2.3 Likelihood of a CGA Configuration finding superior fitness than SAGA

Table 4.12 shows the percent of the 1,228 total CGA configurations that return superior median fitness than the best SAGA configuration. SAGA matches or surpasses the median fitness on Sphere, Ackley, Rosenbrock, and Knapsack, thus, there are no longer any CGA configurations that find a superior median fitness than SAGA for those four problems. For TSP and N-Queens, the percent of CGA configurations that find a superior median fitness than SAGA is lower here than it was when both algorithms ran for 500 generations. Here, 4.75% and 5.26% of CGA configurations find superior median fitnesses than SAGA compared to 9.23% and 8.99% for TSP and N-Queens, respectively. Thus, on TSP and N-Queens, although the best CGA configuration outperforms the best SAGA configuration, it is still very difficult to find the superior CGA configurations with only a small percentage of CGA configurations outperforming SAGA.

Table 4.12: Percentage of CGA configurations that find superior median fitness than the best SAGA configuration with SAGA running for extended generations.

Problem	% CGA configurations superior to best SAGA configuration
Sphere	0.0%
Ackley	0.0%
Rosenbrock	0.0%
Knapsack	0.0%
TSP	4.75%
N-Queens	5.26%

4.4.3 Summary of Results

Overall, SAGA performs as well or better than CGA on the continuous problems and one of the discrete problems we tested. In each of the continuous problems, a superior fitness result, as measured by median fitness, is found with significantly less total fitness evaluations than CGA. Thus, we achieve our stated goal of easing parameter tuning without sacrificing fitness performance on a majority of the static problems. SAGA is less effective on discrete problems. SAGA finds equivalent median fitness results on the Knapsack problem, but fails to do so on TSP and N-Queens. SAGA on TSP and N-Queens, however, still executes significantly less fitness evaluations than CGA. Through changes to SAGA, such as a different representation mapping mechanisms, or an intelligent restarting mechanism, such equivalent results could most likely be found whilst keeping the number of fitness evaluations lower than CGA.

4.5 Analysis of the Self-Adaptive Parameters

Now that we've established that SAGA is competitive with CGA in generating solutions, we next examine whether adaptation is occurring within SAGA and if it is adapting parameters appropriately. We look into the evolved parameter values and how they vary over time and across the different problems. Understanding how self-adaptation is performing in SAGA is the first step towards any future improvements for SAGA.

One comparison we want to make throughout this analysis, is to compare the static operators and rates used by the best CGA configuration to the evolved operators used in the best SAGA configuration in order to see if both algorithms find the same preferred parameter settings or if SAGA finds synergy between different operators. Previous studies have shown that the inclusion of multiple operators affects which operators perform best Contreras-Bolton and Parada 2015; Hong, Wang, Lin, et al. 2002; Yoon and Moon 2002 due to the emergent synergy between certain operators. For ease of reference, we re-list the parameters used by the best CGA configurations in Table 4.13.

Table 4.13: Parameters of the best CGA configuration on each problem.

Problem	Crossover		Mutation		Parent Selection
	Operator	Rate	Operator	Rate	Torunament Size
Sphere	SBC	0.8	Swap	0.05	4
Ackley	SBC	0.8	Swap	0.05	4
Rosenbrock	Blend	0.4	Swap	0.05	2
Knapsack	SBC	0.8	Gaussian	0.001	2
N-Queens	Blend	0.8	Gaussian	0.001	4
TSP	Uniform	0.8	Gaussian	0.001	2

The following analysis looks at the parameters of the best SAGA configuration on each problem. We look at the best SAGA configuration instead of all SAGA configurations, as we want to look at how the parameter values are evolving when SAGA is performing at its best.

Recall that there are five self-adaptive parameters: mutation rate, crossover rate, mutation operator, crossover operator, and parent selection tournament size. This analysis splits these parameters into three groups: the parent selection tournament size, the operator rates of crossover and mutation, and how often each of the four mutation operators and seven crossover operators are used.

4.5.1 SAGA Parent Selection Tournament Size

The first of the self-adaptive parameters to analyse is the parent selection tournament size. We compare the evolved tournament sizes across the problems to see if SAGA is adapting differently to each problem. We also compare the evolved tournament sizes to the tournament sizes utilized by the best CGA configuration, to see if there is agreement between the algorithms or not and if this can help explain the differences in the fitness results of CGA and SAGA.

Figure 4.4 shows, for each problem, the mean parent selection tournament size for SAGA along with a 95% confidence interval. the y-axes show the mean tournament size and the x-axes show generations.

The variety of parent selection tournament sizes between problems demonstrates SAGA's ability to adapt this parameter to the problem at hand. On most of the problems, we see a trend of the tournament size, which always starts at 2 initially, rise up in the initial generations to a fairly stable value. Thus, we immediately see that on every problem SAGA does not stay at the minimum possible tournament size of 2. Sphere, Ackley, and Knapsack all settle around 5, TSP settles around 6, and N-Queens around 8. Rosenbrock is unique in that the tournament size only stays consistent for a limited number of generations, before increasing again in the later generations. Rosenbrock's behavior here is most likely due to the two-phase characteristic of the Rosenbrock problem, where the first phase is much easier than the second. Towards the later generations, SAGA is increasing exploitation through increasing the parent selection pressure, optimizing its ability to hone in on the optimal location. N-Queens and TSP find larger confidence intervals than the other problems, showing that SAGA has a harder time determining an appropriate parent selection tournament size on these two problems. While this variance of evolved values show adaptation to the problem, whether SAGA is evolving the "correct" value cannot be definitively determined through this analysis alone.

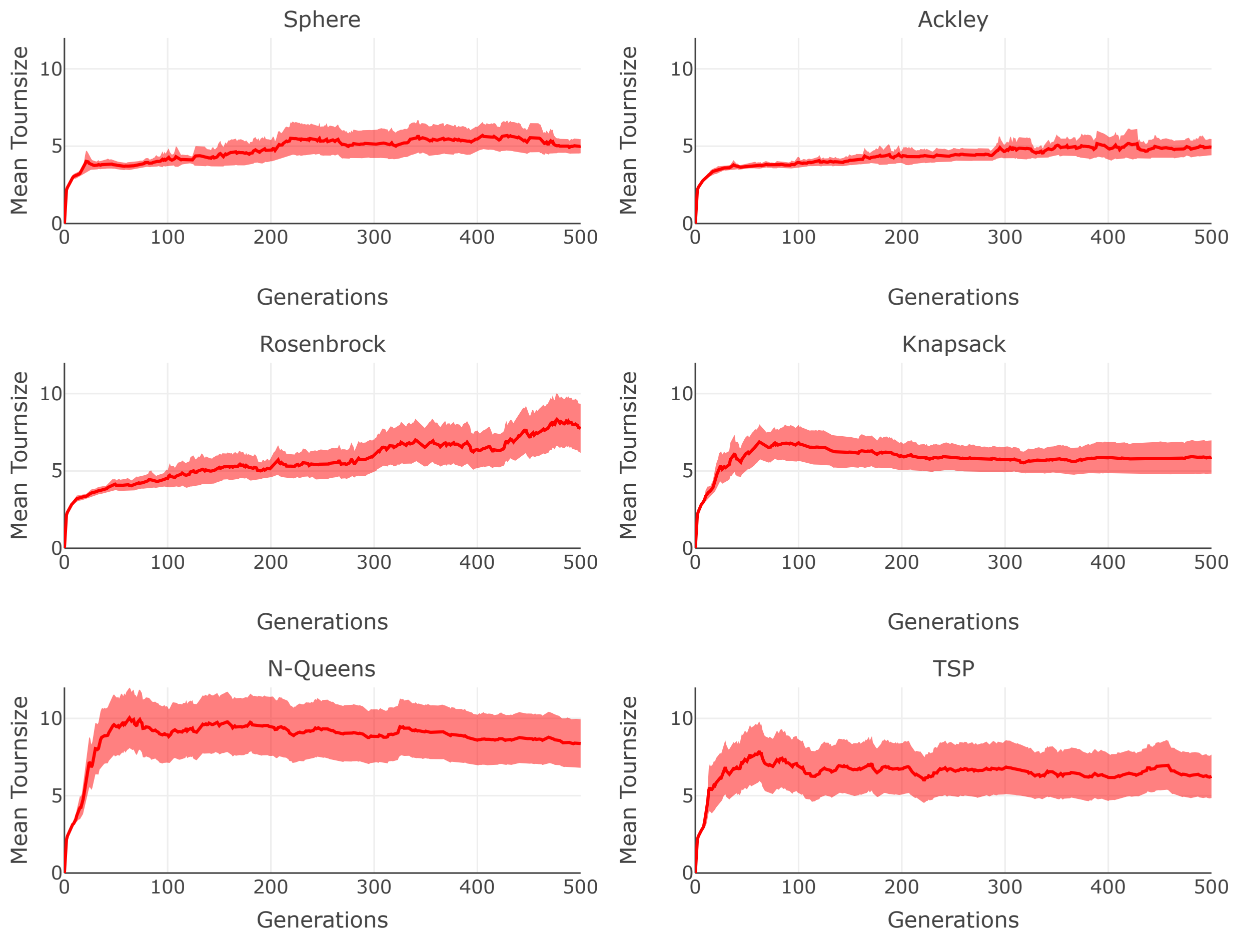

Figure 4.4: Mean parent selection tournament size across all runs of the best SAGA configuration on the static problems.

Comparing SAGA to CGA shows that SAGA evolves higher tournament sizes than CGA on all tested problems. Rosenbrock, Knapsack, and TSP all use a size of 2 on CGA. while SAGA never retains a tournament size so low. Sphere and Ackley are the only problems where SAGA and CGA are similar with a tournament size of 4 for CGA and around 5 for SAGA. N-Queens is very different with a tournament size of 4 for CGA and around 8 for SAGA. These findings show that through the tool of parent selection, SAGA favors a larger focus on exploitation over exploration than does CGA. The correct balance of exploitation vs exploration is problem-dependent, so SAGA's higher focus on exploitation across all problems may explain why SAGA performs superior than CGA on some problems, such as Rosenbrock, and worse than CGA on others, such as TSP.

4.5.2 SAGA Operator Rates

The next of the self-adaptive parameters to analyse is the genetic operator rates. We compare the evolved rates across the problems to see if SAGA is adapting differently to each problem. This section looks at the rates of both crossover and mutation, so we first look at each individually in turn, and then look into the trends across both rates together. We also compare the evolved rates to the tournament sizes utilized by the best CGA configuration, to see if there is agreement between the algorithms or not and if this can help explain the differences in the fitness results of CGA and SAGA.

Figure 4.5 shows the mean crossover and mutation rates evolved across the generations on each problem for the best SAGA configuration. The top plot is for crossover and the bottom plot is for mutation. The y-axes are the rate and the x-axes are generations; the y-axis of the mutation plot is on a logarithmic scale. Recall that the population is initialized such that the initial average crossover rate for the population is roughly 0.75 and the initial average mutation rate is roughly 0.25.

For the crossover rates, we see that over the course of the generations, the continuous problems generally evolve higher crossover rates between 0.5 and 0.7, while the discrete problems evolve to lower crossover rates, between 0.3 and 0.5. N-Queens, especially, drops its crossover rate very quickly in the early generations and remains low throughout its execution. Beyond the initial generations, the crossover rate does not see any sudden changes; either the rate remains relatively stable with only minor perturbations, or follows a slight but steady slope. For instance, Rosenbrock shows a steady upward slope, while Knapsack shows a steady downward slope.

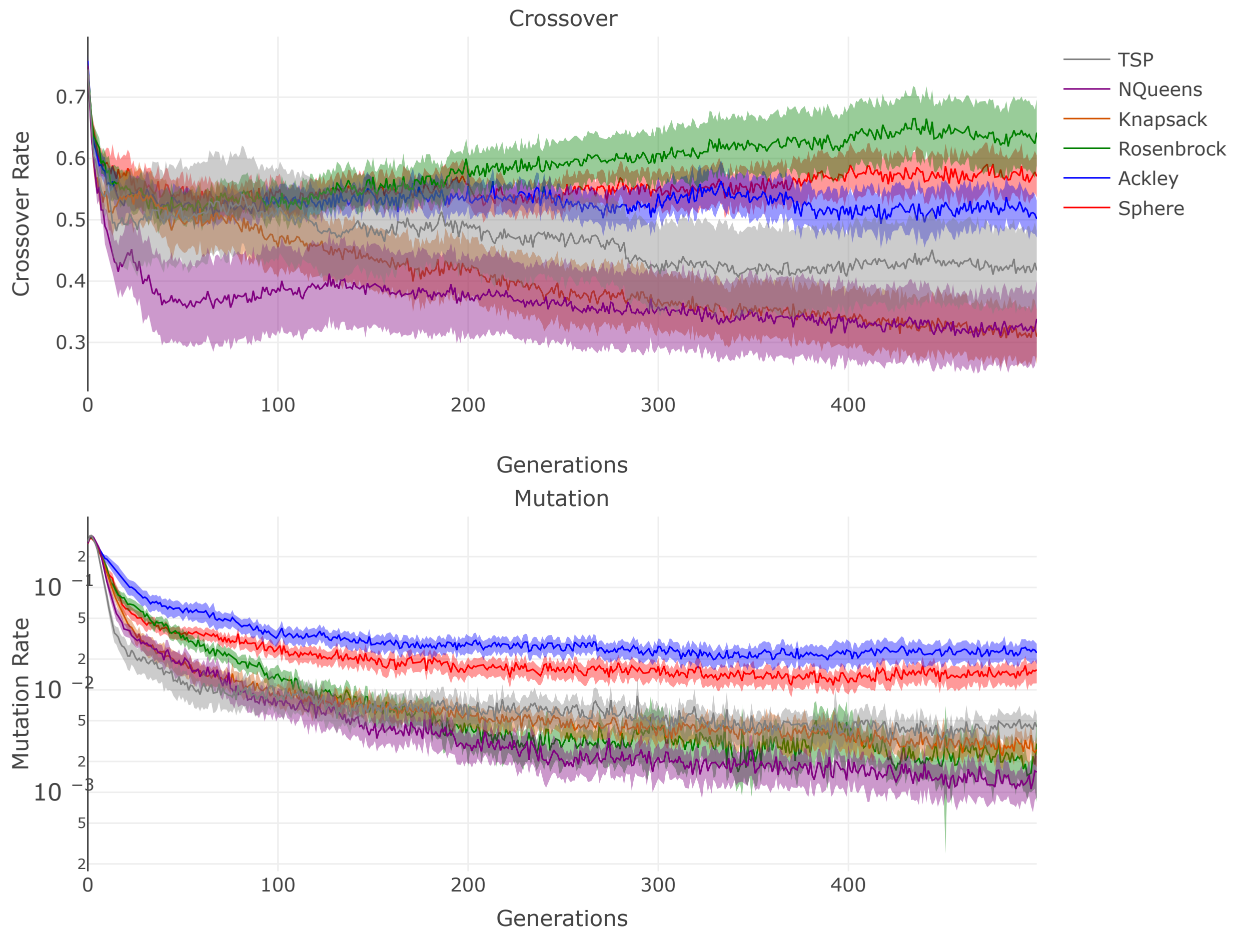

Figure 4.5: Mean crossover and mutation rates across all runs of the best SAGA configuration on the static problems.

On mutation, most problems follow the general trend of decreasing mutation rate over time before settling at a steady rate. The speed of the rate's decrease at the value at which it settles is different for each problem. Sphere and Ackley settle at higher mutation rates. All three discrete problems quickly drop to lower mutation rates. And Rosenbrock ends up with similar mutation rates as the discrete problems, but decreases more slowly than the discrete problems in the earlier generations.

For both crossover and mutation, the most distinct difference between problems is between the continuous and discrete problems. Generally, the continuous problems use higher rates for both crossover and mutation than the discrete problems. Recalling from the results, SAGA does not perform as well on the discrete problems as it does on the continuous problems, particularly N-Queens and TSP. The lower rates on the discrete problems are indicative of the difficulty SAGA had on these problems. On the discrete problems, it is likely that the genetic operations of mutation and crossover were often developing detrimental changes. Thus, the individuals who had high crossover and mutation rates, and were therefore undergoing many changes, often ended up with lowered fitness scores. These lowered fitness scores meant that these individuals were less likely to be selected as a parent of the next generation and were therefore removed from the population, resulting in overall lower rates in the population.

The rates evolved by the best SAGA configuration are sometimes similar to the rates used by the best CGA configuration and sometimes different. CGA uses a high crossover rate of 0.8 on all problems except Rosenbrock, which is lower at 0.4. SAGA, meanwhile, evolves lower crossover rates than CGA on all problems except for Rosenbrock. In fact, SAGA evolves a higher crossover rate on Rosenbrock than it does on any other problem, which is the exact opposite of CGA. On the mutation rates, the absolute values evolved by SAGA and used by CGA are different, but the difference in magnitude of the rates between the continuous and discrete problems is present on both algorithms. Both algorithms use higher mutation rates on continuous problems than discrete problems. Again, however, Rosenbrock is an exception; while SAGA starts with a high mutation rate comparable to the other continuous problems, it later evolves to a rate comparable to the discrete problems.

It is noteworthy that the biggest difference of operators rates between SAGA and CGA is on the

Rosenbrock problem, as Rosenbrock is the problem on which SAGA obtains its best results. SAGA varies both the crossover rate and mutation rate of Rosenbrock over the generations. This variation of rates shows that on Rosenbrock, SAGA is finding and exploiting useful adaptations of the rates that CGA is incapable of matching.

4.5.3 SAGA Operator Usage

Recall that in SAGA, self-adaptation is applied to the crossover and mutation operators by having multiple operators for each and selecting one operator to use whenever crossover or mutation occurs. Every operator is encoded within every individual, but these encoded values do not directly denote how often the operator is used; these encoded values are run through the operator selection process to determine how often the operators are used. We refer to how often an operator is used as its *usage*. We analyse the outcome of this operator selection process and look at the quantity of how often each crossover and mutation operator is used each generation. This analysis is done on the best SAGA configuration for each problem. First we look at the crossover operator usages, followed by the mutation operator usages.

It is important to remember, as was shown in Figure 3.3 that the operator selection method used by SAGA is highly influential in the how often each operator is used. Therefore, it is useful to keep in mind which operator selection method the best SAGA configuration uses on each problem. For ease of reference, Table 4.14 re-lists the operator selection methods used by the best SAGA configuration on each problem.

Table 4.14: Operator selection method of the best SAGA configuration on each problem.

Problem	Operator Selection Method
Sphere	Tournament, winner probability = 0.8, size = all
Ackley	Tournament, winner probability = 0.9, size = all
Rosenbrock	Tournament, winner probability = 0.8, size = 4
Knapsack	Tournament, winner probability = 0.9, size = 3
TSP	Tournament, winner probability = 0.8, size = 2
N-Queens	Tournament, winner probability = 1.0, size = 2

4.5.3.1 Crossover Operator Usage

In the analysis of the crossover operator usage, we are looking into the relative differences of usage between each of the operators to see how strongly SAGA favors certain operators over others. We compare the evolved rates across the problems to see if SAGA is adapting differently to each problem, which we do in two parts: continuous problems and discrete problems. We make a special note of the Rosenbrock problem, since it is the problem on which SAGA performs best and it shows unique characteristics in crossover operator usage. We also compare the evolved rates to the tournament sizes utilized by the best CGA configuration, to see if there is agreement between the algorithms or not and if this can help explain the differences in the fitness results of CGA and SAGA.

Figure 4.6 contains six plots, one for each problem, showing the mean number of times each crossover operator is used each generation along with a 95% confidence interval. The y-axes are the average number of times an operator is used and the x-axes are generations.

On the three continuous problems, we see two phases in behavior. Initially, Simplex is the most used operator for the first 50 to 100 generations. Afterwards, Simplex drops in usage and other operators rise in its place. Uniform and Blend in particular become highly used operators after the first 100 generations. The operators of Arithmetic, Linear, and PCX are used very rarely on the continuous problems past the first few generations. These operator usage patterns show that

SAGA is adapting its crossover operator choice to the current situation in a problem. The operators of Simplex, Uniform, Blend, and SBC are identified as the most effective crossover operators on these problems. Furthermore, in the early generations, SAGA finds that Simplex is the single most useful crossover operator for early fitness improvements. The effectiveness of Simplex later diminishes once the population has made some progress towards the optima, so SAGA switches to more heavily use other operators.

The discrete problems show very little to no adaptation form generation to generation in operator usages. On all three problems, there is very little relative differences in the usage of the operators, although Knapsack has the highest relative difference of the three. There also is no clear shifting of which operators are used most often over the generations as was the case on the continuous problems. The best-performing operator selection method for these three problems all have low selection pressure due to having small tournament sizes of 2 or 3. Of the three discrete problems, SAGA performs best on Knapsack, so the fact that Knapsack shows the most relative difference in usage between operators shows that SAGA was more effectively able to identify which operators are effective on Knapsack than on N-Queens or TSP. These results on N-Queens and TSP means that either the operators we included in SAGA are not particularly effective on these problems, or that our strategies for mapping the encoded data on the problems on N-Queens and TSP can be improved. The mapping of data can have unintended side effects on how the operators interact with the data.

SAGA's crossover usage on Rosenbrock is slightly different from the other two continuous problems of Sphere and Ackley. On Rosenbrock, there are smaller relative differences in the usages of the operators. These smaller differences are because Rosenbrock uses an operator selection method with a lower selection pressure. It may at first seem like a lower selection pressure is inherently undesirable, as it is intuitive that the algorithm should try and identify the best operator and use only that operator. SAGA's fitness results are superior on Rosenbrock than Sphere and

Ackley when compared to CGA, however, showing that this idea of only favoring the best single operator is not always the best strategy. Instead, on Rosenbrock SAGA is able to effectively find synergistic combinations of a larger number of operators.

We see that SAGA and CGA favor different operators in that the operator used by the best CGA configuration and the most heavily used operators of the best SAGA configuration are different. The best CGA configurations use SBC for Sphere, Ackley, and Knapsack while on SAGA, SBC is in the middle of the pack in terms of usage on all three of these problems. On Rosenbrock, there is agreement between the algorithms on Blend being heavily used, but for SAGA, Blend's usage is contested by Uniform, so it is not a wholehearted agreement. On N-Queens, Blend is again the operator used by CGA, and is near the top of the pack in usage for SAGA, but with substantial overlap of usage by nearly all other operators. TSP uses all operators nearly equally in SAGA, thus there is no single operator to compare to CGA's operator of Uniform. The comparison of SAGA and CGA shows that on the problems SAGA performs well on, SAGA is not merely making the same operator choice as CGA, rather it is finding synergistic combinations of multiple operators.

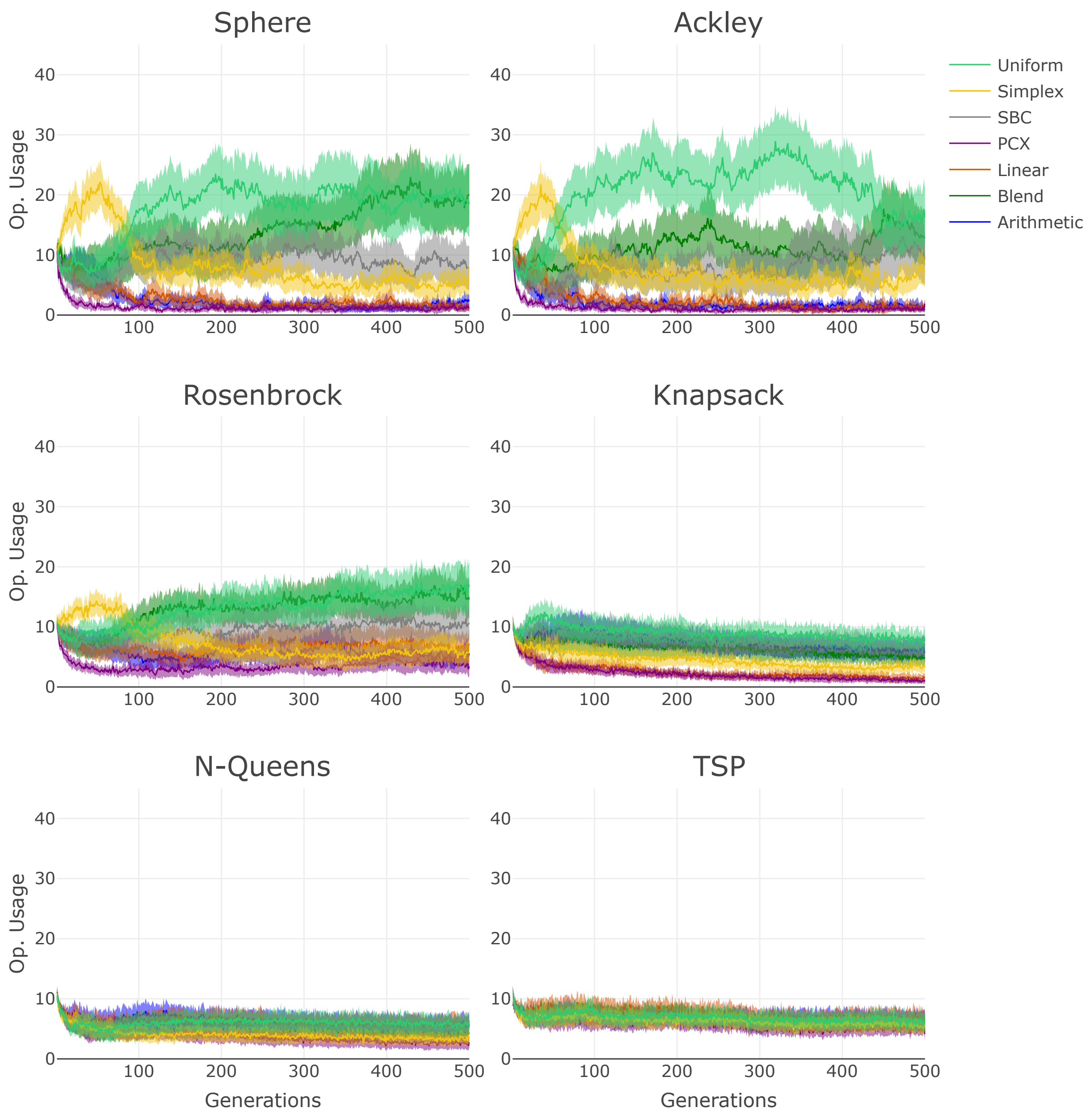

Figure 4.6: Mean crossover operator usages across all runs of the best SAGA configuration on the static problems.

4.5.3.2 Mutation Operator Usage

The analysis of mutation operator usage generally follows the same steps as that of the crossover operator usage. We are looking into the relative differences of usage between each of the operators to see how strongly SAGA favors certain operators over others. We compare the evolved rates across the problems to see if SAGA is adapting differently to each problem, which we do in two parts: continuous problems and discrete problems We also compare the evolved rates to the tournament sizes utilized by the best CGA configuration, to see if there is agreement between the algorithms or not and if this can help explain the differences in the fitness results of CGA and SAGA. Finally, we make a special note of the swap operator, due to it's high usage on the continuous problems.

Figure 4.7 contains six plots, one for each problem, showing the average number of times each mutation operator is used each generation along with a 95% confidence interval. The y-axes are the average number of times an operator is used and the x-axes are generations. The y-axes use a logarithmic scale.

Across all of the continuous problems we see that Swap is the most used mutation operator. On Sphere, the other three operators are used equally throughout the generations, on Ackley, Gaussian is briefly the second most used operator within the first 100 generations, before dropping to equal usage of Uniform and Polynomial. Rosenbrock keeps the same general hierarchy of operator usage as Sphere, but sees Swap's usage drop to near the levels of the other operators by the later generations. Similar to the crossover operator usages, the smaller relative difference between Swap and other operators on Rosenbrock is due to the lower operator selection pressure the best SAGA configuration uses on that problem.

There is some variety in the frequency of operator usage amongst the discrete problems. Knapsack doesn't find a single most used operator, but does find a single least used operator in Uniform. SAGA is, therefore, Able to identify that Uniform is an unhelpful or even detrimental operator on Knapsack. N-Queens and TSP then both behave very similar here to their behavior for the crossover operator usage in that there is very little relative difference in the operator usages. Unlike in the continuous problems, Swap is no longer clearly favored on any of the discrete problems.

The favoritism of Swap on the continuous problems is in agreement with the static mutation operator of the best CGA configurations which also uses Swap on these three problems. On the discrete problems, the best CGA configuration always uses Gaussian. On SAGA there is no clear favored mutation operator on any of the discrete problems. Similar to the crossover usage, the well-performing problems find clearer preferences of operators, while the poorly performing problems are not able to identify which mutation operators should be favored.

An unexpected finding was the usefulness of the Swap mutation operator on the continuous problems. We included this operator to have a simple mutation operator that operates in a fundamentally different manner than the other operators of uniform, Gaussian, and polynomial. In both types of GA, there are crossover operators that create new floating-point values in the chromosomes. Mutation then, is not required to create new data and so a mutation operator that does not create new data and instead only exchanges data within the chromosome is viable and apparently effective. Furthermore, these continuous problems each have an optima where every value in the solution chromosome is identical. If the optimal solution consists of a single repeated value, then this value only has to be found by the population a single time and then Swap mutation, combined with crossover, can spread this optimal value throughout the population. The usage of Swap mutation shows that SAGA is able to detect and exploit the characteristics of a problem, which is further reinforced by the fact that on the problems where the optimal solutions do not consist of a single repeated value, Swap was no longer the most used mutation operator. This unexpected usefulness of Swap is also an example where the inclusion of multiple operators within SAGA can be useful for non-experts, as they may not know before-hand which operators are more effective on which problem.

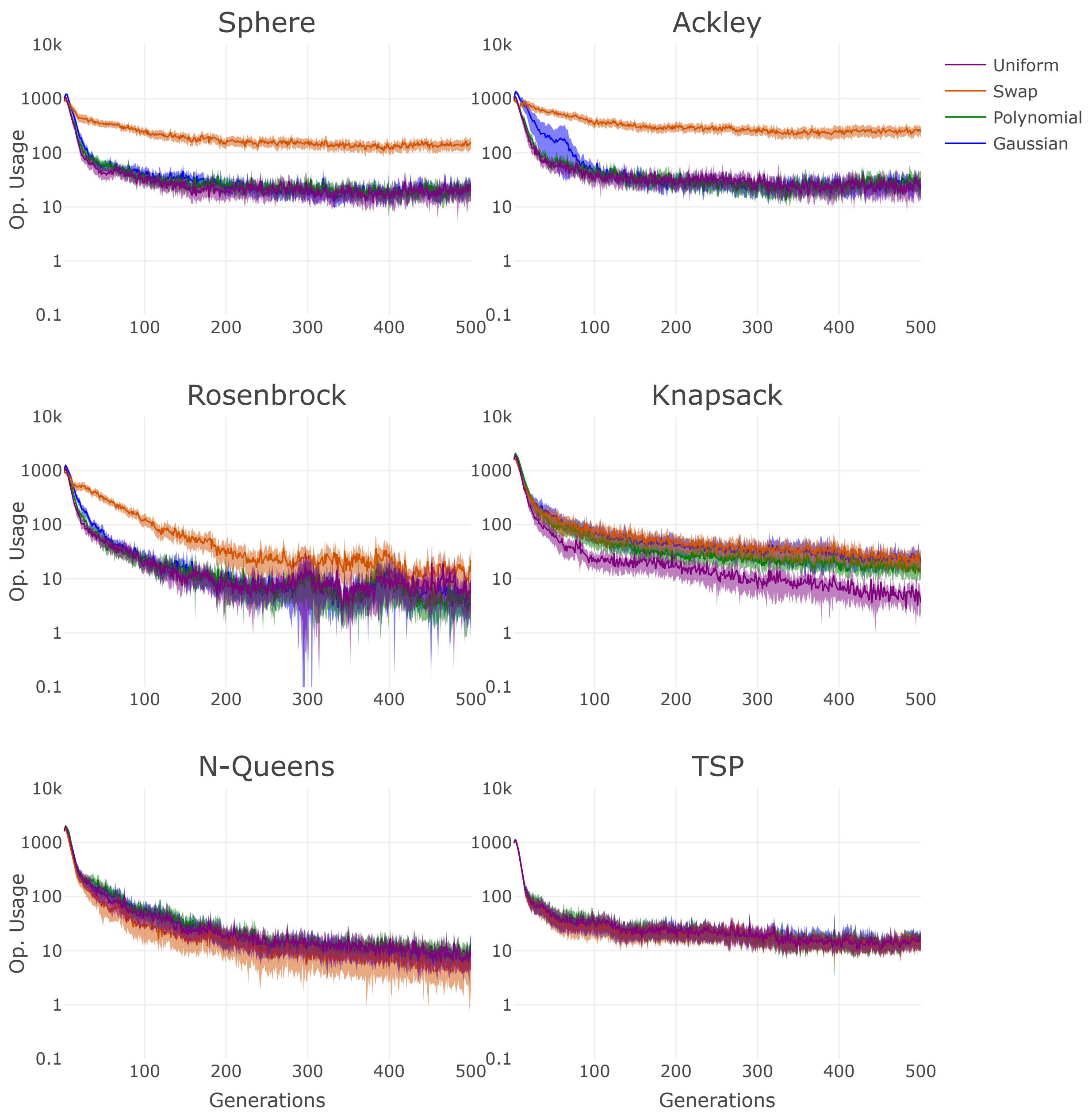

Figure 4.7: Mean mutation operator usages across all runs of the best SAGA configuration on the static problems.

4.6 Conclusions for SAGA on Static Problems

For our first experiment, we test SAGA on a suite of static problems. The problems are split between continuous and discrete cases. We compare SAGA to a well-tuned CGA to compare both the fitness results and algorithm runtimes, as measured by the number of fitness evaluations. Our goal for SAGA is to find equivalent fitness results to thee well-tuned CGA, while improving usability through a reduction parameter tuning, and therefore a reduction of the number of the number of fitness evaluations.

The experiments found that the proposed SAGA is capable on finding equivalent fitness results as measured by median fitness in substantially less function evaluations when compared to an equivalent manually-tuned CGA on the continuous problems we tested. The amount of tuning required is massively reduced while still retaining equally good fitness results on continuous optimization problems. Of course, we cannot say this will remain true on all continuous problems, but the fact that the results are consistent across the continuous problems we do test shows promise in the continuous problem domain. Our implementation does not remove the need for all tuning; other operators such as population size, generations, elitism, etc. are not self-adaptive and can still be tuned as needed. Further, a new parameter to tune, operator selection method is introduced.

SAGA is less capable on the discrete problems we tested. SAGA does find equivalent median fitness to CGA on Knapsack, but does not do so on N-Queens or TSP; though it must be stated that we did limit the maximum number of generations SAGA ran for in this experiment. TSP in particular performed quite poorly throughout all the experiments. The Random Keys representation we use appears to not translate well into our GA's framework. The primary reason for keeping both chromosomes as floating-point values was to keep the logic for the self-adaptive operators as simple as possible; but it now appears that this approach is not sufficient for all discrete problems.

We find that SAGA does indeed make problem-specific adjustments to its self-adaptive parameters. These adjustments are made over the course of the generations such that the self-adaptive parameters vary over the generations. The problems in which SAGA performs well on particularly show clear signs of this adaption of the self-adaptive parameters.

Based on these results, we can recommend SAGA for continuous problems, but we cannot confidently recommend it on all problems of other representations. Future work can be done to investigate alternative representation strategies for the self-adaptive GA that can better facilitate these types of problems. On the continuous problems, we achieve our goal of matching the fitness of a well-tuned CGA while drastically reducing the burden of parameter tuning.

CHAPTER 5: DYNAMIC PROBLEMS

In addition to improving the usability of GAs, SAGA may also be effective on dynamic problems. We define a dynamic problem as a problem in which the fitness landscape of the problem changes over the course of an algorithm's execution. The inherent variability of self-adaptive parameters gives SAGA the ability to adapt to a problem's changes during a run. This experiment consists of a suite of dynamic test problems, multiple GA variants, and metrics designed for dynamic problems. Once everything has been explained we go through the results of each problem followed by an analysis of the behavior of SAGA on these dynamic problems.

5.1 Motivation

Dynamic problems are an important class of problems to investigate; real life problems are often dynamic, as many real-life situations have requirements or goals that change over time. A system that can accurately account for these shifting goals is invaluable. Dynamic problems are not a commonly investigated category of problems within the field of EAs; while there has been some work done in this area Cobb 1990; Cobb and Grefenstette 1993; Grefenstette 1992; Pétrowski 1996; Mc Ginley et al. 2011, the majority of EA research has focused on static problems, where the fitness landscape is fixed.

At first glance, GAs appear to be a natural fit for solving dynamic problems. The existence of a population of multiple diverse candidate solutions spread across the fitness environment should be beneficial to keeping track of a moving target in a dynamic environment. There is a fatal flaw for canonical GAs however; while the population is initially spread out across the environment, the population converges upon a single point as the generations progress. Once the population has converged, it loses its diversity and has difficulty finding an optimal that moves out of the area of convergence. Crossover between two near-identical individuals creates children that are also nearly identical to their parents. Mutation on multiple near-identical individuals search over near-identical areas. A converged population, therefore, loses much of its exploration capabilities. On static problems, this loss of exploration is fine, as the population should be converging upon the optima such that exploitation is desired over exploration. In dynamic problems, however, changes in the problem landscape may cause an unexpected increase in the importance of exploration at any point in time. Multiple methods have been developed for GAs to address this problem; usually through the means of preserving population diversity. Self-adaptation is a competitive alternative to the existing approaches.

5.2 Experimental Setup

The problems are designed to cover a variety of dynamic behaviors to see if self-adaptation is only viable on certain types of dynamic problem landscapes. We bring in alternative GA methods because we want to compare SAGA against GAs that are competitive on dynamic problems and we expect the CGA to perform poorly due to its lack of population diversity preservation. These alternative GAs all have mechanisms for dealing with the difficulty of dynamic problems. Lastly, we introduce two new metrics for the measurement of fitness and population diversity.

5.2.1 Problems

We describe our test suite of nine dynamic problems and how these problems are generated. The fitness landscape of each of our problems is a multi-dimensional field of cones generated through the DF1 Test Problem Generator Morrison and Kenneth A. De Jong 1999. The DF1 Test Problem

Generator allows for a large amount of flexibility in problem setup; the number of cones, cone heights, cone locations, and cone slopes are all modifiable and can be dynamically changed over the course of a run. We use DF1 to create a diverse suite of dynamic problems that cover a variety of problem dynamics and difficulties.

We do not provide a detailed description of the inner workings of DF1; rather, we refer the reader to Morrison and Kenneth A. De Jong (1999) for such a description. We do, however, describe the user-defined DF1 parameters that we use to generate each of the problems to allow for recreation. There are three relevant DF1 parameters for our study; Ac, $cstepscale$, and R. Ac is a number in the range [1.0, 4.0] that controls the complexity of the cone movements through a bifurcation diagram. An $Ac < 3.0$ results in a constant movement size. As Ac increases over 3.0, the movement sizes become increasingly random. The $cstepscale$ controls the size of the cone movement, where a larger $cstepscale$ results in a larger cone movement. The $cstepscale$ is meant to scale the movement generated by Ac to a size appropriate for the bounds of the problem at hand. R is the slope of the cones, where a larger R results in a steeper slope.

Some parameters are kept constant across all problems; these are:

- Number of Cones: 50
- Number of Dimensions: 30
- Optimal Cone Height: 50
- Non-optimal cone heights: 30 - 48
- Cone Slope (R): 7

Fifty cones are initialized randomly within the search space. The number of cones is fixed throughout the problem's duration. All thirty dimensions are floating-point numbers in the range [0.0, 1.0].

A single cone is initialized at the optimal height of 50 and all other cones are initialized with a random height between 30 and 48. We keep the slope of all cones equal and fixed at $R = 7$. We do not vary slope at all in any experiment in order to keep the number of dynamic variables to a manageable level.

Table 5.1 lists our dynamic problem test suite. Our test problems can be separated into two broad categories: steady change and sudden change. The table lists an ID number for each problem, the category of the problem (steady or sudden) a brief description of its dynamics, the interval of its dynamics in generations, the total number of generations executed, and the relevant DF1 parameters that are used. Generations is the time-step of the problem dynamics and any problem dynamics occur at the end of a generation. From this point forward, we refer to each problem by their ID number.

The first test problem is a baseline test on a fixed field of cones. By doing this first non-dynamic test problem, we get a baseline performance on the field of cones landscape allowing us to see how each form of dynamics affects the difficulty of this basic field.

For the steady change problems, 2 through 6, the position of every cone moves around the landscape every generation; the landscape is never stationary. The size of the movement is controlled through the DF1 parameters of Ac and $cstepscale$ which are all set to 2.0 and 0.05, respectively, on each of these problems. These two parameters result in a constant movement size of the cones. The difficulty of the steady problems is controlled by varying the number of dimensions in which the cones move. The more dimensions in which a cone moves, the more difficult the problem. The dimensions in which the cones move is selected randomly on each movement. Upon initialization, the direction along each dimension, positive or negative, in which a cone will move is determined randomly. This direction remains constant, however, until the cone hits the boundary of that dimension, upon which it will reverse. In effect, the cones bounce off the edges of the search space.

Table 5.1: Dynamic test problems.

ID	Category	Description	Generations	Relevant DF1 Parameters
1	No Change	Stationary Landscape	300	n/a
2	Steady	Move all peaks in 1 dimension every gen	300	Ac = 2.0, cstepscale = 0.05
3	Steady	Move all peaks in 15 dimensions every gen	300	Ac = 2.0, cstepscale = 0.05
4	Steady	Move all peaks in [1, 5, 10, 15] dimensions every gen. Change the number of dimensions every 100 gen	1000	Ac = 2.0, cstepscale = 0.05
5	Steady	Move all peaks in [1, 5, 10, 15] dimensions every gen. Change the number of dimensions every 50 gen	500	Ac = 2.0, cstepscale = 0.05
6	Steady	Move all peaks in [1, 5, 10, 15] dimensions every gen. Change the number of dimensions every 20 gen	400	Ac = 2.0, cstepscale = 0.05
7	Sudden	Re-initialize landscape every 100 gens.	500	n/a
8	Sudden	Re-initialize landscape every 50 gens.	500	n/a
9	Sudden	Re-initialize landscape every 20 gens.	500	n/a

We use a constant direction along each dimension because fully random movement can result in cones shaking within a small area due to moving back and forth constantly.

The specifics of each of the dynamic problems are as follows. In problem 2, each cone moves in a single dimension. In problem 3 each cone moves in 15 dimensions, which is 50% of the total number of dimensions. Problems 4 through 6 are more complex and vary the number of dimensions in which the cones move during a run. Every generation, the cones move in either 1, 5, 10, or 15 dimensions. The number of dimensions in which the cones move follows the pattern [1, 5, 10, 15, 10, 5] which repeats until the end of a run. Each value in the pattern is active for *x* generations, where *x* is the “Change Interval” column in Table 5.1. For example, on problem 4, the cones move in 1 dimension for 100 generations, then in 5 dimensions for the next 100 generations, and so on following the pattern. With this pattern, the problems alternate between steadily increasing the difficulty and steadily decreasing the difficulty. The difference between problems 4, 5, and 6 is the frequency in which the number of dimensions changes.

For the sudden change problems, 7 through 9, the problem landscape is stationary for the vast majority of generations and large changes occur every *x* generations, where *x* is given in the "Change Interval" column in Table 5.1. In these problems, the field of cones is completely re-initialized at every change interval. This re-initialization results in both the heights and positions of every cone changing. The rules for the original initialization still hold for each re-initialization, so there is always a single cone of height 50 and the other rules listed earlier are still adhered to. These sudden change problems are designed to show how the GAs handle an abrupt, large-scale change that occurs both while the population is in the process of converging on a single peak or once the population has already converged on a single peak. We test three change intervals of 100, 50, and 20 generations. The population will be more converged at 100 generations than at 50 or 20 generations which may mean it harder to find the new optimal. On the other hand, 100 generation intervals gives the population more time to re-adjust and find the new optimal before the next change.

5.2.2 Algorithms

We test six algorithms on the dynamic problems. In total, there are six algorithms: SAGA, CGA, Triggered Hypermutation (HM) Cobb 1990; Cobb and Grefenstette 1993; Morrison and Kenneth A. De Jong 2000, Random Immigrants (RI) Cobb and Grefenstette 1993; Grefenstette 1992, Clearing Pétrowski 1996, and ACROMUSE Mc Ginley et al. 2011. HM, RI, Clearing, and ACROMUSE are all GA variants that are specifically designed for dynamic problems. These six unique GAs do share some commonalities; they all use the same problem representation, have a population size of 200, and execute 50 runs per configuration. First we describe this common representation, then we describe each of the algorithms in turn.

5.2.2.1 Problem Representation

For each GA method, the solution for each problem is represented as a list of thirty floating-point values. Each of the floating-point values are within the range [0.0, 1.0]. The length of the chromosome, thirty, is equal to the number of dimensions. The floating-point values represent the position along a dimension. We use this representation on all GA variants in this chapter. In the case of SAGA, all of this applies to the solution chromosome, while the parameter chromosome is unchanged from the previous chapters. This common representation ensures that all algorithms are searching the same solution space and that any differences in the results of each algorithm are not due to different representationss.

5.2.2.2 Self Adaptive Genetic Algorithm

The setup of SAGA is unchanged from the previous chapter. We run 15 configurations of SAGA, one for each of the operator selection methods. Table 5.2 lists the best SAGA configuration for each problem.

5.2.2.3 Canonical Genetic Algorithm

We use the CGA as a benchmark. The CGA requires the tuning of many parameters; we conduct this tuning on problem 1, the static landscape problem. We test the same range of parameter values and each genetic operator as was done in the previous chapter and listed in Table 4.2. The values found from the tuning are used by the CGA in the subsequent dynamic test problems. Table 5.3

Table 5.2: Parameters for CGA on the dynamic problems.

Problem	Operator Selection Method
1	Proportional
2	Tournament: size = all, winner probability = 0.8
3	Tournament: size = 4, winner probability = 0.9
4	Tournament: size = 4, winner probability = 1.0
5	Tournament: size = 4, winner probability = 1.0
6	Best
7	Rank
8	Proportional
9	Tournament: size = all, winner probability = 0.8

lists these final tuned parameters. These parameters may not necessarily be the optimal parameters for each problem but tuning 1,228 configurations for all non-SAGA methods on all 9 problems is prohibitively time consuming; this is the crux of the problem we wish to address with self-adaptation.

Table 5.3: Parameters for CGA on the dynamic problems.

Problem	Crossover		Mutation		
	Operator	Rate	Operator	Rate	Args
1 - 9	Blend	0.8	Gaussian	0.001	0.25

5.2.2.4 Triggered Hypermutation

HM is a GA with two different mutation rates. On any generation, the GA will use either one of the two mutation rates. The first rate is the standard mutation rate which is used by default. While using the standard mutation rate, there is no difference between HM and a CGA. The other rate is the *hypermutation rate*, which is a higher value than the standard mutation rate. If, in a generation, the time-averaged best fitness decreases, then hypermutation is triggered for the next generation. The time-averaged best fitness is an average of the best fitnesses over the last N generations. These two new parameters of hypermutation rate and N are both tuned for every test problem and are listed in Table 5.4. For the rest of the GA parameters, HM uses the same parameters as CGA which are listed in Table 5.3. The idea behind hypermutation is that a change in the problem's landscape will lead to a drop in the time-averaged best fitness, which will trigger hypermutation to temporarily increase exploration.

Table 5.4: Parameters for Hypermutation on the dynamic problems.

Problem	Hypermutation Rate	Timed-average generations
1	0.1	10
2	0.1	50
3	0.2	10
4	0.1	5
5	0.1	10
6	0.1	25
7	0.4	10
8	0.5	5
9	0.5	5

5.2.2.5 Random immigrants

RI is a GA with the addition of a new immigration mechanism. Every generation, a random subset of the population is replaced by new randomly generated individuals. In effect, a small portion of the population is re-initialized every generation. The number of individuals to replace each generation is determined by a replacement rate parameter. The replacement rate is tuned for every problem and is listed in Table 5.5. For the rest of the GA parameters, RI uses the same parameters as CGA which are listed in Table 5.3. The idea of RI is to preserve a level of population diversity through the constant input of a small number of random individuals.

Table 5.5: Parameters for RI on the dynamic problems.

Problem	Immigration Rate
1	0.01
2	0.005
3	0.2
4	0.005
5	0.005
6	0.01
7	0.05
8	0.1
9	0.2

5.2.2.6 Clearing

Clearing is a GA that encourages diversity through fitness manipulation. In Clearing, if there are individuals that are in close proximity to each other in the search space, then only a subset of those individuals will be assigned a proper fitness score and the others will be set to the worst possible fitness score. Clearing, therefore, separates the population into multiple niches, wherein only a few individuals can have a good fitness within each niche. The radius around an individual in which this clearing check is done is determined by a dissimilarity threshold parameter. This algorithm introduces two new parameters, the dissimilarity threshold and the number of individuals allowed a proper fitness per niche. We tune both of these parameters for each problem and list the tuned values in Table 5.6. For the rest of the GA parameters, Clearing uses the same parameters as CGA which are listed in Table 5.3. The idea of Clearing is to encourage the population to spread out and find new niches elsewhere in the environment. Clearing is similar to other niching methods such as Fitness Sharing Agoston E. Eiben and J. E. Smith 2003 and Crowding Kenneth Alan De Jong 1975. We use Clearing instead of these other methods because a recent study found Clearing to have better results on a set dynamic problems than these other two methods Hughes 2016.

Table 5.6: Parameters for Clearing on the dynamic problems.

Problem	Dissimilarity Threshold	Number of individuals per niche
1	0.1	5
2	0.01	25
3	0.2	5
4	0.2	10
5	0.01	1
6	0.2	25
7	0.25	5
8	0.25	10
9	0.05	25

5.2.2.7 ACROMUSE

While the other three alternative methods are relatively minor modifications of a CGA, ACROMUSE is an entire specialized algorithm designed for dynamic problems. ACROMUSE is still a GA at its core but with wholly re-worked genetic operations. It is also an adaptive algorithm; however, unlike most adaptive GAs, the adaptation mechanisms in ACROMUSE are not designed to improve fitness, but rather to maintain a high level of population diversity. ACROMUSE has two genetic operator paths, one focusing on exploration and the other on exploitation. Whenever an individual is to be changed, it determines whether to use either the exploration operations or the exploitation operations. This determination and how the operations are applied are adaptively controlled through two diversity measurements, one of which takes the fitness of the individual into account and one of which does not. By including these two diversity measurements, ACROMUSE considers both chromosomal diversity and fitness diversity. ACROMUSE does not have any extra parameters to tune; the authors of ACROMUSE use specific values for their parameters, and so we use the same. A detailed description of ACROMUSE can be found in Mc Ginley et al. (2011).

5.2.3 Dynamic Metrics

The results of dynamic problems are more difficult to analyse than static problems. The standard EC metrics for static problems alone do not give enough information for sufficient analysis. Because of this, we introduce two metrics to aid in our analysis: *Collective Mean Fitness* Morrison 2003 and *Moment of Inertia* Morrison and Kenneth A. De Jong 2001

5.2.3.1 Collective Mean Fitness

Collective mean fitness (CMF) is an alternative fitness metric that is more useful than the traditional fitness metrics on dynamic problems. In static problems, the average fitness of a GA population is expected to steadily improve over the course of its run. In dynamic problems, this is no longer true; the changes in the fitness landscape can cause disturbances or sharp drops in the fitness measurements depending on the kind of dynamics at play. Therefore, simply looking at typical fitness values at the final generation is not sufficient for accurately determining how the results of the algorithms compare. For instance, on Problem 7, looking at the fitness results of the final 500th generation doesn't tell you the overall best performing method, it merely tells you which method happened to have the highest fitness after 5 sudden landscape changes. It doesn't tell you which method had the best fitness after the 2nd, 3rd, or 4th landscape change, nor which would be best if there were a 6th, nor does it tell you which method recovers the quickest after each landscape change. Furthermore, looking at a typical fitness over generations plot for such a problem can appear confusing, with no single method consistently doing better than others over the whole duration. A more wholistic view is needed that observes the fitness values across all generations, which is where CMF comes in. The equation for CMF is:

$$F_C = \frac{\sum_{m=1}^{M}(\frac{\sum_{g=1}^{G} F_{BG}}{G})}{M} \approx F_T \tag{5.1}$$

where F_C is the collective mean fitness, M is the number of runs, G is the number of generations, F_{BG} is the best fitness of a generation of run m, and F_T is the *total mean fitness* (TMF). The total mean fitness resolves to a constant when $G = \infty$. Given a sufficiently large G, F_C approximates this constant F_T. In short, this equation monitors the running best fitness that is averaged across the runs with CMF being the running best fitness at the final generation. The effects of this equation are that the change to the running average fitness from each generation is diminished as generations increase, such that the variations in fitness from problem dynamics eventually smooth out to a steady constant value which is descriptive of the algorithm's performance across all prior generations.

5.2.3.2 Moment of Inertia Diversity Measurement

Moment of Inertia (MoI) is a measurement of the diversity of a population in an EA Morrison and Kenneth A. De Jong 2001. As has been discussed, population diversity is very important for the effectiveness of a GA on a dynamic problem. Thus, we wish to be able to quantify population diversity for analysis purposes. MoI is designed to be a computationally efficient method for computing the diversity of a population for an EA. The equation for MoI is:

$$I = \sum_{i=1}^{N} \sum_{j=1}^{P} (x_{ij} - c_i)^2 \tag{5.2}$$

Where c_i is the i^{th} coordinate of the centroid point of the population and x_{ij} is the i^{th} coordinate of the j^{th} individual. The centroid point is calculated using every individual in the population weighted equally. In the case of SAGA, which has two chromosomes, we calculate two separate

MoIs, one for each chromosome. We could instead calculate a single MoI for SAGA, which would be a combination of both chromosomes, but we wish to be able to observe if one chromosome has a noticeably different amount of diversity than the other. Furthermore, the diversity of SAGA's solution chromosome is directly comparable to the chromosomes of every other method, since each other method's single chromosome is effectively a "solution chromsome".

5.3 Results

We compare the results of each of the six algorithms on the nine dynamic problems. We go over each of the problems in turn and display the same data for each problem. For each problem, we display one table and three plots. Before we go into each problem, we first describe what is contained in the table and plots for each problem. Generally, the table reveals which method performs best, while the plots illustrate why.

Each problem's table shows, for every algorithm, the CMF with a 95% confidence interval and *Best Individual Percent* (BIP). BIP is a measure of the percent of generations in which an algorithm finds a superior best fitness than all of the other algorithms. While the averaged metric of CMF is more likely to be of interest to researchers, this BIP is more likely to be of interest to those with practical real-world applications.

We refer to the three plots of each problem as the fitness plot, CMF plot, and MoI plot. The first plot for each problem shows the fitness of each of the algorithms across the generations with fitness on the y-axis and generations on the x-axis. In this fitness plot, the solid lines represent the best of generation fitness averaged across the runs, along with a shaded 95% confidence interval. The dotted lines represent the single best fitness of that generation in any run. BIP is derived from this dotted line; where BIP is the percent of generations where the an algorithm's dotted line is superior all others. The second plot is for CMF and shows the running average best of generation fitness on the y-axis and generations on the x-axis. Each algorithm has a single line for the second plot, along with a shaded 95% confidence interval. The second plot is an illustration of the CMF calculation and the final generation of the plot is equal to the CMF in the table. The third plot for each problem shows the MoI of each of the algorithm across the generations. Recall that MoI is a measurement of the population's diversity. The y-axis shows MoI on a logarithmic scale and the x-axis shows generations. In the MoI plot, each algorithm has a single line for the MoI with the exception of SAGA, which has two lines; a solid line for the solution chromosome's MoI and a dotted line for the parameter chromosome's MoI.

Figure 5.1 and Table 5.7 show the results for problem 1. This problem gives us a baseline performance on an easy stationary landscape generated by DF1. With this problem we can see the standard behavior of each of the methods.

Table 5.7 shows that each method finds similar CMF results, with CGA and HM performing the best. CGA's good performance on this problem is not surprising, the diversity mechanisms are of no use in a static problem and can actually slow down convergence on the optimal. The table also shows that SAGA finds a BIP of 87.38%, with Clearing in second at a distance 3.24%. Although SAGA doesn't stand out here in the CMF results, it is better at finding a single high performing individual.

The fitness plot in figure 5.1 shows that each method sees similar fitnesses for both the averaged best fitness (solid line) and best fitness (dotted line) except for ACROMUSE and Clearing. ACROMUSE and Clearing find worse average best fitnesses than the other methods. This means that the diversity preservation mechanisms of these two algorithms is actually detrimental when the problem is stationary. Overall though, the fitness plot shows smaller differences between the methods than we will see in the subsequent problems. Due to the static nature of this particular problem, the CMF plot is very similar to the fitness plot, and thus doesn't highlight any new important information that is not already visible in the fitness plot. Of more interest on this problem is the MoI plot. Each method finds a steady level of MoI. As expected, CGA has the lowest diversity. We can see when hypermutation is active in HM in the MoI plot. When HM has a higher MoI than CGA, hypermutation is active, when HM has equal MoI to CGA (or HM's MoI is dropping down to CGA's from a previous hypermutation period) then hypermutation is inactive. Surprisingly, hypermutation is triggered for much of the duration of the run, from around 70 to 240 generations. Hypermutation seems unnecessary on a static problem, but it does not appear to be detrimental in this case. Also of note is SAGA's parameter chromosome, which has a noticeably higher MoI than SAGA's solution chromosome. The high diversity of the parameter chromosome makes sense; there is only a single optimal solution, but there may be many different parameter combinations that are all effective at finding this optimal solution. In fact, we will see that SAGA's parameter chromosome finds a higher MoI than the solution chromosome on all nine problems.

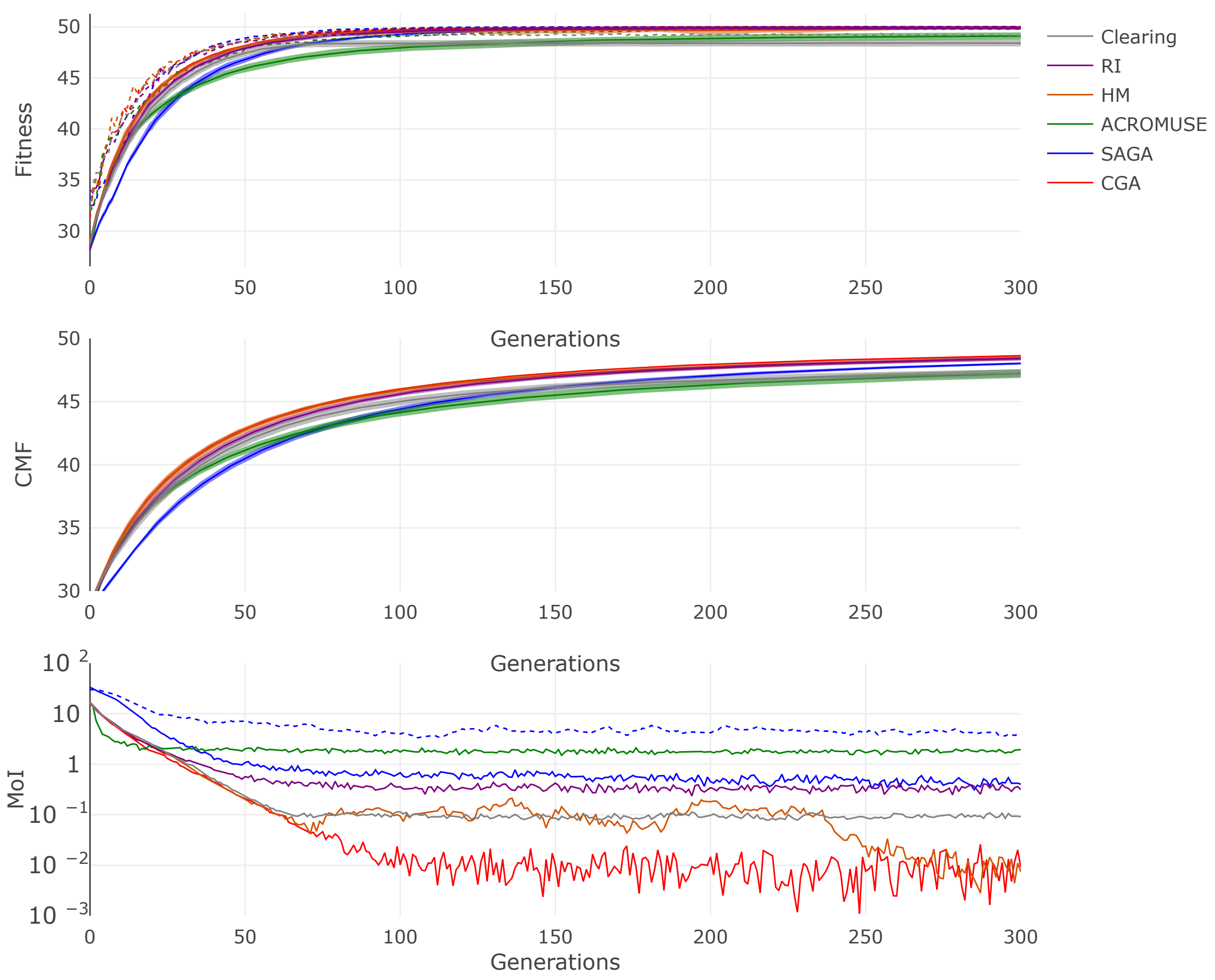

Figure 5.1: Problem 1 fitness, CMF, and MoI plots.

Table 5.7: CMF and BIP results for problem 1.

Method	CMF	BIP
CGA	**48.61 ± 0.03**	2.66%
SAGA	48.02 ± 0.09	**87.38%**
HM	**48.44 ± 0.16**	7.64%
RI	48.41 ± 0.15	0.33%
Clearing	48.05 ± 0.26	1.66%
ACROMUSE	47.22 ± 0.35	0.33%

Figure 5.2 and Table 5.8 show the results for problem 2. On this problem, every cone makes a

constant small movement in a single random dimension every generation. This problem is the easiest of the steady change problems.

Table 5.8 shows that CGA, SAGA, RI, and HM all find the best CMF with overlapping confidence intervals. At this low level of problem dynamics, CGA is still competitive. The table also shows that SAGA finds a BIP of 90.03%, with CGA second at a distant 4.32%. Taken together, these results show that SAGA is very effective on this easy dynamic problem.

The fitness plot in figure 5.2 shows strange behavior for HM, ACROMUSE lagging behind the best methods, and very similar lines for all the other methods. HM sees large drops in fitness at around 100 and 270 generations. What is happening on HM is that hypermutation is being triggered when it is not necessary. For HM, the fitness plateaus early on in the run, then a very slight downward perturbation in the fitness triggers hypermutation. Hypermutation then causes a mutation rate that is too high for the slow moving landscape, causing fitness to drop further, which then keeps hypermutation on the next generation, reinforcing the downward trend. This cycle keeps hypermutation on until the fitness bottoms out long enough for hypermutation to turn back off, allowing fitness to rise again until the process repeats. On the CMF plot, HM's inconsistency is shown as a drop of HM's CMF to settle near the CMF of ACROMUSE. On the MoI plot, CGA and HM are the only two methods that have a noticeably different MoI than in problem 1. It appears as though the small movement is small enough that a noticeable increase in population diversity is not necessary. HM's triggering of hypermutation is also visible in the MoI plot. Judging by ACROMUSE's high MoI, it seems as though the poor CMF of ACROMUSE is because it is simply creating too much diversity for the slow-moving environment. A very high level of exploration is not necessary and can actually be detrimental on this problem.

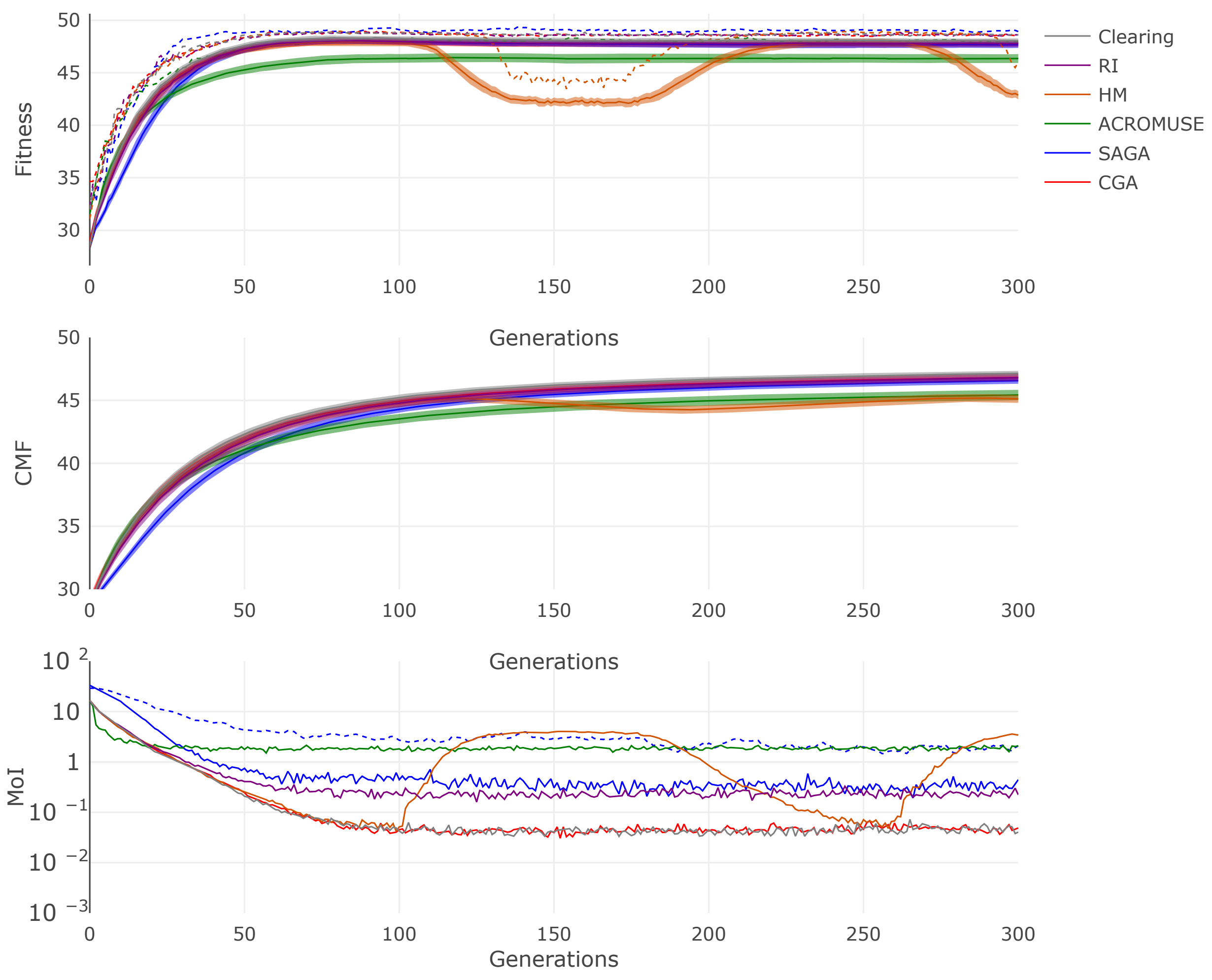

Figure 5.2: Problem 2 fitness, CMF, and MoI plots.

Table 5.8: CMF and BIP results for problem 2.

Method	CMF	BIP
CGA	**46.81 ± 0.30**	4.32%
SAGA	**46.57 ± 0.25**	**90.03%**
HM	45.12 ± 0.32	1.66%
RI	**46.81 ± 0.28**	1.99%
Clearing	**47.15 ± 0.20**	1.66%
ACROMUSE	45.43 ± 0.41	0.33%

Figure 5.3 and Table 5.9 show the results for problem 3. On this problem, every cone makes a

constant small movement in 15 random dimensions every generation. This problem is much more difficult version of problem 2.

Table 5.9 shows that HM finds the best CMF. There is also a clear split between the CMF of each of the methods, with SAGA, HM, and ACROMUSE in the superior group and CGA, Clearing, and RI in the inferior group. The CMF numbers are also much lower across the board than the previous problems. now that the amount of movement has increased, all methods have some difficulty inN finding the optimal peak. There is more competition over BIP than in the previous problems; SAGA again has the largest at 55.81%, with HM in second at 30.90%. Together, these results show that SAGA finds fairly good results; while it is outperformed in CMF, it still retains the highest BIP.

The fitness plot in figure 5.3 illustrates the competition over BIP, with the best fitness going back and forth between SAGA, HM, and ACROMUSE. Hypermutation triggering on and off is visible again, though it now seems to be beneficial for this high-movement problem. The fitness plot also shows that each method's average best fitness peaks before 50 generations, before settling to a fairly steady rate for the remainder of the run. The CMF plot illustrates the clear split in the algorithm's results that was seen in the table. The CMF plot also shows that CGA and Clearing both start off with competitive CMFs, before dropping due down to match RI. The MoI plot shows that the large movements of the cones have a noticeable impact on population diversity; each method, except ACROMUSE, has much higher MoIs than in the previous problems. In fact, ACROMUSE always returns the same level of diversity on all of the steady change problems; it is designed to preserve a minimum amount of population diversity regardless of the problem's dynamics. The MoI plot also helps explain why the fitness of each method peaks in the first 50 generations. On all methods, the MoI is at it's highest at the very beginning of the run. Even though the MoI levels remain high on this problem, they never reach the levels of diversity in the early generations. At the same time, the MoI plot shows that population diversity alone is not sufficient for ensuring good results, as RI has the highest MoI, but one of the lowest CMF.

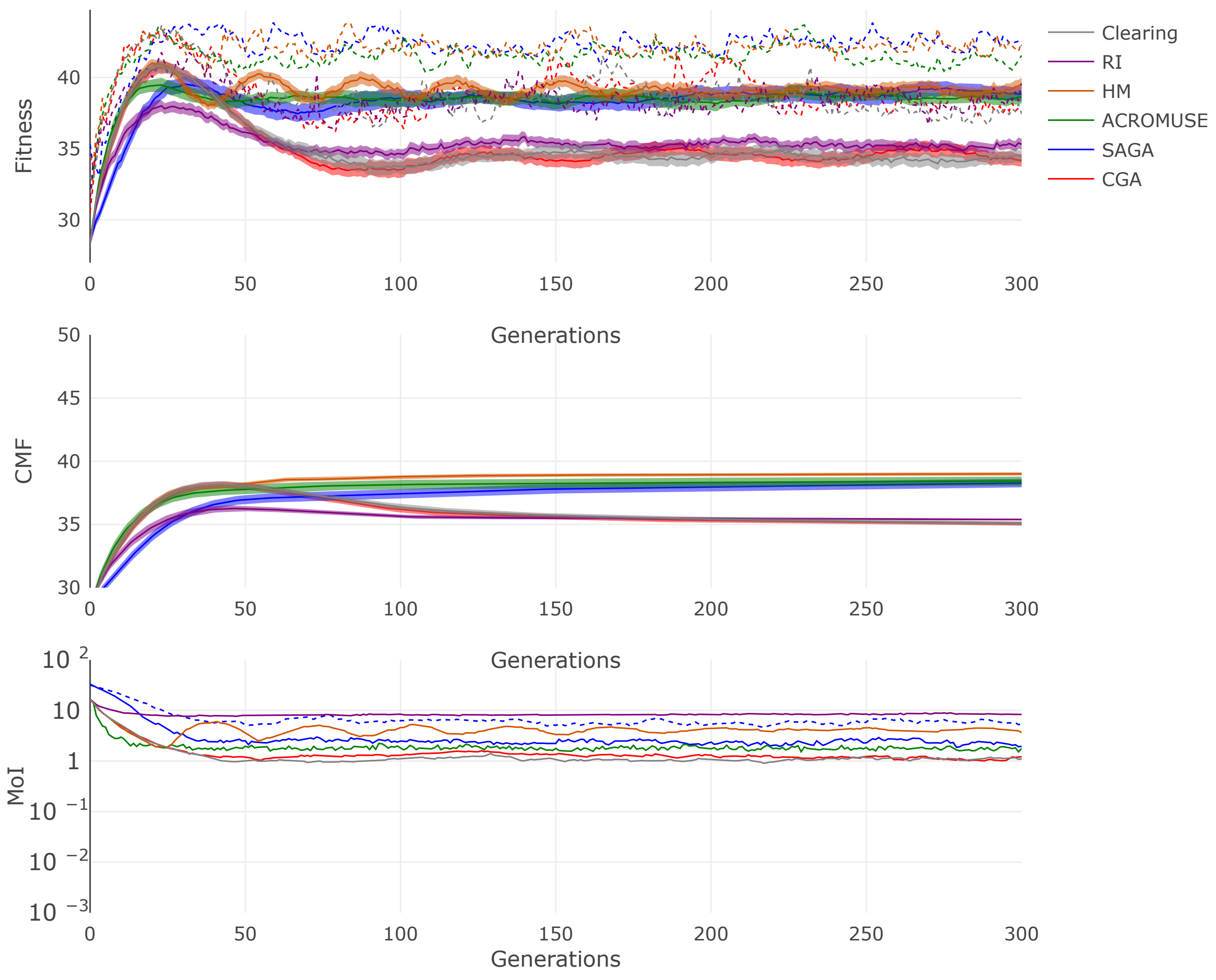

Figure 5.3: Problem 3 fitness, CMF, and MoI plots.

Table 5.9: CMF and BIP results for problem 3.

Method	CMF	BIP
CGA	35.07 ± 0.15	1.66%
SAGA	38.27 ± 0.34	**55.81%**
HM	**39.00 ± 0.13**	30.90%
RI	35.39 ± 0.06	0.33%
Clearing	35.07 ± 0.16	1.99%
ACROMUSE	38.43 ± 0.35	9.30%

Figure 5.4 and Table 5.10 show the results for problem 4. On this problem, every cone makes a

constant small movement in a varying number of dimensions. Every 100 generations the number of dimensions in which to move changes. The problem's dynamics follows a pattern in a repeating pattern of steadily increasing then steadily decreasing the number of dimensions in which to move. Compared to the previous steady change problems, this problem requires the methods to be able to adjust to the changing difficulty of the problem.

Table 5.10 shows that SAGA and HM return the highest CMFs, CGA returns the lowest, and the remaining methods all return CMFs in a similar range of each other. The table also shows that SAGA finds a BIP of 95.20% of generations, with HM second at 3.70%. SAGA performs very well here, with only HM as competition in CMF, and with a noticeably superior BIP.

The fitness plot in figure 5.4 shows the varying difficulty of the problem, where the fitness values are inversely related to the number of dimensions in which the cone movements occur. We also see that HM is one of the best methods during the more difficult portions of the problem. But again, HM occasionally triggers hypermutation when it should not, causing occasional fitness drops during the easier portions of the problem. The CMF plot is now very useful for sorting out the chaos of the fitness plot. The CMF plot shows that while many methods start off well, over the generations, a split forms that separates SAGA and HM from the other methods. The MoI plot shows that the population diversity of most of the methods matches the amount of landscape dynamics of the problem. The periods of higher landscape dynamics matches up with the periods of higher MoIs. ACROMUSE is again the exception here, with a constant level of diversity throughout the generations.

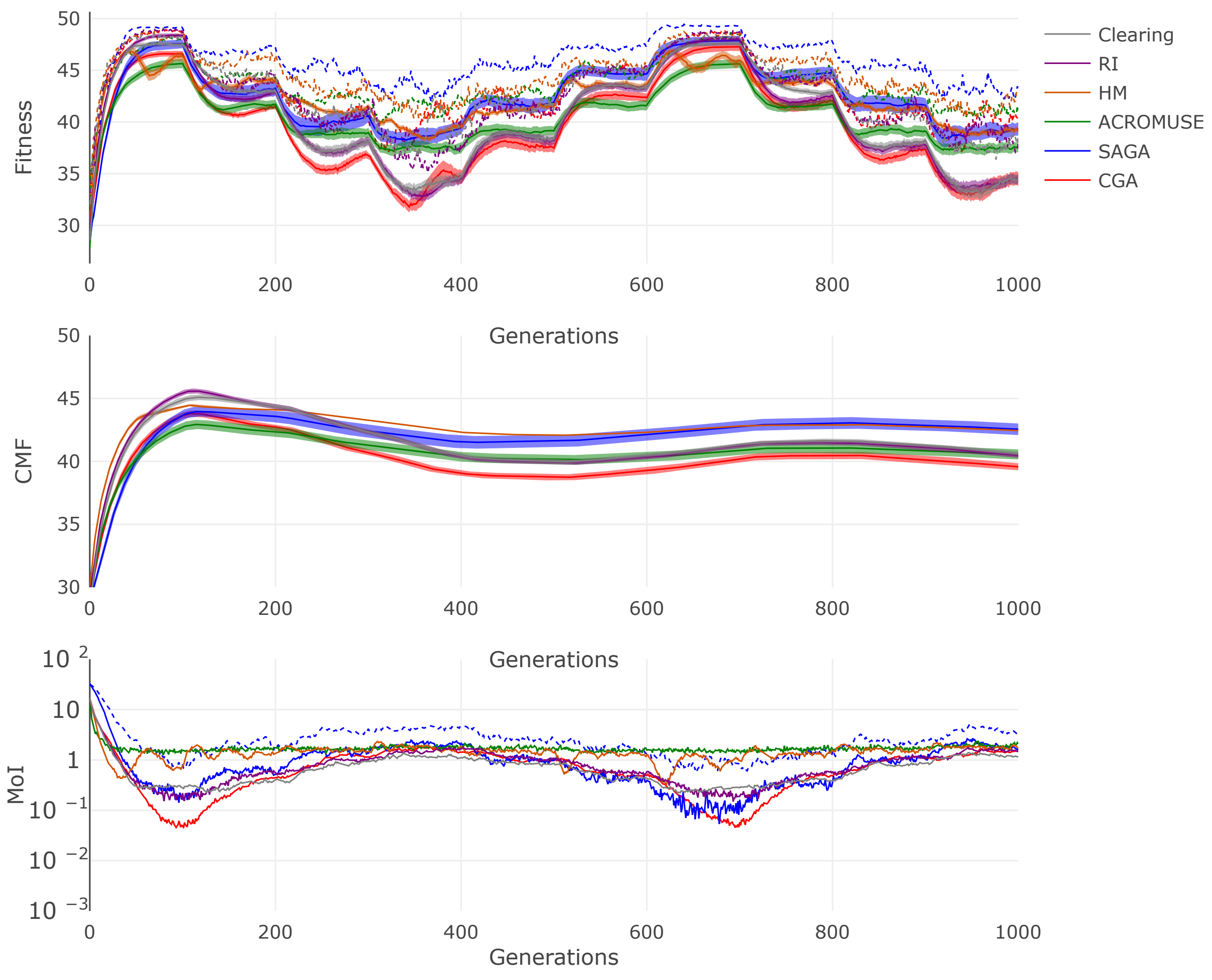

Figure 5.4: Problem 4 fitness, CMF, and MoI plots.

Table 5.10: CMF and BIP results for problem 4.

Method	CMF	BIP
CGA	39.57 ± 0.28	0.00%
SAGA	$\mathbf{42.55 \pm 0.46}$	**95.20%**
HM	$\mathbf{42.45 \pm 0.02}$	3.70%
RI	40.47 ± 0.19	0.10%
Clearing	40.62 ± 0.16	0.90%
ACROMUSE	40.55 ± 0.40	0.10%

Figure 5.5 and Table 5.11 show the results for problem 5. This problem has the same setup as

problem 4, except that the change interval is 50 generations instead of 100. Compared to problem 4, the methods now have less time to converge and less time to adjust to the changing dynamics of the problem.

Table 5.11 shows similar results to the last problem. SAGA and HM still return the best CMFs with overlapping confidence intervals. The only noteworthy change is ACROMUSE, which sees improved CMF on this problem as compared to the previous problem. The table also shows that SAGA finds a BIP of 92.61% and HM finds it on 3.70%. SAGA again performs very well on this problem; shortening the problem's change interval from 100 to 50 generations does not have a noticeable effect on SAGA's performance.

The fitness plot in figure 5.5 shows that all three plots show similar behavior as in problem 4 for all methods. The fitness plot is again very chaotic, such that it is easier to draw conclusions from the CMF plot. The CMF plot shows that CGA, RI, and Clearing are all clearly outperformed by the other methods. Overall, however, it doesn't show any new information that isn't already gleaned from the table or from the previous problem. The MoI plot is also similar to the previous problem, but it does show that the methods of CGA and Clearing find lower MoIs that in the previous problems during the difficult portions of the problem. This lower diversity on those methods can be explained as being due to the shorter change intervals not allowing these methods as much time to adjust to the problem's changing dynamics.

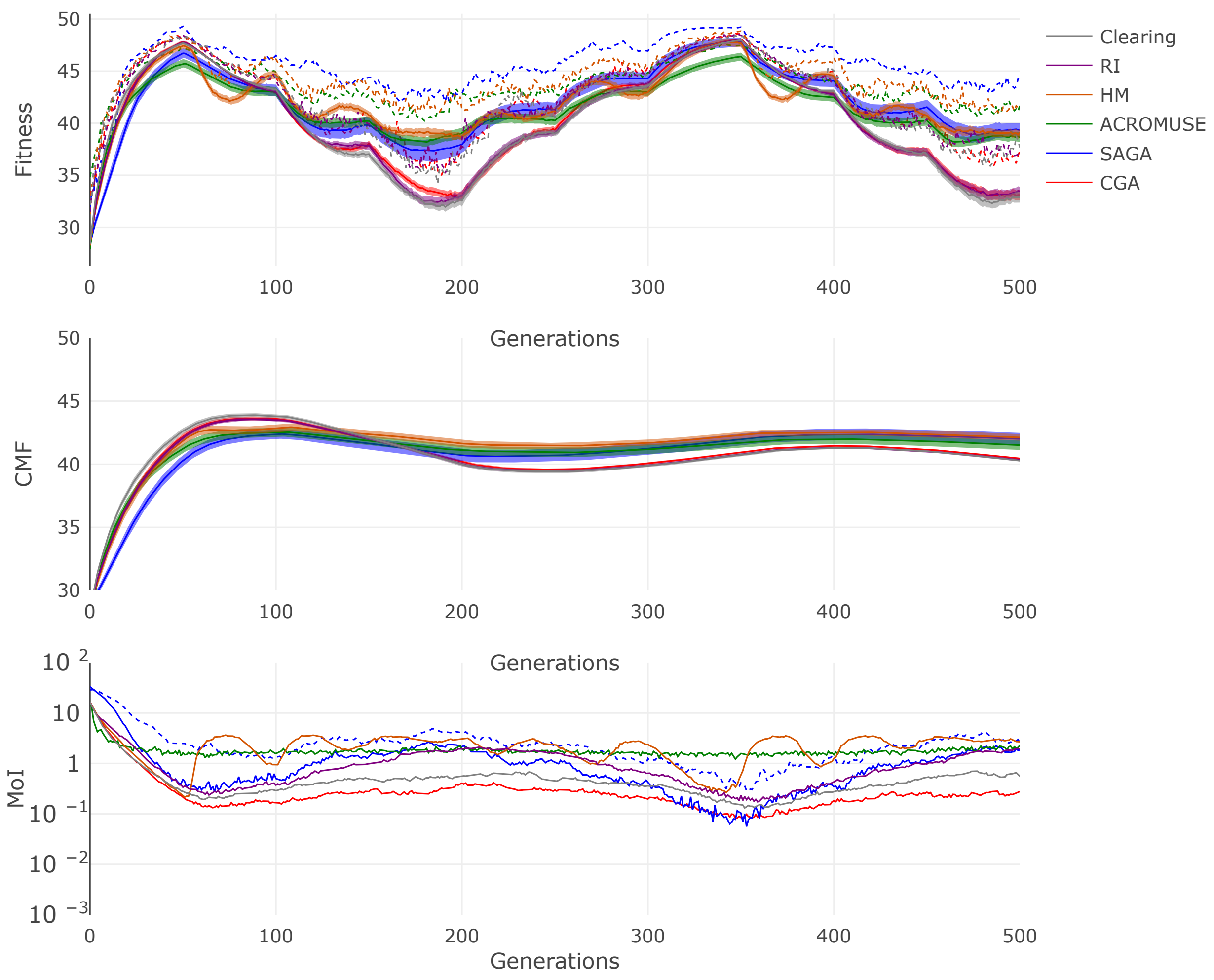

Figure 5.5: Problem 5 fitness, CMF, and MoI plots.

Table 5.11: CMF and BIP results for problem 5.

Method	CMF	BIP
CGA	40.46 ± 0.06	0.00%
SAGA	**42.01 ± 0.48**	**92.61%**
HM	**42.13 ± 0.26**	3.70%
RI	40.34 ± 0.10	0.10%
Clearing	40.29 ± 0.07	0.90%
ACROMUSE	41.52 ± 0.38	0.10%

Figure 5.6 and Table 5.12 show the results for problem 6. This problem has the same setup as

problems 4 and 5, except that the change interval is now only 20 generations. Compared to problems 4 and 5, each method now has very little time to adjust to converge or adjust to the changing dynamics of the problem.

Table 5.12 shows that HM now returns a clearly higher CMF than the other methods with SAGA and ACROMUSE in second. Nevertheless, SAGA still finds a BIP of 92.52% with Clearing in second at only 3.24%. These results for SAGA are fairly good and similar to problems 1, 3, and 3, where SAGA's CMF is outperformed by HM but SAGA has the best BIP

The fitness plot in figure 5.6 shows the same trends witnessed in the previous two problems. The CMF plot is also shows very similar to the previous two problems, with the exception that HM now clearly has the highest CMF through most generations. In the MoI plot, we can see why HM does very well on this problem. With the short intervals of this problem, HM is able to keep hypermutation on during the entire period of increasing problem difficulty, then turn it off for the entire period of decreasing difficulty. The MoI of the other methods are very similar to the previous two problems.

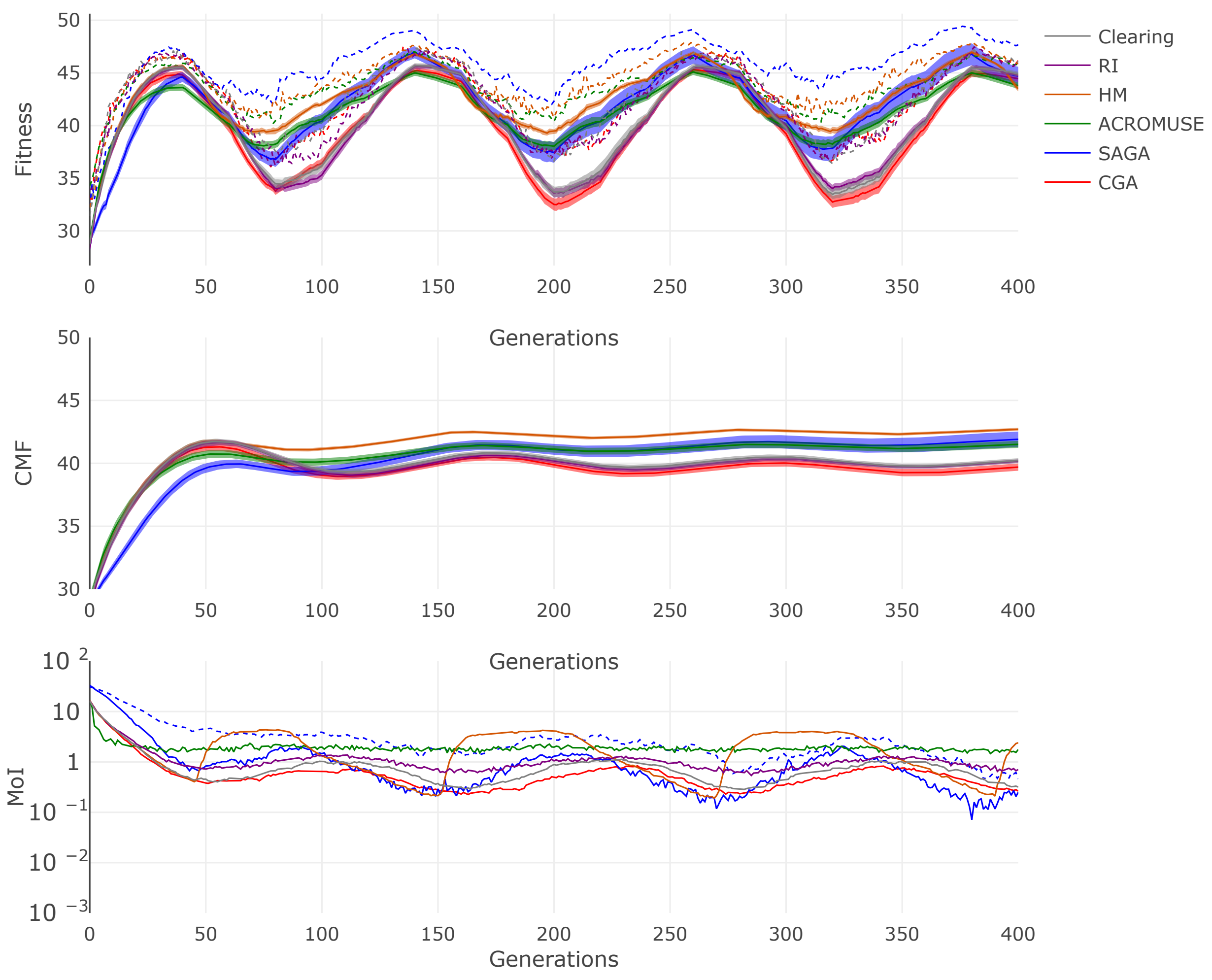

Figure 5.6: Problem 6 fitness, CMF, and MoI plots.

Table 5.12: CMF and BIP results for problem 6.

Method	CMF	BIP
CGA	39.70 ± 0.29	1.50%
SAGA	41.91 ± 0.62	**92.52%**
HM	$\mathbf{42.71 \pm 0.12}$	1.00%
RI	40.17 ± 0.14	0.50%
Clearing	40.15 ± 0.22	3.24%
ACROMUSE	41.51 ± 0.28	1.25%

Figure 5.7 and Table 5.13 show the results for problem 7. On this problem, the fitness landscape

is static but undergoes a complete re-initialization every 100 generations, causing a sudden and drastic change in the landscape.

Table 5.13 shows that HM returns the highest CMF with SAGA in second. SAGA finds a BIP of 85.23% with ACROMUSE in second at 7.98%. SAGA performs fairly well on this problem, with the second best CMF and the largest BIP.

The fitness plot in figure 5.7 shows a very clear and sharp drop in the fitness of all methods every time the fitness landscape is re-initialized. After each landscape re-initialization, the fitnesses then rise back up over the next 100 generations to around the level they were before the re-initialization. This fitness rise is clearly slower in CGA and Clearing. In the CMF plot, we see that the slow rising fitness of CGA and Clearing is very detrimental to their CMF. The CMF of CGA and Clearing are competitive with the other methods only up until the first landscape change at 100 generations. In the MoI plot, we see that every method sees an increase in population diversity at the every landscape change. Most methods see a very sharp and immediate increase in MoI, while CGA and Clearing see a more gradual increase, explaining their slower fitness rise. Even ACROMUSE shows an increase of MoI after each change, whereas on the steady dynamic problems, ACROMUSE always remained at a constant MoI. HM is well-suited to these category of dynamic problems, as the drastic change every 100 generations is a clear signal to trigger hypermutation.

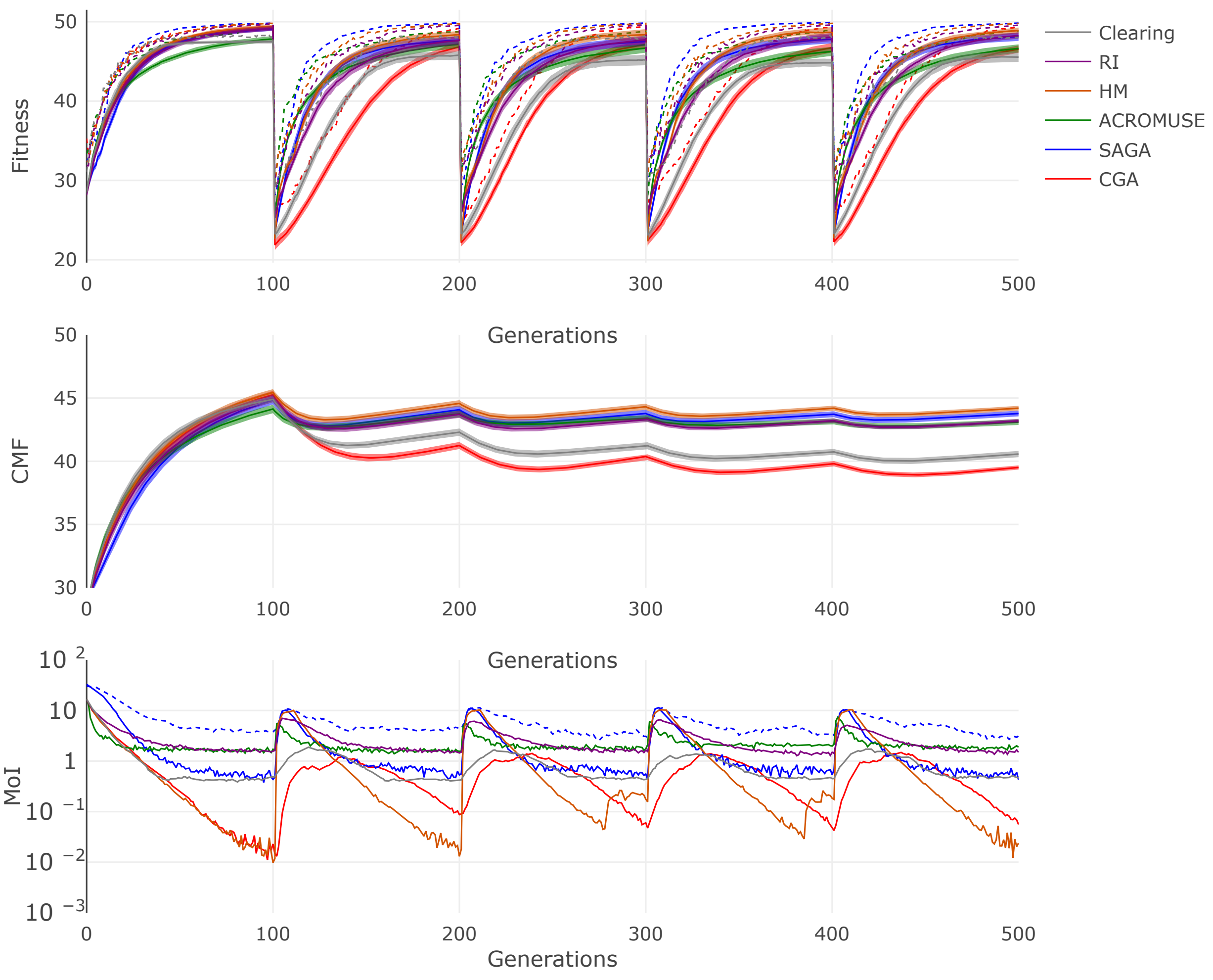

Figure 5.7: Problem 7 fitness, CMF, and MoI plots.

Table 5.13: CMF and BIP results for problem 7.

Method	CMF	BIP
CGA	39.51 ± 0.17	1.00%
SAGA	43.78 ± 0.23	**85.23%**
HM	**44.20 ± 0.18**	2.79%
RI	43.20 ± 0.15	0.60%
Clearing	40.82 ± 0.20	2.40%
ACROMUSE	43.10 ± 0.23	7.98%

Figure 5.8 and Table 5.14 show the results for problem 8. This problem has the same setup as

problem 7, except that the landscape change occurs every 50 generations. Compared to problem 7, the methods now have less time to converge and less time to adjust to the changing dynamics of the problem.

Table 5.14 shows that HM, SAGA, and ACROMUSE all find the best CMFs with overlapping confidence intervals. SAGA finds a BIP of 69.66%, a noticeable drop from the previous problem, but still the highest BIP with ACROMUSE in second at 15.17%. SAGA perfoms very well on this problem with a competitive CMF and the best BIP.

The fitness plot in figure 5.8 shows similar results to the previous problems. There is a clear drop in the fitness of every method every 50 generations and CGA and Clearing have a slower fitness increase than the other methods. The CMF plot is again very similar to the previous problem, with a couple differences. First, SAGA has the worst CMF before the first fitness change, but then surpasses the other methods over the remaining generations. Second, RI's CMF is competitive for the first few landscape changes, but eventually falls behind HM, SAGA, and ACROMUSE. The MoI plot shows similar behavior to the previous problem for each method, with increases after each landscape change, though now the MoI of every method all have higher diversity levels throughout.

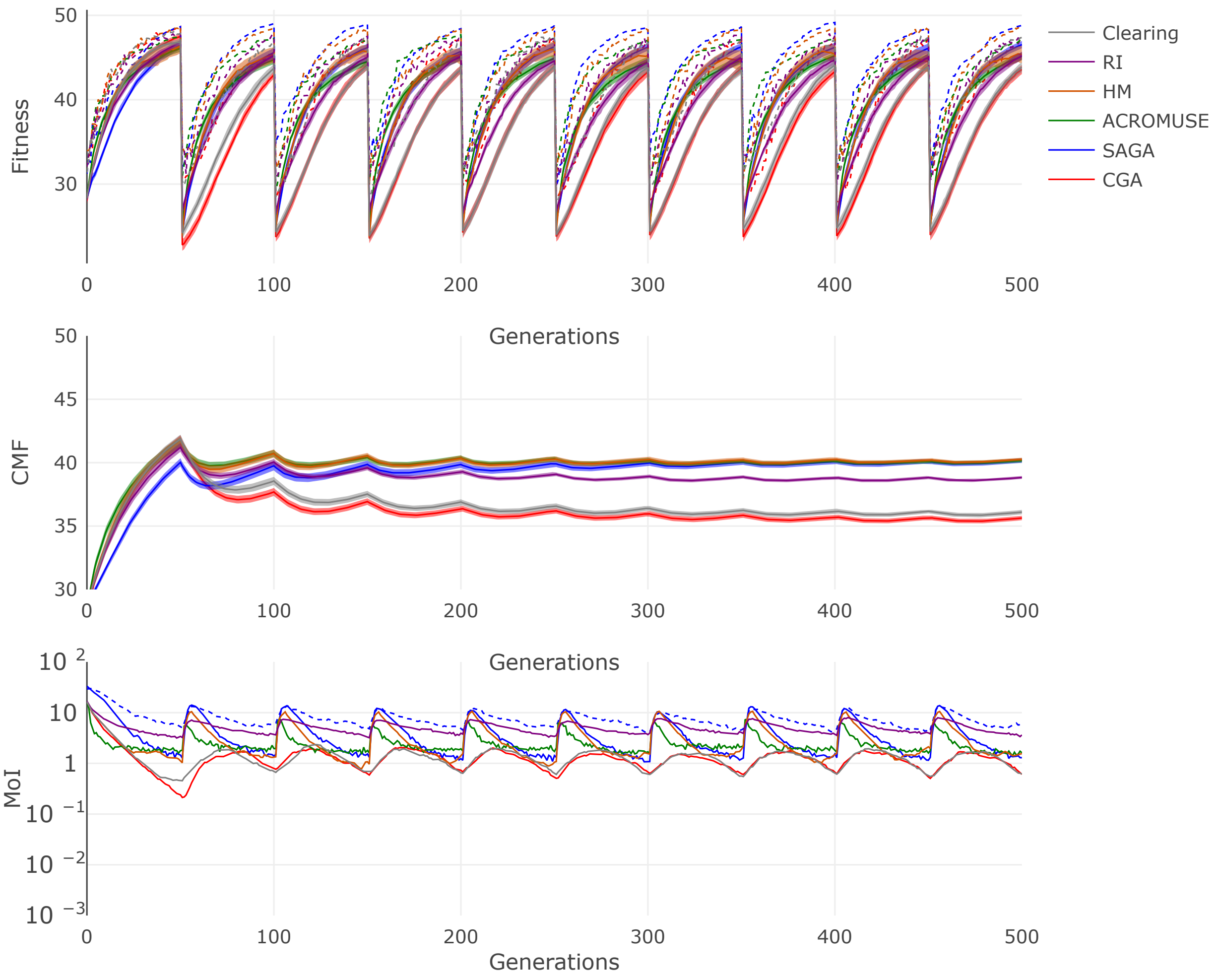

Figure 5.8: Problem 8 fitness, CMF, and MoI plots.

Table 5.14: CMF and BIP results for problem 8.

Method	CMF	BIP
CGA	35.64 ± 0.17	2.00%
SAGA	**40.17** $\pm$ **0.18**	**69.66%**
HM	**40.27** $\pm$ **0.13**	9.38%
RI	38.83 ± 0.11	1.20%
Clearing	36.10 ± 0.17	2.59%
ACROMUSE	**40.02** $\pm$ **0.15**	15.17%

Figure 5.9 and Table 5.15 show the results for problem 9. This problem has the same setup as

problems 7 and 8, except that the landscape change occurs every 20 generations. Compared to problems 7 and 8, each method now has very little time to adjust to converge or adjust to the changing dynamics of the problem.

Table 5.15 shows significantly different results from the previous two sudden change problems. HM and ACROMUSE return the best CMF while SAGA's CMF has dropped considerably from the previous sudden change problems such that it only slightly outperforms CGA. The BIP of SAGA also drops to 0.0% while HM finds the highest BIP of 78.84%. SAGA performs poorly on this problem with both HM and ACROMUSE clearly outperforming it on both metrics.

The fitness plot in figure 5.9 is very chaotic and hard to read, but we still include it for the sake of consistency. The CMF plot is, therefore, very useful on this problem. The CMF plot shows that starting from the first landscape change at 20 generations, HM and ACROMUSE find superior CMF to any other method. CGA start off well, but then drops in CMF once the landscape begins changing at 20 generations. In the CMF plot, SAGA is steadily in the middle of the pack as compared to the other methods. In the MoI plot, we see very high levels of diversity, as compared to previous problems, across all generations. Even though the methods see diversity decreases after each landscape change, the interval is short enough that the MoI doesn't drop much before the next landscape change, keeping overall diversity levels high across all generations.

This is the only dynamic problem on which SAGA performs poorly. SAGA's poor performance is most likely due to the fact that the short change interval of this problem means that SAGA does not have sufficient time to adjust parameters sufficiently before the next change occurs. The addition of the parameter chromosome on SAGA means that SAGA has a larger search space than the other methods, resulting in SAGA commonly having slower initial fitness improvement than the other methods for a few generations after a landscape change. In the 50 and 100 interval problems, this slower start is fine as these intervals are sufficient long for SAGA to make up for it's slower start and surpass the other methods before the next landscape change occurs. The short interval of 20 generations, however, proves to be too short of a period such that SAGA never catches up to the other methods before the next landscape change.

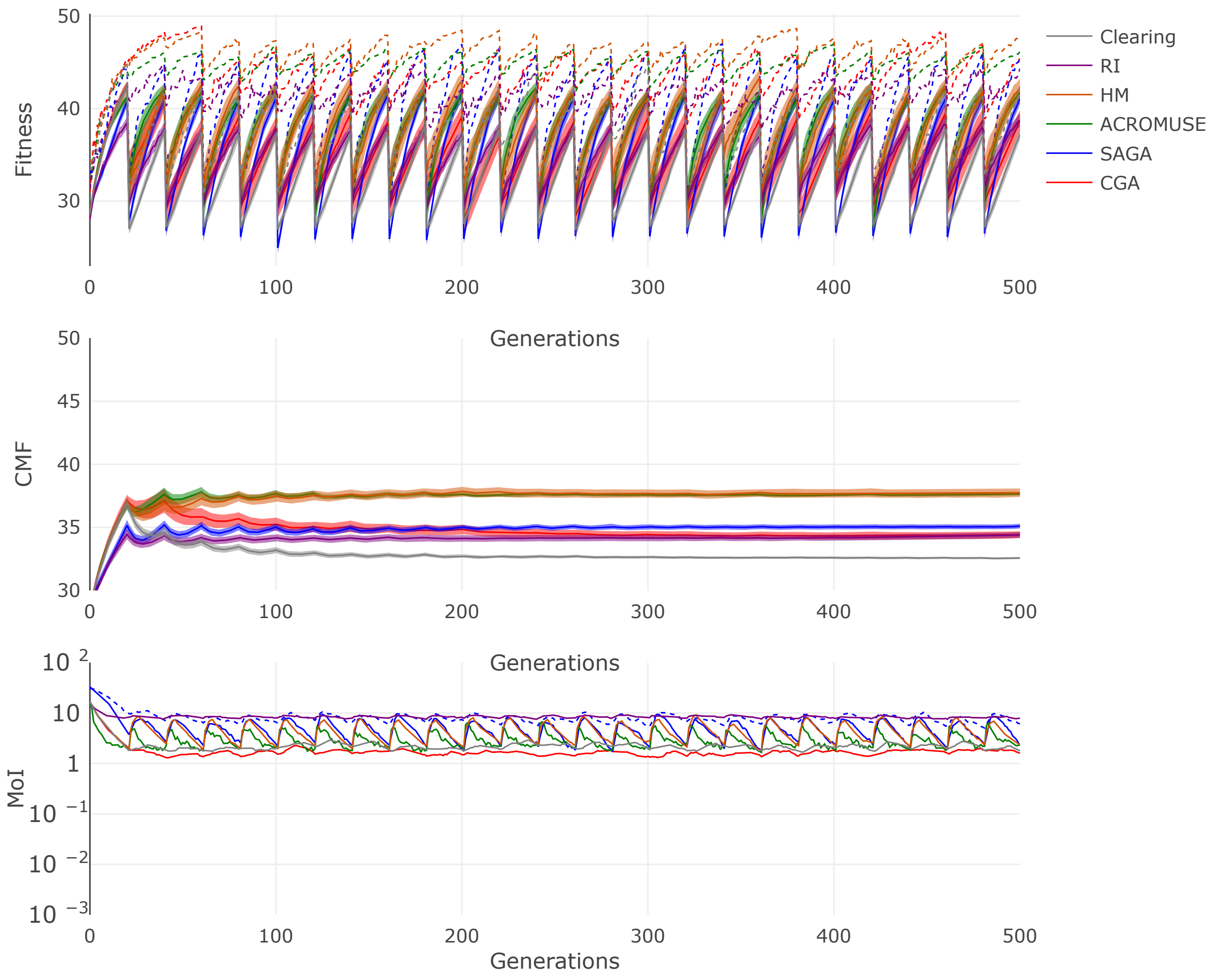

Figure 5.9: Problem 9 fitness, CMF, and MoI plots.

Table 5.15: CMF and BIP results for problem 9.

Method	CMF	BIP
CGA	34.41 ± 0.27	13.57%
SAGA	35.10 ± 0.18	0.00%
HM	**37.75** ± **0.34**	**78.84%**
RI	34.40 ± 0.23	0.00%
Clearing	32.57 ± 0.09	0.80%
ACROMUSE	**37.53** ± **0.11**	6.79%

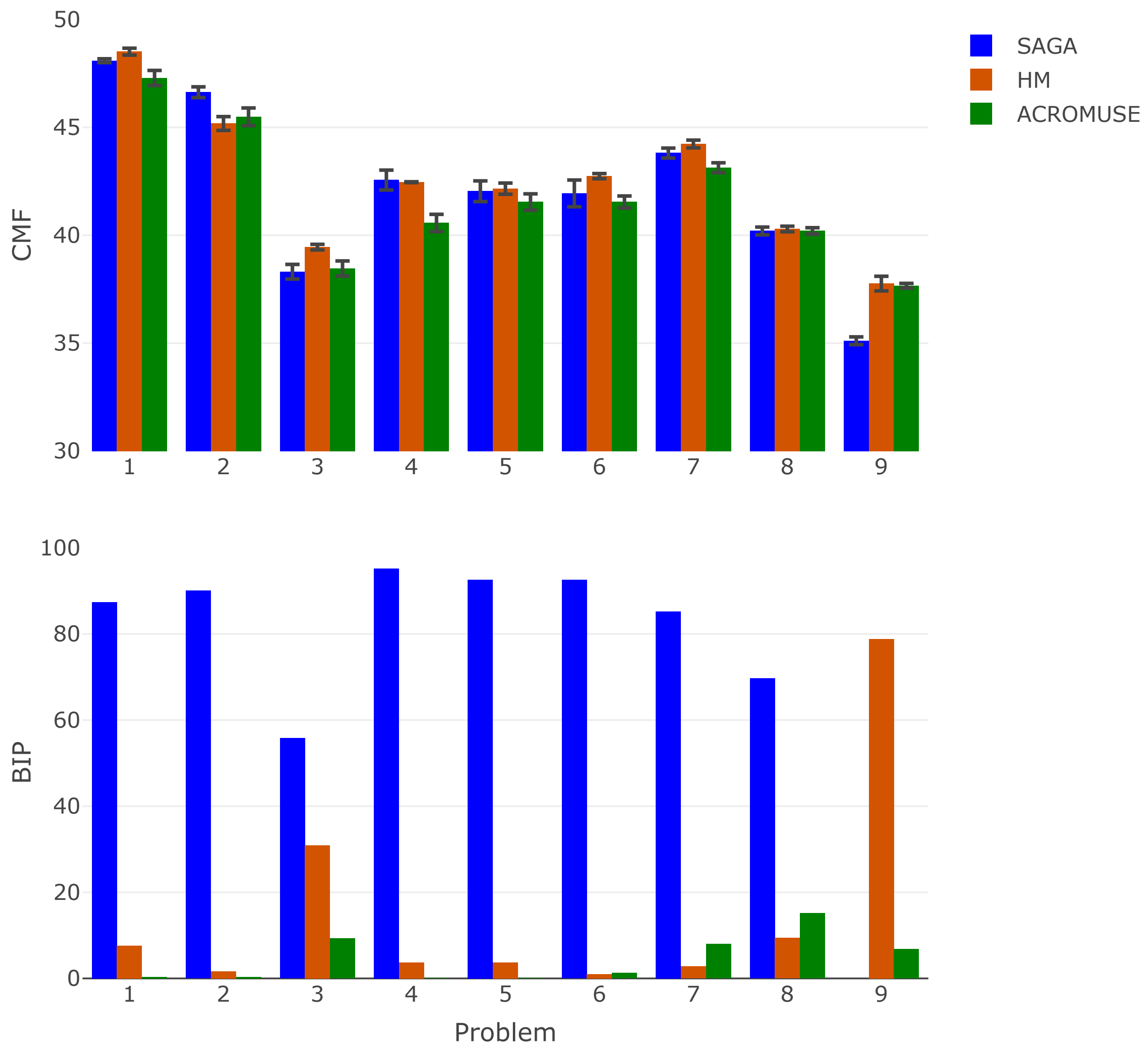

Figure 5.10: Summary of the results of SAGA, HM, and ACROMUSE on dynamic problems.

5.3.1 Summary of Results

We give a recap of the most important results from the nine dynamic problems. The plots in Figure 5.10 show a summary of the CMF and BIP results of SAGA, HM, and ACROMSUE in bar chart form. This figure displays the data from Tables 5.7 through 5.15. The methods of CGA, RI, and

Clearing are convincingly outperformed by SAGA, HM, and ACROMUSE, so we exclude them here to avoid excessive clutter in the plots. In the CMF plot, we see that each of the three methods find similar CMFs across most problems. On many of the problems, there is no single method that significantly outperforms the others in CMF. The one noteworthy standout is problem 9, where SAGA under-performs. Problem 9's fitness landscape changes are too frequent for SAGA to be able to adapt its parameters before the next landscape change. In the BIP plot, we see that SAGA vastly out-performs the other two methods on all but problem 9, where SAGA itself is vastly out-performed by HM. It should be noted that the BIP metric is quite stark in its results because there are no points for coming in second place. A method simply either has the best individual or it does not, and only a single method can have this distinction on a given generation. Nevertheless, for practical real-world uses of these algorithms, a single best result is what the user desires. Thus, from these results we can confidently say that SAGA is a very capable GA method for dynamic problems, because on all but one problem, SAGA finds the best or near best CMF and the best BIP.

5.4 Analysis of the Self-Adaptive Parameters

We analyze whether adaptation is occurring within SAGA and if it is adapting parameters appropriately. In Chapter 4, we already observed that SAGA tends to evolve specific rates and operators over others depending on the problem at hand. Instead of repeating that analysis, in this section, we are interested to see if there are any changes to SAGA's parameters that line up with the problem's dynamics, particularly on the problems where the problem dynamics are not constant through all generations. First, we look into the parent selection tournament size, then the crossover and mutation rates, followed by the crossover and mutation operator usage.

In the analysis of SAGA in this section, our interest is in the differences in parameter adaptation that are due solely to the different dynamics of each problem. We know that SAGA's operator selection method has a significant influence on how SAGA's parameters adapt. Therefore, in order to isolate the differences in parameter adaptation that are solely due to the differences in the dynamics of each problem, we must use a consistent operator selection method across all problems in this analysis. The operator selection method we use is tournament with $size = all$ and $winnerprobability = 0.9$.

5.4.1 SAGA Parent Selection Tournament Size

We take a look into the self-adaptive parent tournament selection size to see if it is adapting the selection pressure according to the changing environments of the dynamic problems. Recall that the parent tournament selection size controls the selection pressure of the algorithm such that a larger tournament size equates to a higher selection pressure. Figure 5.11 shows the parent selection tournament size evolved by SAGA on each of the nine dynamic problems. There are nine plots, one for each problem. The plots show the mean parent selection tournament size with a 95% confidence interval. The y-axes show the mean tournament size and the x-axes show generations. The plots for problems 4 through 9 also show vertical dotted lines. These dotted lines represent when a change in the problem's dynamics occurs. In the case of problems 4 through 6, these lines represent when there is a change in the number of dimensions in which the cones move, and the numbers along the top of the plots denote the number of dimensions in which the cones are moving. For problems 7 through 9, the dotted lines represent when the fitness landscape is re-initialized.

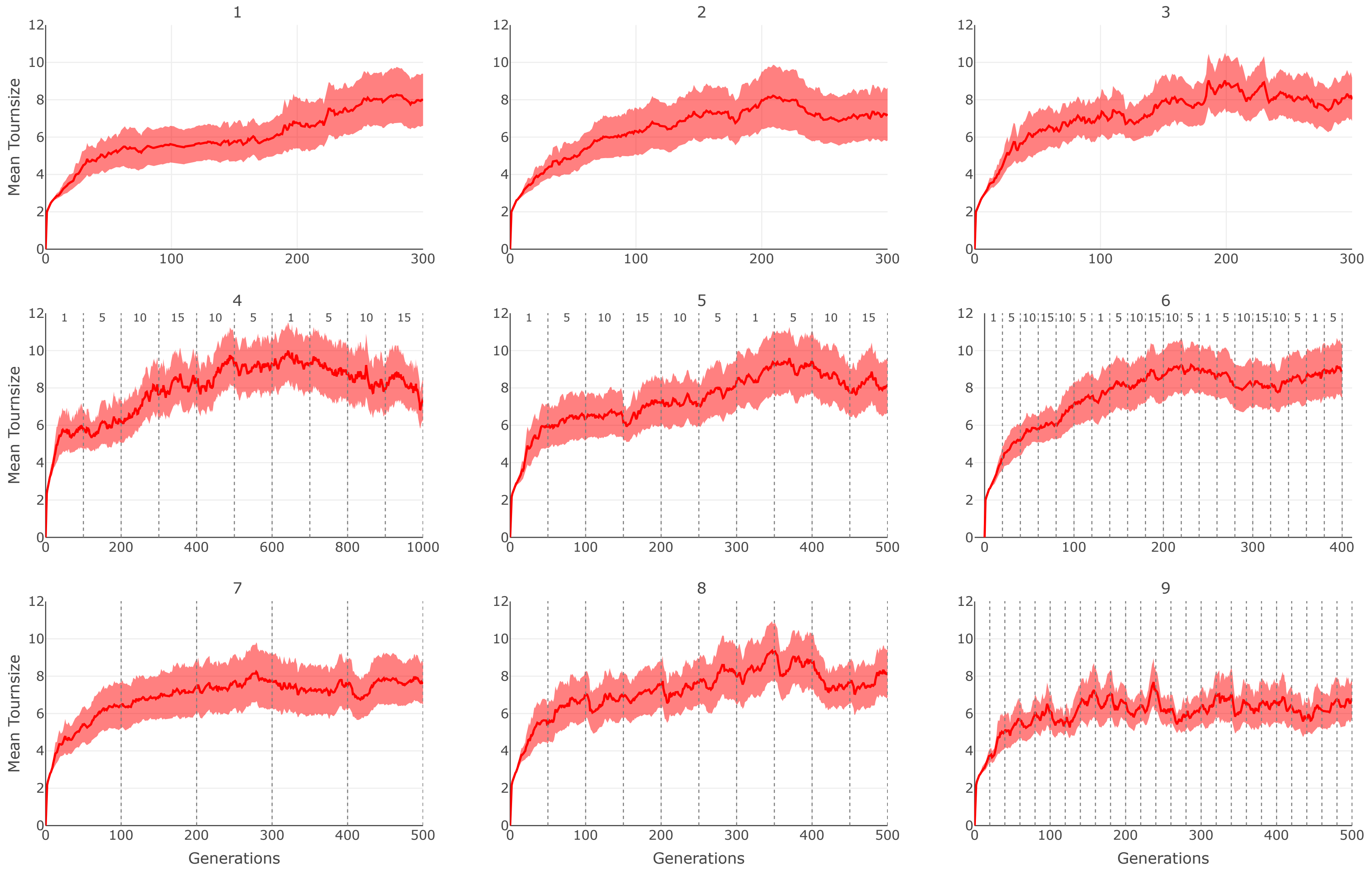

Figure 5.11: Mean parent selection tournament size across all runs of the best SAGA configuration on the dynamic problems.

The plots are very similar on all of the steady change problems, 2 through 6. There is no consistent increase or decrease of the tournament size on the variable steady problems of 4, 5, and 6 to match the varying difficulty of the problem. There is some adaptation of the tournament size on the sudden change problem. On problems 8 and 9 a small decrease of the tournament size can be observed at many of the change intervals. This drop, however, is very small and inconsistent; the drop does not occur at every change interval. In order to make a confident assertion that parent tournament size is adapting to the problem dynamics, we would have to see a clear and consistent change at every drop interval. The inconsistent drop that we sometimes observe does not satisfy this criteria. Thus, we cannot claim that parent tournament size is a primary tool of adaptation on dynamic problems for SAGA.

5.4.2 SAGA Operator Rates

We take a look into the crossover and mutation rates to see if these rates increase or decrease according to the problems' dynamics. Figures 5.12 and 5.13 show the crossover rates and mutation rates, respectively, evolved by SAGA on each of the nine dynamic problems. The figures each contain nine plots, one for each problem. The plots show the mean rates with a 95% confidence interval. The y-axes show the rate and the x-axes show generations. In Figure 5.13, the mutation rate figure, the y-axes are on a logarithmic scale. The plots for problems 4 through 9 also show vertical dotted lines. These dotted lines represent when a change in the problem's dynamics occurs. In the case of problems 4 through 6, these lines represent when there is a change in the number of dimensions in which the cones move, and the numbers along the top of the plots denote the number of dimensions in which the cones are moving. For problems 7 through 9, the dotted lines represent when the fitness landscape is re-initialized.

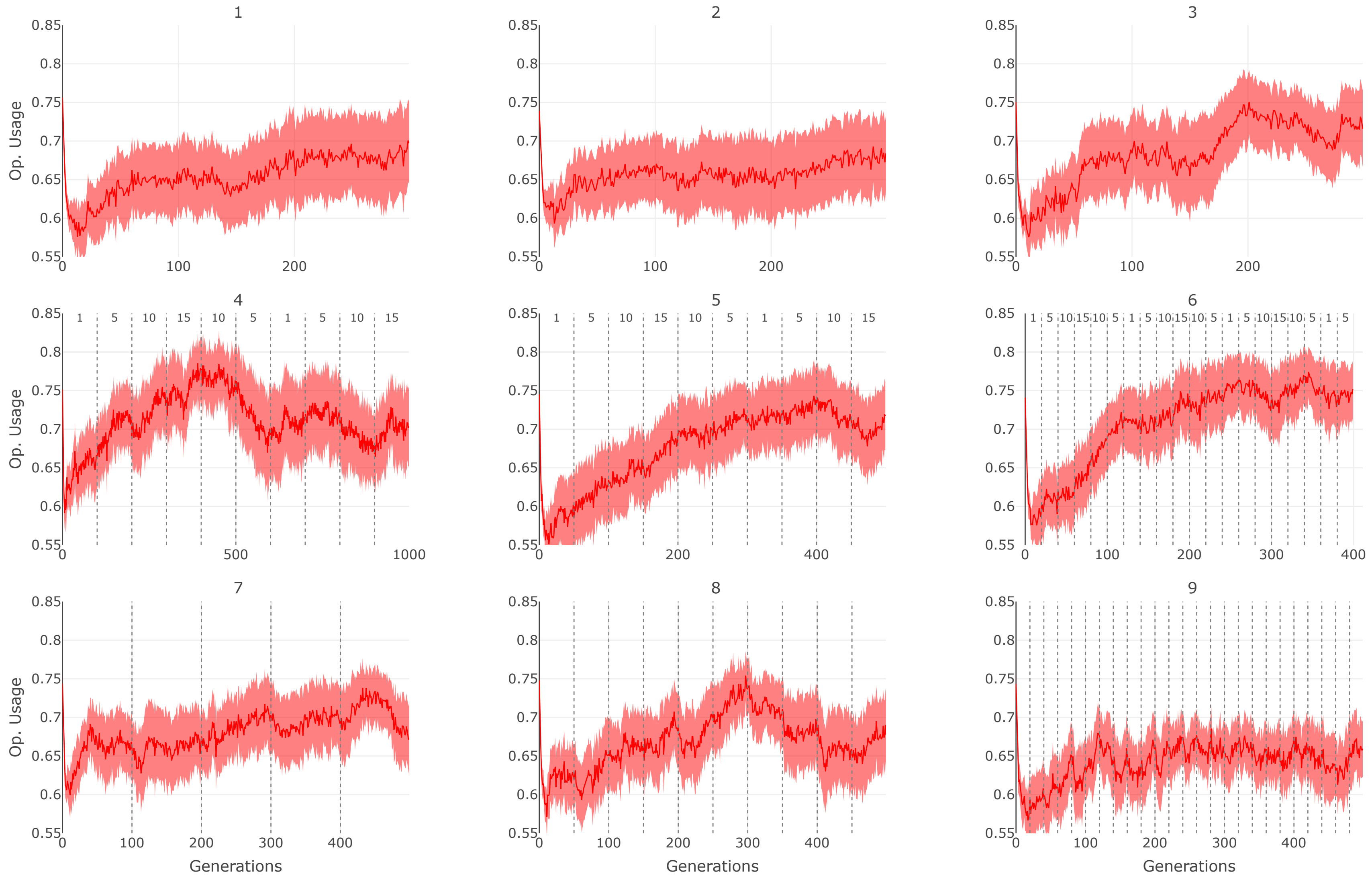

Figure 5.12: Mean crossover rates across all runs on the dynamic problems.

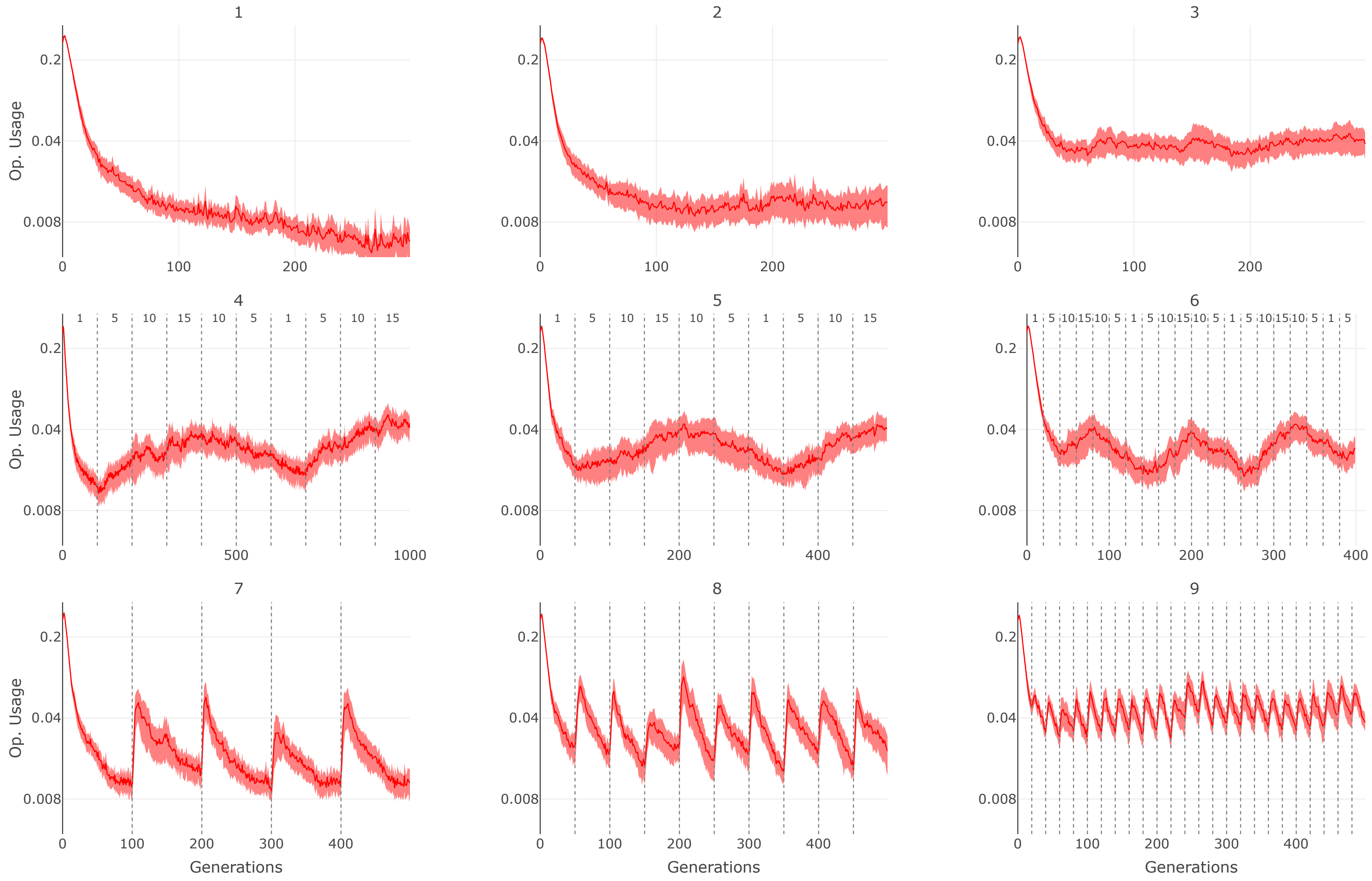

Figure 5.13: Mean mutation rates across all runs on the dynamic problems.

First, we look at Figure 5.12, the crossover rates figure. Problem 1 is a static problem and so adaptation of crossover rate neither expected nor observed. Problems 2 and 3 are steady change problems, but the rate of change is constant throughout, so again, adaptation of crossover rate is neither expected nor observed. Problems 4 through 6 are steady change problems where the number of dimensions in which the cones move vary over the generations. In these three problems we may expect adaptation in crossover rate to correspond to the problem's changing dynamics. This adaptation, however, is not observed, there is no correlation between the crossover rate and the number of dimensions in which the cones are moving. Problems 7 through 9 are the sudden change problem, where the fitness landscape is totally re-initialized at each change interval. On these sudden change problems, there appears to be some adaptation to crossover rate due to the sudden fitness landscape changes, but it is not consistent. There is occasionally a small drop of crossover rate when the landscape is changed. This drop happens often enough to suggest that it may be more than just noise, but not often enough that it is a reliable adaptation that SAGA is making to handle the problem's dynamics. Even if this drop is more than noise, the change is so small such that it is not expected to have a significant impact on the evolutionary behavior of SAGA.

Next we look at Figure 5.13, the mutation rates figure. On problem 1 there is a steadily decreasing mutation rate, much like in the static problems of Chapter 4. On problems 2 and 3, we see mutation decrease to a certain value, where it remains steady for the remainder of the generations. On these first three problems, we see that the magnitude of the mutation rate is correlated to the amount of change occurring in the problem. Problem 3 has more cone movement than problem 2, which has more than problem 1. Similarly, Problem 3 settles on a higher mutation rate than problem 2, which settles on a higher mutation rate than problem 1. On problems 4 through 6, we do see a wave-like variation in the mutation rates, where the rates alternate between increasing and decreasing over the generations. These rates correlate to the number of dimensions in which the cones are moving;

the mutation rate increases when the number of dimensions in which the cones move increases, and vice-versa. This adaptation of the mutation rate holds true on each of these three problems. On problems 7 through 9, every time the fitness landscape undergoes a drastic change, SAGA adapts through a sudden and temporary increase of the mutation rate. The mutation rate then gradually decreases to its prior levels. On these three problems, SAGA is acting similarly to HM, where there is a sudden increase of mutation rate when certain conditions are met. Unlike HM, however, SAGA achieves this behavior without the need to hard-code in a trigger for the increased mutation rate, nor tune for the parameters that control this trigger. SAGA also allows the mutation rate to gradually decrease after each spike, whereas HM is limited to two discrete rates.

From these two figures combined, we can see that SAGA is generally not adapting crossover rate for the problem dynamics, but it is adapting the mutation rate. The evolved crossover rates are uncorrelated to the dynamics of the problems, with some potential minor and inconsistent exceptions on the sudden change problems. Mutation rate, on the other hand, shows a clear correlation between the evolved mutation rates and the dynamics of the problem on all nine problems. Thus, the parameter of mutation rate is a primary tool in which SAGA is adapting to dynamic problems.

5.4.3 SAGA Operator Usage

We examine crossover and mutation operator usage to see if the operators that SAGA uses most often vary according to the problems' dynamics. Figures 5.14 and 5.15 show how often each crossover operator and mutation operator, respectively, are used every generation on each of the nine dynamic problems. The figures each contain nine plots, one for each problem. The plots show the mean operator usages with a 95% confidence interval. The y-axes show operator usage and the x-axes show generations. The plots for problems 4 through 9 also show vertical dotted lines. These dotted lines represent when a change in the problem's dynamics occurs. In the case of problems

4 through 6, these lines represent when there is a change in the number of dimensions in which the cones move, and the numbers along the top of the plots denote the number of dimensions in which the cones are moving. For problems 7 through 9, the dotted lines represent when the fitness landscape is re-initialized.

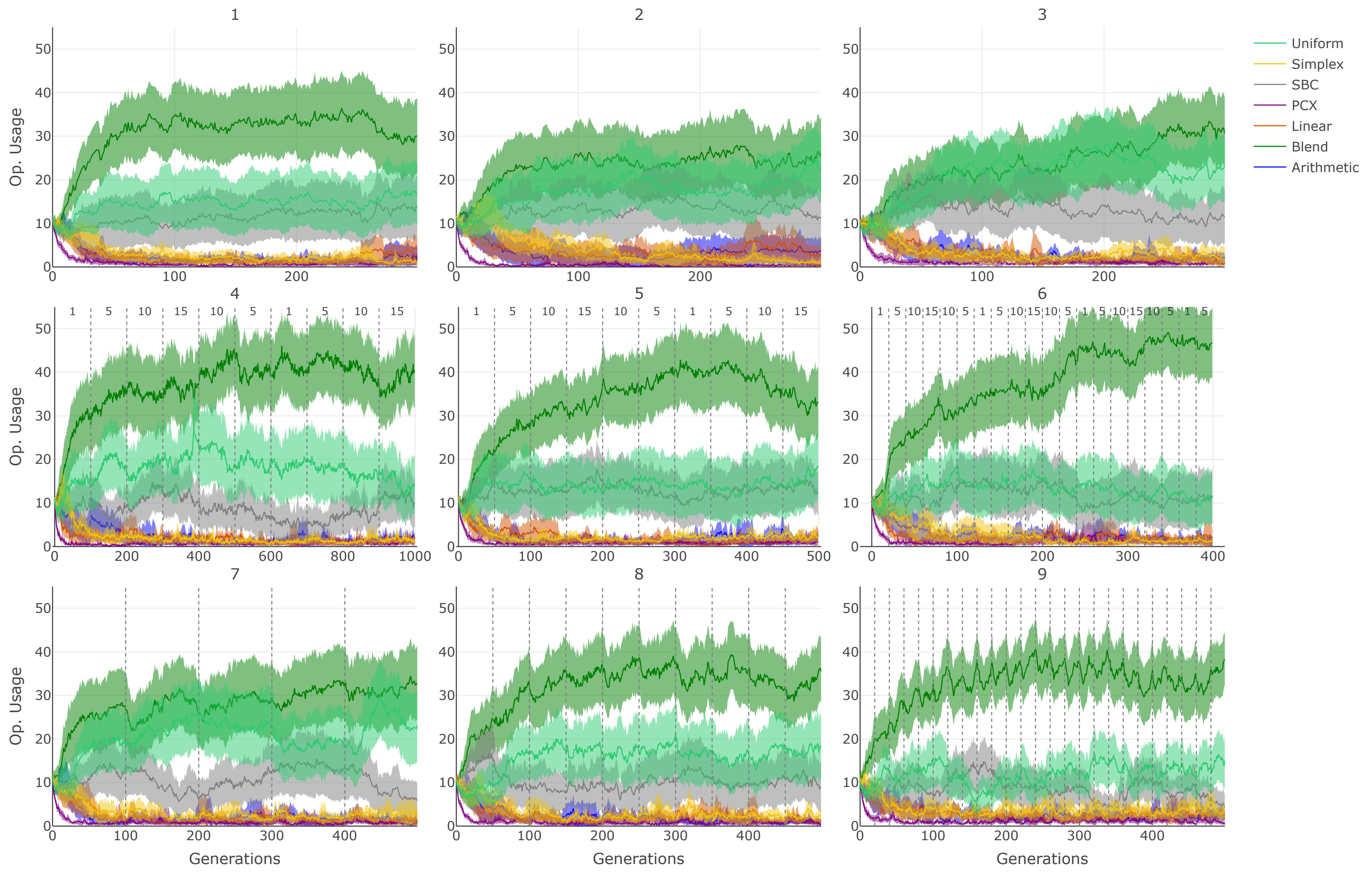

Figure 5.14: Mean crossover operator usage across all runs of the best SAGA configuration on the dynamic problems.

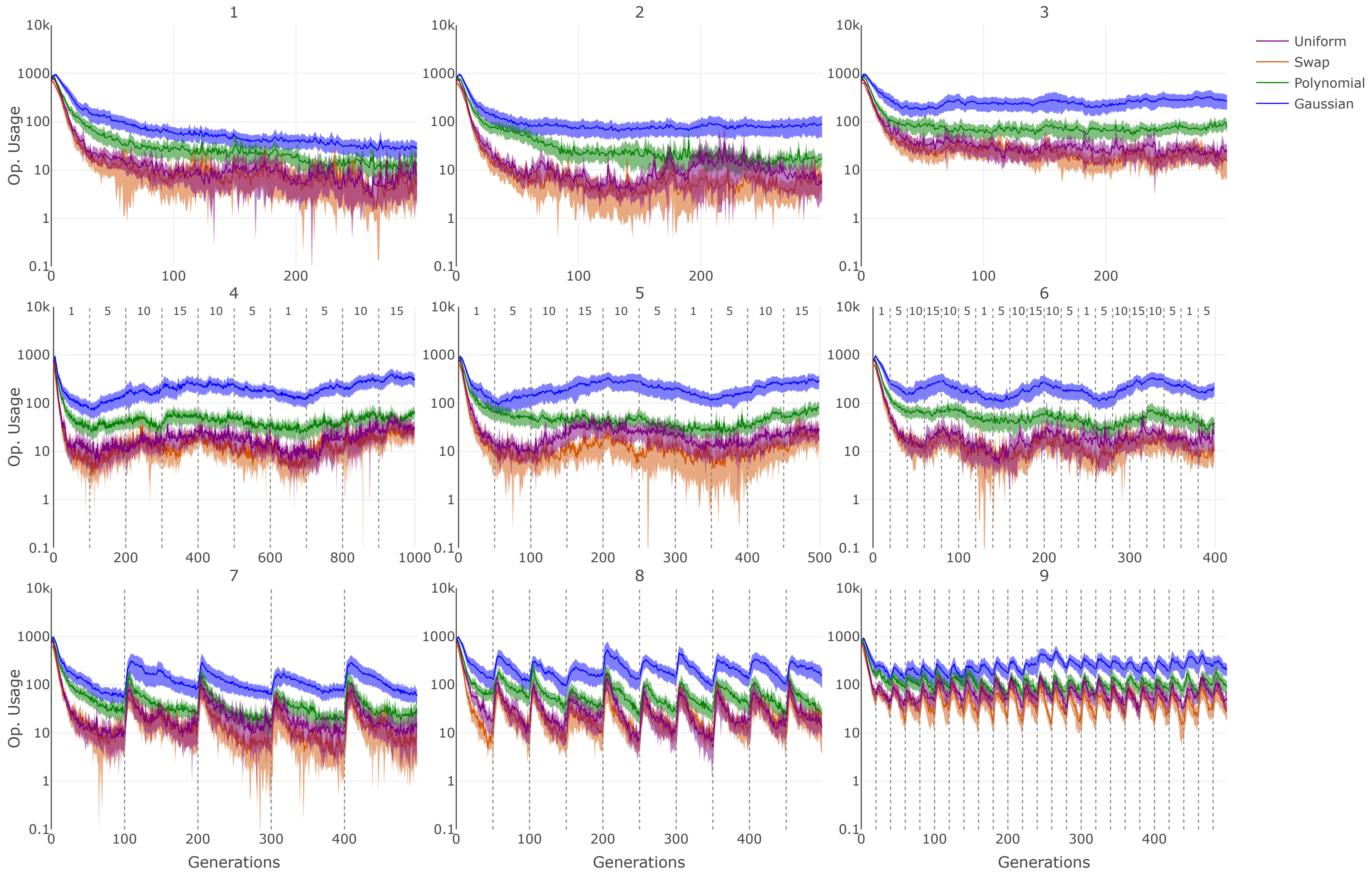

Figure 5.15: Mean mutation operator usage across all runs of the best SAGA configuration on the dynamic problems.

First, we look at Figure 5.14, the crossover operator usage figure. Problem 1 is a static problem and so adaptation of crossover operator usage is neither expected nor observed. Problems 2 and 3 are steady change problems, but the rate of change is constant throughout, so again, adaptation of crossover operator usage is neither expected nor observed. On these two problems, we do not see any clear distinctions between the two plots, meaning that the level of steady change in the problem does not have an effect on the crossover operator usage. Problems 4 through 6 are steady change problems where the number of dimensions in which the cones move vary over the generations. On these problems, there is no correlation between the crossover operator usages and the changes in the problems' dynamics. The ordering of the operators remains fairly constant throughout and does not vary according to the changes in the problems' dynamics. Problems 7 through 9 are the sudden change problem, where the fitness landscape is totally re-initialized at each change interval. On these problems, we see small temporary dips in the usage of Blend crossover on a majority of the change intervals. In the crossover rate plots 5.12, we saw that there is occasionally a drop in the crossover rate at the change intervals. This plot helps explain the inconsistency, while the usage for Blend drops fairly consistently at each interval, the other operators' usages do not change in a consistent manner. Blend crossover is the most used crossover operator, therefore, SAGA is slightly decreasing the usage of only the most-used operator at each change interval. We can deduce that the spike of mutation rate that also occurs at the change intervals is increasing exploration amongst the crossover operators which may, in turn, be causing this minor drop in the usage of the most-heavily used operator and inconsistency amongst the remaining operators. Therefore, we cannot call this small drop in Blend usage a significant adaptation to the problems' dynamics.

Next we look at Figure 5.15, the mutation operator usage figure. On problems 1 through 3, the mutation operator usages are stable and do not show any adaptation over the generations. The amount of mutation usages is correlated to the mutation rates observed in Figure 5.13. On problems

4 through 6, we see the mutation usages varying up and down as the dynamics of the problem changes, corresponding to the similarly varying mutation rate seen in Figure 5.13. The ordering of the mutation operator usages, however, remains constant throughout the fluctuations of the mutation rate. On problems 7 through 9, the sharp spikes of mutation operator usages are clearly visible, again, consistent with the spikes of mutation rate observed earlier. The ordering of the mutation operator usages remains constant throughout the spikes. These mutation operator usages show us that it is purely the mutation rate that SAGA is adapting, not the mutation operator usages. The mutation operator that is used most often, Gaussian, is always used most often across all problems and across all changes to problem dynamics.

From these two figures combined, we can see that SAGA is not adapting either crossover or mutation operator usages for the problem dynamics. There is no clear or consistent correlation between the ordering of the operator usages and the dynamics of the problems. Which operators are used most often remains constant for both crossover and mutation across all nine problems. This means that SAGA is choosing its preferred operators based upon the general field of cones problem, rather than basing the preferred operators on the specific dynamics of each particular problem instance.

5.5 Conclusions for SAGA on Dynamic Problems

Overall, SAGA has shown itself to be very proficient on dynamic problems. The problems cover a variety of dynamic behaviors, so as to test the robustness of SAGA on dynamic problems. Such robustness is of particular importance on dynamic problems, as the dynamics within the problem may not be fully known before-hand, or the dynamics may change over time. Of the methods tested, SAGA performs best or near best on 8 of the 9 problems as measured by CMF and BIP. Thus, we achieve our goal of improving GA usability without sacrificing fitness result quality. We did not design SAGA specifically for dynamic problems, so the fact that it is better or competitive to other methods that are designed specifically for dynamic problems is very encouraging.

Our results indicate that SAGA is adapting to the dynamics of the problems primarily through the mutation rate parameter. While the other self-adaptive parameters show no correlation or inconsistent correlation to the different problem dynamics, mutation rate shows clear and distinct adaptations to match the different problem dynamics on all problems. On the steady moving problems, 2 through 6, the magnitude of mutation rate on any given generation is correleated to the magnitude of the problem's dynamics. In the sudden change problems, 7 through 9, SAGA evolves a spike in mutation rate after each environmental change. Through these adaptation to mutation rate, SAGA is able to adjust itself to increase exploration and population diversity when appropriate.

HM is best performer of the alternative methods and also performs very well across most of the problems. HM, however, has a downside in the heavy reliance upon the two new parameters that must be accurately tuned. If these parameters are not tuned optimally then hypermutation can be triggered at detrimental moments. Looking at Table 5.4, which lists the best hypermutation parameter values of each problem, we see that there is a large variety in these parameters across the problems. Furthermore, these static parameters cannot change over the course of a run to adapt to changing problem dynamics, as SAGA is capable of doing on problems 4 through 6. Due to the potential unknown dynamics of a dynamic problem, this reliance on proper static settings can be quite detrimental to the practical application of HM.

It also should be noted that SAGA is not mutually exclusive with most of the alternative methods tested. SAGA can be combined with RI, Clearing, or HM such that a GA that combines the strengths of both methods can be achieved. Combining SAGA with HM is the easiest option, and based on the results, probably the most fruitful. Combining SAGA with HM may seem redundant at first, since both SAGA and HM use a variable mutation rate to handle the dynamics of a problem, but hypermutation can increase the speed at which the GA adjusts to a sudden environment change, which is where SAGA failed on problem 9. At the same time, self-adaption can remove the need to manually tune the HM parameters and make the HM parameters adaptive so that they can adjust to variable problem dynamics during a run. Combining SAGA with RI or Clearing is also possible, though the parameters of RI and Clearing are global parameters so more thought would be required on how to apply self-adaptation to these parameters. Combining SAGA with ACROMUSE would be very complex. ACROMUSE is already an adaptive GA designed from top-to-bottom for dynamic problems with custom genetic operations; modifying it to to be self-adaptive, while not impossible, would require a significant amount of design work.

In the end, we see that SAGA can successfully be applied to dynamic problems. SAGA is competitive with or outperforms other methods designed specifically for dynamic problems. And SAGA achieves this performance whilst staying true to our original goal of improving GA usability through self-adaptation.

CHAPTER 6: REAL-WORLD MEDICARE DATA EXPERIMENT

Being that our objective is to improve SAGA usability in order to promote a GA's applicability in real-world problems by users who are not within the field of EA, we need a real-world problem to solidify the relevance of this work. The problem we use is the prediction of whether the annual standardized Medicare payments of physical therapists (PT) will be over or under the national industry median. The standard method for this kind of problem within the field of healthcare informatics is logistic regression (LR) and will be the benchmark with which we compare SAGA. We continue to compare SAGA to CGA as well. We provide the motivation behind this particular real-world problem, explain the problem and how it is implemented, show the prediction accuracy results, and finally analyse said results. This chapter is focused specifically on this singular problem and is more focused on results than the other chapters which have a heavy focus on SAGA's self-adaptive behavior and usability.

6.1 Motivation

The problem we address is a predictive classification problem in which we use PT practice parameters to predict if the annual Medicare payment to a PT will be above or below the industry median. Medicare is the primary source of health insurance for individuals aged 65 or older in the United States. Physical therapy services, which can improve and maintain mobility, strength, and general health, are covered by the Medicare fee-for-service (FFS) program American Physical Therapy Association 2014. Due to the aging baby boomer population and to recent declines in birth rate, the 65+ segment of the US population is expected to increase from 15% in 2014 to a projected 24% in 2060 Mather, Jacobsen, and Pollard 2015. As this segment of the US population increases, demand for physical therapy services is also expected to rise. PTs receive payments from

Medicare for the clients they serve who are on Medicare. A better understanding of the factors associated with receiving above or below median Medicare payments can inform PTs and policy makers about practice variability as well as provide potential strategies and policies for improving access to physical therapy services for Medicare FFS beneficiaries.

EAs have been successfully applied to predictive classification problems for healthcare applications Fidelis, Lopes, and Freitas 2000; Wu, Liu, and Norat 2019 as well as other classification applications such as multi-criteria inventory classification Guvenir and Erel 1998 and prediction of a country's natural gas usage Kovačič and Dolenc 2016. In addition to these real-world examples, previous work have shown effective results when testing EA classification on various benchmarking data sets Dehuri et al. 2008; Fernández et al. 2010. GAs in particular, are often used in conjunction with other ML algorithms to identify the relevant features of a classification problem. Once the GA has selected the relevant features of a problem, other algorithms such as support vector machines Min, J. Lee, and Han 2006, random forest classifiers Paul et al. 2017, neural networks Jefferson et al. 2000; Shrivastava et al. 2017, linear discriminant analysis Höglund 2017, or LR Vandewater et al. 2015; Zhang et al. 2018 perform the classification using the features selected by the GA.

6.2 Experimental Setup

For this experiment, we must first describe the problem at hand and the algorithms that will run against it. We give a brief description of LR since it used as our performance benchmark. Finally, the specifics of how this problem is encoded within a GA is described.

6.2.1 Problem

The data that we use to make this classification comes from the 2014 Medicare Provider Utilization and Payment Data: Physician and Other Supplier Public Use File (PUF) and the 2015-2016 Area Health Resources File (AHRF). These data can be split into two groups; provider level variables and county level variables. The provider level variables come from the PUF and give data for 40,662 specific Medicare service providers from across the country. The county level variables come from the AHRF and give data on the counties in which the Medicare service providers reside. Each data point consists of 25 independent variables representing provider and county practice parameters and a corresponding dependent variable representing whether the PT received an above median Medicare payment. Table 6.1 lists the independent variables and their scopes. Figure 6.1 shows a histogram of the distribution of all 40,662 dependent variables of standardized Medicare payments in our data set, along with the overall median value.

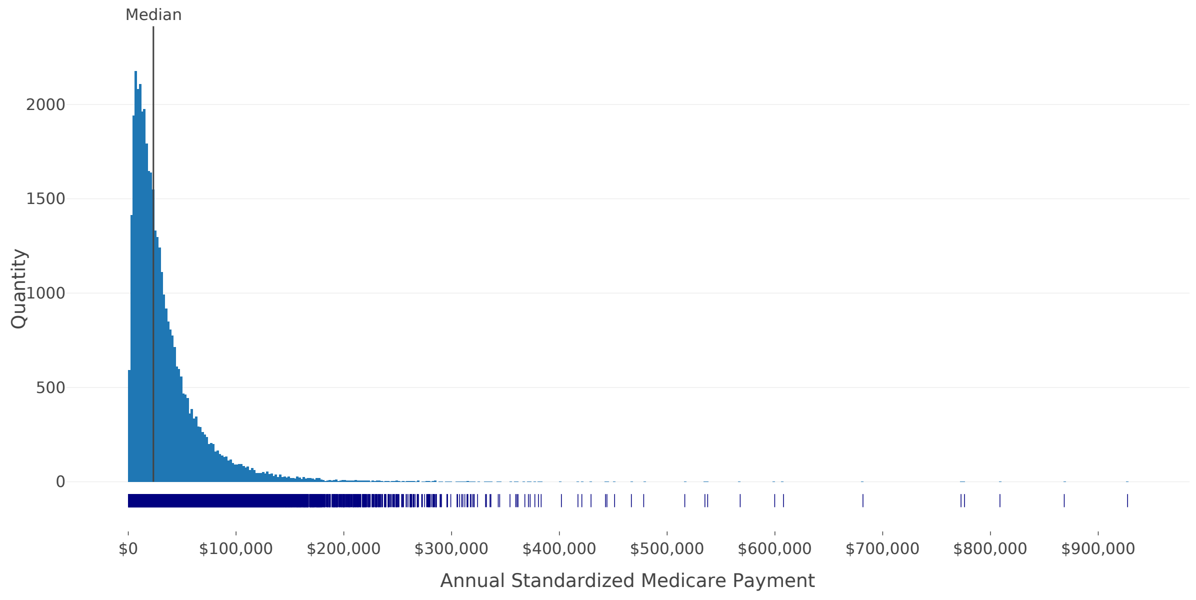

Figure 6.1: Histogram of the distribution of the annual standardized Medicare payments. The Median is $23,296.85. Outliers exist on the x-axis up to $900,000.

We evaluate the performance of SAGA on the problem of classifying whether or not the Medicare standardized payment for physical therapists is above the national median and compare its performance to that of LR as a benchmark. We also compare SAGA's performance to that of a CGA to

Table 6.1: Independent variables of the Medicare problem.

ID	Independent Variable	Unit of Observation
1	Female Gender	Provider
2	Doctor of Physical Therapy degree	Provider
3	Number of HCPCS/CPT codes billed	Provider
4	Number of beneficiaries	Provider
5	Charge to Medicare allowed amount ratio	Provider
6	Average Medicare standardized payment per beneficiary	Provider
7	Proportion of physical agents	Provider
8	Percent of therapeutic procedures	Provider
9	Number of new patients	Provider
10	Average age of beneficiaries	Provider
11	Average Hierarchical Condition Category (HCC) risk score	Provider
12	Small Metro Practice location	Provider
13	Mid-sized Metro Practice location	Provider
14	Non-metro Metro Practice location	Provider
15	Standardized Risk-Adjusted Per Capita Medicare Costs	County
16	Primary care physicians per 10,000 population	County
17	PTs per 10,000 population (2009)	County
18	Percent of beneficiaries eligible for Medicaid	County
19	Average age of beneficiaries	County
20	Percent female beneficiaries	County
21	Average HCC risk score	County
22	Beneficiaries percent of population	County
23	Median household income	County
24	Percent of persons 65 or older in deep poverty	County
25	PTs per 10,000 beneficiaries	County

verify that SAGA can perform at least as well as a CGA. Our evaluation metric is the percentage of correct classifications. We split the input data into a training set and a test set, where the algorithm is trained on only the training set but is evaluated against both. We run the problem on five data sets, each consisting of a different ratio of training and test data; these training:test ratios are: 50:50, 25:75, 10:90, 5:95, and 1:99. The training set sizes are no larger than 50% because a previous study Wu, Liu, and Norat 2019 showed no significant different in results for this same problem while using training sets ranging from 50% to 85%.

We experiment with using both the raw input data and standardized input data. The ranges of the independent variables vary greatly; standardization is used to equalize these ranges such that the larger range variables do not have an artificially large impact on the result. Although we do give SAGA and CGA a mechanism for adjusting the magnitude of the coefficients, we do not know how effectively that mechanism will be used by SAGA and CGA. We therefore test five methods: SAGA on raw data (RD-SAGA), SAGA on standardized data (SD-SAGA), CGA on raw data (RD-CGA), CGA on standardized data (SD-CGA), and LR.

The standardized independent variables are calculated as follows:

$$v'_{i,j} = \frac{v_{i,j} - \vec{v}_i}{\sigma_i} \tag{6.1}$$

where $v_{i,j}$ is the i^{th} independent variable from data point j,

$$\vec{v}_i = \frac{1}{40662} \sum_{r=1}^{40662} v_{r,i} \tag{6.2}$$

is the average of the i^{th} independent variable across all 40,662 data points, and σ_i is the standard deviation of $\vec{v}_i$.

6.2.2 Logistic Regression

LR is a commonly used benchmark for predictive classification problems of this kind in the field of health informatics De Vasconcelos et al. 2001; Min, J. Lee, and Han 2006; Serban, Kupraszewicz, and Hu 2011; Chaurasia and Pal 2014; Thornblade, Flum, and Flaxman 2018. LR makes a prediction on whether a data point j belongs to a class x based on the logistic function:

$$p_{x,j} = \frac{1}{1 + e^{-(c_0 + \sum c_i v_{i,j})}} \tag{6.3}$$

where $p_{x,j}$ is the probability that data point j belongs to class x, c_0 is the intercept, c_i is the coefficient for the independent variable i, and $v_{i,j}$ is the independent variable i of data point j. For our implementation, we use the LogisticRegression class from the scikit-learn library in Python.

6.2.3 Genetic Algorithm Implementation

Table 6.2 lists our chosen parameters for the CGA. Unlike the previous chapter, the tuning for the CGA here is not an exhaustive combinatorial search of over a thousand parameter values. Because, this is a real-world problem, we approach it as such. In the real-world there usually is not sufficient time nor resources for exhaustive tuning of thousands of configurations on multiple problem sets; which further illustrates the difficulties of parameter tuning that is at the heart of our research. So instead, we find a set of well performing parameters through empirical experimentation.

Table 6.2: CGA parameters for Medicare problem.

Parameter	Value
Stopping Condition	200 generations
Population size	100
Crossover rate	0.9
Crossover operator	two-point
Mutation rate	0.2
Mutation operator	uniform random
Parent Selection method	Tournament, size 10

6.2.3.1 *Fitness Function*

In our problem, the fitness of an individual is calculated as follows: first, the GA calculates a linear summation, $prediction_j$, for each data point j to be classified:

$$prediction_j = c_0 + \sum_{i=1}^{25} c'_i v_{i,j} \tag{6.4}$$

where $v_{i,j}$ is the i^{th} independent variable, $i = 1,...,25$, of the j^{th} data point, c_0 is the intercept, and c'_i is the i^{th} modified coefficient as calculated by:

$$c'_i = \begin{cases} c_i^1 & \text{if } -1.0 \leq c_{i+25} < -0.5 \\ c_i^2 & \text{if } -0.5 \leq c_{i+25} < 0.0 \\ c_i^3 & \text{if } 0.0 \leq c_{i+25} < 0.5 \\ c_i^4 & \text{if } 0.5 \leq c_{i+25} < 1.0 \end{cases} \tag{6.5}$$

The reason for raising the coefficients c_1 through c_{25} by a power specified by the coefficients c_{26} through c_{50}, respectively, is that the ranges of the independent variables in our data set vary greatly, with some variables having ranges that are magnitudes larger than others. This modification gives the GAs a means to account for the wide variability of ranges. Note that in addition, we also test standardizing the data, as will be described in the experimental setup.

Next, we apply the linear summation of Equation 6.4 to each of the data points in the training set and determine a classification based on:

$$above\_median_j = \begin{cases} 1 & \text{if } prediction_j > 0.0 \\ 0 & \text{if } prediction_j \leq 0.0 \end{cases} \tag{6.6}$$

Both SAGA and CGA use this fitness function.

6.3 Results

Our results consist of two measurements. The first and primary measurement is accuracy; the number of correct classifications. The second measurement is a breakdown of the accuracy into the number of true positive, true negative, false positive, and false negative classifications.

Tables 6.3 and 6.4 show the accuracy results of our study on the training and test data sets, respectively. The tables show the accuracy of each of the algorithms for each data set. We execute 50 runs of SAGA and CGA, due to their pseudo-random nature; therefore, the tables show the best single run and the average of all 50 runs, along with the 95% confidence interval. In addition, the test data table, Table 6.4, highlights the best accuracy in bold.

Table 6.3: Training data accuracy results for the Medicare problem.

Data Set	RD-SAGA		SD-SAGA		RD-CGA		SD-CGA		LR
	Best	Mean	Best	Mean	Best	Mean	Best	Mean	
Train 50%	93.10	77.82 ± 3.79	94.17	93.45 ± 0.16	93.26	89.37 ± 0.96	91.37	90.46 ± 0.16	93.11
Train 25%	93.65	80.89 ± 3.42	94.14	93.58 ± 0.15	93.25	89.87 ± 0.77	91.53	90.27 ± 0.17	92.95
Train 10%	93.33	83.52 ± 2.88	94.45	93.00 ± 0.55	92.97	90.00 ± 0.66	91.69	90.37 ± 0.13	92.94
Train 5%	94.10	81.38 ± 3.26	95.47	93.86 ± 0.58	94.44	90.14 ± 0.91	93.51	91.14 ± 0.17	93.85
Train 1%	94.34	77.63 ± 3.31	96.06	92.16 ± 0.75	93.10	88.53 ± 0.87	91.87	90.12 ± 0.21	92.86

Table 6.4: Test data accuracy results for the Medicare problem.

Data Set	RD-SAGA		SD-SAGA		RD-CGA		SD-CGA		LR
	Best	Mean	Best	Mean	Best	Mean	Best	Mean	
Test 50%	92.92	77.80 ± 13.61	**93.84**	93.15 ± 0.54	93.12	89.35 ± 0.96	91.32	90.40 ± 0.17	92.90
Test 75%	93.20	80.92 ± 12.47	**93.86**	93.25 ± 0.49	93.15	90.08 ± 0.74	91.44	90.40 ± 0.17	93.02
Test 90%	93.23	83.06 ± 10.56	**93.80**	92.40 ± 1.92	93.47	89.82 ± 0.69	91.85	90.21 ± 0.17	92.93
Test 95%	92.76	80.09 ± 11.68	**93.40**	91.91 ± 1.89	93.17	89.16 ± 0.87	91.61	90.07 ± 0.20	93.01
Test 99%	91.74	75.60 ± 12.14	91.95	87.92 ± 3.02	**92.10**	86.23 ± 0.97	91.45	87.90 ± 0.46	90.98

SD-SAGA shows the best overall accuracy across the data sets, outperforming LR and the other SAGA and CGA methods on both test and training sets. The only instance in which SD-SAGA is not the best performer is the 99% test data set, in which RD-CGA slightly outperforms it by 0.15%. The RD-CGA appears to be the second best performer overall, followed the RD-SAGA, LR, and lastly SD-CGA.

Interestingly, the methods that use standardized data are both the best and worst performing of the SAGA and CGA methods. In the case of SAGA, standardization provides a clear advantage, with SD-SAGA outperforming RD-SAGA on every instance, while the opposite is true for the CGA methods. One consistent effect standardization has on both SAGA and CGA is that it lowers the variation from run to run, as shown by the smaller confidence intervals and smaller distance between average accuracy and best accuracy on both of the standardized data methods in Tables 6.3 and 6.4. Increased consistency across runs is neither necessarily a benefit nor a detriment in and of itself. Increased consistency can mean that the CGA/SAGA more consistently converges at the optimal (or near optimal) solution, or it can mean that it more consistently converges prematurely, on a sub-optimal solution.

Figure 6.2 shows the accuracy of the test sets for each algorithm. The x axis is the training set size and the y axis is test set accuracy. The figure shows that there is little degradation in the classification accuracy on the test data as the training set size decreases. Only when the training set is down to 1% of the input data, equal to around 400 data points, is there a clear and consistent drop in accuracy on test data across the algorithms when compared to all other training set sizes. On this particular problem, it appears that large training data sets are not required or that there are sufficient data points that a small percent is a large enough data set.

In Table 6.5, we record and display the number of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) classifications accrued, where a positive classification is when the Medicare standardized payment is above the median. In each column the best result is in bold; for TP and TN, this means the largest value, while for FP and FN it means the smallest value. These results are on the test data sets, and in the case of the SAGA and CGA methods, uses the best run as shown in Table 6.4.

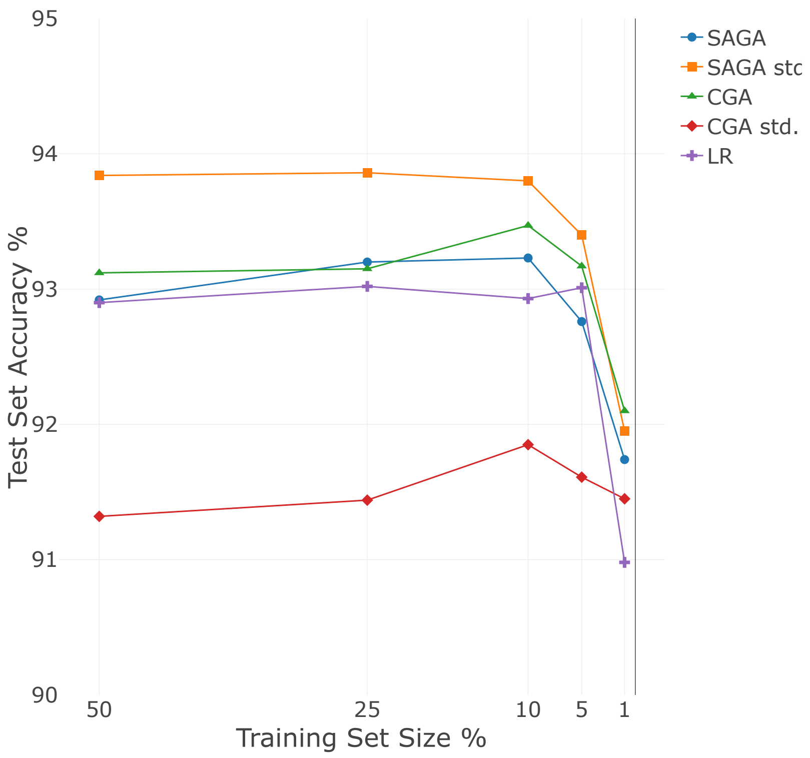

Figure 6.2: Test set accuracy of each algorithm.

Table 6.5 has a few noteworthy observations. The first is that despite having the best accuracy in all but the 99% test set, SD-SAGA rarely scores the best in any one category of TP, TN, FP, or FN. Rather, SD-SAGA begets the best overall accuracy by getting good enough results on each of the four categories. Another observation is that the SAGA and CGA methods tend to favor positive classifications, having a larger ratio of TP to TN and FP to FN. The only exception is the SD-CGA which produces a more even ratio of positive and negative classifications on all but the 99% test set. LR, on the other hand, is much more balanced in regards to the amount of positive and neg-

Table 6.5: Statistical results of test data sets. Best results are in bold.

50:50 Test Set				
	TP	TN	FP	FN
RD-SAGA	9698	9231	983	419
SD-SAGA	**9829**	9211	906	**348**
RD-CGA	9647	9285	832	567
SD-CGA	9352	9215	902	862
LR	9424	**9470**	**647**	790

25:75 Test Set				
	TP	TN	FP	FN
RD-SAGA	**14625**	13798	1424	**649**
SD-SAGA	14559	13982	1240	715
RD-CGA	14540	13867	1355	734
SD-CGA	13952	13935	1287	1322
LR	14120	**14232**	**990**	1154

10:90 Test Set				
	TP	TN	FP	FN
RD-SAGA	17672	16447	1845	**631**
SD-SAGA	17643	16604	1688	660
RD-CGA	**17646**	16558	1734	657
SD-CGA	16620	16991	1301	1683
LR	16870	**17151**	**1141**	1433

5:95 Test Set				
	TP	TN	FP	FN
RD-SAGA	18707	17005	2293	623
SD-SAGA	18339	17568	1730	991
RD-CGA	**18818**	17172	2126	**512**
SD-CGA	17753	17635	1663	1577
LR	17990	**17932**	**1366**	1340

1:99 Test Set				
	TP	TN	FP	FN
RD-SAGA	19046	17882	2245	1082
SD-SAGA	18976	17802	2325	1152
RD-CGA	18724	18351	1776	1404
SD-CGA	**19196**	17616	2511	**932**
LR	18208	**18406**	**1721**	1920

ative classifications; in fact, LR consistently has the lowest number of false positives in all test sets, resulting in to LR having the best record for correctly classifying negative classifications. The SAGA and CGA methods, however, are better at avoiding incorrect negative classifications, leading to higher correct positive classifications. Because the dividing point between our two classes is the median of the medicare standardized payments, and a median is by definition in the middle, our data is evenly split between the two classes. This balance of positive and negative classes means that LR is actually correct in making roughly equal positive and negative classifications, however, in total, LR makes more incorrect classifications; hence, both SAGA methods, despite disproportionately favoring positive classifications, return higher overall accuracies.

6.4 Analysis of Results

We take a closer look at the results of SAGA in order to better understand why it evolves as it does. While our focus is on SAGA, we similarly examine the CGA to see if the self-adaptive mechanisms cause any noticeable divergences in evolutionary behavior when compared to a CGA. We begin by examining the evolved coefficient values to see if they provide any insight on how the independent variables are used to make a classification. To help with this analysis, we examine how the characteristics of the independent variables from the input data relate to the coefficients. Our observations suggest that SAGA and the CGA may be performing feature selection in conjunction with finding appropriate coefficient values, so we further explore the ability of SAGA and CGA to effectively enact feature selection during evolution.

6.4.1 Coefficient Analysis

Here we take a deeper look into the evolved coefficients. Figures 6.3 and 6.4 are both organized in the same manner. Figure 6.3 shows the evolved coefficients from the best individual of every run for both RD-SAGA and RD-CGA. Figure 6.4 shows the same for SD-SAGA and SD-SAGA. The y-axis displays the value of the coefficient and the x-axis shows the ID number of the independent variable to which each coefficient corresponds. There are 50 lines in each coefficient group along the x-axis corresponding to each of the 50 runs. The x-axis starts with a 0^{th} value; this is the intercept and is marked as 0 so that the other coefficient numbers align properly with the IDs in Table 6.1.

First, we see that there is a larger difference between standardized data and raw data than there is between SAGA and CGA. The evolved coefficients are very similar for SD-SAGA and SD-CGA and the evolved coefficients are also very similar for the RD-SAGA and RD-CGA. This consistent similarity suggests that the self-adaptive mechanisms that are added to SAGA to dynamically control genetic operator parameters do not significantly change the evolution of candidate solutions as compared to a CGA.

The two raw data methods, RD-SAGA and RD-CGA, (Figure 6.3) both show very inconsistent and chaotic results. For the majority of the independent variables, the evolved coefficients are all over the place. Not only are the magnitudes of the coefficients very different from run to run, but the signs are as well; it is not uncommon to see runs with coefficients near both extremes of 1 and -1 for the same independent variable. Despite this noise, there are a few independent variables that are evolved to consistent values in almost every run; four, six, nine, fifteen, and twenty-three, which according to Table 6.1 correspond to “Number of beneficiaries”, “Average Medicare standardized payment per beneficiary”, “HCC risk score”, “Primary care physicians per 10,000 population (county)”, and “Median household income (county)”. Four, six, and nine are

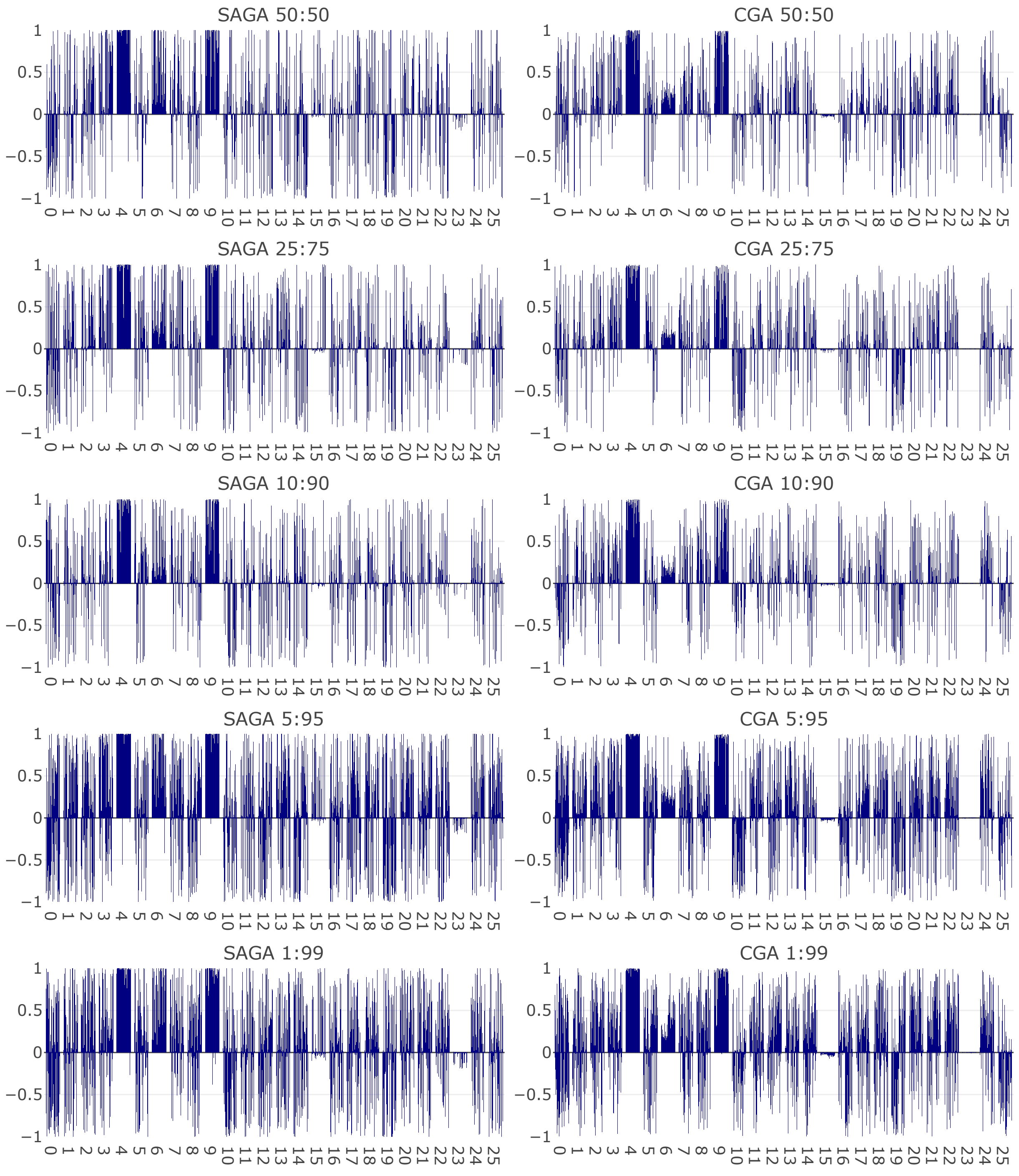

Figure 6.3: Plots for each training:test ratio of the best coefficients on raw data for SD-SAGA (left column) and SD-CGA (right column) on all 50 runs.

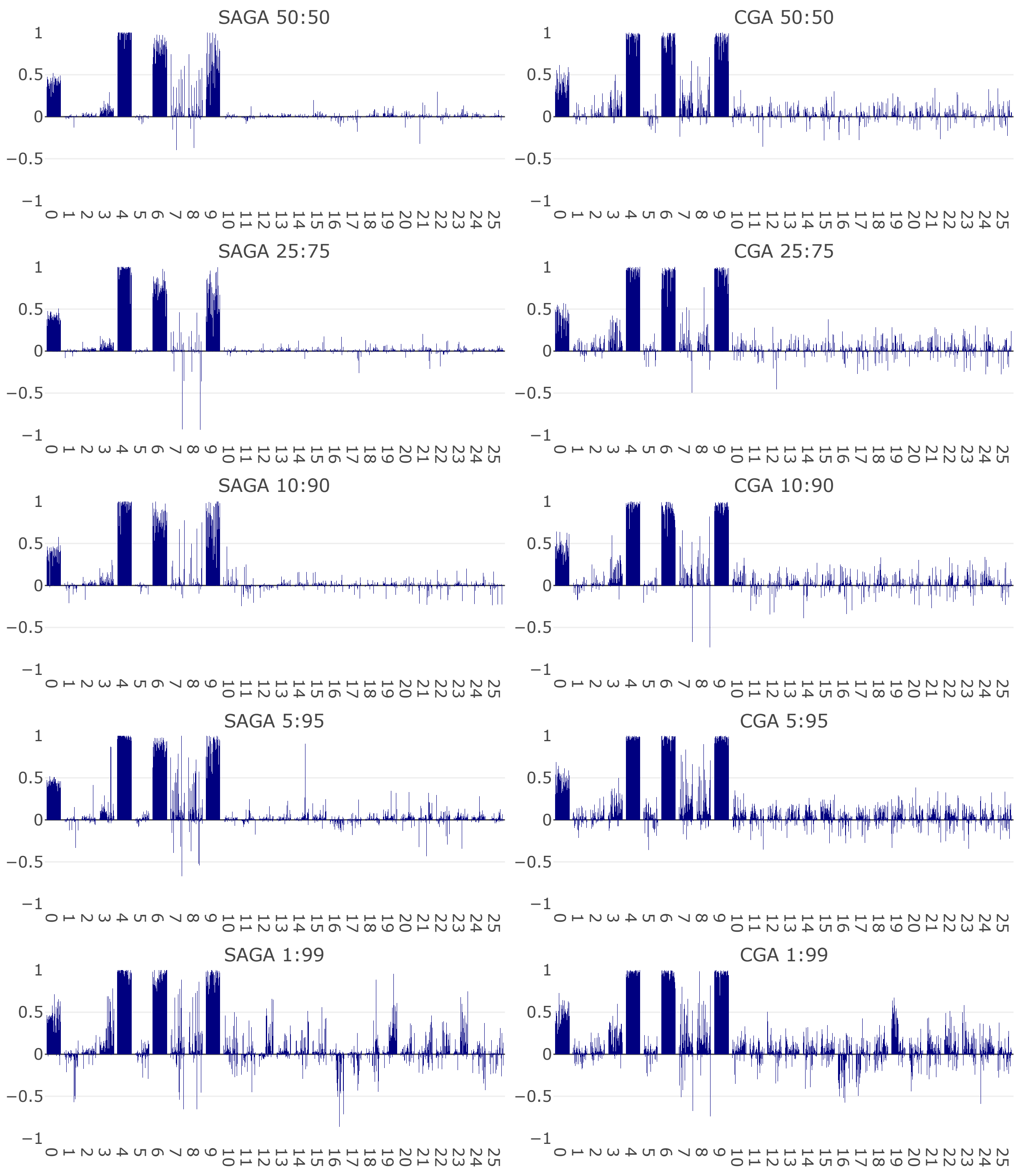

Figure 6.4: Plots for each training:test ratio of the best coefficients on standardized data for SD-SAGA (left column) and SD-CGA (right column) on all 50 runs.

consistently positive, usually with a medium to large magnitude while fifteen and twenty-three are consistently near zero. Variables four, six, and nine appear to be the most important in determining the outcome while fifteen and twenty-three appear to be disruptive and are thus evolved to zero to remove their influence entirely; this is further reinforced when we analyze both the coefficients of SD-SAGA and SD-CGA and the input data.

The plots for the standardized data methods, SD-SAGA and SD-CGA, in Figure 6.4 show more consistent coefficients as compared to RD-SAGA and RD-CGA. Similar to RD-SAGA and RD-CGA, we see that the coefficients at indices four, six, and nine all have highly positive magnitudes, signalling their significance. These three coefficients are even more consistent here than in the raw data methods, and nearly always evolve to magnitudes near the maximum of 1. The variables that were consistently evolved to zero in the raw data methods, fifteen and twenty-three, are again almost always very near zero; however, they no longer stand out, as nearly all other coefficients are similarly evolving to zero.

Furthermore, the standardized data methods drastically reduce the magnitude of the majority of the coefficients. These reduced magnitudes are especially prevalent on SD-SAGA, where most coefficients evolve to near zero. By evolving coefficients near zero, SD-SAGA is effectively removing the associated independent variable from the calculation, since any number multiplied by a sufficiently small number will be nearly zero. Thus, SD-SAGA and SD-CGA are effectively enacting simultaneous feature selection and classification through its evolutionary mechanisms. These small magnitudes are less prevalent on the smaller training set sizes, where apparently there is not enough information to fine tune these coefficients as effectively; the lower accuracy on the 99% test set is most likely due to this. Further exploration of this idea of GAs for simultaneous feature selection and classification is worth investigating in the future.

6.4.2 Input Analysis

We need more background information to understand why the coefficients evolve as they do, so we take a look at the input data and its relation to the evolved coefficients. Figure 6.5 shows histograms of each of the independent variables. The x-axes show the value of the independent variable and the y-axes show the frequency of that value. These plots are generated from the full input data set of 40,662 data points.

First, we can see why standardization can have a stabilizing effect on the evolution. There is a very large difference in the ranges of these variables; some are binary, 0 or 1, some are continuous with very small ranges, from 0 to less than 1, and others still are continuous from 0 up to tens or hundreds of thousands.

All of the independent variables have non-negative ranges, which further explain some of the differences between the raw data and standardized data methods. Thus, when running on the raw data, to push towards a negative classification, there must be some negative coefficients or a negative intercept; this can be observed in Figure 6.3. The process of standardization transforms the data such that there are both positive and negative inputs. Thus, negative coefficients are no longer required to obtain negative classifications, and in fact the plots of Figure 6.4 show nearly exclusively positive coefficients and intercepts.

We identify five independent variables that have ranges larger than [0,100]: four, six, nine, fifteen, and twenty-three. Without standardization, these five variables will have the largest impact on the outcome because of their larger ranges. These five variables match perfectly with the five variables identified earlier as the only consistent variables on the raw data methods. The large magnitudes of these five variables explains the inconsistent and chaotic nature of the coefficients on the remaining variables on RD-SAGA and RD-CGA. Relative to these variables, the other variables are too small to have a significant effect on the outcome and so their coefficient does not matter.

Based on Figures 6.3 and 6.4, we identify variables four, six, and nine as the influential variables for making a correct classification since they are important on both the standardized data and raw data methods. These three variables are also among the five identified as having larger ranges, which leads to the conclusion that the raw data methods benefited from the fact that these influential variables had large ranges. If the opposite were true such that the influential variables had small ranges and the non-influential variables had large ranges, then RD-SAGA and RD-CGA may have had much more difficulty. In conclusion, despite the fact that RD-SAGA and RD-CGA perform well here, GA methods, self-adaptive or otherwise, should follow the norms of other classification ML algorithms in standardizing their input data.

6.4.3 Significant Variables Analysis

In order to further test the simultaneous feature selection and classification abilities of SAGA, we run the problem two more times; first using only the significant independent variables as identified earlier (4, 6, 9) and second using all but those three independent variables. With this extra experimentation, we want to see how correct SAGA is in its identification of the three significant variables (4, 6, 9); do the SAGA and CGA methods focus on these three because they are required for high accuracy, or can similar accuracy be reached without them? We also include LR again to continue its role as a comparison benchmark.

Tables 6.6 and 6.7 show the results of the experiment when using only the three significant variables (4, 6, 9) on training and test data, respectively. Similarly, Tables 6.8 and 6.9 show the results when using the all variables except the three significant variables on training and test data, respectively. Comparing the results of these tables, we can clearly see that every method scores significantly higher when using the significant variables, with accuracies that are 20% to 30% higher across the board. The insignificant independent variables alone are not sufficient to accurately classify this problem through either SAGA, CGA, or LR. These results validate SAGA's capabilities for simultaneous feature selection and classification on this data set, as the variables deemed significant by SAGA are indeed the only variables that are necessary to finding high accuracy results.

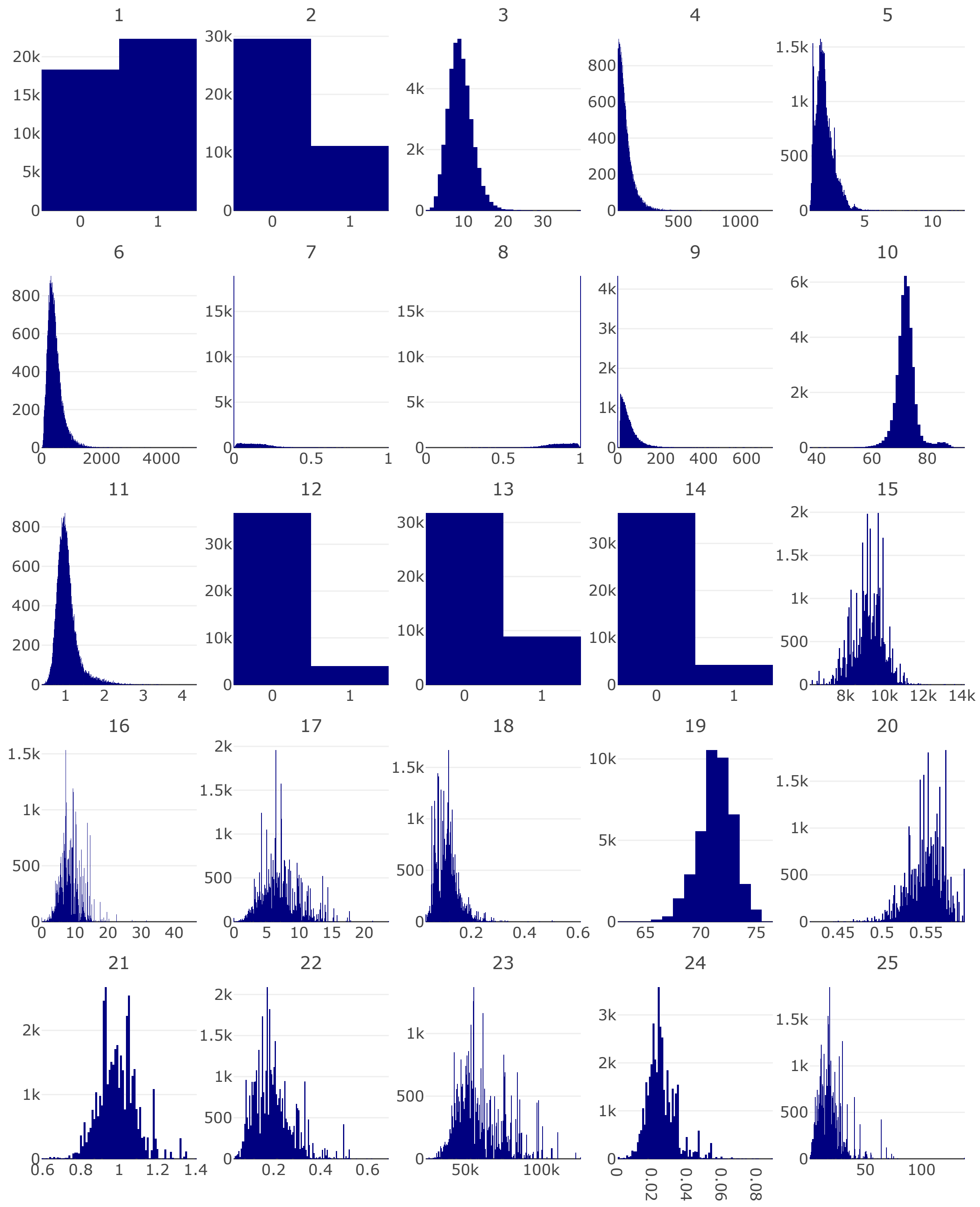
Figure 6.5: Histograms of each of the independent variables for the Medicare problem.

Table 6.6: Training data accuracy results for the Medicare problem using only significant variables.

Data Set	RD-SAGA		SD-SAGA		RD-CGA		SD-CGA		LR
	Best	Mean	Best	Mean	Best	Mean	Best	Mean	
Train 50%	94.09	83.30 ± 3.06	94.11	94.04 ± 0.03	94.00	91.94 ± 1.22	94.09	94.07 ± 4.0e-5	93.15
Train 25%	93.99	82.04 ± 2.98	94.03	93.98 ± 0.01	93.93	90.21 ± 1.99	94.03	93.99 ± 4.9e-5	93.02
Train 10%	94.05	83.23 ± 2.95	94.10	93.90 ± 0.04	93.80	90.52 ± 1.79	94.10	94.00 ± 0.03	92.75
Train 5%	94.88	76.22 ± 3.18	94.98	94.81 ± 0.04	94.84	91.98 ± 1.70	94.98	94.86 ± 0.02	94.20
Train 1%	95.07	78.85 ± 3.62	95.07	94.86 ± 0.08	94.83	91.70 ± 1.81	95.07	94.99 ± 0.04	91.40

Table 6.7: Test data accuracy results for the Medicare problem using only significant variables.

Data Set	RD-SAGA		SD-SAGA		RD-CGA		SD-CGA		LR
	Best	Mean	Best	Mean	Best	Mean	Best	Mean	
Test 50%	**93.84**	83.74 ± 2.94	93.82	93.69 ± 0.03	93.72	91.88 ± 1.17	93.80	93.70 ± 0.01	92.92
Test 75%	93.92	82.04 ± 2.99	93.92	93.82 ± 0.01	93.87	90.10 ± 1.98	**93.94**	93.82 ± 0.01	93.02
Test 90%	93.80	82.86 ± 2.98	93.88	93.67 ± 0.05	**93.89**	90.48 ± 1.83	93.86	93.75 ± 0.03	93.12
Test 95%	93.76	78.35 ± 3.18	**93.90**	93.70 ± 0.05	93.82	91.18 ± 1.70	**93.90**	93.73 ± 0.04	93.28
Test 99%	**93.85**	80.07 ± 3.14	93.49	92.19 ± 0.14	93.83	91.05 ± 1.62	92.64	92.26 ± 0.07	92.92

Table 6.8: Training data accuracy results for the Medicare problem using only insignificant variables.

Data Set	RD-SAGA		SD-SAGA		RD-CGA		SD-CGA		LR
	Best	Mean	Best	Mean	Best	Mean	Best	Mean	
Train 50%	67.38	56.09 ± 0.91	71.76	71.20 ± 0.08	65.09	55.83 ± 0.63	71.02	70.57 ± 0.04	69.84
Train 25%	66.97	55.36 ± 0.57	71.53	70.91 ± 0.11	65.17	55.78 ± 0.66	70.52	70.27 ± 0.04	69.07
Train 10%	61.29	55.49 ± 0.39	71.77	70.87 ± 0.14	66.01	57.97 ± 1.10	70.59	70.31 ± 0.04	68.85
Train 5%	62.03	55.03 ± 0.30	72.21	71.08 ± 0.20	64.39	55.96 ± 0.66	71.08	70.60 ± 0.05	67.60
Train 1%	62.81	56.50 ± 0.37	77.59	75.07 ± 0.43	66.50	57.49 ± 0.64	75.62	74.91 ± 0.09	69.78

Table 6.9: Test data accuracy results for the Medicare problem using only insignificant variables.

Data Set	RD-SAGA		SD-SAGA		RD-CGA		SD-CGA		LR
	Best	Mean	Best	Mean	Best	Mean	Best	Mean	
Test 50%	63.12	49.62 ± 1.25	**70.88**	70.23 ± 0.10	65.51	51.38 ± 0.80	70.58	70.02 ± 0.08	69.76
Test 75%	63.91	50.73 ± 1.40	**71.09**	70.42 ± 0.11	60.16	50.46 ± 1.15	70.67	70.28 ± 0.06	69.21
Test 90%	63.26	50.45 ± 0.65	**70.72**	69.98 ± 0.14	65.68	51.56 ± 1.27	70.61	69.99 ± 0.09	68.99
Test 95%	66.85	49.94 ± 1.11	**70.63**	69.26 ± 0.18	59.63	50.47 ± 0.97	70.25	69.49 ± 0.11	68.80
Test 99%	59.38	49.88 ± 1.04	**68.98**	66.92 ± 0.32	65.19	50.18 ± 1.47	68.63	67.21 ± 0.22	67.96

Using only significant variables has a large improvement on the performance of SD-CGA, no change in performance for SD-SAGA, and a slight improvement on the remaining three methods. SD-CGA gains 2.48% in performance from 91.32% in Table 6.4 to 93.80% in Table 6.7 on the 50% test data set, with similar improvements on the other data sets. The same improvement is not made on SD-SAGA, which was the best performance method when all variables are used. Using only significant variables results in a SD-SAGA performance of 93.82% on the 50% test data set, which is nearly identical to the 93.84% performance with all variables from Table 6.4. RD-SAGA, RD-CGA, and LR all gain 1% or less on most of the data sets. The 99% test data set is the one exception, where RD-SAGA, RD-CGA, and LR gain around 2% when using only significant variables. The results are also somewhat equalized across the methods, with multiple ties for best accuracy across the SAGA and CGA methods. These results show that SD-SAGA is the most capable at handling the insignificant variables; reinforced by the coefficient figures in Section 6.4.1, where SD-SAGA finds the most consistent coefficients for the insignificant variables. The inclusion of more variables increases the search space, increasing the difficulty of the problem; this increase in difficulty is not a factor for SD-SAGA, which is effective enough to find excellent results on both sets of variables.

Figure 6.6 shows the plots of the evolved coefficients on the 50:50 training:test set using only the significant variables. We see a very large difference between the evolved weights of the standardized data methods and raw data methods for both SAGA and CGA. On the raw data, the intercept is negative, coefficient 4 is very small in either direction, coefficient 6 is very small and negative, and coefficient 9 is positive with either a very small or very large magnitude. Closer inspection reveals that RD-SAGA finds two different coefficient configurations that are viable. In one, both coefficient 4 and 9 are slightly positive. In the other configuration, coefficient 4 is slightly negative, and coefficient 9 is highly positive. Both of these configurations are very different from the more consistent results of the standardized data methods, wherein every coefficient and the intercept is positive. RD-CGA appears to find only the former configuration in all runs. Because the CGA has identical constant parameters across all runs, it is much more likely for each run to follow a similar evolutionary path as compared to SAGA with its dynamic parameters that are initialized randomly and may evolve differently in each run. Therefore, the evolved individuals are more likely to be more consistent in CGA than in SAGA, leading to CGA being unable to discover both of the viable coefficient configurations. In terms of the accuracy of SAGA and CGA for this particular problem, this consistency is irrelevant, as both configurations find equivalent results. On other problems, the increased variation across the runs of SAGA may lead to improved results if CGA gets stuck in a local optima in every run.

Figure 6.7 shows the plots of the evolved coefficients on the 50:50 data set using only the insignificant variables, defined as every variable except four, six, and nine. The plots of the insignificant variables are much more varied across runs, even on the standardized data methods. A few mostly consistent trends are found, the most prominent being a highly positive coefficient for variable 3, "Number of HCPCS/CPT codes billed", but this is not as impactful on accuracy as the significant variables are. With only the insignificant variables, neither SAGA nor CGA methods are able to find a clear strategy for obtaining accurate results.

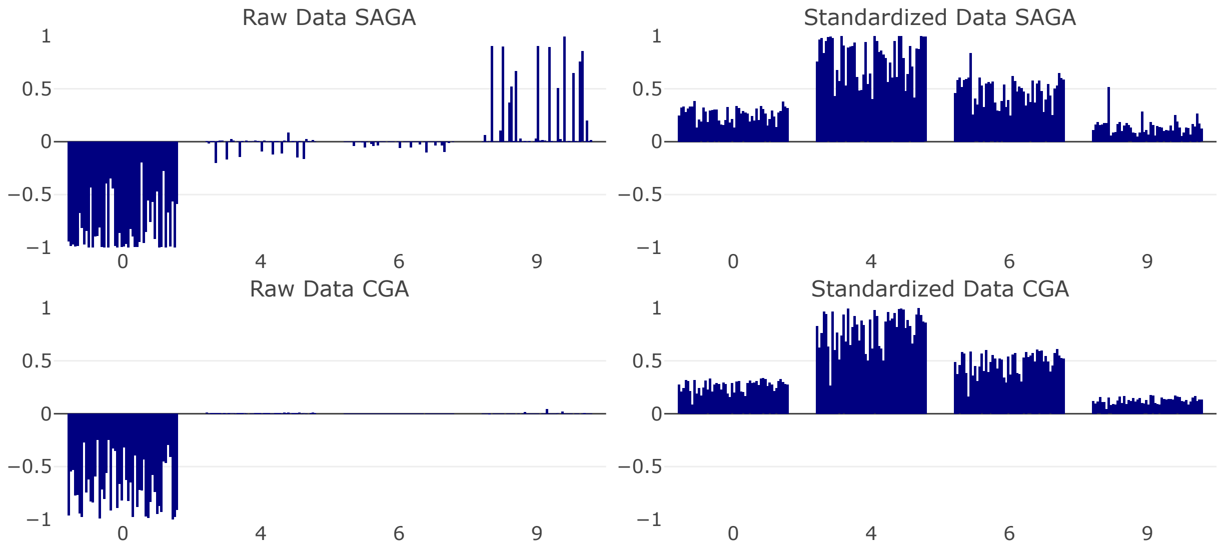

Figure 6.6: CGA and SAGA coefficients using only the significant variables on the 50:50 training:test set.

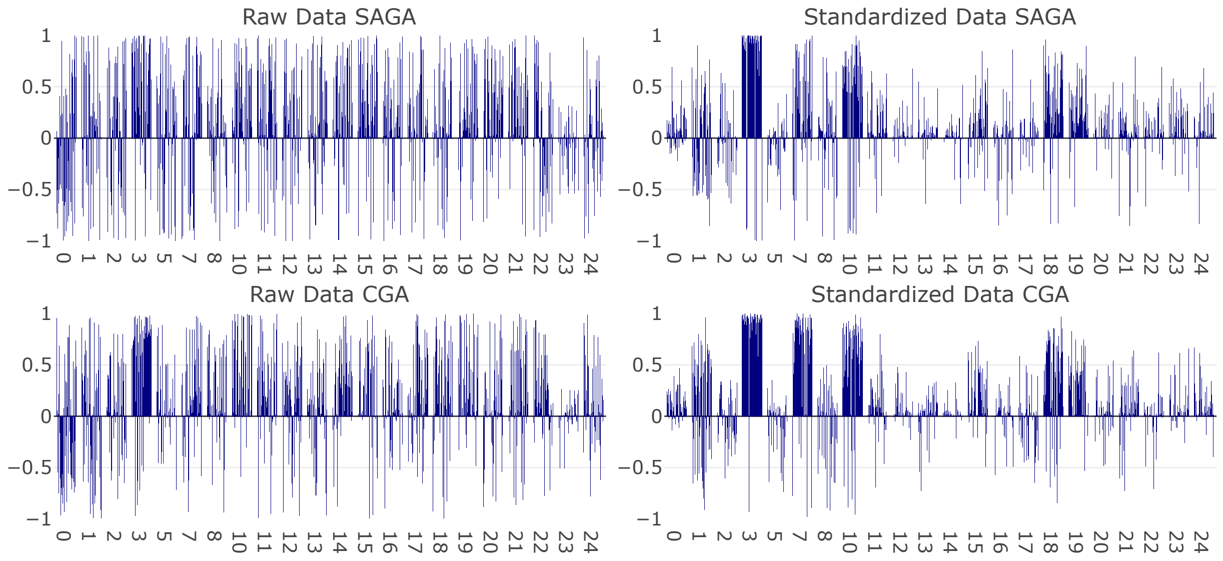

Figure 6.7: CGA and SAGA coefficients using only the insignificant variables on the 50:50 training:test set.

6.5 Conclusions for SAGA on Real-World Medicare Problem

The results show that SAGA's performance of the standardized input data yields the highest accuracy of all methods tested. All but one of the SAGA and CGA variants perform equivalent to or superior to LR. SAGA and CGA tended to give more positive classification than negative classifications, which is an incorrect ratio since the data set is even split between the two classes. LR on the other hand, found roughly equal number of positive and negative classifications. Despite the positive bias, SAGA and CGA still tended to return a higher accuracy, particularly for the self-adaptive SAGA on standardized data. Across all five of the methods, there is no appreciable degradation of accuracy as the size of the training set decreases until the training size reaches just 1% of the input data.

We test SAGA and CGA on both standardized data and raw data to see if standardization results in any noticeable differences. Results indicate that standardization produces a small improvement in accuracy for SAGA and a small decrease in accuracy for CGA. For this particular problem, the nature of the ranges of the independent variables is beneficial to running the algorithms with raw data, since the influential variables are also amongst the variables who have the largest ranges. Nonetheless, the best results are found with standardized data. Our results suggest that standardization is indeed recommended for SAGAs and GAs on these kind of classification problems, much like in other ML methods.

Analysis of the behaviors of SAGA and CGA indicate that they are effectively enacting simultaneous feature selection and classification. This result means that, in addition to learning how to make accurate classifications from training data, our SAGA and CGA approach can also provide information on which features are most relevant for making an accurate classification. There is potential for future work for a GA that is designed around this idea of simultaneous feature selection and classification.

In conclusion, we show that SAGA is a competitive method for this real-world predictive classification problem within the field of Healthcare informatics. Our results suggest that SAGA, in addition to evolving appropriate coefficient values for the classification task, may also be able to simultaneously perform feature selection. SAGA achieves these results with very little parameter tuning; showing the capability of SAGA to be quickly and easily applied to novel real-world problems.

CHAPTER 7: CONCLUSIONS

We implement a new self-adaptive genetic algorithm in which the parameters of mutation rate, mutation operator selection, crossover rate, crossover operator selection, and parent selection pressure are made self-adaptive. SAGA is designed with the intent of reducing the amount of computation time needed to tune a GA, while finding fitness results that are at least equal to a canonical GA. Reducing tuning and computation time greatly improves the practicality and usability of GAs, particularly for those who are not experts in the field. Usability of algorithms is often overlooked in the pursuit of improved fitness results; however, if an algorithm is too difficult or cumbersome to use, then it's use in real-world scenarios will be greatly diminished.

This SAGA is designed with two chromosomes per individual, a solution chromosome containing the solution of the problem, and a parameter chromosome containing the self-adaptive parameters. Both chromosomes evolve over the course of a run through the genetic operators. The parameter chromosome contains all of the self-adaptive parameter values. Crossover and mutation rate are made self-adaptive by encoding the rates directly into the parameter chromosome. The selection of crossover and mutation operators is made self-adaptive by encoding a usage rate and performing a probabilistic selection based on this usage rate. All additional arguments required by any of the operators are also encoded and self-adaptive.

We compare SAGA to CGA on static problems, dynamic problems, and a real-world medicare problem. The static problems consist of both continuous and discrete problems. The focus of the static problems experiment is on the usability improvements of SAGA. The dynamic problems consist of both steady problem dynamics as well as sudden problem dynamics. In the dynamic problem experiment we also compare SAGA to other GA methods that are designed specifically for dynamic problems. The focus of the dynamic problem experiment is to see if the self-adaptive parameters can react appropriately to a changing fitness landscape and be competitive with the methods specialized for dynamic problems. The real-world medicare problem is a single classification problem. In the medicare problem experiment we also compare SAGA to logistic regression. The focus of the medicare problem experiment is to validate SAGA's viability in a real-world problem scenario; this is important as our overall goal is the improvement of GA usability for real-world scenarios.

We find good results for SAGA on static continuous problems. Equivalent fitness results to a CGA are found on all of the continuous problems with greatly reduced total fitness evaluations. Increased generations for SAGA are required for most static continuous problems to match the fitness results of the CGA; however, the number of fitness evaluations are still significantly smaller than fully tuning a CGA due to the large tuning reductions. Mixed results are found on the static discrete problems. SAGA finds equivalent fitness to a CGA on the Knapsack problem. The results for the N-Queens problem results are mediocre and never match the best fitness results of a tuned CGA. Our system performed poorly on TSP and does not appear to be a viable problem for this implementation of a self-adaptive GA at present. An analysis of SAGA's self-adaptive parameters reveal that SAGA's adaptations do respond to the characteristics of the problems; however, they seem to do so more decisively on continuous problems than on discrete problems.

SAGA performs quite well on dynamic problems despite not being specifically designed for such problems. We apply SAGA to dynamic problems as we saw the potential for the process of parameter self-adaptation in aid in tracking a moving optimal fitness. The results for SAGA on dynamic problems are comparable to or superior to other GA methods that are designed specifically for dynamic problems depending on which performance metric is considered. The self-adaptive GA achieves this with very little parameter tuning, upholding our goal of tuning simplification. The type of dynamics determines how effective SAGA will be; while SAGA was capable on most dynamic problems, the one case it was not able to handle was the drastic fitness landscape changes that occurs every 20 generations. Furthermore, SAGA can be combined with many of the other dynamic-problem GA methods, such as Triggered Hypermutation, combining the benefits of both methods.

Good results are found on the real-world Medicare payment classification problem. This Medicare problem is a real-world problem within the field of healthcare informatics. Testing SAGA on such a problem verifies SAGA's applicability on real-world problems. This test was done with equivalent number of generations to the CGA. SAGA's results are comparable to or better than the CGA results. The results are also superior to logistic regression, a common method used in the field. We also highlight the ability for GAs in general, both self-adaptive and otherwise, to simultaneously perform feature selection and classification.

Future work can further examine the poorly performing problems to find improvements that can be made to SAGA to improve performance on these problems. This future work includes investigating alternative representation strategies to try to expand and improve SAGA on other problem types such as TSP. Another path to investigate is developing new operator selection methods to improve fitness performance. The operator selection method has a noticeable influence on how the self-adaptive parameters evolve. We used selection methods based upon common selection methods used for GA parent, but the selection methods tested are not exhaustive of all known selection operators and new, novel methods designed specifically for operator selection may prove more effective. Expanding self-adaption to other parameters to further reduce tuning is also potentially a very useful expansion.

We achieved our goal of developing an algorithm, SAGA, that can improve GA usability without sacrificing fitness performance on a majority of the problems tested. The problems on which SAGA did prove a success show massively reduced computation times and tuning effort, which means results can be obtained much more quickly and easily. This improvement in usability allows novice users to more effectively use GAs while also allowing experts to spend their time more efficiently by reducing the burden of parameter tuning.

LIST OF REFERENCES

[1] Adnan Acan et al. “A genetic algorithm with multiple crossover operators for optimal frequency assignment problem”. In: *Congress on Evolutionary Computation*. Vol. 1. IEEE, 2003, pp. 256–263.

[2] American Physical Therapy Association. *Guide to Physical Therapist Practice 3.0*. Alexandria, VA. Available at: http://guidetoptpractice.apta.org/. Accessed December 4, 2014. 2014.

[3] Thomas Bäck. “Self-adaptation in genetic algorithms”. In: *Proceedings of the First European Conference on Artificial Life*. MIT Press, 1992, pp. 263–271.

[4] Thomas Bäck. “The interaction of mutation rate, selection, and self-adaptation within a genetic algorithm”. In: *Proceedings of the 2nd Conference of Parallel Problem Solving from Nature*. Elsevier Science Publishers, 1992.

[5] James C. Bean. “Genetic algorithms and random keys for sequencing and optimization”. In: *ORSA Journal on Computing* 6.2 (1994), pp. 154–160.

[6] Hans-Georg Beyer. “Toward a theory of evolution strategies: Self-adaptation”. In: *Evolutionary Computation* 3.3 (1996), pp. 311–347.

[7] Stephan Blum et al. “Adaptive mutation strategies for evolutionary algorithms”. In: *Weimar Optimization and Stochastic Days*. 2005.

[8] Vikas Chaurasia and Saurabh Pal. “Data mining techniques: to predict and resolve breast cancer survivability”. In: *International Journal of Computer Science and Mobile Computing* 3.1 (2014), pp. 10–22.

[9] Helen G. Cobb. *An investigation into the use of hypermutation as an adaptive operator in genetic algorithms having continuous, time-dependent nonstationary environments*. Tech. rep. Naval Research Lab, 1990.

[10] Helen G. Cobb and John J. Grefenstette. *Genetic algorithms for tracking changing environments*. Tech. rep. Naval Research Lab, 1993.

[11] Carlos Contreras-Bolton and Victor Parada. “Automatic combination of operators in a genetic algorithm to solve the traveling salesman problem”. In: *PLOS ONE* 10.9 (2015).

[12] Maria José Perestrello De Vasconcelos et al. “Spatial prediction of fire ignition probabilities: comparing logistic regression and neural networks”. In: *Photogrammetric Engineering & Remote Sensing* 67.1 (2001), pp. 73–81.

[13] Kalyanmoy Deb and Ram Bhushan Agrawal. “Simulated binary crossover for continuous search space”. In: *Complex Systems* 9.2 (1995), pp. 115–148.

[14] Kalyanmoy Deb and Debayan Deb. “Analysing mutation schemes for real-parameter genetic algorithms”. In: *International Journal of Artificial Intelligence and Soft Computing* 4.1 (2014), pp. 1–28.

[15] Kalyanmoy Deb, Dhiraj Joshi, and Ashish Anand. “Real-coded evolutionary algorithms with parent-centric recombination”. In: *Congress on Evolutionary Computation*. Vol. 1. IEEE, 2002, pp. 61–66.

[16] Satchidananda Dehuri et al. “Application of elitist multi-objective genetic algorithm for classification rule generation”. In: *Applied Soft Computing* 8.1 (2008), pp. 477–487.

[17] Benjamin Doerr, Carsten Witt, and Jing Yang. “Runtime analysis for self-adaptive mutation rates”. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2018, pp. 1475–1482.

[18] A. E. Eiben, Robert Hinterding, and Zbigniew Michalewicz. “Parameter control in evolutionary algorithms”. In: *IEEE Transactions on Evolutionary Computation* 3.2 (1999), pp. 124–141.

[19] A. E. Eiben and Selmar K. Smit. “Evolutionary algorithm parameters and methods to tune them”. In: *Autonomous Search*. Springer, 2011, pp. 15–36.

[20] A. E. Eiben and Selmar K. Smit. “Parameter tuning for configuring and analyzing evolutionary algorithms”. In: *Swarm and Evolutionary Computation* 1.1 (2011), pp. 19–31.

[21] Agoston E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. Vol. 53. Springer, 2003.

[22] Larry J. Eshelman and J. David Schaffer. “Real-coded genetic algorithms and interval-schemata”. In: *Foundations of Genetic Algorithms*. Vol. 2. Elsevier, 1993, pp. 187–202.

[23] Alberto Fernández et al. “Genetics-based machine learning for rule induction: state of the art, taxonomy, and comparative study”. In: *IEEE Transactions on Evolutionary Computation* 14.6 (2010), pp. 913–941.

[24] Marcos Vinicius Fidelis, Heitor S. Lopes, and Alex A. Freitas. “Discovering comprehensible classification rules with a genetic algorithm”. In: *Proceedings of the Congress on Evolutionary Computation*. Vol. 1. 2000, pp. 805–810.

[25] Terence C. Fogarty. “Varying the probability of mutation in the genetic algorithm”. In: *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers Inc., 1989, pp. 104–109.

[26] Tobias Friedrich et al. “Analysis of diversity-preserving mechanisms for global exploration”. In: *Evolutionary Computation* 17.4 (2009), pp. 455–476.

[27] John J. Grefenstette. “Genetic algorithms for changing environments”. In: *Parallel Problem Solving from Nature*. Vol. 2. 1992, pp. 137–144.

[28] John J. Grefenstette. “Optimization of control parameters for genetic algorithms”. In: *Transactions on Systems, Man, and Cybernetics* 16.1 (1986), pp. 122–128.

[29] H. Altay Guvenir and Erdal Erel. “Multicriteria inventory classification using a genetic algorithm”. In: *European Journal of Operational Research* 105.1 (1998), pp. 29–37.

[30] David Hadka and Patrick Reed. “Borg: An auto-adaptive many-objective evolutionary computing framework”. In: *Evolutionary Computation* 21.2 (2013), pp. 231–259.

[31] Nikolaus Hansen and Andreas Ostermeier. “Completely derandomized self-adaptation in evolution strategies”. In: *Evolutionary Computation* 9.2 (2001), pp. 159–195.

[32] Georges R. Harik and Fernando G. Lobo. “A parameter-less genetic algorithm”. In: *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation*. Vol. 1. Morgan Kaufmann Publishers Inc., 1999, pp. 258–265.

[33] Jürgen Hesser and Reinhard Männer. “Towards an optimal mutation probability for genetic algorithms”. In: *Parallel Problem Solving from Nature*. Springer, 1990, pp. 23–32.

[34] Robert Hinterding. “Gaussian mutation and self-adaption for numeric genetic algorithms”. In: *Conference on Evolutionary Computation*. Vol. 1. IEEE, 1995, pp. 384–389.

[35] Robert Hinterding. “Self-adaptation using multi-chromosomes”. In: *Conference on Evolutionary Computation*. IEEE, 1997, pp. 87–91.

[36] Robert Hinterding, Zbigniew Michalewicz, and A. E. Eiben. “Adaptation in evolutionary computation: A survey”. In: *Proceedings of the International Conference on Evolutionary Computation*. IEEE, 1997, pp. 65–69.

[37] Henrik Höglund. “Tax payment default prediction using genetic algorithm-based variable selection”. In: *Expert Systems with Applications* 88.1 (2017), pp. 368–375.

[38] John Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[39] Tzung-Pei Hong, Hong-Shung Wang, and Wei-Chou Chen. “Simultaneously applying multiple mutation operators in genetic algorithms”. In: *Journal of Heuristics* 6.4 (2000), pp. 439–455.

[40] Tzung-Pei Hong, Hong-Shung Wang, Wen-Yang Lin, et al. “Evolution of appropriate crossover and mutation operators in a genetic process”. In: *Applied Intelligence* 16.1 (2002), pp. 7–17.

[41] Matthew Hughes. “Investigating the effects Diversity Mechanisms have on Evolutionary Algorithms in Dynamic Environments”. PhD thesis. 2016.

[42] Miles F. Jefferson et al. “Comparison of a genetic algorithm neural network with logistic regression for predicting outcome after surgery for patients with nonsmall cell lung carcinoma”. In: *Cancer: Interdisciplinary International Journal of the American Cancer Society* 79.7 (2000), pp. 1338–1342.

[43] Kenneth Alan De Jong. “Analysis of the behavior of a class of genetic adaptive systems”. PhD thesis. 1975.

[44] Bryant A. Julstrom. “Adaptive operator probabilities in a genetic algorithm that applies three operators”. In: *Proceedings of the Symposium on Applied Computing*. ACM, 1997, pp. 233–238.

[45] Giorgos Karafotias, A. E. Eiben, and Mark Hoogendoorn. “Generic parameter control with reinforcement learning”. In: *Proceedings of the Annual Conference on Genetic and Evolutionary Computation*. ACM, 2014, pp. 1319–1326.

[46] Giorgos Karafotias, Mark Hoogendoorn, and A. E. Eiben. “Parameter control in evolutionary algorithms: trends and challenges”. In: *IEEE Transactions on Evolutionary Computation* 19.2 (2015), pp. 167–187.

[47] Miha Kovačič and Franjo Dolenc. “Prediction of the natural gas consumption in chemical processing facilities with genetic programming”. In: *Genetic Programming and Evolvable Machines* 17.3 (2016), pp. 231–249.

[48] Nga Lam Law and Kwok Yip Szeto. “Adaptive genetic algorithm with mutation and crossover matrices”. In: *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*. 2007, pp. 2330–2333.

[49] Wen-Yang Lin, Wen-Yuan Lee, and Tzung-Pei Hong. “Adapting crossover and mutation rates in genetic algorithms”. In: *Journal of Information Science and Engineering* 19.5 (2003), pp. 889–903.

[50] Fernando G. Lobo and David E. Goldberg. “The parameter-less genetic algorithm in practice”. In: *Information Sciences* 167.1 (2004), pp. 217–232.

[51] Mark Mather, Linda A. Jacobsen, and Kelvin M. Pollard. “Aging in the United States”. In: *Population Bulletin* 70.2 (2015).

[52] H. David Mathias and Vincent R. Ragusa. “An empirical study of crossover and mass extinction in a genetic algorithm for pathfinding in a continuous environment”. In: *Congress on Evolutionary Computation*. IEEE, 2016, pp. 4111–4118.

[53] Brian Mc Ginley et al. “Maintaining healthy population diversity using adaptive crossover, mutation, and selection”. In: *IEEE Transactions on Evolutionary Computation* 15.5 (2011), pp. 692–714.

[54] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Berlin: Springer, 1992.

[55] K. L. Mills, J. J. Filliben, and A. L. Haines. “Determining relative importance and effective settings for genetic algorithm control parameters”. In: *Evolutionary Computation* 23.2 (2015), pp. 309–342.

[56] Sung-Hwan Min, Jumin Lee, and Ingoo Han. “Hybrid genetic algorithms and support vector machines for bankruptcy prediction”. In: *Expert Systems with Applications* 31.3 (2006), pp. 652–660.

[57] Ronald W. Morrison. “Performance measurement in dynamic environments”. In: *GECCO Workshop on Evolutionary Algorithms for Dynamic Optimization problems*. 2003, pp. 5–8.

[58] Ronald W. Morrison and Kenneth A. De Jong. “A test problem generator for non-stationary environments”. In: *Congress on Evolutionary Computation*. Vol. 3. 1999, pp. 2047–2053.

[59] Ronald W. Morrison and Kenneth A. De Jong. “Measurement of population diversity”. In: *International Conference on Artificial Evolution*. Springer, 2001, pp. 31–41.

[60] Ronald W. Morrison and Kenneth A. De Jong. “Triggered hypermutation revisited”. In: *Proceedings of the Congress on Evolutionary Computation*. Vol. 2. IEEE, 2000, pp. 1025–1032.

[61] Tadahiko Murata and Hisao Ishibuchi. “Positive and negative combination effects of crossover and mutation operators in sequencing problems”. In: *Proceedings of the International Conference on Evolutionary Computation*. IEEE, 1996, pp. 170–175.

[62] Desbordes Paul et al. “Feature selection for outcome prediction in oesophageal cancer using genetic algorithm and random forest classifier”. In: *Computerized Medical Imaging and Graphics* 60 (2017), pp. 42–49.

[63] Alain Pétrowski. “A clearing procedure as a niching method for genetic algorithms”. In: *Proceedings of the International Conference on Evolutionary Computation*. IEEE, 1996, pp. 798–803.

[64] David Pisinger. “Where are the hard knapsack problems?” In: *Computers & Operations Research* 32.9 (2005), pp. 2271–2284.

[65] Gerhard R. 1997. URL: `http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html`.

[66] Krzysztof L. Sadowski, Dirk Thierens, and Peter A. N. Bosman. "GAMBIT: A parameterless model-based evolutionary algorithm for mixed-integer problems". In: *Evolutionary Computation* 26.1 (2018), pp. 117–143.

[67] J. David Schaffer et al. "A study of control parameters affecting online performance of genetic algorithms for function optimization". In: *Proceedings of the Third Internation Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, 1989, pp. 51–60.

[68] Ramona Serban, Andrzej Kupraszewicz, and Gongzhu Hu. "Predicting the characteristics of people living in the south USA using logistic regression and decision tree". In: *IEEE International Conference on Industrial Informatics*. 2011, pp. 688–693.

[69] Martin Serpell and James E. Smith. "Self-adaptation of mutation operator and probability for permutation representations in genetic algorithms". In: *Evolutionary Computation* 18.3 (2010), pp. 491–514.

[70] Prashant Shrivastava et al. "A survey of nature-inspired algorithms for feature selection to identify Parkinson's disease". In: *Computer Methods and Programs in Biomedicine* 139 (2017), pp. 171–179.

[71] Jim Smith and T. C. Fogarty. "Self adaptation of mutation rates in a steady state genetic algorithm". In: *Conference on Evolutionary Computation*. IEEE, 1996, pp. 318–323.

[72] William M. Spears. "Adapting crossover in evolutionary algorithms." In: *Evolutionary Programming*. 1995, pp. 367–384.

[73] Dirk Thierens. "Adaptive mutation rate control schemes in genetic algorithms". In: *Proceedings of the Congress on Evolutionary Computation*. Vol. 1. IEEE, 2002, pp. 980–985.

[74] Lucas W. Thornblade, David R. Flum, and Abraham D. Flaxman. “Predicting future elective colon resection for diverticulitis using patterns of health care utilization”. In: *Journal for Electronic Health Data and Methods* 6.1 (2018), pp. 1–8.

[75] Shigeyoshi Tsutsui, Masayuki Yamamura, and Takahide Higuchi. “Multi-parent recombination with simplex crossover in real coded genetic algorithms”. In: *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation*. Vol. 1. Morgan Kaufmann Publishers Inc., 1999, pp. 657–664.

[76] Luke Vandewater et al. “An adaptive genetic algorithm for selection of blood-based biomarkers for prediction of Alzheimer’s disease progression”. In: *BMC Bioinformatics* 16.18 (2015), S1.

[77] David H. Wolpert and William G. Macready. “No free lunch theorems for optimization”. In: *IEEE Transactions on Evolutionary Computation* 1.1 (1997), pp. 67–82.

[78] Alden H. Wright. “Genetic algorithms for real parameter optimization”. In: *Foundations of Genetic Algorithms*. Vol. 1. Elsevier, 1991, pp. 205–218.

[79] Annie S. Wu and Ivan Garibay. “The proportional genetic algorithm: gene expression in a genetic algorithm”. In: *Genetic Programming and Evolvable Machines* 3.2 (2002), pp. 157–192.

[80] Annie S. Wu, Xinliang Liu, and Reamonn Norat. “A genetic algorithm approach to predictive modeling of Medicare payments to physical therapists”. In: *Proceedings of the Florida Artificial Intelligence Research Society Conference*. 2019.

[81] Hyun-Sook Yoon and Byung-Ro Moon. “An empirical study on the synergy of multiple crossover operators”. In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 212–223.

[82] Zhongheng Zhang et al. “Variable selection in logistic regression model with genetic algorithm”. In: *Annals of Translational Medicine* 6.3 (2018), p. 45.