Missouri S&T Library and Learning Resources

Scholars' Mine

Masters Theses

Student Theses and Dissertations

Summer 2009

Automated offspring sizing in evolutionary algorithms

André Chidi Nwamba

Follow this and additional works at: https://scholarsmine.mst.edu/masters_theses

 Part of the Computer Sciences Commons

Department:

Recommended Citation

Nwamba, André Chidi, "Automated offspring sizing in evolutionary algorithms" (2009). *Masters Theses*. 6784.
https://scholarsmine.mst.edu/masters_theses/6784

This thesis is brought to you by Scholars' Mine, a service of the Missouri S&T Library and Learning Resources. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

AUTOMATED OFFSPRING SIZING IN EVOLUTIONARY ALGORITHMS

by

ANDRÉ CHIDI NWAMBA

A THESIS

Presented to the Faculty of the Graduate School of

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

2009

Approved by

Dr. Daniel Tauritz, Advisor
Dr. Ralph Wilkerson
Dr. Ron Frank

Copyright 2009
André Chidi Nwamba
All Rights Reserved

ABSTRACT

Evolutionary Algorithms (EAs) are a class of algorithms inspired by biological evolution. EAs are applicable to a wide range of problems; however, there are a number of parameters to set in order to use an EA. The performance of an EA is extremely sensitive to these parameter values; setting these parameters often requires expert knowledge of EAs. This prevents EAs from being more widely adopted by non-experts. Parameter control, the automation of dynamic parameter value selection, has the potential to not only alleviate the burden of parameter tuning, but also to improve performance of EAs on a variety of problem classes in comparison to employing fixed parameter values. The science of parameter control in EAs is, however, still in its infancy and most published work in this area has concentrated on just a subset of the standard parameters. In particular, the control of offspring size has so far received very little attention, despite its importance for balancing exploration and exploitation.

This thesis introduces three novel methods for controlling offspring size: Self-Adaptive Offspring Sizing (SAOS), Futility-Based Offspring Sizing (FuBOS), and Diversity-Guided Futility-Based Offspring Sizing (DiGFuBOS). EAs employing these methods are compared to each other and a highly tuned, fixed offspring size EA on a wide range of test problems. It is shown that an EA employing FuBOS or DiGFuBOS performs on par with the highly tuned, fixed offspring size EA on many complex problem instances, while being far more efficient in terms of fitness evaluations. Furthermore, DiGFuBOS does not introduce any new user parameters, thus truly alleviating the burden of tuning the offspring size parameter in EAs.

ACKNOWLEDGMENT

First and foremost, I would first like to thank my advisor, Dr. Daniel Tauritz, for his advice and support; this thesis would not have been possible without it. I would also like to thank Dr. Ron Frank and Dr. Ralph Wilkerson for their participation on my thesis committee. I want to thank the Missouri S&T Deparment of Computer Science for the financial support it provided. Finally, I want to thank my family, especially my Mom and Dad, for all the obvious reasons.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

NOMENCLATURE

Symbol	Description
λ	Offspring Size
μ	Population Size

1. INTRODUCTION

1.1. MOTIVATION

Computers are becoming increasingly common in many fields, allowing people to solve problems that would not have been feasible before. However, the problems that need to be solved are growing in complexity. There is a widening gap between how much computational power current technology yields, and the amount of computational power needed to solve current problems in a reasonable amount of time. One of the ways to reduce this "computation gap" is to create more computationally efficient algorithms. One often used way of improving the efficiency of an algorithm is to take problem-specific information into consideration when designing the algorithm. However, it takes considerable time and effort to create problem-specific algorithms, and they usually can only be modified to solve similar problems. A method for solving these complex problems that requires little modification to solve a wide range of problems would be very useful, and such methods exist, e.g., hill climbing-algorithms, simulated annealing, and local beam search. The aforementioned methods perform well when the search space is well-behaved, like the search space shown in Figure 1.1. Unfortunately, real world problems typically have an ill-behaved search space. An ill-behaved search space is one that has certain undesirable traits, such as discontinuities and an exponentially increasing number of local optima (see Figure 1.2) in respect to problem size. On such problems, these algorithms often perform poorly. For those search spaces, we need more robust solution methods. One class of such methods is Evolutionary Algorithms.

An Evolutionary Algorithm (EA) is a stochastic, population-based, optimization algorithm that is inspired by concepts from biological evolution, such as recombination, mutation, and natural selection. The EA starts off with a pool of potential solutions to a problem. From this pool of potential solutions, new solutions are created. All these potential solutions undergo natural selection; the solutions compete with one another and the "weaker" ones die out. This process is repeated until some user-defined termination condition is met.

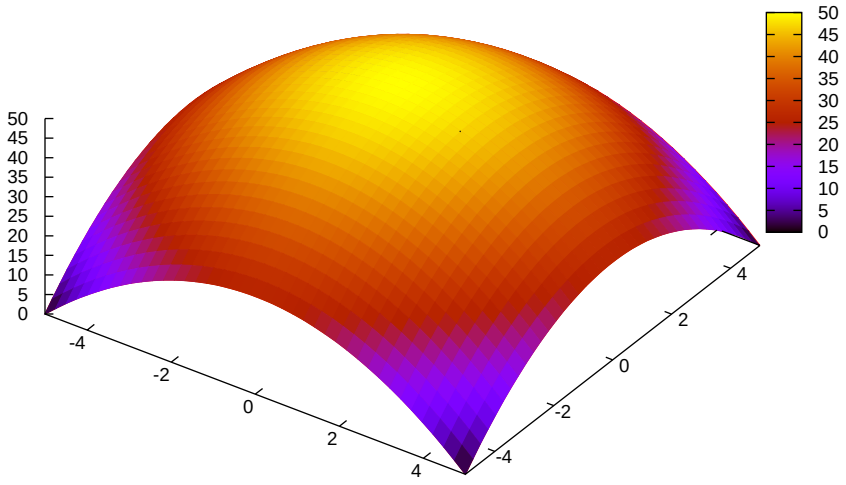

Figure 1.1: An example of a well-behaved search space

One of the major problems with EAs is the large number of strategy parameters the user must set. The performance of EAs is extremely sensitive to how these parameters are set, and setting these parameter values is a non-trivial task. The optimal value for these parameters is problem specific: there does not exist an optimal configuration for all EAs. Furthermore, there is interaction among parameters: the effect that one parameter has on the performance of the EA is dependent on what the other parameters are.

Parameter values can either remain fixed throughout the execution of the EA or change and adapt during the execution of the EA. Typically, standard EAs use fixed parameter values; manual tuning is the usual approach one takes when determining how to set these values. This process often involves executing test runs of an EA with different configurations to find a "good" set of parameters. Parameter control involves introducing an algorithm that changes and adapts the value used for a parameter during the execution of the EA. Using parameter control often requires less manual

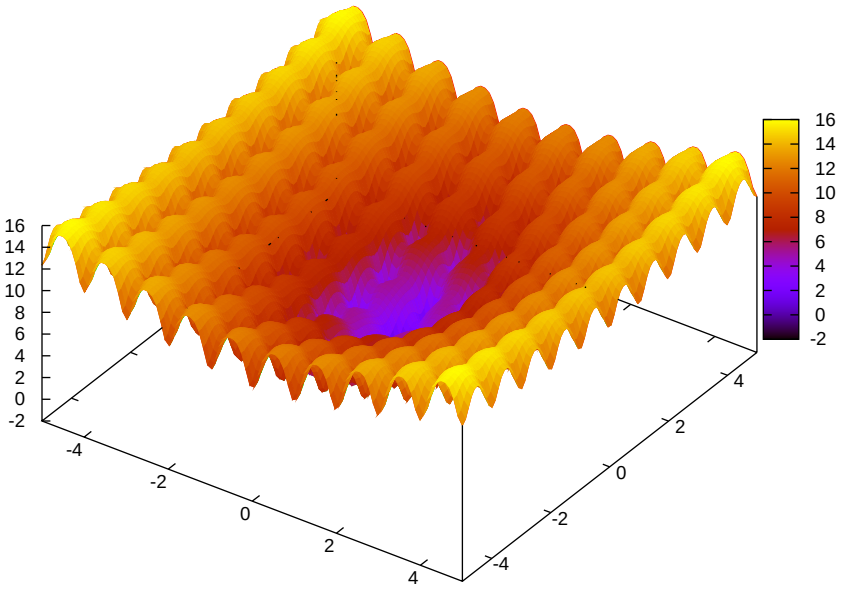

Figure 1.2: The Ackley Function [1] on $\mathbb{R}^2$: an example of a search space with an exponentially increasing number of local optima

tuning than using fixed parameter values. One reason for this is the performance of an EA using parameter control is typically less sensitive to the initial values used for those parameters than in an EA using fixed parameter values.

Parameter control in EAs has been both empirically and theoretically shown to be potentially beneficial in terms of improved performance on a variety of problem classes in comparison to the use of fixed parameter values, due to different parameter values being optimal at different stages of the evolutionary process [7]. An additional benefit is that parameter control has the potential to alleviate the burden of parameter tuning, either by removing the need to set a parameter altogether or by replacing the original parameter to be tuned with a new parameter to which performance is less sensitive. This is a significant benefit because, despite the efforts of many researchers in the field, parameter tuning remains to this day a very difficult and time-consuming task, often requiring significant expertise and thus preventing EAs from being more

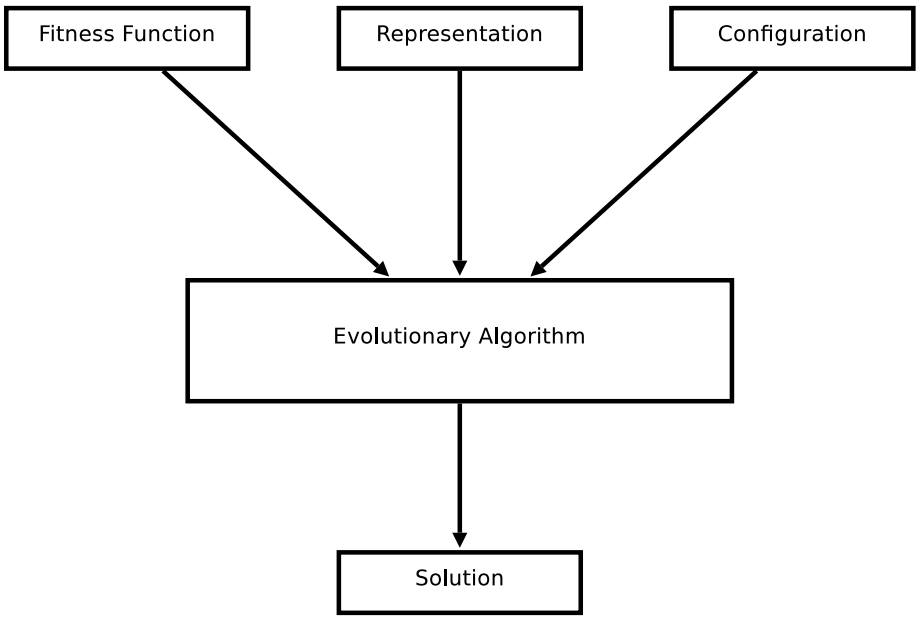

Figure 1.3: EA Overview

widely adopted by practitioners.

1.2. EVOLUTIONARY ALGORITHMS

One can think of an EA as a black box with three separate inputs, as shown in Figure 1.3. Two of these inputs are pieces of problem specific information: a representation and a fitness function. A representation is simply a way to represent a candidate solution to your problem. Commonly used representations are bitstrings, real-valued vectors, and trees. For instance, if the problem is finding the maximum value of $f(\vec{x}) = \Sigma_{i=1}^{n} x_i$, then a candidate solution to this problem could be a real-valued vector of length n. The representation of a candidate solution is analogous to the genes of an organism. A fitness function maps a candidate solution to a linear scale based on how well it solves the problem, with the fitness value begin proportional to the quality of the solution. The fitness function must have some gradient to the values it assigns, i.e., it can not assign a fitness value of 1 to the optimal solution and a fitness value of 0 to any other solution as that would result in the EA doing no better than random search. The last input required for the EA is a set of strategy parameters. Examples of strategy parameters include population size, offspring size,

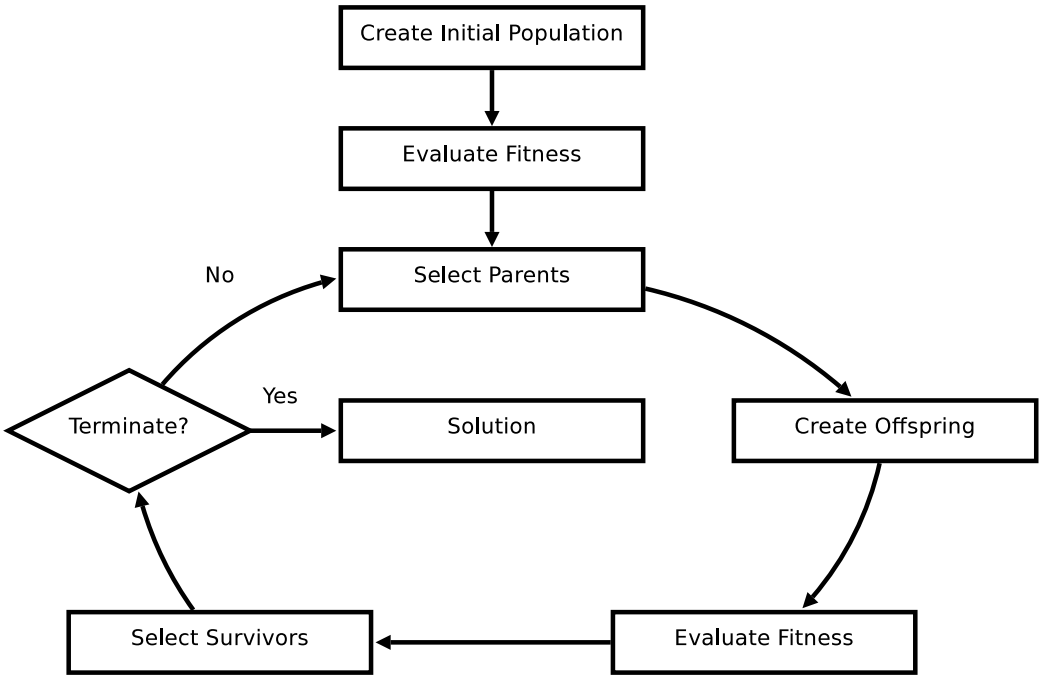

Figure 1.4: Evolutionary Cycle

mutation rate, termination condition, choice of recombination operators, and choice of selection operators.

Looking inside the black box, one will find that the basic structure of an EA is very simple, as shown in Figure 1.4. The first task an EA performs is the creation of the initial population, a set of individuals. An individual contains a candidate solution and possibly additional genes or attributes. For an example of when an individual would have additional genes, see Section 2.2. Each candidate solution in the initial population can be generated in a variety of ways, including randomly, using a heuristic, and reusing results from a previous run. Once the initial population is created, the EA evaluates each individual in the population and assigns them a fitness value using the fitness function. Once every individual in the population has a fitness value, the EA enters the evolutionary cycle and parents are selected. There are numerous ways for a parent to be selected, and they usually are biased towards picking fitter individuals, i.e., individuals with higher fitness values. Once the parents have been selected, they undergo recombination with a certain probability. Through recombination, new individuals, referred to as the offspring, are created by using

genes from each parent. How these offspring are created depends partially on the representation of the individual; there are numerous recombination methods for a given representation. Once created, the offspring undergo, with a certain probability, mutation; the genes of an individual get slightly modified. Too much mutation and EAs degrade into random search. Too little and the EA can get "stuck" on a suboptimal solution. Once created, the offspring get assigned fitness values via the fitness function. Once all the offspring have been created, the EA needs to determine which individuals survive to form the new population. There are numerous methods to select the survivors, and they usually are biased towards killing off weaker individuals, i.e., individuals with lower fitness values. The survivors will form the new population, and the EA will have completed one generation. After each generation, the EA checks some termination criteria. This could be based on a variety of things, such as the number of fitness evaluations that have been performed, the number of generations that have occurred, and the average fitness of the population reaching a certain threshold. If the termination criteria has not been met, then parents are selected and another generation occurs. If the termination criteria has been met, then the EA terminates. The solution typically is the best individual in the population at the end of the final generation.

1.3. OVERVIEW

The science of parameter control in EAs is still in its infancy and most published work in this area has concentrated on just a subset of the standard parameters, in particular mutation step size [14, 25], population size [2, 26, 9], choice of recombination from a predefined set [4], and parent selection [15, 27, 16]. For a detailed review, see for instance [10, §8.5] or [5]. Yet, the control of the offspring size parameter, λ, has so far received very little attention, despite its importance for balancing exploration and exploitation [5].

The focus of this thesis is controlling λ. Section 3 introduces three novel methods to achieve this goal: Self-Adaptive Offspring Sizing (SAOS), Futility-Based Offspring Sizing (FuBOS), and Diversity-Guided Futility-Based Offspring Sizing (DiGFuBOS). SAOS takes a more natural approach to controlling λ. The choice of λ, normally determined at the population level in EAs, is determined at the individual level.

Each individual chooses how many offspring they want to have. Each individual is penalized based on how many children they have had, and their fitness is adjusted accordingly. FuBOS controls λ on a per generation basis without even requiring the user to set an initial offspring size value. This is done by observing the change in average fitness of all the offspring with each offspring created. This change is compared to a threshold value to determine if it would be useful to generate additional offspring in the current generation, thus dynamically adjusting the balance between exploration and exploitation. DiGFuBOS extends the concept of FuBOS by dynamically changing the threshold value every generation based on the genetic diversity of the population, the population size, and the problem size.

2. RELATED WORK

2.1. PARAMETER CONTROL

As mentioned in Section 1, parameter control is desirable due to its potential to improve the performance of EAs and to alleviate the burden of manually tuning parameters. Most published work has focused on only a few strategy parameters. In Evolutionary Strategies (ES), much progress has been made in controlling mutation step size. Covariance Matrix Adaptation (CMA), a method for controlling mutation step sizes, is currently one of the most powerful techniques for single-objective, real-valued, numerical optimization [17]. The basic idea is to use the "path" the EA travels on through the search space to update different CMA parameters. These parameters control the mutation step size.

Other components of an EA can be controlled also. For example, penalty functions are often used when dealing with constrained optimization problems. Usually, one defines a penalty function using the method of static penalties [21] because an existing EA requires very little modification to use this method. However, the method of static penalties introduces $2c + 1$ new parameters, where c is the number of constraints. Eiben et al. [8] showed that the parameters involved with defining a penalty function can be successfully controlled without the EA "cheating" by minimizing the parameter values associated with the penalty function instead of satisfying the constraints.

2.2. ADAPTIVE AND SELF-ADAPTIVE PARAMETER CONTROL

One commonly used method for parameter control in EAs is adaptation. Adaptation works by having a function, whose inputs are some form of feedback from the search, determine the magnitude and/or direction of the change to the parameters. One classic example of this is Rechenberg's 1/5 success rule [23] for controlling the mutation step size in Evolutionary Strategies. This rule states that one-fifth of mutations should be successful, i.e., one-fifth of the offspring created should be fitter than their parents. When the ratio between successful mutations and all mutations drops

below one-fifth, then the mutation step size is increased to widen the width of the search; conversely, if the ratio is higher than one-fifth, then the mutation step size decreases to concentrate the search around the current solutions.

Another method used for parameter control in EAs is self-adaptation. Self-adaptation works by encoding the parameters to be adapted into the genes of an individual. The encoded parameters undergo mutation and recombination along with the encoded candidate solution. The idea is that high quality individuals are created under ideal circumstances; in other words, when the EA's parameters are optimal considering the current state of the search. Self-Adaptation has been successfully used to control mutation step sizes [14]. However, using self-adaptation in EAs can lead to premature convergence [24], i.e., the individuals in the population converge to suboptimal solutions. Regardless, self-adaptation is known to be a highly robust, state-of-the-art adaptation technique that works well in real-valued search spaces [5].

2.3. EXPLORATION VS. EXPLOITATION

For an EA to be effective, it must balance exploration vs. exploitation. Exploration refers to creating new individuals in unvisited areas of the search space. In general, when EAs spend more time exploring the search space, this leads to the EA finding higher-quality solutions. Less exploration tends to result in premature convergence. Exploitation refers to creating new individual near areas where known good solutions have been found. When too little exploitation occurs, the EA does not take advantage of known good solutions and the progress of the search suffers. When too much exploitation occurs, the EA does not explore the search space thoroughly enough, leading to premature convergence.

2.3.1. Offspring Size Control. Determining the appropriate value for λ at any given point of an EA's execution is a difficult task; the optimal value for λ varies based on the problem, the other EA strategy parameters used, the EA's progress through the search space, various observed variables such as the rate of change of the average population fitness, and the termination condition. When λ is set too low, the possibility for premature convergence increases, especially for problems with a ill-behaved fitness landscape; while setting λ too high results in wasted computational effort [18].

λ is inversely proportional to the number of generations that occur during the execution of an EA, assuming a fixed value for λ and a predetermined number of fitness evaluations to be performed. Having more offspring per generation will result in a decreased number of generations, which means there will be less exploitation of the population [5]. Conversely, having less offspring per generation will result in more generations, but the number of offspring produced per generation is directly related to how much the EA explores the search space. So less offspring per generation implies less exploration performed, which can lead to premature convergence [5], in particular on more complex fitness landscapes. So the goal is to balance the number of offspring per generation and the number of generations, i.e., to balance exploration and exploitation. Note, also, that the optimal balance can shift during the execution of the EA because different values for λ may be appropriate at different points in an EA's execution [7].

2.3.2. Diversity Guided EA. Ursem [28] created a method for controlling the balance between exploration and exploitation in EAs. The Diversity Guided EA (DGEA) alternates between two different modes: exploration and exploitation. The DGEA starts in exploitation mode, and to determine which mode the DGEA is in, a diversity metric is compared to two manually-tuned threshold values, d_{low} and d_{high}. While the DGEA is in exploitation mode, it behaves like a typical EA. Diversity-reducing operators, such as survivor selection, are used by the DGEA until the diversity drops below a certain level specified by d_{low}, then the DGEA switches to exploration mode. While in exploration mode, the DGEA mutates the population until the diversity is above d_{high}. Then it switches back to exploitation mode. The diversity metric used by the DGEA on an N-dimensional problem is:

$$diversity(P) = \frac{1}{|L|\,|P|} \sum_{i=1}^{|P|} \sqrt{\sum_{j=1}^{N} (s_{ij} - \bar{s}_j)^2}, \tag{1}$$

where P is the population, s_{ij} is the j-th gene of the i-th individual, $\bar{s}_j$ is the average of the j-th gene over the population, and $|L|$ is the length of the diagonal of the search space, $\mathbb{R}^N$. Assuming each gene, x_i, is bounded above and below by x_{max} and x_{min}

respectively, $|L|$ is defined as follows:

$$|L| = \sqrt{N \cdot (x_{max} - x_{min})^2}. \tag{2}$$

2.3.3. Prior Work. There have been many successful attempts at using parameter control to reduce or eliminate the burden of selecting parameters a priori while maintaining or improving the performance of an EA [5]. However, very little work has been done in regard to dynamically controlling λ. Hansen et al. [13] devised a method for adjusting λ for (1,λ) Evolutionary Strategies (ESs) based on the second best individual created, the λ used last generation, and a manual-tuned parameter controlling adaptation speed. The goal of this strategy is to maximize the local serial progress-rate, i.e., the expected fitness gain per fitness evaluation. However, maximizing the local serial progress-rate is equivalent to maximizing the convergence rate, which often leads to premature convergence on complex fitness landscapes.

Jansen et al. [18] created a method for adjusting λ for (1+λ) ESs based on the number of offspring that are fitter than their parent. When none of the offspring created during a generation are fitter than their parent, λ is doubled; otherwise, λ is divided by the number of offspring that are fitter than the parent. The idea being that λ is increased quickly when it appears to be too small, and decreased based on the current success rate when it appears to be too large. While this approach was empirically shown to work well for less complex fitness landscapes, it had problems with more complex fitness landscapes that require a larger value for λ. On a complex fitness landscape, a larger λ is required to ensure that successful offspring lie on the path to the global optimum. However, it is impossible to know how many offspring lie on this path without a priori knowledge of the fitness landscape. So while the method proposed by Jansen et. al. does allow for λ to increase quickly, they concluded that there is no way of knowing how large λ needs to grow a priori without using knowledge of the problem. Thus, λ grew to smaller, suboptimal values instead.

The aforementioned methods were created for ESs with a population size, μ, of one. Although it is possible that these methods could be generalized for any population size, these methods inherently have drawbacks that prevent them from being well-suited for complex fitness landscapes. The Hansen method prematurely

converges due to its goal of maximizing the convergence rate, and the Jansen method tends to grow λ to smaller, suboptimal values. These deficiencies are independent from μ.

In [13], the concept of a "cut-off point" is introduced. This is the point during the mating process where having additional offspring will result in additional cost with negligible benefit to the search, i.e., the additional search space explored by having an additional offspring is not worth the computational effort required to create the offspring and evaluate its fitness. While determining the cut-off point can be done a priori employing asymptotical analysis for certain situations [13], it is usually far more difficult and time-consuming to do this than to manually tune λ. However, this concept can be used to determine how many offspring should be created during any given generation.

3. METHODOLOGY

3.1. SELF-ADAPTIVE OFFSPRING SIZING

EAs are biologically inspired, so it makes sense to look at nature for inspiration for how to control λ. In standard EAs, λ is fixed throughout its execution with λ offspring being created each generation; however, this is not what happens in nature. In nature, λ is determined at the individual level: each individual decides how many offspring it wants to have. An individual is free to have as few or as many offspring as it desires, but there are consequences. There is a limited amount of resources in any environment, such as food and space. If too many offspring are created, there will not be enough resources for each individual. In many animal species, parents support their offspring. There is only a finite amount of support a parent can give; more offspring for the parent to support results in less support for each individual offspring.

Individuals in typical EAs have no intelligence to make decisions with. However, by encoding the amount of offspring an individual wants to have into the gene of that individual, self-adaptation can be used to control the desired amount of offspring that individual wants to produce. Self-Adaptive Offspring Sizing (SAOS) uses this idea to control λ. SAOS allows each individual to specify how many living offspring they would like to have. During each generation, individuals mate until each individual has as many living offspring as it desires. Since the amount of offspring an individual wants to have is encoded into the genes of that individual, this desire gets passed down from parent to offspring. However, implementing this by itself has one main problem. An individual who wants to have many offspring will have all the offspring it desires and will pass down that desire to each of its offspring. These offspring will also have many offspring, passing down the desire to their offspring. The individuals who have many offspring will overwhelm those individuals who want to have few offspring with their sheer numbers. This means that there is the possibility of λ growing over time without bound. As mentioned in Section 1, when λ is too high, convergence speed decreases. Since each offspring created needs to be evaluated, this results in

the EA using its fitness evaluations inefficiently. One way of preventing unbounded growth of λ is to penalize each individual for each offspring it produces. This penalty is analogous to the support parents provide for their offspring seen with many higher organisms.

Algorithm 1 Self-Adaptive Offspring Sizing EA

```
initialize pop
for each P ∈ pop do
  P_offWant = 1
end for
evaluate fitnesses of pop
while termination condition has not been met do
  off ← ∅
  while there exist unsatisfied parents in pop do
    if there are k or more unsatisfied parents in pop then
      select k unsatisfied parents
    else
      clone all unsatisfied parents
    end if
    parents mate to create offspring o
    set o_offWant to average of parents' offWant
    o_offWant ← o_offWant + N(0, .2)
    evaluate fitness of o
    off ← o ∪ off
  end while
  select survivors using score instead of fitness value
end while
```

SAOS, outlined in Algorithm 1, gives each individual in the population, *pop*, an additional gene: $offWant$. $offWant$ indicates the number of living offspring an individual wants to have. Each individual's $offWant$ can be no less than 1 and no greater than μ. The lower bound on $offWant$ ensures that there are individuals in the population that want offspring. Without the lower bound, evolution can come to a halt due to no individual in the population wanting to have offspring. The upper bound on $offWant$ helps prevent λ from growing without bound. Based on emperical results, μ was determined to be a good value for the upper bound. Due

to this restriction, there is an upper bound on the amount of offspring created per generation, μ^2. For example, if $\mu = 3$, then the largest λ can be is 9 regardless of the problem size or the other parameter values. So the value of μ must be large enough in order to not restrict λ from growing to an appropriate value; thus, it would be impractical to use SAOS on an EA with $\mu = 1$.

For each generation, k "unsatisfied" parents are selected and have a single offspring together. A parent is considered unsatisfied if the amount of offspring they have created this generation plus the amount of offspring they have in the current population are less than their $offWant$. The offspring's $offWant$ is the average value of the parents' $offWant$. Each offspring's $offWant$ is then mutated by adding $N(0, .2)$ to it, where $N(0, .2)$ is a random value from a normal distribution with a mean of 0 and a variance of .2. The random value is chosen from a normal distribution because there is a high probability of the random value being close to 0, and a low probability of it being far from 0. There is a small chance for a large amount of mutation, and a large chance for a small amount of mutation. In general, this is a desirable way to perform mutation. The value of .2 was determined experimentally to be a good value to use for the variance, although the value to use for the variance might be dependent on other parameters, such as μ. This process is repeated until every individual is "satisfied". An individual is considered satisfied if it is not unsatisfied. At the end of each generation, all of the offspring, off, and individuals in *pop* get assigned a score based on a score function. The score of the individual is used in lieu of their fitness values during survivor selection.

The use of different score functions was examined in order to find one that best balances the need to have offspring to explore the search space and the need to keep the number of offspring being created in check:

$$score_1(ind) = (f(ind) - \min_{ind \in pop \cup off} f(ind)) \cdot (\mu - ind_{offWant}), \tag{3}$$

$$score_2(ind) = \frac{f(ind) - \min_{ind \in pop \cup off} f(ind)}{ind_{offWant}}, \tag{4}$$

and

$$score_3(ind) = (f(ind) - \min_{ind \in pop \cup off} f(ind)) \cdot (\mu - ind_{offWant}) \cdot (\mu - ind_{siblings}) \quad (5)$$

where $f(i)$ is the fitness value of individual i, $i_{siblings}$ is the number of individuals in $pop \cup off$ that share a parent with i, and μ is the population size. $score_3$ penalizes large "families" as an additional measure to keep λ in check. As shown in Section 5, each *score* function has its respective problems.

3.2. FUTILITY-BASED OFFSPRING SIZING

Futility-Based Offspring Sizing (outlined in Algorithm 2) was created in response to the problems present in SAOS, which shall be discussed in Section 6. The change in genetic diversity of the offspring with each new offspring generated so far in a particular generation can be used as a metric for determining how much the EA benefits from having additional offspring. As the change in genetic diversity approaches zero, the amount of reachable search space not explored by the offspring approaches zero too. While calculating genetic diversity is trivial for simple representations (e.g., bitstrings and real-valued vectors), this can be a problem for more complex representations (e.g., tree structures). Using the change in average fitness of the offspring is a less accurate metric for determining how much search space is being covered by the offspring, but it can be applied regardless of representation. Also, methods for calculating genetic diversity tend to be computationally expensive (e.g., pairwise Hamming distances), but calculating the change in average fitness of the offspring can be done in constant time.

FuBOS determines when it is futile to create more offspring, i.e., when the computational effort required to create additional offspring outweighs how much these additional offspring contribute to the progress of the search. To do this, it uses the offspring created during a given generation to decide if the mating process should continue. At the beginning of the mating process, two offspring, o_1 and o_2, are created. The average fitness of the offspring before o_2 was created is compared to the average fitness of the offspring after o_2 was created. If the difference between those fitness values becomes sufficiently small, the mating process stops; otherwise,

Algorithm 2 Futility-Based Offspring Sizing EA

```
initialize pop
evaluate fitnesses of pop
while termination condition has not been met do
  off ← ∅
  while |offspring| < 2 and Inequality 6 is true do
    select parents
    parents mate to create offspring o
    evaluate fitness of o
    off ← o ∪ off
  end while
  select survivors
end while
```

another offspring is created and the change in the average fitness is examined again. When the mating process stops, it is because o_n has a negligible small impact on the average fitness of the offspring. Thus, it is assumed that the additional search space explored by creating o_{n+1} is not worth the computational effort required to create o_{n+1} and evaluate its fitness. This means we have reached the cut-off point, and having additional offspring will not be beneficial for the search.

For the mating process to continue, FuBOS checks to see if the following condition is satisfied after an offspring has been created:

$$\frac{min\left(\left|\frac{\sum_{i=1}^{n} F(o_i)}{n}\right|, \left|\frac{\sum_{i=1}^{n-1} F(o_i)}{n-1}\right|\right)}{max\left(\left|\frac{\sum_{i=1}^{n} F(o_i)}{n}\right|, \left|\frac{\sum_{i=1}^{n-1} F(o_i)}{n-1}\right|\right)} < 1 - \epsilon \tag{6}$$

where $\epsilon = .000001$, o_i is the i-th offspring created this generation, and n is the number of offspring created this generation so far. Inequality 6 compares the average fitness of the offspring with the average fitness of the offspring excluding the most recently created offspring, o_n. The .000001 value for ϵ was initially determined experimentally using the DTRAP problem described in Section 4. Other values tested for ϵ resulted in a difference in performance of less than 1%, leading us to conclude that performance is far less sensitive to ϵ than λ. The sensitivity of the performance of FuBOS to ϵ was also examined using another problem and also found to be less sensitive than λ. Further analysis of the sensitivity of the performance of FuBOS to ϵ will need to be

performed in the future.

The left side of Inequality 6 produces a ratio (between 0 and 1 inclusive) between the average fitness of the offspring before the last offspring was created and the average fitness of the offspring after the last offspring was created. This is done to normalize the fitness values, as to mitigate the effects that the range of the fitness function could have when attempting to determine the magnitude of the change of the average fitness of the offspring. This ratio also indicates how the average fitness of the offspring has changed: if the ratio is close to 1, that means the change in the average fitness of the offspring is small.

If Inequality 6 is no longer true, this implies that o_n has a negligible small impact on the average fitness of the offspring. Based on this, it can be said that FuBOS attempts to maximize the amount of search space covered by the offspring while minimizing the wasted computational effort associated with producing too many offspring. By doing this, FuBOS can find the appropriate balance between exploration and exploitation.

3.3. DIVERSITY-GUIDED FUTILITY-BASED OFFSPRING SIZING

The biggest problem with FuBOS lies in the choice of ϵ. Although ϵ is a less sensitive parameter than λ, it would still be beneficial for the choice of ϵ to be automated. Diversity-Guided Futility-Based Offspring Sizing (DiGFuBOS) is a method for controlling λ that involves no additional parameters. DiGFuBOS (outlined in Algorithm 3) is similar to FuBOS, but differs from it in two ways. First, instead of looking at the change in average fitness, DiGFuBOS looks at the change in genetic diversity by actually looking at the genes of individuals. This change was made because different individuals can have the same fitness value, and that might cause FuBOS to terminate the mating process early. Genetic diversity was measured using "moment-of-inertia", a diversity measurement introduced by Morrison and De Jong [22].

The formula used to calculate the genetic diversity of the offspring, $offdiversity_N$,

is as follows:

$$offdiversity_N = \sum_{i=1}^{L} \sum_{j=1}^{N} (x_{ij} - c_i)^2, \tag{7}$$

where L is the length of the genotype (the number of bits in a bitstring or elements of a real-valued vector), N is the amount of offspring created so far during the current generation, x_{ij} is the i-th gene of the j-th offspring, and c_i, the average of the i-th gene over the offspring, is defined as:

$$c_i = \frac{\sum_{j=1}^{N} x_{ij}}{N}. \tag{8}$$

The computation of the moment-of-inertia has $O(n^2)$ complexity, while the traditional method for calculating the pair-wise hamming/Euclidean distance has $O(n^3)$ complexity. For DiGFuBOS, genetic diversity is calculated iteratively with each offspring created, so the complexity of each method is reduced to $O(n)$ and $O(n^2)$, respectively. Note, though, that the moment-of-inertia is equal to the pair-wise hamming distance divided by the population size when applied to bitstrings. By examining the genetic diversity of the offspring by looking at their genes, the applicability of DiGFuBOS is reduced to EAs using bitstrings and real-valued vectors for representations; however, this allowed for the "ϵ problem" of FuBOS to be neatly solved.

The second difference between FuBOS and DiGFuBOS is that ϵ is not set a priori and fixed throughout the EA's execution, but controlled during its execution without the need to set an initial value for ϵ. ϵ affects how many offspring are created. When ϵ is low, it is likely that more offspring will be created; the opposite is true when ϵ is high. Consider the following situations. When μ is low, there is need to have many offspring due to the fact that little genetic information is carried over between generations. When μ is high, the need for offspring is reduced due to the large amount of genetic information being saved between generations. Genetic diversity and λ are similarly related. When there is very little genetic diversity, it would be beneficial to have more offspring in order to increase exploration of the search space and to help prevent premature convergence. Also, larger problems need larger values of λ due to how much more additional search space there is to explore.

Algorithm 3 Diversity-Guided Futility-Based Offspring Sizing EA

```
initialize pop
evaluate fitnesses of pop
while termination condition has not been met do
  if popdiversity ≠ 0 then
    set ϵ according to Equation 9
  end if
  off ← ∅
  while |offspring| < 2 and Inequality 13 is true do
    select parents
    parents mate to create offspring o
    evaluate fitness of o
    off ← o ∪ off
  end while
  select survivors
end while
```

DiGFuBOS sets ϵ using the following equation:

$$\epsilon = \frac{\mu \cdot popdiversity}{L}, \tag{9}$$

where L is the length of the bitstring or real-valued vector and $popdiversity$ is a modified version of the diversity metric used in the Diversity Guided EA [28]. $popdiversity$ has a separate definition for bitstrings and real-valued vectors:

$$popdiversity_{bit} = \frac{2}{L\mu} \sum_{i=1}^{\mu} \sqrt{\sum_{j=1}^{L} (s_{ij} - \bar{s}_j)^2}, \tag{10}$$

$$popdiversity_{real} = \frac{2}{D\mu} \sum_{i=1}^{\mu} \sqrt{\sum_{j=1}^{L} (s_{ij} - \bar{s}_j)^2}, \tag{11}$$

where s_{ij} is the j-th gene of the i-th individual, $\bar{s}_j$ is the average of the j-th gene over the population, and D is defined as follows:

$$D = \sqrt{L \left(\max_{1<i<\mu} \max_{1<j<L} s_{ij} - \min_{1<i<\mu} \min_{1<j<L} s_{ij} \right)}. \tag{12}$$

The rationale behind using a different formula for bitstrings lies in the fact that each bit in a bitstring does not actually represent a number, but represents true or false. 0 and 1 are often used for values of a bitstring, but any two values could just as easily be used. The co-domain of both $popdiversity_{bit}$ and $popdiversity_{real}$ is $[0, 1]$.

For the mating process to continue, DiGFuBOS checks to see if the following condition is satisfied after the N-th offspring has been created:

$$\frac{offdiversity_{N-1}}{offdiversity_N} < 1 - \epsilon \tag{13}$$

Inequality 13 checks to see if the change in genetic diversity has become small. When $offdiversity_N = 0$, DiGFuBOS terminates the mating process due to the lack of diversity in the offspring. If $\epsilon = 0$, then Inequality 13 will always hold if there exist two offspring that have different genes. A proof of this statement is presented in the appendix. Since Inequality 13 always holds when $\epsilon = 0$, offspring will be produced indefinitely and the current generation will never be completed. $\epsilon = 0$ only when the population has no diversity (e.g., $popdiversity = 0$), and this can happen when selective pressure is high enough on a complex fitness landscape. However, the population of an EA tends to converge slowly over time. If the diversity of the current population is 0, then it is likely that the diversity of the previous generation's population is close to 0. This would mean the previous value of epsilon would be a good value to use. So when $popdiversity = 0$, then epsilon is not changed.

4. EXPERIMENTAL DESIGN

4.1. EXPERIMENTS

To determine how the methods described in Section 3 effect the performance of an EA, EAs using each of the methods described in Section 3 were compared to a standard EA with a manually tuned value for λ, the Optimal Fixed Offspring Size EA (OFOS-EA). Although OFOS-EA is impractical due to the large number of fitness evaluations required to tune it (see Table 4.6), it provides an upper bound for the performance of an EA using fixed values for λ. Also, the aforementioned methods were compared to an EA using a random value for $\lambda(\in [1, 1000])$ for each generation, the Random Offspring Sizing EA (ROS-EA). This was done to examine the effect of just changing λ per generation; it is possible that the fact that these methods change λ every generation has more effect on the performance of an EA than how λ changes.

Also, FuBOS-EA, OFOS-EA, and ROS-EA were compared to an EA using the offspring sizing method proposed by Hansen et al. (Hansen-EA) [13] and the method proposed by Jansen et al. (Jansen-EA) [18]. Hansen-EA and Jansen-EA assume $\mu = 1$, so this set of experiments was conducted with $\mu = 1$. DiGFuBOS requires $\mu > 1$, so DiGFuBOS-EA was not compared with Jansen-EA and Hansen-EA. Also, since both Hansen-EA and Jansen-EA use elitist survivor selection methods, FuBOS-EA, OFOS-EA, and ROS-EA all use Truncation for survivor selection. Due to the constraints placed on the maximum number of offspring each individual can have when using SAOS, the EAs using SAOS were also not compared with Jansen-EA and Hansen-EA.

The EAs listed in Table 4.1 (excluding Hansen-EA and Jansen-EA) were all tested using different population sizes to determine how well each λ control method scales, in terms of performance, with μ. Due to the authors' computation time constraints, 1000 was chosen for the largest population size. Based on that, values of 100 and 500 were chosen for the other population sizes. All the EAs used the same parameters on each of the 4 test problems (shown in Table 4.2, Table 4.3, Table 4.4, and Table 4.5 respectively), to ensure fairness. These parameters were manually tuned

Table 4.1: Aliases for EAs

Alias	Description
OFOS-EA	EA using manually tuned value for λ
ROS-EA	EA using random value for λ
FuBOS-EA	EA using FuBOS
DiGFuBOS-EA	EA using DiGFuBOS
$SAOS_1$-EA	EA using SAOS with $score_1$ function
$SAOS_2$-EA	EA using SAOS with $score_2$ function
$SAOS_3$-EA	EA using SAOS with $score_3$ function
Jansen-EA	EA using method described in [13]
Hansen-EA	EA using method described in [18]

Table 4.2: EA parameters used in the experiments on ONEMAX problem

Parameter	Value
Initialization	Each bit is initialized to either 0 or 1 with a uniform probability
Parent Selection	6-1 Tournament
Survivor Selection	Truncation when $\mu = 1$, 3-1 Tournament otherwise
Recombination	Uniform Crossover
Mutation Rate	$1/l$ (l :=the length of the bitstring)
Termination Condition	Optimal solution found

using OFOS-EA.

The following is an explanation of some of the parameters used in the experiments. k-1 tournament selection is performed by picking k individuals and selecting the "winner" of the tournament: the fittest individual out of those k individual. So in terms of parent selection, the winner of the tournament is selected as a parent. In terms of survivor selection, the winner of the tournament becomes part of the population for the next generation. Truncation survivor selection means the survivors are the top μ fittest individuals that are either newly-created offspring or currently in the population.

The λ used by OFOS-EA for each experiment (which can be found in Table 4.7) was determined by using the following method (which is outlined in Algorithm 4).

Table 4.3: EA parameters used in the experiments on 3-SAT problem

Parameter	Value
Initialization	Each bit is initialized to either 0 or 1 with a uniform probability
Parent Selection	Random
Survivor Selection	Truncation
Recombination	Uniform Crossover
Mutation Rate	$1/l$ (l :=the length of the bitstring)
Termination Condition	300,000 evals

Table 4.4: EA parameters used in the experiments on DTRAP problem

Parameter	Value
Initialization	Each bit is initialized to either 0 or 1 with a uniform probability
Parent Selection	Random
Survivor Selection	Truncation when $\mu = 1$, 4-1 Tournament
Recombination	2-point crossover
Mutation Rate	$1/l$ (l :=the length of the bitstring)
Termination Condition	200,000 evals

Table 4.5: EA parameters used in the experiments for GRIEWANK problem

Parameter	GRIEWANK
Initialization	Each $\vec{x}_i$ is chosen from $[-600, 600]$ with a uniform probability
Parent Selection	3-1 Tournament
Survivor Selection	Truncation when $\mu = 1$, 6-1 Tournament
Recombination	Arithmetic crossover
Mutation Rate	Self-Adaptive Mutation
Termination Condition	200,000 evals

First, the EA is executed with $\lambda = 1$ and the average of the performance metric used for each problem (as previously specified in this subsection) over 30 runs and the standard deviation of that metric is recorded. Then the EA is executed using $\lambda = 25i$

Table 4.6: Number of fitness evaluations used to tune OFOS-EA

Population Size	1	100	500	1000
DTRAP	42,000,000	192,000,000	174,000,000	60,000,000
ONEMAX	6,673,542	4,887,021	14,942,448	21,668,019
GRIEWANK	42,000,000	90,000,000	84,000,000	36,000,000
3-SAT	54,000,000	279,000,000	369,000,000	657,000,000

Table 4.7: Values for offspring size found when manually tuning OFOS-EA

Population Size	1	100	500	1000
DTRAP	25	675	525	75
ONEMAX	1	1	75	225
GRIEWANK	1	275	100	25
3-SAT	1	425	600	1775

for $i \in \mathbb{Z}^+$, with the λ_{max} being the value for λ that gave the best results. For each value of λ, the EA is executed 30 times, or until it is determined that using that value for λ gives significantly different results than when using λ_{max}. This process terminates once the best λ recorded is significantly better than 3 larger values for λ. The point of this method is to not only find the best value for λ, but to also reduce the amount of computation time required by looking for a downward trend in performance as λ increases due to the decreasing number of generations. The reason why λ increases by 25 and 3 strikes are used is to reduce the amount of computation time required to manually tune OFOS-EA while still performing a thorough search. The performance achieved by the EA when using different values of λ are compared using the two-sample t-test with a significance level of .05 assuming unequal variances. OFOS-EA required a significant amount of fitness evaluations to manually tune it, as shown in Table 4.6.

4.2. TEST SUITE

All EAs were compared using test problems of various fitness landscapes: the "counting ones" problem (ONEMAX), the 4-bit bounded deceptive trap problem

Algorithm 4 Manual Tuning Algorithm

```
λ = 1
MBF_best = MBF_λ = EA(λ)
λ = 25
strikes = 3
repeat
  MBF_λ = EA(λ)
  if MBF_λ > MBF_best then
    MBF_best = MBF_λ
    strikes = 3
  end if
  if MBF_best statistically significantly better than MBF_λ then
    strikes = strikes − 1
  end if
  λ = λ + 25
until strikes = 0
```

(DTRAP) [6], the Griewank function (GRIEWANK) [12], and a special case of the boolean satisfiability problem (3-SAT). To maximize the differentiating power of the experiments, the largest values were chosen for the problem sizes within the authors' computation time constraints.

For ONEMAX, the goal is simply to maximize the number of bits in a bitstring that are 1. For the experiments, a bitstring of length 2000 was used. This problem is an easy problem for not only EAs, but most optimization algorithms. This is due to the the fact that there is only one local optima: the global optima. As long as mutation is applied to the offspring, there is a very low probability of premature convergence.

For the DTRAP problem, a bitstring of length $4n$ composed of n ($\in \mathbb{N}$) 4-bit trap functions is used. The trap function defined for a bitstring with x ones is as follows:

$$f_{dtrap}(x) = \begin{cases} 4 & \text{if } x = 4, \\ 3 - x & otherwise. \end{cases} \tag{14}$$

For example, for the bitstring 0011 1011 1111, the values of each trap are 1, 0, and

4 respectively. The DTRAP problem was chosen due to its "deceptive" nature. This is an especially hard problem for EA solvers because a candidate solution's fitness is increased as its distance to the optimum increases. For the experiments, a bitstring of length 1000 was used.

For the GRIEWANK problem, the goal is to minimize the following function (which is plotted in Figure 4.1):

$$f_{Griewank}(\vec{x}) = \sum_{i=1}^{n} \frac{\vec{x}_i^2}{4000} - \prod_{i=1}^{n} cos\left(\frac{\vec{x}_i}{\sqrt{i}}\right) + 1. \tag{15}$$

where $\vec{x}$ is a real-valued vector of length n. For these experiments, a real-valued vector of length 200 is used. The Griewank function is a well-known multi-modal function that is widely used to test optimization algorithms. The number of local optima increases exponentially as the number of dimensions of the problem increases [20].

For the 3-SAT problem, instances with 500 variables were used. All EAs were also tested on the 3-SAT problem with clause to variable ratios: 2, 3, 4, and 6 with $\mu = 500$. A clause to variable ratio of 4 was used for all other values of μ. Using different clause to variable ratios tests how well the performance of the EAs scale in respect to the difficulty of the problem. The 3-SAT instances used were in conjunctive normal form. To ensure fairness, sixty 3-SAT instances were randomly generated, and each instance was used for a single run of all EAs. 3-SAT is a classic problem in computer science. It was one of the first problems shown to be NP-Complete and a wide range of problems can be reduced to an instance of 3-SAT [19, 3].

4.3. PERFORMANCE METRICS

For the ONEMAX problem, the most obvious fitness function for a bitstirng is simply the number of ones it contains. GRIEWANK tries to minimize the Griewank function, so the obvious fitness value for a real-valued vector $\vec{x}$ is $-f_{Griewank}(\vec{x})$. A candidate solution for the DTRAP problem is a bitstring which is the concatenation of n 4-bit trap functions. So a good choice for the fitness function to use for DTRAP would be the sum of the individual trap functions. For the 3-SAT problem, the fitness function is the number of clauses which a candidate solution satisfies. This is

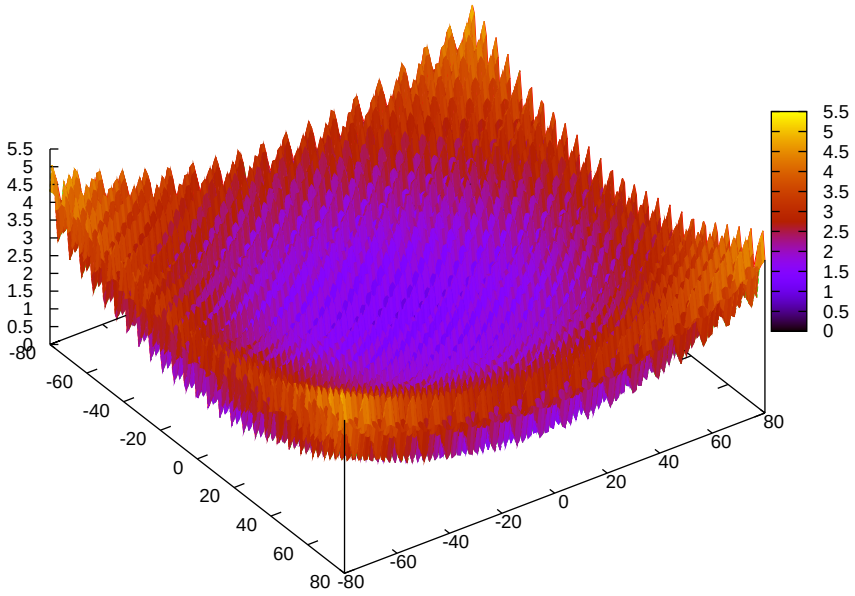

Figure 4.1: The Griewank function on $\mathbb{R}^2$

a commonly used fitness function for EAs solving the 3-SAT problem [11].

For the 3-SAT, DTRAP and GRIEWANK problems, performance was measured using the Mean Best Fitness (MBF) averaged over 60 runs. The Mean Best Fitness is the average of the fitnesses of the best individual ever created during multiple runs of an EA. This metric was chosen in order to measure the average performance one can expect from the EAs tested, since no two runs of any stochastic algorithm will necessarily yield the same result. For ONEMAX, all EAs have a high probability of converging to the optimal solution based on the parameters used as shown in Table 4.2, so using MBF as a performance metric would not yield much information about the performance of the EA. So in lieu of the MBF, the number of fitness evaluations used to obtain the optimal solution averaged over n runs, was used as a performance metric for the ONEMAX problem. Due to the stochastic nature of EAs, some statistical test needs to be performed in order to determine to some degree of certainty whether one EA will outperform another EA on average. The EAs were

compared, with their respective performance metric, using the two-sample t-test with a standard significance level of .05 assuming unequal variances.

5. RESULTS

The results, as shown in Table 5.1, were quite surprising. On the DTRAP problem, the SAOS EAs performed poorly in most cases. They were all outperformed by OFOS-EA for all values of μ, and performed on par with the other EAs when $\mu = 1$. The performance of $SAOS_1$-EA and $SAOS_3$-EA decreased as μ increased, especially when compared to OFOS-EA. While $SAOS_2$-EA did not have the same problems as the other two SAOS EAs, it was still consistently outperformed by OFOS-EA, FuBOS-EA, and DiGFuBOS-EA by a statistically significant margin. FuBOS-EA and DiGFuBOS-Ea faired far better than SAOS-EAs, performing on par with OFOS-EA when $\mu = 100, 1000$. However, there was a dip in performance when $\mu = 500$ for which they were outperformed by OFOS-EA by a statistically significant margin. ROS-EA actually performed unexpectedly well. It outperformed all of the other EAs tested when $\mu = 100, 500$; however, its performance dropped when $\mu = 1000$ for which it was outperformed by OFOS-EA, FuBOS-EA, and DiGFuBOS-EA by a statistically significant margin.

Another thing to look at besides performance is how λ changes during the EA's execution for FuBOS, DiGFuBOS-EA, and the SAOS EAs on the DTRAP problem. Figure 5.1 shows λ growing much larger for DiGFuBOS-EA than it does for FuBOS-EA on the DTRAP problem with $\mu = 100$. Figure 5.2 shows how λ changed during the execution of the SAOS EAs. λ grows large for $SAOS_1$-EA and $SAOS_3$, but shrinks halfway through its execution. When looking at how the average fitness of the population (shown in Figure 5.3) changes during $SAOS_1$-EA and $SAOS_3$-EA's execution, it can be seen that λ starts falling as the population's fitness values start to converge. λ barely grows for $SAOS_2$-EA; it quickly drops to a smaller, suboptimal value and remains near that value until it terminates. As shown in Figure 5.4, when $\mu = 500$, λ decreases in the beginning then increases towards the end of FuBOS-EA's execution, while λ for DiGFuBOS-EA increases in the beginning, decreases, then stays around 175 on average. When $\mu = 1000$, FuBOS and DiGFuBOS shrink and grow respectively, with λ to a certain range of values for both (as can be seen in Figure 5.6). The SAOS EAs exhibited similar behavior when $\mu = 500$ and when

Table 5.1: Performance of EAs on various problems in terms of the metrics specified in Section 4 (standard deviation in parentheses, best results are bolded)

Population Size	100	500	1000
DTRAP (in terms of MBF)			
FuBOS-EA	831.117 (6.08849)	864.933 (6.88928)	882.867 (7.4352)
DiGFuBOS-EA	833.233 (6.64421)	863.017 (5.99859)	883.1 (7.93452)
OFOS-EA	834.183 (5.46959)	868.9 (6.48511)	**884.467 (8.06736)**
ROS-EA	**834.8 (7.50511)**	**869.667 (7.06085)**	873.083 (8.63576)
$SAOS_1$-EA	831.283 (6.54495)	833.25 (8.9752)	789.883 (7.91221)
$SAOS_2$-EA	823.083 (7.17238)	855.567 (6.98896)	873.217 (6.93564)
$SAOS_3$-EA	831.833 (7.1278)	830.833 (7.30791)	788.917 (7.82792)
ONEMAX (in terms of average number of fitness evaluations used to find optimal solution)			
FuBOS-EA	59008.8 (5223.58)	49280.9 (5125.93)	**50801.3 (5115.64)**
DiGFuBOS-EA	551042 (30322.4)	66471.8 (5241.88)	55918.4 (4335.16)
OFOS-EA	**35410.2 (3964.47)**	**48049 (3710.87)**	53297.8 (5426.09)
ROS-EA	145273 (11628.9)	70526.3 (6039.66)	64622.8 (5474.98)
$SAOS_1$-EA	158540 (12930.4)	180857 (10384.5)	314670 (7293.65)
$SAOS_2$-EA	75961 (8980.85)	94652.8 (8218.94)	129282 (5512.72)
$SAOS_3$-EA	167568 (16284)	179243 (12275.2)	314727 (7381.58)
GRIEWANK (in terms of MBF)			
FuBOS-EA	-0.160202 (0.175902)	-0.0121147 (0.00834563)	-0.0868882 (0.0348113)
DiGFuBOS-EA	-0.540971 (0.503606)	-0.0119955 0.0328575)	-0.0392013 (0.0190165)
OFOS-EA	**-0.146447 (0.17793)**	**-0.00409469 (0.0108856)**	**-0.0390263 (0.0172252)**
ROS-EA	-1.14187 (0.450214)	-0.116357 (0.0467904)	-0.455314 (0.086007)
$SAOS_1$-EA	-0.395596 (0.200672)	-2.75214 (0.679632)	-6.29603 (1.77951)
$SAOS_2$-EA	-0.348629 (0.13839)	-1.19705 (0.048372)	-1.63009 (0.13839)
$SAOS_3$-EA	-0.38458 (0.209234)	-2.78335 (0.723493)	-6.5625 (1.75996)
3-SAT (in terms of MBF)			
FuBOS-EA	1986.15 (2.61932)	1988.63 (2.39421)	1985.83 (2.15381)
DiGFuBOS-EA	1988.48 (2.56575)	1989.33 (2.14994)	**1990.53 (1.97878)**
OFOS-EA	**1989.4 (2.31084)**	**1990.1 (2.19621)**	1990.37 (1.72208)
ROS-EA	1986.45 (2.59117)	1989.98 (2.10152)	1990.18 (2.17172)
$SAOS_1$-EA	1982.8 (3.646)	1988.63 (2.50311)	1988.38 (2.18397)
$SAOS_2$-EA	1984.93 (3.03791)	1989.55 (2.04471)	1990.38 (1.98403)
$SAOS_3$-EA	1982.42 (3.24187)	1988.87 (2.41155)	1990.28 (1.73293)

$\mu = 1000$. Figure 5.5 shows λ monotonically increasing during the execution of $SAOS_1$-EA and $SAOS_3$, and Figure 5.7 shows λ growing to even larger values. $SAOS_2$-EA grows λ for approximately the first 10% of the its execution, although λ does not grow nearly as large as it does for $SAOS_1$-EA and $SAOS_3$-EA. After the initial growth, λ drops to a lower value and remains near that value until $SAOS_3$-EA terminates.

On the ONEMAX problem, the SAOS EAs perform poorly for all values of μ when compared to OFOS-EA. They perform increasingly worse as μ increases, but so does OFOS-EA. Just like on the DTRAP problem, the performance of $SAOS_1$-EA and $SAOS_3$-EA on ONEMAX is worse than $SAOS_2$-EA. FuBOS-EA performance monotonically increased as μ increased, outperforming all other EAs tested when $\mu = 100$. DiGFuBOS-EA's performance increased in a similar manner. The performance

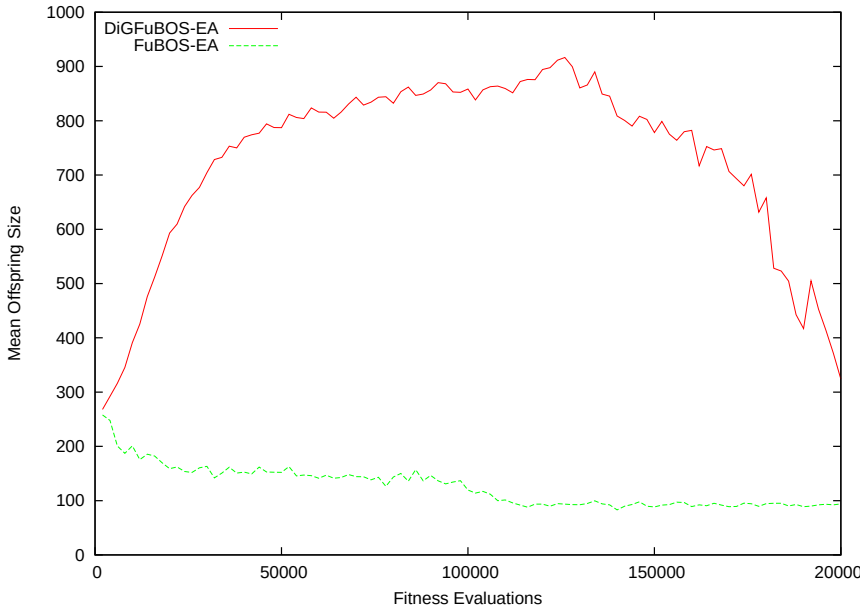

Figure 5.1: λ over the course of the execution of FuBOS-EA and DiGFuBOS-EA on the DTRAP problem with $\mu = 100$ averaged over 60 runs

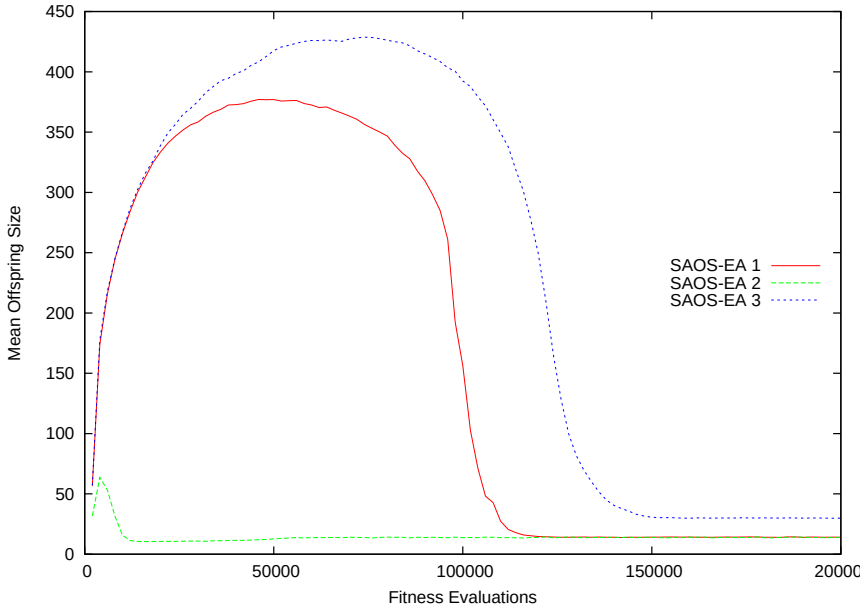

Figure 5.2: λ over the course of the execution of the SAOS EAs on the DTRAP problem with $\mu = 100$ averaged over 60 runs

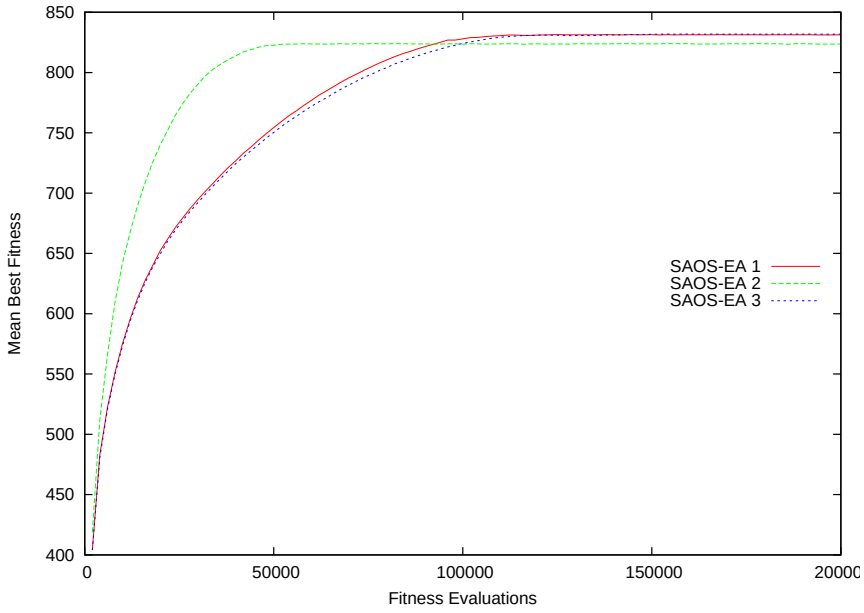

Figure 5.3: Fitness of best individual over the course of the execution of the SAOS EAs on the DTRAP problem with $\mu = 100$ averaged over 60 runs

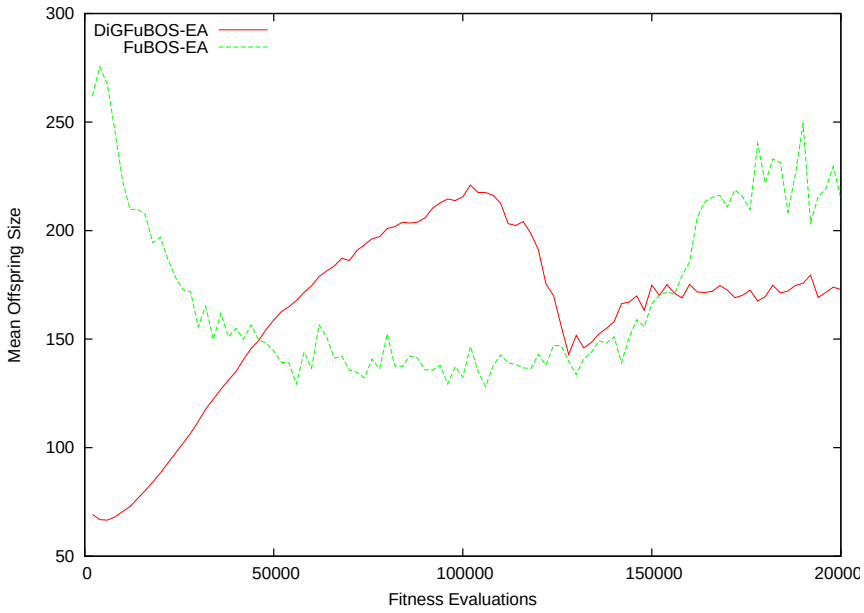

Figure 5.4: λ over the course of the execution of FuBOS-EA and DiGFuBOS-EA on the DTRAP problem with $\mu = 500$ averaged over 60 runs

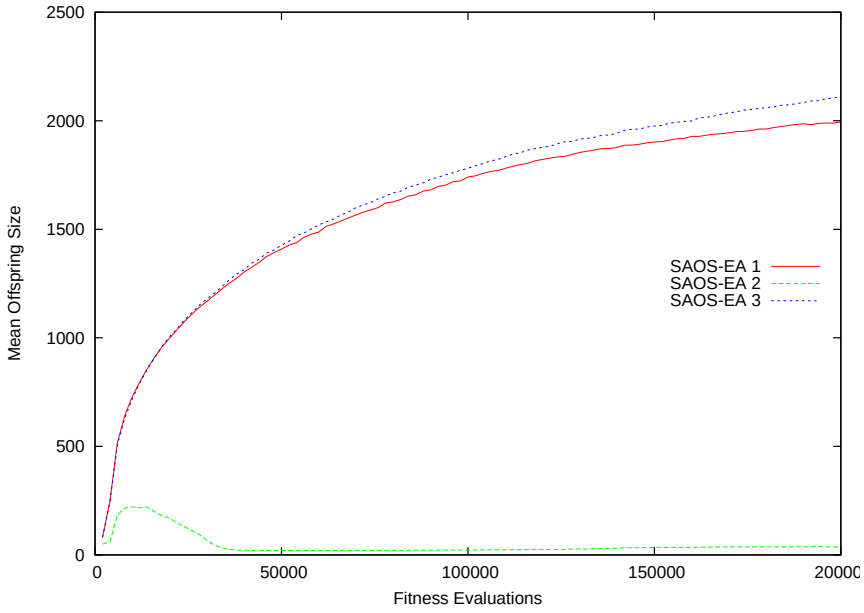

Figure 5.5: λ over the course of the execution of the SAOS EAs on the DTRAP problem with $\mu = 500$ averaged over 60 runs

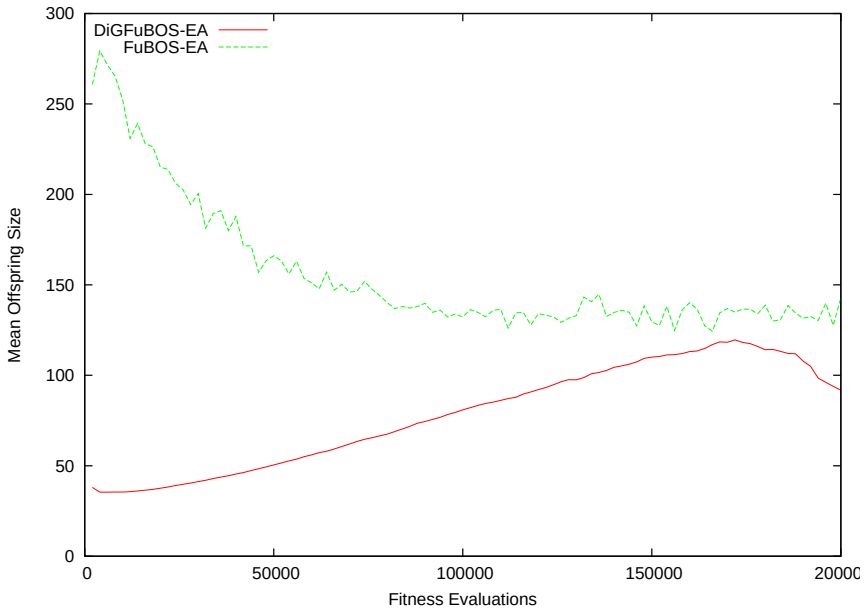

Figure 5.6: λ over the course of the execution of FuBOS-EA and DiGFuBOS-EA on the DTRAP problem with $\mu = 1000$ averaged over 60 runs

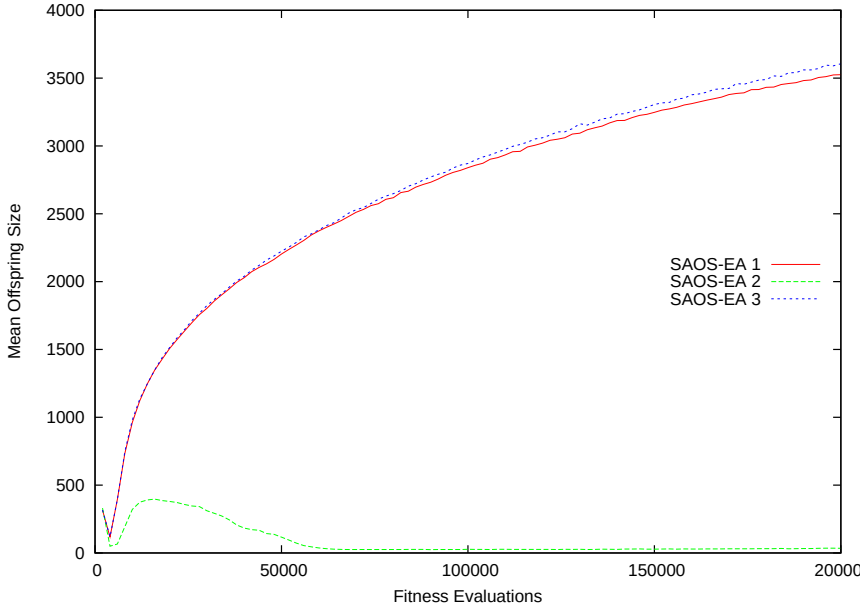

Figure 5.7: λ over the course of the execution of the SAOS EAs on the DTRAP problem with $\mu = 1000$ averaged over 60 runs

of DiGFuBOS-EA when $\mu = 100$ was the worst out of all EAs tested, yet was only outperformed by OFOS-EA and FuBOS-EA when $\mu = 500, 1000$. ROS-EA performed poorly, being outperformed by OFOS-EA, FuBOS-EA, and DiGFuBOS-EA when $\mu = 500, 1000$ and being outperformed by OFOS-EA and FuBOS-EA when $\mu = 100$.

Looking at the GRIEWANK problem, a pattern starts to emerge. The SAOS EAs perform poorly on this problem too when compared to OFOS-EA. While the performance of the SAOS EAs degrade as μ increases, $SAOS_2$-EA's performance degrades less than the performance of the other two SAOS EAs. FuBOS-EA was outperformed by OFOS-EA for all values of μ by a statistically significant margin. DiGFuBOS-EA was outperformed by OFOS-EA by a statistically significant margin when $\mu = 100, 500$; however, it was only outperformed by OFOS-EA by a statistically insignificant margin when $\mu = 1000$. Also, DiGFuBOS-EA was outperformed by not only FuBOS-EA, but the SAOS EAs when $\mu = 100$. ROS-EA was outperformed by all other EAs when $\mu = 100$; however, it outperformed the SAOS EAs, FuBOS-EA,

Table 5.2: Performance of EAs on the 3-SAT problem for different clause to variable ratios (standard deviation in parentheses, best results are bolded) with $\mu = 500$

Ratio	2	3	4	6
FuBOS-EA	**1000 (0)**	1498.43 (1.10101)	1988.63 (2.39421)	2941.5 (3.52846)
DiGFuBOS-EA	999.983 (0.128019)	1498.5 (0.991632)	1989.22 (2.26636)	2940.13 (3.6399)
OFOS-EA	**1000 (0)**	1498.4 (0.934523)	**1990.1 (2.19621)**	2941.62 (3.63818)
ROS-EA	**1000 (0)**	**1498.63 (0.948098)**	1989.98 (2.10152)	**2942.03 (3.08203)**
$SAOS_1$-EA	999.983 (0.128019)	1498.5 (0.991632)	1988.63 (2.50311)	2940.13 (3.6399)
$SAOS_2$-EA	**1000 (0)**	1498.6 (1.05198)	1989.55 (2.04471)	2941.77 (3.28819)
$SAOS_3$-EA	**1000 (0)**	1498.22 (1.03427)	1988.87 (2.41155)	2939.8 (4.13254)

and DiGFuBOS-EA when $\mu = 500$, and the SAOS EAs when $\mu = 1000$.

On the 3-SAT problem, the performances of the EAs were unexpectedly similar. When $\mu = 100$, OFOS-EA outperformed all other EAs, except DiGFuBOS-EA, by statistically significant margins. FuBOS-EA, ROS-EA, and DiGFuBOS-EA performed on par with each other, while the SAOS EAs performed the worst. When $\mu = 500$, each of the MBFs were close to one another, with FuBOS-EA being outperformed by all other EAs. FuBOS-EA performed worse when $\mu = 1000$ compared to the other EAs by a wider margin than when using smaller values of μ. While the other EAs performed similarly well, DiGFuBOS-EA actually outperformed all other EAs. When the clause to variable ratio varies, the results (shown in Table 5.2) are similar. When the ratio was 2, all EAs except for $SAOS_2$-EA and DiGFuBOS-EA solved every 3-SAT instance. When the ratio was 3, the performances of the EAs were quite even. Significant differences between the performances of the EAS were still not seen when the clause to variable ratio was 6.

FuBOS-EA and OFOS-EA were compared to the previous offspring sizing EAs, Jansen-EA and Hansen-EA; the results of these experiments are in Table 5.3. On the DTRAP problem, FuBOS-EA outperformed Hansen-EA and Jansen-EA by a statistically significant margin, while being outperformed by OFOS-EA. OFOS-EA outperformed all other EAs on the ONEMAX problem, while FuBOS-EA performed the worst. OFOS-EA outperformed all other EAs on the 3-SAT problem. In fact, when $\mu = 1$, OFOS-EA outperformed all other EAs on the 3-SAT problem for all values of μ. On the GRIEWANK problem, Hansen-EA outperformed the other EAs, while OFOS-EA performed better than Jansen-EA and FuBOS-EA.

Table 5.3: Performance of some offspring sizing EAs on various problems in terms of metrics specified in Section 4 with $\mu = 1$ (standard deviation in parentheses, best results are bolded)

Population Size	DTRAP	ONEMAX	3-SAT	GRIEWANK
FuBOS-EA	815.35 (7.03758)	98253.1 (6371.98)	1980.47 (4.39874)	-2565.73 (480.756)
OFOS-EA	**818.483 (7.07223)**	**39687.5 (9969.84)**	**1996.82 (1.52197)**	-1860.75 (776.034)
Hansen-EA	789.6 (5.35164)	42558.4 (7388.13)	1987.97 (2.76265)	**-203.854 (91.5486)**
Jansen-EA	810.067 (6.51119)	64282.8 (10290)	1944.52 (56.6764)	-2702.76 (695.39)

6. DISCUSSION

6.1. RANDOM OFFSPRING SIZING

The most surprising results of the experiments are the fact that ROS-EA worked very well on the DTRAP problem, outperforming all other EAs when $\mu = 1,500$, which supports the theory that using dynamic parameter values has the potential to the performance of EAs [7]. This is likely due to the range of values ROS-EA chooses λ from. Since it chooses λ from $[1, 1000]$ with uniform probability, the expected value of λ is approximately 500. As shown in Table 4.7, the values OFOS-EA used for λ when $\mu = 100, 500$ on the DTRAP problem were 675 and 525, respectively; however, the value OFOS-EA used for λ when $\mu = 1000$ was 75. The difference between the λ used by OFOS-EA and the expected value of λ for ROS-EA when $\mu = 1000$ is much greater than the difference beteween these two values when $\mu = 100, 500$. This is why ROS-EA performs poorly when $\mu = 1000$. Another reason might simply be luck. OFOS-EA used a fixed value for λ that might be suboptimal during parts of the EA's execution. Since ROS-EA uses a wide range of values for λ with each generation and completely ignores the state of the search when choosing a new value, ROS-EA might simply make a lucky guess. This "strategy" can yield great success or great failure, which would explain why ROS-EA has large standard deviations on the DTRAP problem.

ROS-EA also performed well on the 3-SAT problem. In fact, the difference in performance of all the EAs on 3-SAT was small. 3-SAT is deceptive like DTRAP, but in a different way. With DTRAP, the entire bitstring does not need to be changed to escape from a local optimum, only the bits of the "stuck" traps need to be changed. For 3-SAT, escaping from a local optimum can require changing every bit of the bitstring due to the depenencies between the variables caused by the 3-SAT instance being solved. So having enough offspring to perform a proper search of the search space does not have much impact on the search since getting caught in a local optimum can require huge changes to the genes of the individuals to escape. This would explain

why there was not much variation in the performance of the EAs on 3-SAT.

6.2. SELF-ADAPTIVE OFFSPRING CREATION

One of SAOS's major problems is the fact that λ is heavily dependent on μ. As the results in Section 5 show, the performance of the SAOS EAs, when compared to the other EAs, decreased as μ increased on the GRIEWANK, DTRAP, and ONEMAX problems. This can be attributed to a variety of reasons. Consider the initial population of an EA using SAOS. Every individual in the initial population has an $offWant$ value of 1. Since SAOS allows every individual to have as many living offspring as it wants, there are $\frac{\mu}{2}$ offspring created that generation. When the average $offWant$ of the population grows, the amount of offspring being produced grows at a factor of $\frac{\mu}{2}$.

An individual is penalized for how many living offspring it has, to prevent the average $offWant$ from growing to large, sub-optimal values, and thus preventing λ from growing too large. However, each score function either over-penalizes or under-penalizes for the amount of offspring had. Consider the DTRAP problem. The rate at which λ grew for $SAOS_1$-EA and $SAOS_3$-EA increased as μ increase. When the values for λ grew to be much larger than what was used for OFOS-EA, $SAOS_1$-EA and $SAOS_3$-EA performed poorly. These large values of λ caused $SAOS_1$-EA and $SAOS_3$-EA to explore more than necessary, causing these EAs to perform poorly due to lack of exploitation. For $SAOS_2$-EA, the problem was the opposite; λ did not grow large enough to properly explore the search space. This is why $SAOS_2$-EA performed better than the other SAOS EAs, but was outperformed by FuBOS-EA, OFOS-EA, and DiGFuBOS-EA. The performance of the SAOS EAs on the GRIEWANK and ONEMAX problems is evidence that the SAOS EAs might exhibit this behavior in general.

The problem is balancing the need to prevent λ from growing too large with the need to have offspring to explore the search space. This balance is heavily influenced by μ and the problem the EA is trying to solve. The introduction of constants to the score functions could possibly fix this issue, but there are some problems with this. Initial tests show that performance is not less sensitive to any constant introduced to a score function than it is to λ. This means that tuning this constant would not be

any easier to do than tuning λ. Also, it might be possible that adding a constant will not completely solve the balance issue: μ still places a lower bound on λ. To solve this problem, it might be necessary to control μ and λ simultaneously. This way, the effect μ has on λ can be taken into consideration when choosing μ. Another way to solve this problem would be to not allow individuals to have as many offspring as they want. An unsatisfied parent could possibly have no offspring during a generation; thus, an increase in μ would not necessarily result in an increase in λ.

6.3. FUTILITY-BASED OFFSPRING SIZING

FuBOS solves one of the main problems with SAOS: the strong connection between λ and μ. By making λ a choice made at the population level instead of at the individual level, μ has less of an influence on λ. This opens up the possibility of having a method for controlling λ that can scale well with increasing μ. FuBOS's goal is to maximize the amount of offspring created per generation and minimize wasted computational effort associated with creating too many offspring. This is to help prevent premature convergence by exploring the search space as much as possible. On a simple problem with no non-global optima, like ONEMAX, there is less emphasis on exploration and more emphasis on exploitation due to how difficult it is to prematurely converge. This would explain why OFOS-EA, Hansen-EA, and Jansen-EA reached the optimal solution faster than FuBOS-EA on the ONEMAX problem.

FuBOS performed well on the experiments for larger values of μ. One reason for this might be due to how FuBOS benefits from a diverse pool of parents and a parent selection operator that has low selection pressure. Under such conditions, the parents selected are more likely to differ, thus creating more diverse offspring. Since FuBOS attempts to maximize the number of offspring created per generation, there is likely to be a large pool of diverse offspring under these conditions. For the experiments in Section 5, a large population resulted in a large pool of potential parents. A larger, randomly seeded initial population has more diversity than a smaller, randomly seeded initial population.

Another reason might be due to how FuBOS decides when the mating process should stop. As previously mentioned, FuBOS terminates the mating process if the

change in average fitness of the offspring drops below a certain threshold. Therefore, FuBOS terminates prematurely whenever successive offspring are generated which happen to have similar fitness values, even though there may still be an abundance of search space left to explore. This is more likely to happen with a smaller population, because there will be a smaller pool of parents to choose from. Also, if the λ to μ ratio is high (which seemed to be beneficial for smaller populations), there will be more competition for a spot in the population simply due to the amount of individuals vying for a single spot.

Another problem with FuBOS is that it is a serial method; the offspring must be created and examined one at a time. This means that any EA using FuBOS cannot evaluate the fitness of the offspring in parallel. This also means that the order in which the offspring are created matters. There are situations where FuBOS can be fooled into terminating the creation of offspring; however, further analysis must be done in order to see how likely such situations occur during the actual execution of an EA. FuBOS's biggest problem lies in determining when the change in average fitness is negligible, which is equivalent to determining what value to use for ϵ. While the value used for ϵ worked well for the test problems utilized, this value might result in degraded performance on a wide range of different problem classes. The fitness function itself might also cause problems for FuBOS. A fitness function whose range is small (e.g., $(0, 1)$) or varies in magnitude could have a negative effect on the performance of an EA using FuBOS. This might explain the results of the experiments using the GRIEWANK function. However, it is important to note that FuBOS-EA outperforms OFOS-EA in terms of performance per fitness evaluations, considering how many fitness evaluations were required to tune OFOS-EA (see Table 4.6).

6.4. DIVERSITY-GUIDED FUTILITY-BASED OFFSPRING SIZING

DiGFuBOS attempts to rectify some of the problems present in FuBOS. The biggest problem with FuBOS was the choice of ϵ, DiGFuBOS automates this decision. Based on the results presented in Section 5, DiGFuBOS preforms quite well on a wide range of problems. Furthermore, the results of the experiments show that the performance of DiGFuBOS-EA does not degrade as μ increases; on the contrary, the performance often increased when compared to the other EAs. However, DiGFuBOS-

EA performed very poorly when compared to OFOS-EA and even FuBOS-EA on the ONEMAX and GRIEWANK problems when $\mu = 100$. DiGFuBOS looks at the genetic diversity of the population when setting ϵ and determining when to stop the mating process. Because of this, DiGFuBOS-EA tends to allow more offspring to be created when the diversity is low. While this is a good thing in general, this is a problem with the ONEMAX problem. The ONEMAX problem is a very simple problem; since there are no local optima that are suboptimal, there is no need to worry about premature convergence. Thus, there is no need to allow more offspring to be had as diversity decreases. Since the only way DiGFuBOS can measure a problem's difficulty is the length of the bitstring or real-valued vector, DiGFuBOS wastes fitness evaluations doing more exploration than necessary.

Using fitness values as a measure of diversity can be problematic because a one-to-one mapping from fitness values to genotypes does not necessarily exist. By examining the genes instead of the fitness values of individuals, DiGFuBOS is able to make a better informed decision about when to stop the mating process than FuBOS. This allows DiGFuBOS to outperform FuBOS on more than two-thirds of the experiments ran on the DTRAP, GRIEWANK, and 3-SAT problems. Unfortunately, generality is sacrificed for performance: an EA using DiGFuBOS is restricted to using bitstrings and real-valued vectors for representations of candidate solutions. Also, DiGFuBOS has a requirement of $\mu \geq 2$ since diversity is always zero when $\mu = 1$; DiGFuBOS has no real effect on λ in that situation. Even though DiGFuBOS is more restrictive than FuBOS, DiGFuBOS tunes λ without the addition of any user-set parameters. Furthermore, the performance of DiGFuBOS-EA is comparable in performance to OFOS-EA on a wide variety of test problems, and surpasses OFOS-EA in terms of performance per fitness evaluations.

7. CONCLUSIONS

EAs are robust methods for solving difficult problems; however, the number of parameters that must be tuned a priori and the expertise required to set these parameters hinders the adoption of EAs by non-experts. While methods for controlling the parameters of an EA have been proposed, little has been done towards controlling the offspring size of an EA. This thesis explored controlling λ, proposed three novel methods for controlling λ, and compared their performance on a diverse set of test problems.

The first method proposed, SAOS, allowed each individual to specify the number of offspring they would like to have and made each individual "support" their offspring though penalizing individuals based on how many living offspring they have. However, this method performed poorly on most test problems when compared to the highly tuned OFOS-EA. Also, the performance of an EA using SAOS often degrades as μ increases. This is due to SAOS failing to find a proper balance between having offspring to explore the search space and preventing λ from increasing to large, sub-optimal values. While the introduction of parameters could fix this issue, performance is not less sensitive to these parameters.

The second method proposed, FuBOS, solves some of the problems associated with SAOS. FuBOS determines when the computational effort required to create additional offspring outweighs how much these additional offspring contribute to the search. This is done by looking at the change in the average fitness of the offspring with each offspring created. FuBOS-EA was shown to outperform the few previously published offspring sizing control methods for many complex problem instances. Also, FuBOS-EA performed on par with the highly tuned OFOS-EA for many of the test problems using various population sizes, while being far more efficient in terms of fitness evaluations. Another advantage of FuBOS is that it alleviates the user from having to set λ (or even an initial value for λ), instead replacing it with a threshold value, ϵ, to which performance is far less sensitive.

The final method proposed, DiGFuBOS, extends the concept of FuBOS by dynamically changing the threshold value, ϵ, every generation based on the genetic

diversity of the population, the population size, and the problem size. Also, DiGFuBOS looks at the change in genetic diversity of the offspring instead of the change in average fitness. Unlike FuBOS, DiGFuBOS requires a representation-specific diversity metric which can be difficult to define for some representations. This thesis presents such diversity metrics for bitstring and real-valued vector representations. DiGFuBOS-EA was shown to outperform FuBOS-EA for many complex problem instances and perform on par with the highly tuned OFOS-EA for various population sizes on most test problems. One major advantage DiGFuBOS has over FuBOS is the automation of ϵ, making DiGFuBOS far easier to use than FuBOS.

The following studies can be important extensions of the work presented in this thesis:

- EAs are known to be "embarrassingly parallel" algorithms. Embarrassingly parallel algorithms are able to be broken down into a number of smaller, independent subtasks with minimum effort. Often with EAs, the fitness evaluation is the most time consuming portion of the algorithm. In the case of standard EAs, the fitness evaluations to be performed during any generation are independent from one another; thus, they can be performed in parallel. FuBOS and DiGFuBOS are serial methods: they look at each offspring created one at a time. Thus, any EA using FuBOS or DiGFuBOS cannot be parallelized. One way of fixing this is to look at the change in average fitness of the individuals (or genetic diversity in the case of DiGFuBOS) with each k individuals created. Another way of fixing this problem is to consider the offspring as a set of individuals and finding a confidence interval on the mean of the fitness values. When the width of that confidence interval grows small enough, stop producing offspring.

- The performance of SAOS-EA often degrades as μ increases due to the number of offspring being created per generation increasing as μ does. However, different parameter values are optimal at different stages of an EA's execution [7]. It is possible that μ will need to change during the execution of an EA using SAOS in order to keep λ from increasing to large, sub-optimal values. FuBOS and DiGFuBOS might also benefit from μ being controlled. Combining two different

parameter control methods can improve the performance of an EA beyond what each method can do alone [15].

- The main idea behind FuBOS and DiGFuBOS, producing offspring until it becomes futile to produce any more, might be able to be applied to other EA strategy parameters. For example, consider the population size of an EA. One way of applying this idea to μ would be starting EA with $\mu = 1$. After k generations, increase the value of μ by 1. k generations later, compare the rate of change of the average fitness of the population when $\mu = 1$ with the rate of change of the average fitness of the population when $\mu = 2$. If the difference between those rates is large, then increment μ and repeat the process. However, if the difference between those rates is small, then stop incrementing μ.

- DiGFuBOS currently works for bitstring and real-valued vector representations. It would be beneficial to generalize DiGFuBOS to allow it to be applicable to EAs using different representations or simply extending DiGFuBOS to work with more representations. Generalizing DiGFuBOS is a non-trivial task: one undertaking this task would have to consider the different representations that exist (and can possibly exist in the future) and the different diversity metrics each representation may have. It is likely that extending DiGFuBOS to different representations is a more feasible task.

APPENDIX

Lemma 1. *$offdiversity_N = 0$ if, and only if, all offspring have identical genes.*

Proof. Clearly,

$$offdiversity_N = 0 \Leftrightarrow \sum_{i=1}^{L}\sum_{j=1}^{N}(x_{ij} - c_i)^2 = 0 \Leftrightarrow (x_{ij} - c_i)^2 = 0$$

for all offspring j where x_{ij} is that offspring's i-th gene. Thus,

$$(x_{ij} - c_i)^2 = 0 \Leftrightarrow x_{ij} - c_i = 0 \Leftrightarrow x_{ij} = c_i.$$

Since c_i is the average of gene i over the offspring, this means all offspring j have the same i-th gene, which is equivalent to all offspring having identical genes. □

Theorem 1. *If $\epsilon = 0$ and there exist two offspring whose genes differ, then Inequality 13 will always hold.*

Proof. Let x_a and x_b be two offspring who differ in their k-th gene. So $offdiversity_N \neq 0$ by Lemma 1. Thus,

$$\frac{offdiversity_{N-1}}{offdiversity_N} < 1 - \epsilon = 1. \tag{.16}$$

Clearly, $offdiversity_N > 0$, so simplifying yields

$$\frac{offdiversity_{N-1}}{offdiversity_N} < 1 \tag{.17}$$

$$offdiversity_{N-1} < offdiversity_N \tag{.18}$$

$$offdiversity_N - offdiversity_{N-1} > 0 \tag{.19}$$

$$\sum_{i=1}^{L}\sum_{j=1}^{N}(x_{ij} - c_i)^2 - \sum_{i=1}^{L}\sum_{j=1}^{N-1}(x_{ij} - c_i)^2 > 0 \tag{.20}$$

$$\sum_{i=1}^{L}(\sum_{j=1}^{N}(x_{ij} - c_i)^2 - \sum_{j=1}^{N-1}(x_{ij} - c_i)^2) > 0 \tag{.21}$$

$$\sum_{i=1}^{L}(x_{iN} - c_i)^2 > 0 \tag{.22}$$

Since the other offspring cannot be the same as x_a and x_b due to $x_a \neq x_b$, every other offspring differs in their k-th gene from either x_a or x_b. Since c_i is the average of the i-th gene over all offspring, and all offspring differ from at least one other offspring in the k-th gene, then

$$x_{kN} - c_k \neq 0 \tag{.23}$$

$$(x_{kN} - c_k)^2 > 0. \tag{.24}$$

So this means that Inequality .22 will always hold, thus Inequality 13 will always hold. □

BIBLIOGRAPHY

[1] D. H. Ackley. *A Connectionist Machine for Genetic Hillclimbing*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[2] J. Arabas, Z. Michalewicz, and J. Mulawka. GAVaPS-a genetic algorithm with varying population size. In *Proceedings of the First IEEE Conference on Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence*, volume 1, pages 73–78, June 1994.

[3] S. A. Cook. The Complexity of Theorem-Proving Procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[4] L. D. Davis and M. Mitchell. Handbook of genetic algorithms. *Van Nostrand Reinhold*, 1991.

[5] K. De Jong. Parameter Setting in EAs: a 30 Year Perspective. In F. G. Lobo, C. F. Lima, and Z. Michalewicz, editors, *Parameter Setting in Evolutionary Algorithms*, pages 1–18. Springer-Verlag, 2007.

[6] K. Deb and D. E. Goldberg. Analyzing Deception in Trap Functions. In *Proceedings of the Second Workshop on Foundations of Genetic Algorithms*, pages 93–108, July 1992.

[7] A. E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter Control in Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999.

[8] A. E. Eiben, B. Jansen, Z. Michalewicz, and B. Paechter. Solving CSPs using self-adaptive constraint weights: how to prevent EAs from cheating. In *Proceedings of GECCO 2000 - the Genetic and Evolutionary Computation Conference*, pages 128–134, July 2000.

[9] A. E. Eiben, M. C. Schut, and A. R. de Wilde. Is self-adaptation of selection pressure and population size possible? - a case study. In T. P. Runarsson, H.-G.

Beyer, E. K. Burke, J. J. M. Guervós, L. D. Whitley, and X. Yao, editors, *PPSN*, volume 4193 of *Lecture Notes in Computer Science*, pages 900–909. Springer, 2006.

[10] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[11] J. Gottlieb, E. Marchiori, and C. Rossi. Evolutionary Algorithms for the Satisfiability Problem. *Evolutionary Computation*, 10(1):35–50, 2002.

[12] A. O. Griewank. Generalized Descent for Global Optimization. *Journal of Optimization Theory and Applications*, 34(1):11–39, 1981.

[13] N. Hansen, A. Gawelczyk, and A. Ostermeier. Sizing the Population with Respect to the Local Progress in (1,λ)-Evolution Strategies-A Theoretical Analysis. In *Proceedings of CEC 1995 - The IEEE International Conference on Evolutionary Computation*, volume 1, pages 80–85, 1995.

[14] N. Hansen and A. Ostermeier. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[15] E. A. Holdener. *The Art of Parameterless Evolutionary Algorithms*. PhD thesis, Missouri University of Science and Technology, 2008.

[16] E. A. Holdener and D. R. Tauritz. Learning offspring optimizing mate selection. In *Proceedings of GECCO 2008 - the Genetic and Evolutionary Computation Conference*, pages 1109–1110, New York, NY, USA, 2008. ACM.

[17] C. Igel, N. Hansen, and S. Roth. Covariance Matrix Adaptation for Multi-objective Optimization. *Evolutionary Computation*, 15(1):1–28, 2007.

[18] T. Jansen, K. A. D. Jong, and I. A. Wegener. On the Choice of the Offspring Population Size in Evolutionary Algorithms. *Evolutionary Computation*, 13(4):413–440, 2005.

[19] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[20] M. Locatelli. A Note on the Griewank Test Function. *Journal of Global Optimization*, 25(2):169–174, 2003.

[21] Z. Michalewicz and M. Schoenauer. Evolutionary Algorithms for Constrained Parameter Optimization Problems. *Evolutionary Computation*, 4(1):1–32, 1996.

[22] R. W. Morrison and K. A. D. Jong. Measurement of Population Diversity. In *Selected Papers from the 5th European Conference on Artificial Evolution*, pages 31–41, London, UK, 2002. Springer-Verlag.

[23] I. Rechenberg. *Evolutionsstrategie: optimierung technischer systeme nach prinzipien der biologischen evolution.* Frommann-Holzboog, 1973.

[24] G. Rudolph. Self-Adaptation and Global Convergence: A Counter-Example. In P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, editors, *Proceedings of the Congress on Evolutionary Computation*, volume 1, pages 646–651, Mayflower Hotel, Washington D.C., USA, July 1999. IEEE Press.

[25] H.-P. Schwefel. *Numerical Optimization of Computer Models.* John Wiley & Sons, Inc., New York, NY, USA, 1981.

[26] E. A. Smorodkina and D. R. Tauritz. Greedy Population Sizing for Evolutionary Algorithms. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 2181–2187, Sept. 2007.

[27] E. A. Smorodkina and D. R. Tauritz. Toward Automating EA Configuration: the Parent Selection Stage. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 63–70, Sept. 2007.

[28] R. Ursem. Diversity-Guided Evolutionary Algorithms. *Parallel Problem Solving from Nature – PPSN VII*, pages 462–471, 2002.

VITA

André Chidi Nwamba was born on November 21, 1984 in Huntsville, Alabama. He graduated from Hazelwood Central High School in May 2007 and enrolled as an undergraduate at University of Missouri-Rolla (now Missouri University of Science and Technology) in the fall of that year. He received a BS in both Computer Science and Applied Math in May 2007, and enrolled in the Computer Science graduate program later in the fall of that year. He received his master's degree in August 2009.