CO-EVOLUTIONARY AUTOMATICALLY DEFINED FUNCTIONS IN GENETIC PROGRAMMING

by

Anthony Lukas

A thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfillment of

the requirements for the degree of

MASTER OF COMPUTER SCIENCE

School of Computer Science

at

CARLETON UNIVERSITY

Ottawa, Ontario

August, 2008

© Copyright by Anthony Lukas, 2008

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-44109-1
Our file Notre référence
ISBN: 978-0-494-44109-1

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

We show how the addition of co-evolution to genetic programming (GP) overcomes the current limitations of GP as well as GP augmented with automatically defined functions (GP+ADF) with a method called co-evolutionary automatically defined functions (GP+CADF). We demonstrate that GP+CADF requires a lower computational effort to solve the parity, sum of bits, image recognition, lawn coverage and the bumblebee problems. Furthermore, we find that GP+CADF consistently finds solutions in the fewest generations. To further improve GP+CADF, we discover that using elitism and a single best evaluation coupling lowers the computational effort even further. We also discover ways to improve the initial population and initial best individuals used for evaluation, and perform an analysis of the effect of mutation. The solutions discovered by GP+CADF are also analyzed to demonstrate that GP+CADF is able to find reusable subroutines and break problems into smaller sub-problems which are then assembled into the final solution.

Acknowledgements

I would like to thank Professor Oppacher for getting me interested and involved in genetic programming as well as providing invaluable advice, support and feedback in this thesis. I would also take this opportunity to thank my family for supporting me in my further studies and Eve St-Laurent for helping me take ideas in new directions.

Table of Contents

List of Tables

Chapter 4: The Even-n-Parity Problem

Chapter 5: N-S-Sum of Bits Problem

Chapter 6: The Image Recognition Problem

Chapter 7: Lawn Coverage Problems

Chapter 8: Bumblebee Problem

List of Figures

Chapter 3: Co-evolutionary Automatically Defined Functions

Chapter 4: The Even-n-Parity Problem

Chapter 5: N-S-Sum of Bits Problem

Chapter 6: The Image Recognition Problem

Chapter 7: Lawn Coverage Problems

Chapter 8: Bumblebee Problem

Chapter 1: Introduction

1.1 Statement of the Problem

Drawing inspiration from the concept of evolution has proven to be successful in solving challenging problems with genetic programming (GP). A great deal of work has been done to maximize the potential of genetic programming by augmenting it with various new ideas and techniques. The addition of functions or subroutines to GP is one such idea that allows a program to reuse modular functionality. It has been shown that when genetic programming is augmented with automatically defined functions (GP+ADF) we can solve problems beyond the capability of standard GP. This is due to the fact that ADFs exploit underlying regularities and repetitions in the solutions of a problem. However, even with the addition of ADFs some problems eventually become too hard to solve. The problem that we are faced with is that we can not solve increasingly difficult problems with standard GP or even GP augmented with subroutines or ADFs. There is a certain threshold of difficulty where the GP algorithms require too much time and computation to yield an answer in any reasonable amount of time. In other words, the problem is how to keep discovering solutions where GP+ADF is unable to do so. This work will explore how we can answer this challenge with the introduction of co-evolution to genetic programming where we devise multiple populations to work together toward solving a problem.

1.2 Contributions

The contributions of this work will be summarized in this section and discussed in detail in the following chapters. This work's main contribution is the addition of co-

operative co-evolution in genetic programming with a method called co-evolutionary automatically defined functions in genetic programming, abbreviated as GP+CADF. The value of the contribution is that GP+CADF allows us to continue solving difficult problems where GP+ADF is unable to do so. In other words, we discover that for a number of problems GP+CADF requires less computational effort to discover a solution than standard GP or GP+ADF. The technique is implemented in a popular framework for evolutionary computing called evolutionary computing in java or ECJ. A comparative analysis of standard genetic programming (regular GP), genetic programming augmented with automatically defined functions (GP+ADF) and GP+CADF is performed on a wide variety of different problems that include up to 14-even-parity, sum of bits, image recognition, lawn coverage and the bumblebee problems. In addition, methods to improve GP+CADF are discovered which include improvements of initial populations, improvements in best individuals used for evaluation, effect of elitism, effect of multiple best individuals and the effect of mutation. One of the problems studied, sum of bits, was specifically invented to test GP+CADF with a problem harder than Boolean even-n-parity which was claimed to be the hardest Boolean test problem. In addition, the effect of different number of populations and architectures is studied to determine how to best utilize GP+CADF. Lastly, the actual solutions obtained with GP+CADF are analyzed in order to show that GP+CADF discovers reusable subroutines as ADFs and uses them to exploit regularities and repetitions in the program solutions. Similarly, we show that GP+CADF uses the ADFs to break problems into smaller sub-problems and then assembles those parts into the final solution.

1.3 Overview of Results

Performance comparisons are made using a standard method introduced by Koza [1] called computational effort. With this metric, we have been able to show that GP+CADF goes beyond the limitations of regular GP and GP+ADF by requiring a lower computational effort in all of the problems studied. Computational effort tells us how many fitness evaluations we need to perform in order to guarantee that a solution will be found with 99% probability. This means that GP+CADF will require fewer evaluations than other methods in order to guarantee a solution with 99% probability. Consequently, GP+CADF was the only method to yield a solution in some cases. On the parity problem, we have been able to obtain solutions for up to 14-even-parity. By comparison, GP+ADF stopped at 10-even-parity and regular GP stopped at 6-even-parity. Similarly, on the sum of bits problem, GP+CADF was the only method that obtained solutions for the highest difficulty problem of 6-3 sum of bits. Further advantages of GP+CADF, although not as large as in parity and sum of bits, were discovered in the image recognition, lawn coverage and bumblebee problems.

Furthermore, we have shown experimentally that several improvements considerably lower the computational effort required for GP+CADF. These include using elitism, improving the initial population and the initial best individuals. We have also determined that GP+CADF is further improved by sub-tree and rehang mutation operators. Lastly, it is worth mentioning that we have invented a Boolean test function termed 'sum of bits' that was even harder to solve for GP than even-n-parity as our experimental results show.

1.4 Organization of Thesis

This thesis will be organized as follows. Chapter 2 will review the existing work in co-evolutionary GP which includes two main fields; competitive and co-operative co-evolution. We will see where co-evolution currently stands and where our method lies in that spectrum. Chapter 3 will review how GP and ADFs work and some of their shortcomings. Next, GP+CADF will be explained in detail as a co-evolutionary approach to using ADFs in GP and several improvements will be proposed. Chapter 4 is a detailed analysis and comparison of the performance of regular GP, GP+ADF and GP+CADF on the Boolean even-n-parity problem. Improvements for GP+CADF and the effects of mutation will also be presented. Chapters 5 through 8 will analyze results on additional problems that include sum of bits, image recognition, lawn coverage and the bumblebee problem. The conclusion will include a summary of results and proposed future work in adding competitive co-evolution to GP+CADF.

Chapter 2: Related Work in Co-Evolution

2.1 Introduction

Genetic programming was first introduced by Koza in [7] where it was shown that we can evolve a population of program trees to solve a wide range of problems. Since then, the scope of GP has expanded considerably to include many new ideas such as different representations of individuals and new evolving strategies. One of the strategies that was used previously in GAs and introduced to GP was co-evolution. Stated by Oppacher and Morrison in [12] as a broad definition, co-evolution is a strategy where the fitness of an individual depends not only on how well it solves the problem but also on other individuals.

This chapter will discuss work that falls into two categories of co-evolutionary systems. The first of these is based on combining individuals such that they compete against each other and is termed competitive co-evolution. The second combines individuals such that they all work together toward forming a solution and is called co-operative co-evolution. The goal of this chapter will be to survey the existing work in these categories of co-evolution. This will give us a good idea of where co-evolution currently stands in evolutionary computing and will allow us to precisely locate our own approach and contribution to co-evolutionary genetic programming with our GP+CADF method in co-operative co-evolution. The following section will briefly review genetic algorithms, genetic programming and computational effort.

2.2 Review of Genetic Algorithms, Genetic Programming and Computational Effort

Drawing inspiration from the concept of evolution has proven to be successful in solving challenging problems with genetic algorithms (GA). The basic premise of a genetic algorithm is that in order to solve a problem, we work with a population of potential solutions. Each solution can be evaluated on how well it solves a problem which is referred to as that individual's fitness. The goal is to find a solution with the highest fitness that solves the problem.

Initially, all the individuals are initialized randomly and evaluated which is the first generation for our population. To drive the evolution forward, we apply genetic operators such as crossover and mutation. Crossover means that we select two individuals with a probability proportional to their fitness and exchange parts between them to make two offspring. Similarly, mutation involves taking a single individual and changing some part of it. The key is that fitness proportional selection leans toward choosing good individuals but does not exclude even the worst individuals. We produce enough offspring to generate the next generation and continue this process until a solution is found or a certain number of generations is reached. Essentially, the GA algorithm searches through the space of possible solutions by combining and mutating individuals. Genetic programming (GP) is a continuation of GAs where individuals are programs that attempt to solve a problem or perform a task. A program can be simply represented as a program tree as discussed by Koza in [1]. In GP, crossover involves exchanging sub-trees between programs and mutation involves modifying a sub-tree in some way.

A metric used in this thesis to evaluate a GP algorithm and introduced by Koza is computational effort [1]. Computational effort tells us how many individuals we need to evaluate until we are 99% certain that a solution will be discovered. How can we guarantee that a solution will be found with any probability? We rely on the fact that whenever we run GP for a fixed number of generations G there is a constant probability of success for a specific problem. For example, we might find that GP gives us a solution 3 times in 100 runs and this will always be consistent. This would mean a 3% probability of success. What happens when we keep repeating runs of something that has a 3% probability of success? It is a stochastic process, and the probability of failure which is originally 97% will keep dropping. We can calculate k which is how many times we need to run the GP algorithm so the probability of failure is 1% where probability of success p(success) is the problem specific success probability of GP in 1 run:

$$k = \log 0.01 / \log (1 - p(\text{success}))$$

Then, the computational effort E which is defined as the number of individuals we must evaluate to guarantee a solution with 99% probability is simply:

$$E = M * G * k$$

where M is the population size and G is the number of generations. The minimum computational effort is simply the minimum computational effort among all the different generations from 1 to G. Recall that the cumulative success probability will usually be zero at generation 1 and rise toward the maximum at generation G. For example, we may find that the success probability is 2.9% at generation 10 and 3% at generation 60. We might find a lower computational effort at generation 10 even though the success probability is lower than at generation 60. In this thesis, performance graphs will be given

that show the success probability and computational effort for every generation as well as the minimum computational effort found.

2.3 Competitive Co-Evolution

2.3.1 Host Parasite Systems

The first competitive co-evolutionary system was introduced by Hillis in [13] which he coined host-parasite co-evolution to apply to the problem of generating minimal sorting networks. Basically the system consists of two populations: the candidate solution population (hosts) which are the sorting networks and groups of test cases (parasites) which are the sorting problems. The fitness of individuals in the two different populations is mutually determined by pairing a single host and parasite. The fitness of a host depends on how many test cases it successfully solved in a parasite, and the fitness of the parasite is how many test cases were failed by the host sorting network. It is expected that co-evolution would be likened to an arms-race in the competitive case [19].

The benefit reported by Hillis is that the evolutionary process converged less often on a local optimum since a large imperfect subpopulation of solutions would attract parasites that are able to exploit its weakness by providing tests that make it fail. In addition, the search for test cases focuses on those that exploit weaknesses and therefore it is not necessary to test an individual against all test cases but only those that were determined to be hard. Hillis reported that co-evolutionary runs produced consistently better results than those without and with less wasted computational effort.

Williams and Mitchell [14] further improve on the host-parasite paradigm with spatial co-evolution. In spatial co-evolution the hosts and parasites are embedded on a grid with one pair at each cell. The competition that takes place is only with other

individuals in the surrounding neighbourhood. At every generation, each individual is replaced by the fittest individual in a neighbourhood of cells. It is reported that this strategy promotes continued diversity in the population and produces parasites that target specific weaknesses in hosts [14].

2.3.2 Challenges of Competitive Co-Evolution

After the general idea of co-evolution was determined to be feasible, much work has been done to overcome obstacles encountered during competitive co-evolution. Bongard and Lipson [16] describe the problem of disengagement. Disengagement occurs when one of the populations comes to completely dominate another population. For example, the parasite population can be made up of individuals that are all hard for the host population. In this case the fitness gradient is lost in the parasite population, or in other words, the parasite population has lost variety of fitnesses. This brings an end to reciprocal evolution and causes premature convergence.

Another problem that was identified was cycling or intransitivity described by Jong and Pollack [14] and Nolfu and Floreano [18]. It may happen that a population adopts short term strategies that end up forming a loop. For example the host population may determine strategy H_1 in response to parasite strategy P_1. Next, parasites develop strategy P_2 that beats H_1 and hosts develop H_2 that beats P_2. Next, parasites discover that P_1 beats H_2 so hosts again adopt H_1. This process may then cycle indefinitely.

An additional problem discussed by Jon and Pollack [15] is that a complete set of tests may be infeasible to use for evaluation. Some subset of tests must be determined and this may present a problem in itself to yield accurate evaluation with a reduced set.

2.3.3 Reducing Disengagement in Competitive Co-Evolution

In [19] Kim describes an attempt to reduce disengagement with a method called entry fee exchange tournament. Broadly stated, a fitness evaluation involves competing individuals with an entry fee that is proportional to fitness. When an individual wins a competition it gains the loser's entry fee. The result is that an individual receives a larger reward for winning against a strong opponent. It is reported that this method reduces disengagement and keeps population diversity in the sorting network and Nim game problems. In [16] Bongard and Lipson describe a way of alleviating disengagement called managed challenge. The basic idea is to use a test bank. When disengagement occurs because of hard tests that no host can pass, then this test is stored in the bank. When the host population re-engages this test is re-introduced. Similarly, tests that become too easy are withdrawn. It is reported that this method reduces disengagement and the number of tests required.

The work of Lohn and Kraus in [20] is a GA co-evolutionary approach that attempts to manage test difficulty and disengagement using target objective vectors or TOVs which are vectors containing targets for objectives that are to be optimized in a multi-objective optimization problem. The population of TOVs evolves from easy to hard based on how well the host population can solve these tests. Different TOVs lead to different rewards for the hosts that can solve them. For example, the highest reward is for a TOV that only one individual can solve, and this TOV also has the highest fitness. Easy TOVs that most individuals can solve, or impossible ones that no host can solve are assigned low fitness. Experimental results indicate that this co-evolutionary GA outperforms other evolutionary algorithms.

2.3.4 Reducing Required Number of Tests

The work of Pollack and Jong [15] focuses on that idea that pairing a host to all possible parasites is infeasible and it is possible to approximate a Complete Evaluation Set (CES) that is of manageable size and considers all underlying objectives. The CES is a set of tests that makes all distinctions that can be made between hosts. A distinction implies that it can be determined whether one host wins over another. The CES is approximated by taking into account the fact that there are at most H^2 distinctions where H is the number of hosts. Therefore it is sufficient to choose parasites to cover all distinctions. With the CES, we can precisely determine if one individual dominates another [15] and results of experiments with the Numbers Game report that accurate evaluation is possible with CES. Jong continues this work in [21] with the MaxSolve algorithm that uses an archive to guarantee monotonic success. This kind of archive means that with every change to the archive we are closer to the solution concept. Ficici in [17] also presents work that selects tests that best distinguish between hosts to reduce the total number of tests required for accurate evaluation.

2.3.5 Further Applications of Competitive Co-Evolution

Nolfi and Floreano [18] explore an application of competitive co-evolution as applied to evolutionary robotics. They investigated whether evolutionary arms races arise in evolving predator and prey robots. Real Khepera robots with a variety of sensors were used in their experiments. Based on results they concluded that co-evolution has a higher adaptive power than simple evolution. In [22] Khan, Miller and Halliday use competitive co-evolution to accomplish learning in two intelligent agents controlled by a computational network that is based on real neurons and models soma, dendrites, axon

branches and synaptic connections. The authors report that complex dynamics arise from the co-evolutionary competition between the two agents and give rising to learning. In [23] Kotani and Kato use competitive co-evolution for the task of feature extraction. Co-evolution is used in fitness evaluation by specifying that the fitness of an individual is how many pieces of data it can recognize that none of the other individuals can. One consequence is that as a highly fit individual multiplies in the population, its fitness drops. As a result, diversity of the populations is preserved and recognition accuracy is improved. This approach contrasts the other approaches discussed so far that had separate host and test population. Here is an example where we perform competitive co-evolution on a single population.

2.4 Co-operative Co-Evolution

So far we have seen co-evolution where different populations compete against each other like hosts and parasites to mutually improve each other. A different form of co-evolution is where multiple populations co-operate toward solving a problem in a symbiotic relationship. The GP+CADF method presented in this work is one such example of co-operative co-evolution. The fundamental approach in most cases is to have multiple populations that evolve independently but to have individuals from each population combine with the necessary individuals from the other population in order to form a complete solution. The two main challenges are how to decompose a problem such that we may place components of it in each population, and how to choose and combine individuals in order to evaluate them.

2.4.1 Problem Decomposition Methods and Individual Selection

One of the first works in co-operative evolution was presented by Potter and Jong in [24] where they applied co-evolutionary GA to function optimization. The problem consisted of finding the values of each variable in a function in order to minimize it. A population was initialized for each function variable and individuals were evaluated by combining them with the current best individual from all the other populations. The complete set of variable values is then plugged into the target function and the resulting fitness is assigned to the individual. It was reported that this version of co-operative co-evolution outperforms standard GA on a number of target functions.

While this work is a good example of co-operative co-evolution, the challenge of problem decomposition was problem specific. They simply analyzed the problem domain and concluded that variable values in a function optimization problem should constitute separate populations. The question remains, how do we decompose a general problem and even more importantly, how do we decompose an unknown target problem of which we know nothing of in advance. GP+CADF will attempt to answer this challenge with evolving ADF subroutines as parts of the solution.

Secondly, this work gives one approach to combining individuals to form a complete solution that can be evaluated, which is to combine each individual in a population with the current best individual from the other populations. This kind of strategy is reasonable in absence of any other choosing criteria and is a popular choice among other works including GP+CADF. For example, in [26] Cai and Peng apply co-evolution to path planning for multiple robots. Each robot path is encoded as a separate population and to evaluate one path it is combined with the best paths from the other

populations.

Potter and Jong continue their work in [25] where an attempt is made to let problem decomposition be a part of the evolutionary process. They let the appropriate number of interdependent subcomponents that cover specific tasks emerge rather than pre-specifying them. For example, in the binary string covering problem they let populations evolve to cover subsets. In the neural network problem, populations are allowed to control weights of a layer. Layers as well as populations are added and removed dynamically. However, it can still be argued that this work still does not entirely let subcomponents emerge by evolution. The problem domain knowledge is still utilized in determining the structure of the populations. GP+CADF will move toward utilizing subroutines that are not manually distributed beforehand, such as distributing populations among layers in Potter and Jong's neural network problem.

2.4.2 Further Work in Co-Operative Co-Evolution

In [28] Vanneschi and Mauri describe a version of heterogeneous co-operative co-evolution that combines GA and GP. This is done by allowing GP to evolve the main program tree while the GA populations evolve good values for the numerical terminals. The results were encouraging on a number of problems such as symbolic regression. In [27] Bongard explores a variant of co-evolution in GP with populations that work on similar, yet different problems. It was found that mutual exchange between such populations promotes pulling a population from its current local optimum [27]. Work has also been done to analyze the dynamics of co-evolution. Popovici and Jong [29] examined what effect the frequency of interactions between populations has on co-evolution. They concluded that this is problem specific and some problems will require

more interaction than others and characterized it by a property called best–response. In [10] Aler describes a method similar in high level design to our GP+CADF method. In the next chapter we will contrast the differences of our approaches after the details of our method are introduced.

2.5 Combined Competitive and Co-Operative Co-Evolution

Some works combine the ideas of co-operation and competition such as the work of Tulai and Oppacher in [30] to evolve cascade neural networks. Multiple GA populations with different mutation and crossover rates compete against each other until there is one winning species. This is done by absorbing individuals from underperforming populations into more successful populations and gradually phasing out unfit populations until one is left. In the co-operative phase the winning species is co-evolved with the best representatives of previous winning species. It is reported that cascade neural networks for the two spiral problem are obtained where all of the network characteristics and breeding operator rates are evolved and the results outperform those of other GA based attempts. In [31] Oppacher and Tulai extend this work by including the mutation and crossover rates in the actual genome. Tan and Lau perform a comparative analysis of competitive and co-operative co-evolution in [32] as applied to multi-objective optimization and determine that co-operative co-evolution is the better choice for a class of scalable tri-objective problems.

2.6 Conclusion

We have surveyed the literature in both competitive and co-operative co-evolution for both GA and GP approaches. Issues that arise in competitive co-evolution such as

disengagement and determination of the appropriate test set were dealt with in a number of works. Our GP+CADF method stands firmly in the co-operative realm of co-evolution. The main challenges of problem decomposition and methods of individual interaction are explored in existing works. The approach we take to decomposing problems into multiple co-operating populations with subroutines in GP is the main topic of the next chapter.

Chapter 3: Co-evolutionary Automatically Defined Functions

3.1 Introduction

The main idea of genetic programming with co-evolutionary automatically defined functions (GP+CADF) is that we have multiple and separate populations of individuals that co-operate in evolution by combining the best individuals of each population. Stated very broadly, it is a multi population system where each population evolves separately but uses the best members from other populations in order to evaluate and improve its own members. The following sections will review genetic programming, automatically defined functions and what the need and motivation would be for such a multi population system as an improvement to genetic programming with automatically defined functions. Following this, the GP+CADF algorithm will be introduced and we will describe in detail how fitness evaluation is performed in this multi-population system. Lastly, several techniques will be proposed as potential improvements for GP+CADF. The chapter will finish with a review of earlier work by Aler in co-evolutionary GP.

3.2 Review of Automatically Defined Functions

Genetic programming (GP) with automatically defined functions (GP+ADF) described by Koza in [1] is an attempt to improve GP by recognizing that many problems can be more readily solved by decomposing them into sub problems and assembling the sub problems into the final solution. In other words, GP+ADF makes use of subroutines in solving a problem.

For example, consider the following equation

$y = \sin(x^2 + x) + \cos(x^2 + x) + x^2 + x$

Another way that this equation can be represented in a more compact way is by recognizing that one part of the equation is repeated 3 times and can be represented separately as follows:

$$y = \sin (f(x)) + \cos (f(x)) + f(x)$$

where we simply let $f(x) = x^2 + x$.

We see that the more times f(x) is repeated in the equation and the more complex f(x) is, the more complex the final equation.

When working with GP, all individuals are represented as program trees. For this example the program tree of the original equation would be the following:

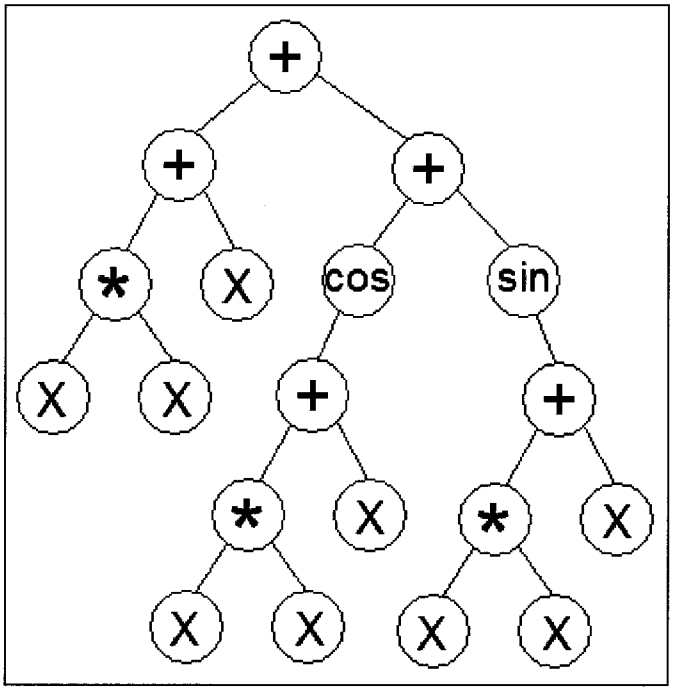

Figure 1 - Program tree for the equation sin $(x^2 + x)$ + cos $(x^2 + x)$ + x^2 + x

To simplify things we will define the function set here to be the mathematical operators +, *, sin and cos and the terminal x.

An automatically defined function (ADF) such as f(x) would be just another program tree

as represented in Figure 2. We see that in order to make use of ADFs we need to separate the traditional program tree representation into separate trees or branches one of which is called the main body and the other of which is called the ADF. Koza refers to these as the result producing branch and function defining branch respectively [1].

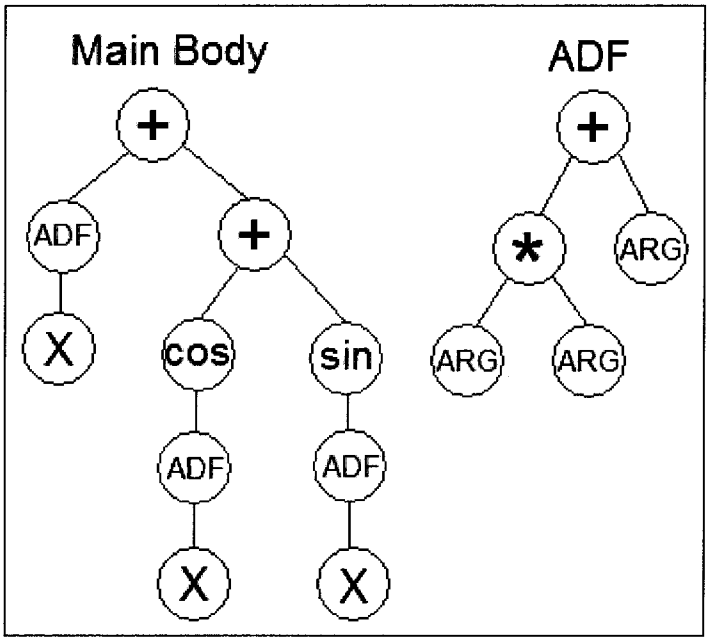

Figure 2 - Program tree for equation sin $(x^2 + x)$ + cos $(x^2 + x)$ + x^2 + x where x^2 + x is represented as a one argument ADF.

Furthermore, we can invoke the ADF just like a function call simply by including it in the function set of the main body. In the setup presented in Figure 2, the main body calls the ADF 3 times and passes one argument to it which is x. The ADF is a function that takes one argument and returns the sum of that argument and the square of the argument. It is clear to see that once the ADF function in Figure 2 is plugged into the main body it is equivalent to the program tree represented in Figure 1.

The important fact here is that the main body and the ADF are part of the same individual and each individual in the population has its own main body and ADF.

Therefore, in a population of M individuals we have M separate and distinct main bodies and ADFs. Once it comes time to evaluate the individual the ADF is simply plugged into wherever the main body calls it. In addition, depending on the architecture that we choose, we can have as many ADFs as we see fit and each one is represented as a separate tree.

3.3 Benefits of ADFs

Automatically Defined Functions allow GP to make use of reusable subroutines and this has been proven by Koza to reduce the computational effort when the problem complexity becomes sufficient to warrant the use of subroutines [1]. Koza refers to this condition as the breakeven point for computational effort where the use of ADFs starts to be beneficial. Considering the program tree shown in Figure 1, we can begin to understand how subroutines can aid in finding a solution. That simple example shows that the regular GP approach would need to discover the same piece of functionality three times in different parts of the tree. Conversely, GP+ADF would need to discover an ADF that performs some subroutine only once and call it three times from the main body. It is not immediately clear from our simple example if discovering

$$\sin (f(x)) + \cos (f(x)) + f(x) \text{ and } f(x)=x^2+x$$

would be easier to discover than

$$\sin (x^2 + x) + \cos (x^2 + x) + x^2 + x.$$

However, an analysis by Koza of the Boolean Parity, Lawnmower, Bumblebee, Impulse Response Function, Artificial Ant, Obstacle-Avoiding Robot, Minesweeper, Letter Recognition and Transmembrane Domain Prediction in Proteins [1] among other problems has demonstrated that it is indeed easier to discover a subroutine and reuse it

many times within a main body, than it is to discover one complete tree as in the traditional GP approach.

3.4 Problems with ADFs

So far we have seen that there are benefits to creating reusable subroutines with GP+ADF. Nevertheless, there may be some problems that arise when subroutines are utilized in genetic programming.

3.4.1 Poor Main Body and Good ADF

It may happen that a useful ADF has been discovered through evolution, but the main body calls it in the wrong place. Referring back to our example, suppose x^2+x has just been discovered as an ADF which is clearly very useful in finding the solution. If the individual to which this ADF belongs has a main body that calls this ADF in the wrong place, then the fitness of the individual will still be determined to be low. There is no partial credit given to the individual based on the fact that this ADF is useful. In point of fact, there is not even any way to determine if this ADF is useful because we would need to couple it with a main body that is able to exploit it to the fullest to determine its usefulness.

In other words, this case applies to the situation where the ADF is called in the wrong place or when the main body is very poor, thus, there is no immediate gain in having discovered a very useful ADF.

3.4.2 Poor ADF and Good Main Body

The previous section explained that it may happen that the ADF is useful but the main body is poor. It may also be the case that the main body is very fit but the ADF does not do anything useful. In this case there is no partial credit given to the good main

body. In fact, without a fitting ADF, it can not even be determined how useful a main body is, assuming the main body relies on using ADFs. Another time, we see the importance of coupling ADFs and main bodies that work well together.

3.4.3 Main Body and ADF Do Not Fit Together

There might be more than one way a problem can be solved and consequently there might be more than just one useful subroutine or main body. These useful parts of a solution may or may not all be interchangeable. It may happen that both the main body and ADF within an individual could potentially be highly fit parts of a solution, but they just do not produce a good individual when coupled together, in other words, they do not fit together. With the proper ADF or main body counterpart, either one of them could produce a highly fit individual. In this case, there is yet again no partial credit given to the individual for having good parts of a solution.

3.4.4 ADF is Not Called At All

There is also no constraint in GP+ADF that forces program trees to use ADFs. An ADF may or may not be called at all within the main body and can simply be ignored. These individuals are essentially the kind we find in regular GP. These individuals may carry useful parts, nevertheless, if they account for a part of the total population and if the problem complexity is such that it warrants the use of subroutines, then it may be wasted effort in evaluating these individuals.

3.4.5 ADF is Passed The Wrong Arguments

In our example, an ADF is defined such that it takes an argument. In Figure 2, the argument that is passed is x and therefore the ADF performs the necessary functionality. However, even if an ADF has the right functionality and the main body calls it in the

right place in its program tree to make it useful, it may happen that the wrong argument is passed. Suppose for example that instead of x, the main body passes x-x or 0. Subsequently, the individual's fitness would be determined to be low even though almost all of the program structure was correct. Therefore, it may be worthwhile trying to pass different arguments to an ADF if we know that it is a useful subroutine.

3.4.6 Conclusion

In all of the cases presented above, it may happen that good parts of a solution are found but they are not given any partial credit in terms of fitness. The consequence is that these good parts may be lost in following generations since their worth was not recognized. We claim that the right *coupling* of an ADF and a main body is just as important as discovering useful subroutines. In addition, due to the architecture of the GP+ADF approach, an ADF is always constrained to a single individual. If a very fit main body or ADF is found in one individual, this discovery may not even be recognized since only one other counterpart is used to couple and evaluate it. Furthermore, these good subroutines and main bodies will not be available to all the other individuals except with crossover exchange of some parts in very few other individuals. With these considerations in mind, we can proceed to describe to co-evolutionary approach and how we will try to overcome these limitations.

3.5 Co-Evolutionary Automatically Defined Functions

The core idea of GP with Co-Evolutionary Automatically Defined Functions (GP+CADF) is that the main bodies and ADFs are no longer part of the same individual. Instead, they are each distinct individuals in their own populations. There will always be

exactly one main body population and as many ADF populations as the number of ADFs we choose to have in our choice of architecture.

3.5.1 Description of the Algorithm

The GP+CADF algorithm works with n populations $P_0 \ldots P_n$ where P_0 is the main body population and $P_1..P_n$ are the ADF populations. There are as many ADF populations as the number of ADFs we choose to have in our architecture. The total number of individuals in all populations is M, and therefore the number of individuals in each population is M/n. Moreover, we can also decide to have an unequal number of individuals in each population as long as the total adds up to M. This could be used in the case where we want to devote more computational effort to the main body or to a particular ADF. For the remainder of this text the number of individuals will be split evenly unless otherwise stated and explained why.

Figure 3 presents the basic GP+CADF algorithm:

1: Let B_i be random individual of P_i for i=0..n

2: For G generations or until solution found:

3: Update Bi to be the best individuals found so far in P_i for i=0..n

4: Evaluate each individual in P_i by coupling it with B_j where j=0..n and $j \neq i$

5: Breed each population and apply genetic operators

Figure 3 - Basic GP+CADF algorithm

Since the main bodies and ADFs are all program trees, we can apply the genetic operators in line 5 such as crossover and mutation to all individuals. In addition, actions such as tree initialization and fitness proportional selection including tournament selection can also be applied in all populations just as in the regular GP approach. However, one problem that becomes apparent is that the individuals in the main body

and ADF populations can not be evaluated just by themselves. We need to have all the ADFs for a main body and we also need a main body and all other ADFs for an ADF in order to couple them together and form a complete individual that can be evaluated as stated in line 4.

3.5.2 Fitness Evaluation of Individuals in the GP+CADF Approach

Suppose that we decide in our architecture that we will have two ADFs. Then, let's say we decide that there will be a total of 3 populations each with M/3 individuals. If we consider all the ways that these individuals can be coupled, there are a total of $M^3/27$ combinations. If we wanted to evaluate all of these combinations at every generation this would quickly become infeasible due to exponentially increasing evaluation times. By contrast, GP+ADF with a single population of M members would evaluate just M individuals at every generation.

A solution to this problem is to couple all of the individuals in a population with just one individual B_i from all other i populations. It is immediately apparent that this choice of B_i is very important since this individual will be used in all fitness evaluations. Furthermore, the fitness will be an evaluation of how well an individual fits with all our chosen B_i. Consequently, there will only be M/3 * 3 = M total evaluations which is the same as GP+ADF. A reasonable assumption is that a good choice for Bi would be the current best individual found so far in each of the populations. This choice makes sense because the best individual has the highest fitness and does most of what is required by our problem. Therefore, these best individuals are the best candidates for useful subroutines and main bodies. Furthermore, since all of the individuals in every population will be coupled with these best individuals we will move toward our goal of finding the

best coupling where the ADFs and main bodies work well together.

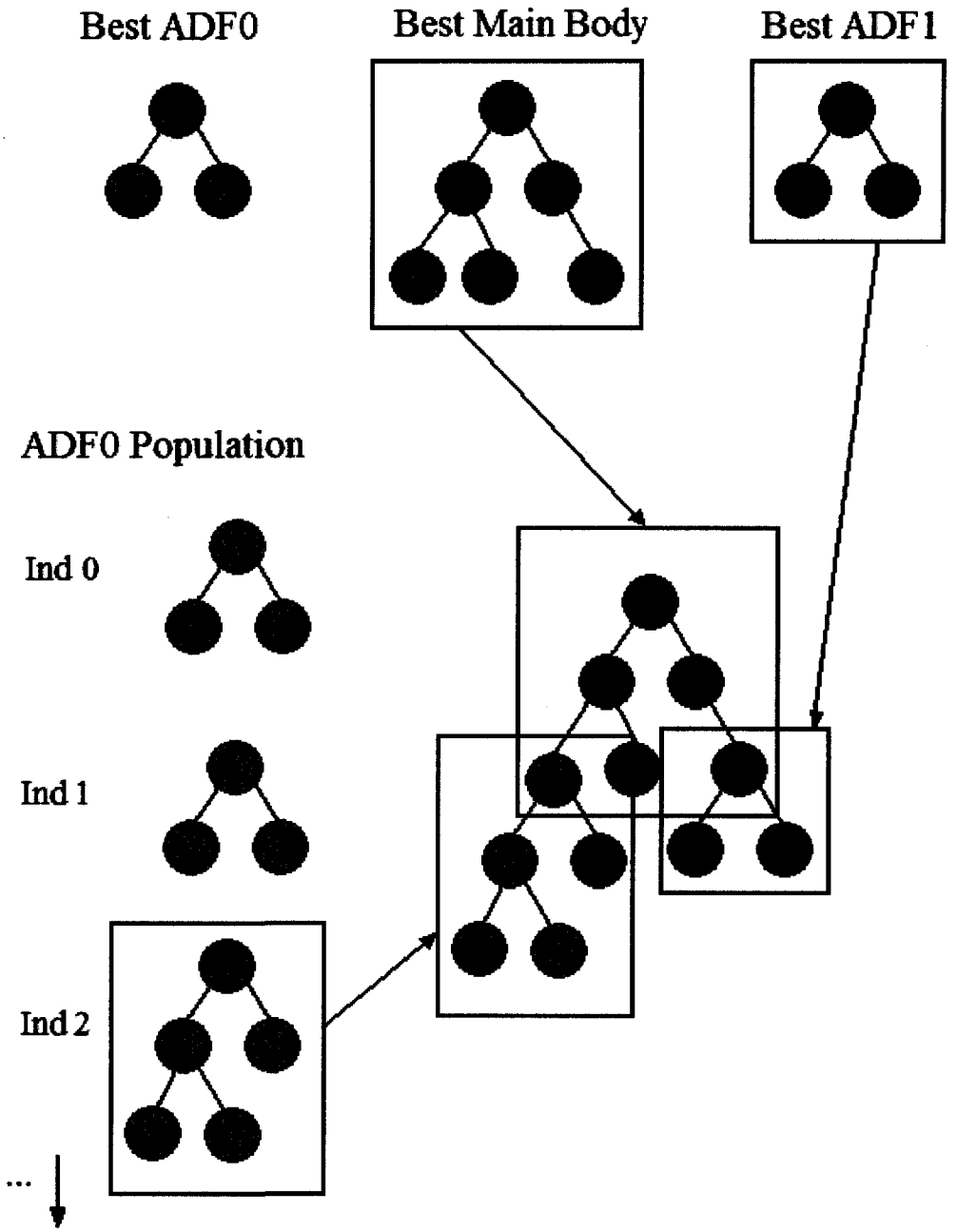

Figure 4 - Evaluating ADFs in a multi-population system. Each ADF from the ADF0 population is plugged into the best main body along with the best ADF1. The complete individual is evaluated and the fitness is assigned to the ADF0 population individual we were trying to evaluate. In this example, individual 2 from the ADF0 population is evaluated by coupling it with the best main body and best ADF1.

3.5.3 How to Evaluate ADFs

Figure 4 shows a diagram of how evaluation of ADFs is done in the co-evolutionary multi-population system. The setup for that example is that we have one main body and two ADFs. Therefore, there will be a total of 3 populations: one for the main body and one for each ADF. In the top row are the best individuals from each population. In the diagram, the best main body is a 6 node program tree and it calls both ADFs in its terminals. The best ADFs are both 3 node program trees in this example. For the sake of simplicity we do not deal with ADF arguments in this example.

The ADF0 population is represented as a column of individuals and for the sake of simplicity there are only the first three individuals shown. The diagram shows how individual 2 would be evaluated: it will be plugged into the best main body along with the best ADF1. This complete individual will be evaluated and the fitness will be assigned to individual 2. The same process will be repeated for all individuals in the ADF0 population. Similarly, this process will be applied to all the individuals from the ADF1 population with Best ADF0 used where ADF0 is called.

3.5.4 How to Evaluate Main Bodies

Figure 5 presents a diagram of how evaluation is done for the main bodies. The best of each population is shown in the top row and the first two individuals from the main body population are shown in a column. To evaluate the main body individual 1, the two best ADFs will simply be plugged into wherever it calls ADF0 or ADF1 and the complete individual's determined fitness will be assigned to main body individual 1. In the example shown, the main body calls both ADFs in its terminals.

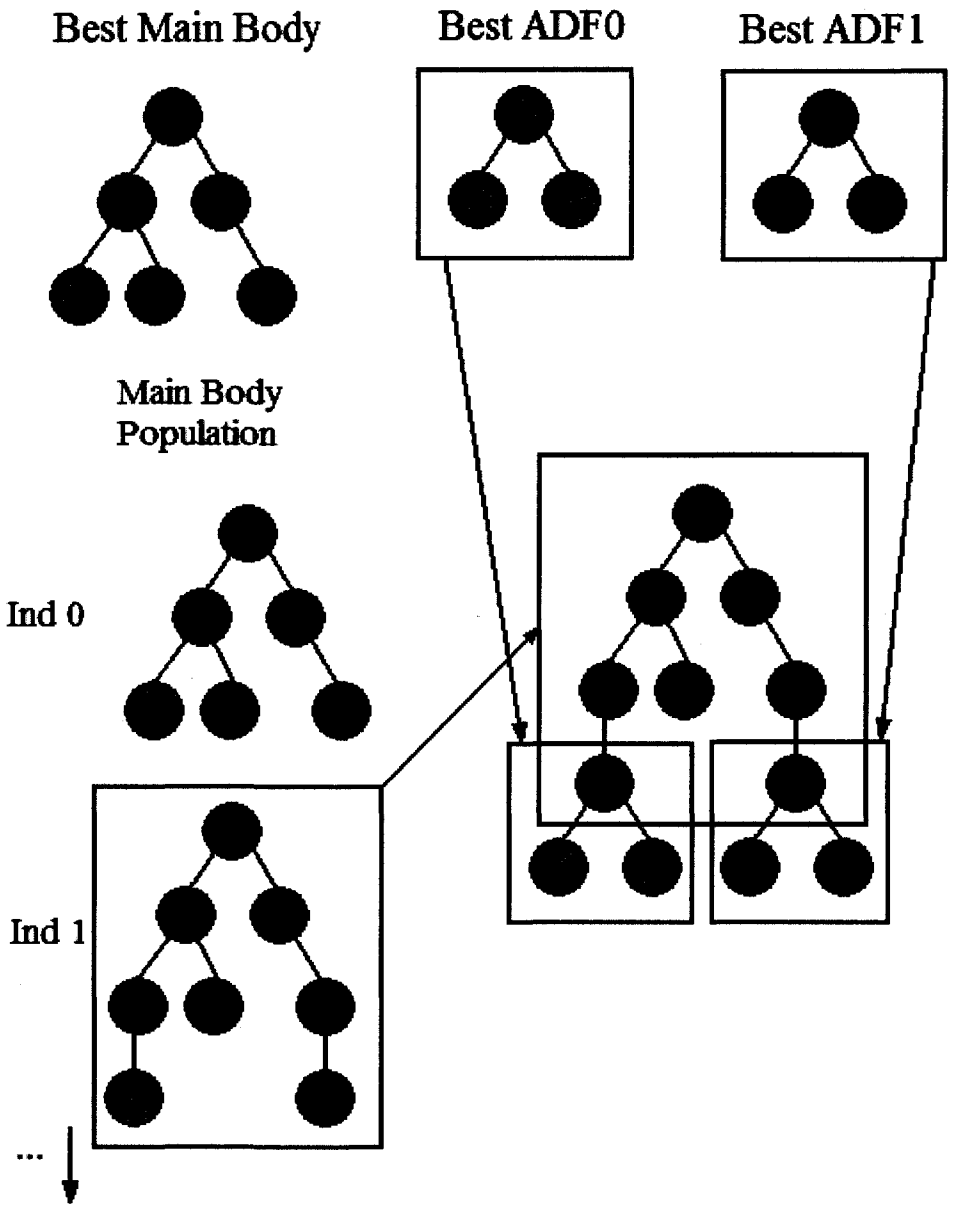

Figure 5 - How evaluation of main bodies is performed in a multi-population system: The best ADFs from each population are plugged into wherever the main body calls them. The determined fitness is then assigned to the main body individual we were trying to evaluate.

3.5.5 Choice of Best Individuals

The previous sections showed that a solution to the problem of evaluating main bodies and ADFs in different populations was to couple all of the individuals in a population with just one individual B_i from all other i populations. Moreover, it was

stated that a good choice for Bi would be the current best individual found so far in each of the populations. However, there are some important design decisions that must be considered when choosing the best individuals.

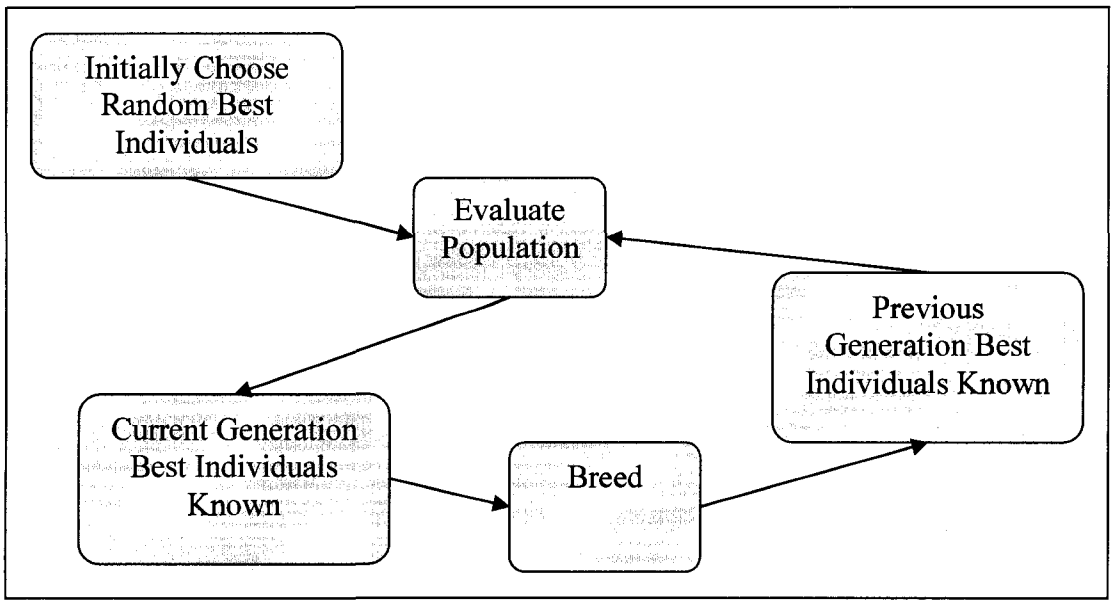

Figure 6 - Flowchart of the co-evolutionary evolution process shows the order that operations must be applied in. The chart shows that knowing the best individuals must precede evaluation and evaluation must precede breeding.

Figure 6 shows a flowchart of the evolution process. The important point is that individuals cannot be evaluated unless we know the best individuals B_i. However, we cannot know the best individuals unless we evaluate the entire population. Therefore, to boot start the whole process all Bi must be determined in a way that does not involve using any Bi, and one way is to choose all B_i randomly as shown in the top left hand corner of Figure 6 and line 1 of the basic algorithm in Figure 3.

Furthermore, the population cannot be bred using fitness proportional selection unless it has been evaluated. Therefore, evaluation must necessarily precede breeding. However, once the population has been bred into the new generation it is once again unevaluated. This is because breeding operators like crossover and mutation will change

individuals and we will not know their fitness unless we evaluate them again in the next cycle. Therefore, after breeding we will only know the previous generation's best individuals. To complete the evolutionary cycle, we are forced to evaluate the current generation using the previous generation's best individuals. In other words, the best individuals B_i from each population will lag behind one generation. Figure 7 illustrates this mechanism. In the figure, an arrow points from a population to be evaluated to the best individual in the other population used for the evaluation. We see that in the first generation a random individual is designated which is used to evaluate generation 0.

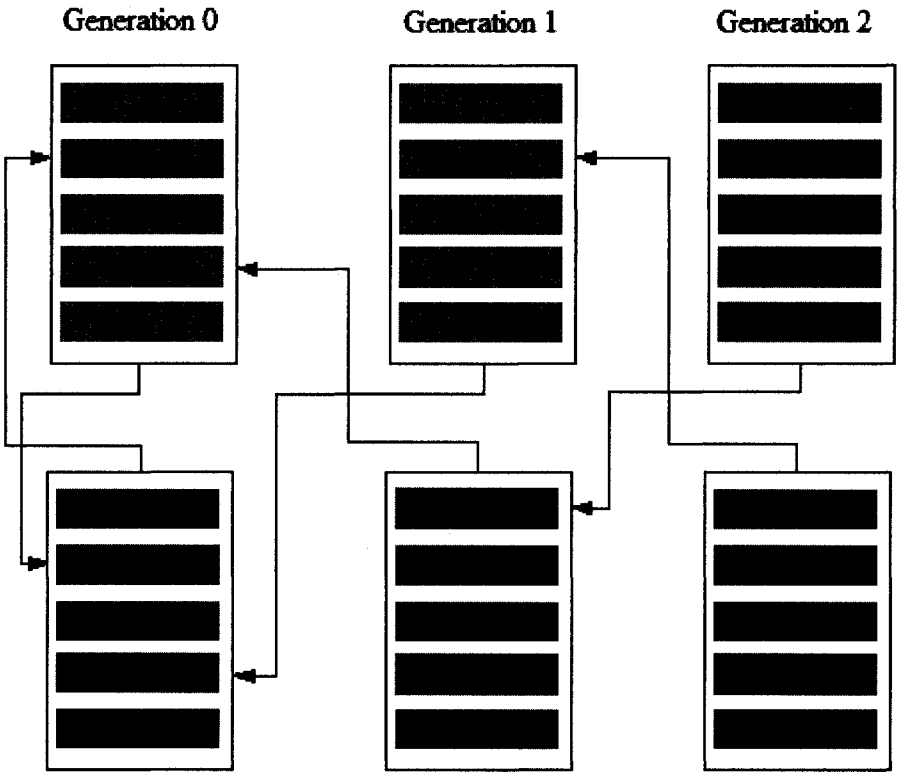

Figure 7 – This is a co-evolutionary, two population system with the ADF population in the top row and the main body population in the bottom row. Initially, random individuals from both populations are used for evaluation. Afterward, the best individuals from the previous generation are used. For simplicity, only the first three generations are shown. The arrow points from a population to the best individual used to couple with and evaluate all individuals in that population.

Generation 1 is evaluated with the best individuals determined in generation 0. Similarly,

generation 2 is evaluated with the best individuals determined in generation 1.

3.6 Proposed Improvements for GP+CADF

The next sections will examine potential improvements for the GP+CADF method. At this point we introduce them as potential improvements because we only make a hypothesis as to how GP+CADF could be improved and we test if there was actually an improvement in chapter 4 with the parity problem. These include using elitism in evaluation, using more than one best individual for evaluation, finding better initial best individuals and generating a better initial population. They are presented as hypothesized improvements or solutions to drawbacks or difficulties and will all be put under test in the next chapter where GP+CADF will be tested extensively with the parity problem.

3.6.1 Co-evolutionary Elitism

Section 3.5.5 showed that one strategy for evaluating individuals is to use the best individuals that were determined from the previous generation. The motivation for this design decision is that the best individuals must be updated as evolution progresses in order to use better, newly evolved ADFs and main bodies and to find the best coupling between them. Initial experimentation with the parity problem confirmed that this was absolutely necessary to get any solution at all.

On the other hand, we could also use an elitist strategy where we only keep track of the best individual found so far in the entire evolutionary run. The motivation here would be that we want to use the absolute best main body and ADF that we can find in the entire run. In addition, another hypothesis would be that this will have a stabilizing effect because the best individuals we are trying to couple and fit with all other

individuals will not change as rapidly. These hypotheses will be tested empirically in the next chapter where we will test the GP+CADF method with the parity problem and will determine if the elitist strategy would fare better.

3.6.2 K-Best Couplings

In the GP+CADF approach as presented so far we evaluate a population by coupling it with only one best individual from all other populations. Instead of just one, we could try coupling it with the k best individuals from the other populations. Essentially, we would be creating k different individuals for every evaluation and would look for the best fitness that was found among those k individuals.

It is not immediately clear if computational effort could be reduced since every increment of k will add M extra evaluations where M is the population size. This possible improvement will be tested in the next chapter with the parity problem.

3.6.3 Better than Random Initial Best Individuals

In Figure 6 we saw that one way to boot start the process of evaluation is to choose the initial best individuals randomly from generation 0. The problem with this approach is that since these individuals are random, they might be particularly unfit individuals and it might not be worthwhile to start the evolution process with these poor individuals. In other words, choosing particularly bad initial best individuals may lead the search down the wrong path.

A different approach would be to initially try M couplings between all individuals and choose as best those individuals that produced the highest fitness when coupled together. Recall that M is the population size. Since we have no prior idea of which couplings would be the best, a simple sequential coupling of the ADFs and main bodies

as they appear in the population is as good as any other random coupling. This is justified by the fact that the initial individuals were generated randomly and not in any particular order.

This would amount to sacrificing one generation of evaluation in order to determine better initial best individuals. This hypothesis will be put to the test in the next chapter with the parity problem to see if this approach produces better results than choosing random best initial individuals.

3.6.4 Checking for ADF Calls

One critique of the GP+ADF approach that was mentioned is that there is no constraint that forces the main bodies to call ADFs at all. We may simply have a main body individual that is of the regular GP type and does not use any subroutines.

Since using ADFs and coupling main bodies and ADFs is the heart of the GP+CADF approach, then it is essential that ADFs be used by main bodies. This is because most of the computational effort in GP+CADF is spent in evaluating couplings between best individuals and populations. It will be wasted effort to couple a main body with all ADFs in an ADF population when an ADF is not even referenced.

Therefore, an improvement to the initial population would be to regenerate a main body that does not call ADFs until one is made with ADF references. It will be tested in the next chapter with the parity problem if this change would lead to an improvement.

3.7 Earlier Work with Co-Evolutionary GP

The idea of co-evolving main bodies and ADFs in GP has already been dealt with in [10] by Aler. While both his work and this deal with a multi population co-

evolutionary system, there are important differences in the implementation, design decisions and testing methodologies.

In our approach, an n population system will be set up to have M/n individuals in each population where M is the total number of individuals desired in all populations. GP+ADF will also have M individuals in comparison with our method. Aler's approach is to have M individuals in each population but run each population for G/n generations where G is the total number of generations desired. Each of the n populations is run for 1 generation until the last population is evaluated at which point we loop back to the first population. In other words, in order to compensate for the larger number of evaluations required in GP+CADF we reduce the number of individuals in each population while Aler reduces the number of generations that each population is run. In all approaches: this, Aler's and GP+ADF, there are the same number of evaluations per generation performed. An important difference is that in Aler's approach, the populations are evolved one by one in a round robin fashion while we evolve all populations at every generation.

The GP settings are also quite different than the experimental settings that we will use. Aler experiments with a small population and long runs, which means that M is 200 and 400, and G is 150. The large number of generations used in his setup also compensates for the design decision that each population is run for G/n generations. Had the number of generations been 60 with 3 populations as we commonly use, then each population would only be run for 20 generations which is below the convergence generation for many problems examined here. In addition, many problems would be infeasible to run with G=150 because of increasing evaluation times for complex problems such as higher order parity, except with a very small population. Furthermore,

due to premature convergence it may happen that we never converge to a solution and it will be more worthwhile to restart a run at an earlier generation than have really long runs. We will also test the GP+CADF with a range of different population sizes and distributions of different population sizes between the populations.

In addition, Aler's approach is elitist meaning that only the best of run best individuals are kept for evaluation. However, none of the other proposed improvements outlined in the previous section are implemented. We will experiment with both the elitist and non-elitist versions. In addition, we will test if k best couplings rather than just a single one would fare better. We will also attempt to improve on the choice of the initial best individuals and attempt to find a better alternative to choosing them randomly as is the case in Aler's work. Moreover, we will also attempt to improve the initial population by regenerating individuals with no ADF calls which may be crucial to the GP+CADF approach. Lastly, we will provide a detailed treatment of the effect of a range of different mutation operators on GP+CADF which has not been studied before.

In terms of experimental tests, Aler's work tests his version of co-evolutionary ADFs with only the 5 and 6-even-parity problems. While GP+ADF was outperformed in those tests, we can not conclude on the basis of these tests alone that GP+CADF is advantageous over GP+ADF. We will test the GP+CADF method with a wide variety of problems that scale up in complexity much further than 6-even-parity and compare the regular GP, GP+ADF and GP+CADF methods in all cases. These problems will include up to 14-even-parity, sum of bits, image recognition, lawn coverage, and bumblebee problems. Our results show that the ratio in computation effort difference between GP+ADF and GP+CADF was larger by a factor of 6 than in Aler's work. In addition,

Aler's test only included a 3 population system; we will test a range of total populations and ADF architectures to determine the relative performance of GP+ADF and GP+CADF under many different conditions. We will also identify problems and cases where using GP+CADF is not advantageous.

Chapter 4: The Even-n-Parity Problem

4.1 Introduction

The even-n Boolean parity problem is the task of recognizing whether a bit string of length n consisting of 1's and 0's contains an even number of 1's. Suppose that we are considering a bit string of length 3 and would like to find a function that determines if there is an even number of 1's in that string. Then, we can fully represent and define this function with a truth table shown in Table 1 that specifies the output for all possible bit strings of length 3.

Bit String	Bit2	Bit1	Bit0	Output
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Table 1 - Truth table for the even-3-parity function.

We see that if even just one bit is changed in the bit string, then the output changes. Additionally, in order to find the output we need to consider all of the inputs. For that reason, a parity check is often used in the transmission of data to detect whether an error has occurred. In addition, since the output is very responsive to changes in any of the inputs and since we need to consider all the inputs, the parity function is hard to learn for machine learning algorithms as discussed in [2]. Minsky and Papert showed that a single layer neural network or perceptron cannot even solve odd-2-parity which is equivalent to the XOR function. The reason is that such a perceptron can only induce

linear functions [3]. Furthermore, Koza reports that not a single program solved the even-3-parity function in a blind random search of 10,000,000 programs that work with 3 input bits and the Boolean functions AND, OR, NAND, NOR [1].

Another flexibility that we have with the parity problem is that we can increase the problem difficulty considerably by increasing n (bit string length) as was shown by Koza in [1]. Consequently, the parity function is often used as a benchmark in machine learning algorithm comparisons and will be the first problem that puts GP+CADF under test.

The main motivation for using the even-n-parity problem to test GP+CADF is to test it with a relatively hard problem that can be scaled up in complexity. In addition, we want to perform a comparative analysis of regular GP, GP+ADF and GP+CADF to determine if we can solve a problem with less computational effort with GP+CADF than with regular GP or GP+ADF. The parity problem will be examined in detail for values of n ranging from 3 to 14. Furthermore, experiments with different mutation operators will be performed to determine what we may gain by adding mutation to GP+CADF. Lastly, we want to test our hypothesized improvements to GP+CADF proposed in the previous chapter that include using elitism in evaluation, using more than one best individual for evaluation, finding better initial best individuals and generating a better initial population.

4.2 Related Work on Parity Problems

The parity problem has been approached with GP in previous works. In [1] Koza gives a detailed treatment of the parity problem with and without ADFs for n = 3 to 11. The function set that Koza used was {AND, OR, NAND, NOR} and ADFs. This function set is computationally complete and unbiased for solving parity. The set is unbiased

because the Boolean functions XOR (Exclusive Disjunction) and EQ (Logical Equality) are very useful when solving parity as stated in [4], in other words, they can be used as building block functions. In fact, XOR and EQ are themselves odd-2-parity and even-2-parity respectively. Omitting them makes the parity problem much harder for GP because it needs to come up with equivalent functionality many times in the program tree [4]. Koza's use of ADFs showed that the ADFs came up with subroutines that were very often parity rules. A function is considered a parity rule if it matches the behaviour of any even or odd n-parity function. Therefore, Koza's work showed that parity is a type of problem that can benefit from the use of subroutines that can be utilized as reusable building blocks. In other words, Koza showed that higher order parity problems can be solved by decomposing them into smaller parity sub problems and then assembling the solutions of the sub problems into the final solution. Koza also showed that the problem complexity quickly scales up as n is increased. Regular GP was unable to solve parity higher than 5 and GP+ADF solved problems up to n=11; Koza was unable to proceed past this due to increasing evaluation times which is not surprising given that this work was published in 1994. Consequently, Koza's experiments do not have many independent runs with n=11 featuring only 4 runs. Meaningful performance curves are only given up to 7-parity. This work will improve on these results by finding results for larger values of n with as many trials as is feasible.

In [4], Poli and Page report solving up to 22-even-parity. However, in their experimental setup they utilized all 16 2-arity Boolean functions including XOR and EQ. In addition, they used Sub-Machine Code GP which allows parallel execution of 64 fitness cases per program execution. Furthermore, they used up to 50 parallel

workstations each with a part of a population called a deme. Similar to our approach, the demes exchange best individuals after every generation, but this is done in a very different way from GP+CADF; best individuals will be inserted into other populations replacing the worst individuals. While obtaining a result for 22-even-parity is the most recorded in any works examined, the experimental setup used was highly tuned for the specific parity problem and parameters were optimized for particular values of n. The goal of testing GP+CADF will not be to optimize it fully to work with the parity problem. Rather, the goal will be to perform a comparative analysis of different methods under the same settings with unbiased parameters.

In [5] Oltean describes a method for solving parity problems that utilizes Multi-Expression Programming (MEP). In MEP, individuals are represented as linear chromosomes where each chromosome is made up of genes. A gene is a function or a terminal. The main feature of MEP is that each individual is decoded from the genotype to phenotype and this decoding can produce multiple solutions. The fitness of the individual is then the fitness of the best solution. In addition, Oltean aims to improve MEP with the use of ADFs and employs the same strategy of [4] of using all 2-arity Boolean functions and Sub-Machine Code GP. The author mentions that a perfect comparison between GP and MEP can not be made due to incompatible representations. Nevertheless, it is reported that a combination of all the improvements yielded a solution for up to 18-parity.

In [6] Gathercole and Ross report experiments with attempting to reduce the total number of evaluations required for parity. In order to fully evaluate a program, all 2^n fitness cases must be evaluated for a bit string of length n. A different strategy is to stop

evaluating an individual if a certain threshold of misclassifications is passed and to judge all remaining test cases as misclassifications, leading to large savings in CPU time. The basic idea is that low fitness individuals do not need to be tested on all fitness cases. The order of fitness case presentations is also dynamically adjusted at run time to present them from easier to harder cases. However, the computational effort is not reported.

4.3 Experimental Setup

Table 2 summarizes all the parameters in the experimental setup for the even-n-parity problem. The choice of architecture for GP+ADF and GP+CADF is to have a total

Problem	Even n-parity
Objective	Find a program that outputs 1 if the number of 1's in a bit string is even, false otherwise
Number of bits (n)	3 to 14
Population Size (M)	1800 for n<=13, 3800 for n=14
Generations (G)	60
Runs	300 for n > 6, 500 for n <=6
CADF Populations	2
Mutation	Rehang Mutation for GP+CADF, Subtree Mutation for GP+ADF (`ec.gp.koza.MutationPipeline`, `ec.gp.breed.RehangPipeline`)
Crossover	Subtree Crossover (`ec.gp.koza.CrossoverPipeline`)
Number of ADFs	2
Function Set (no ADF)	D0 to D13, AND, NAND, NOR, OR
Function Set (with ADF) Main Body	D0 to D13, AND, NAND, NOR, OR, ADF0, ADF1
Function Set for ADF0	AND, OR, NAND, NOR, ARG0, ARG1
Function Set for ADF1	AND, OR, NAND, NOR, ARG0, ARG1, ARG2
Fitness Cases	All 2^n possible bit strings
Fitness	2^n – correct outputs
Hits	correct outputs
Success Predicate	0 (all fitness cases must be perfect for 0 fitness)

Table 2 - Experimental setup for the even-n-parity problem.

of 2 ADFs; one with 2 arguments, and one with 3 arguments. The total population size M is 1800. This implies that GP+CADF will have a total of 3 populations; one for the main body and two for the ADFs each with 600 individuals. These parameters were kept constant for up to 13-parity to be able to see trends across different n with the same settings. For n=14 the settings were changed; the second ADF had 4 arguments instead of 3 and M was 3800. The population size was larger in an attempt to increase the success probability for the highest difficulty problem.

A function set very similar to Koza's setup was used that included the input bits D0 to D13 and the Boolean functions AND, NAND, NOR, OR which are computationally complete. Therefore, the function set for regular GP without ADFs is

F = {D0-D13, AND, NAND, NOR, OR}

with an argument map of

{0,2,2,2,2}.

In the case with ADFs the function set for the main body is

$F_{MainBody}$ = {D0-D13, AND, NAND, NOR, OR, ADF0, ADF1}

with an argument map of

{0,2,2,2,2,2,3}.

It was decided that the ADFs will not have access to the input bits which can only be optionally passed as parameters to the ADFs by the main body.

Therefore the function set for ADF0 is

F_{ADF0} = {AND, NAND, NOR, OR, ARG0, ARG1}

and the function set for ADF1 is

F_{ADF1} = {AND, NAND, NOR, OR, ARG0, ARG1, ARG2}

4.4 Experimental Results

Figure 1 shows the minimum computational efforts E_{min} obtained in the experimental comparison of regular GP, GP+ADF and GP+CADF with the setup described in the previous section. Note that the computational effort scale is logarithmic where every numbered y axis increment increases by an order of magnitude.

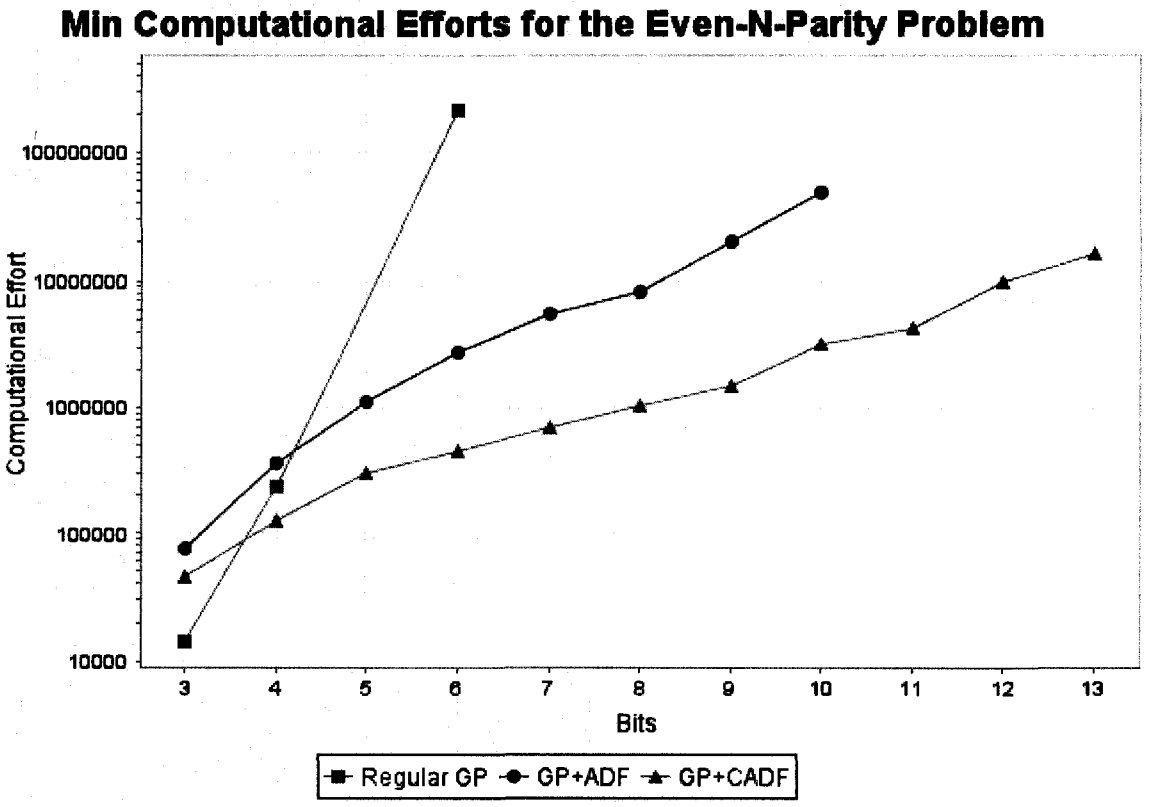

Figure 1 - A graph showing the experimental results of the comparison of regular GP, GP+ADF and GP+CADF. In order to show data in a meaningful way, the y-axis is logarithmic.

There were no solutions obtained for regular GP with 300 runs past even-6-parity, and GP+CADF was the only method to yield solutions after 300 trials past even-10-parity. The immediate conclusion is that as we increase n, the computational effort increases most quickly for regular GP followed by GP+ADF and GP+CADF. Correspondingly, there is always less computational effort required to solve the even-n-parity problem with

GP+CADF than with GP+ADF. The next sections will examine in more detail the performance and solutions obtained for different values of n. Table 3 presents the same data as Figure 1 numerically, along with a column that shows the increase factor in computational effort for every increment of n. The motivation for going through a considerable number of different parity problems is to examine in detail the actual solutions that different methods come up with. In addition, we want to analyze at what point GP, GP+ADF and GP+CADF stop discovering solutions as the problem difficulty increases. Those readers who are interested only in the highest parity problems solved should proceed to 10-even-parity for GP+ADF and 14-even-parity for GP+CADF.

n	Runs	Regular GP		GP+ADF		GP+CADF	
		E_{min}	E_{min}+	E_{min}	E_{min}+	E_{min}	E_{min}+
3	500	14,400	n/a	75,600	n/a	45,000	n/a
4	500	234,000	16.3	365,400	4.83	126,000	2.8
5	500	No solution	n/a	1,107,000	3.00	302,400	2.4
6	500	219,515,400	n/a	2,714,400	2.45	446,400	1.48
7	300	No solution	n/a	5,508,000	2.03	693,000	1.55
8	300	No solution	n/a	8,100,000	1.47	1,044,000	1.50
9	300	No solution	n/a	20,109,600	2.48	1,461,600	1.4
10	300	No solution	n/a	49,572,000	2.47	3,240,000	2.22
11	300	No solution	n/a	No solution	n/a	4,248,000	1.31
12	300	No solution	n/a	No solution	n/a	9,720,000	2.29
13	80	No solution	n/a	No solution	n/a	16,380,000	1.67

Table 3 – Minimum computational efforts E_{min} for the even-n-parity problem for all 3 methods along with an increase factor E_{min}+ in computation effort from the previous n. Naturally, the increase factor E_{min}+ can not be shown where there is no data for the previous n.

4.4.1 Even-3-parity

In low order parity problems such as n=3, the regular GP method has the lowest computational effort. The conclusion is that at n=3 the problem complexity is such that it

does not warrant the use of subroutines. It is easier for GP to find a complete program tree than to find reusable subroutines and call them from the main body. Koza refers to this condition as the problem complexity being below the breakeven point for computational effort [1]. The performance graphs for regular GP, GP+ADF and GP+CADF are given in Figures 2, 3 and 4 respectively.

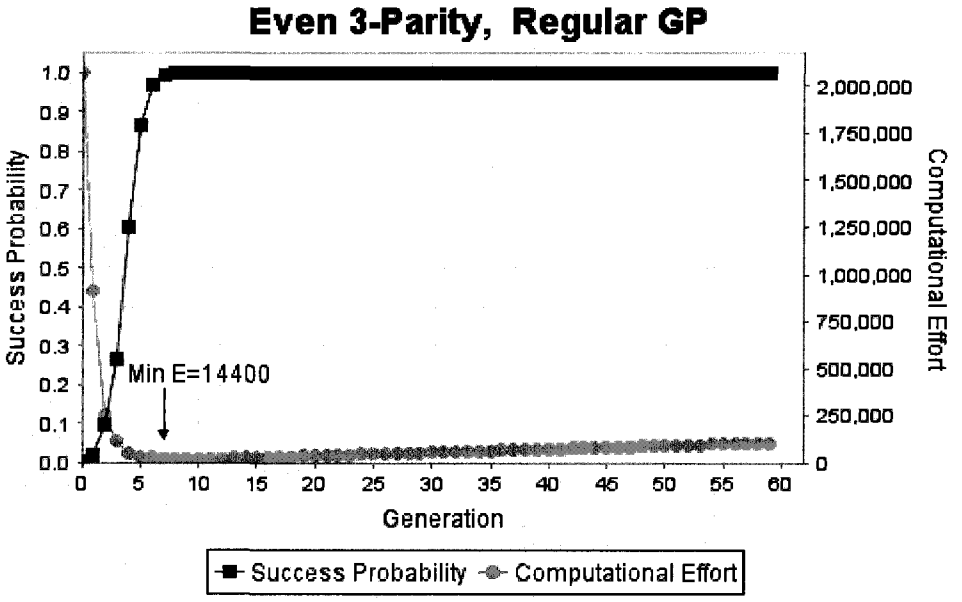

Figure 2 - E_{min} for regular GP was 14,400 at generation 7 with success probability 100%.

We can get an insight into why regular GP is the best method at this level of complexity by analyzing some of the solutions.

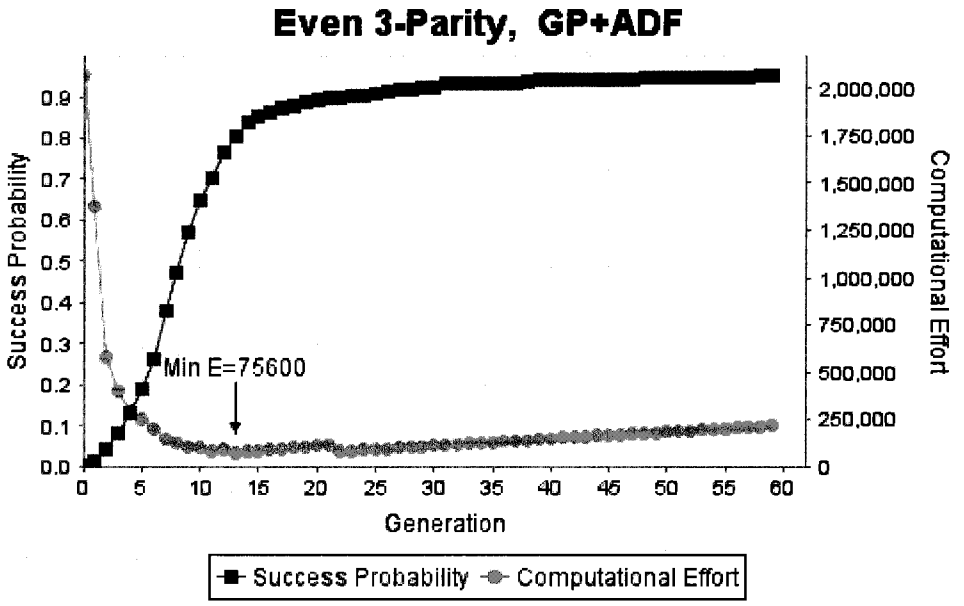

Figure 3 - E_{min} for GP+ADF was 75,600 at generation 13 with success probability 81%.

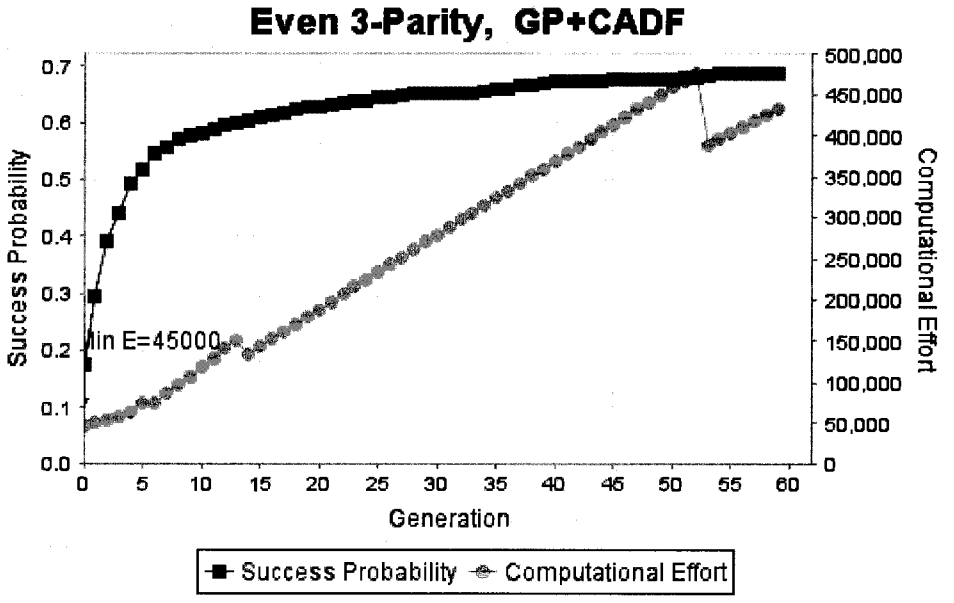

Figure 4 - E_{min} for GP+ADF was 45,000 at generation 0 with success probability 17%.

The following 55 node program is what regular GP evolved to solve even-3-parity:

```
(nor (nand (or (nand D1 (nand D2 D1)) (nor (nand D0 (nand (and (and D2
D1) D2) (nor (and D1 D1) D2))) (nor D1 D2))) (nor (nor (nand D0 D2)
(nor D1 D1)) (and (nor D1 D2) (or D0 D2)))) (nor (and D1 (or D2 D0))
(or (nand (or D2 D1) (nor D1 D1)) D0)))
```

Using all the logical operators and all the input bits a correct solution is found. By contrast, GP+CADF evolved the following 116 node program:

```
Main Body:
 (ADF1[2] D1 D1 (ADF1[2] (nor D2 (ADF1[2] (ADF0[1] D2 D1) (nand D0 D1)
(nor D0 D2))) (ADF0[1] D0 D2) (and D0 D2)))
ADF 0:
 (nand (nor (and (and (or ARG[1] ARG[1]) (or ARG[0] ARG[0])) (and (nand
ARG[1] ARG[0]) (and ARG[1] ARG[1]))) (or (nor (and ARG[1] ARG[0]) (nor
ARG[1] ARG[1])) (and (nor ARG[1] ARG[1]) (nor ARG[0] ARG[1])))) (or (or
(nand (or ARG[1] ARG[0]) (and ARG[0] ARG[0])) (or (and ARG[0] ARG[0])
(nor ARG[1] ARG[0]))) (or (nand (or ARG[0] ARG[1]) (nor ARG[0] ARG[0]))
(and (or ARG[1] ARG[0]) (and ARG[1] ARG[1])))))
ADF 1:
 (nor (or (or (nor ARG[1] ARG[2]) (and ARG[1] ARG[2])) (nor (or ARG[0]
ARG[1]) (nor ARG[1] ARG[1]))) (nand (nor (nor ARG[2] ARG[1])(nor ARG[1]
ARG[2])) (or (or ARG[0] ARG[0]) (and ARG[2] ARG[2]))))
```

Two ADFs are evolved and both are used in the solution. The first ADF is the two argument Boolean rule 10. This rule outputs a 1 whenever the first argument is 0. Oddly, the second ADF is a recreation of the AND function which is included in the function set. We can see that the GP+CADF developed ADFs that were larger than the main body, that the ADFs are called hierarchically and that one of the ADFs was unnecessary since it was a function already included in the function set. Therefore, the GP+CADF method introduced unnecessary complexity by using ADFs.

However, why did the GP+CADF solution use ADFs? In the GP+CADF approach, the main body population is 3 times smaller than in the regular GP approach. This is because we divide the total population between the main body and ADF populations as described in the previous chapter. Most of the evaluation time is spent in coupling main bodies and ADFs and evaluating them individually. Also, it is important to note that these trials were run with improved initial populations as outlined in chapter 2. This means that the initial population main bodies are regenerated if they do not have

ADF calls. Therefore, it is not at all surprising that GP+CADF relied heavily on using ADFs even when it would be simpler to come up with the entire program tree in the main body and not call ADFs.

In conclusion, we see that the level of complexity for even-3-parity is such that regular GP has the best performance with E_{min} at 14,400. It is followed by GP+CADF and GP+ADF with 45,000 and 75,600 respectively. Regular GP was the best approach because the problem complexity was such that ADFs were not needed; however, the GP+CADF method still relied on using them.

4.4.2 Even-4-parity

An interesting observation is that even though the computational effort was higher for GP+CADF then regular GP in even-3-parity, the generation for E_{min} for GP+CADF was 0 with 17% success probability (85 out of 500 trials). By contrast, the success probability was 0% for regular GP at generation 0. This shows that GP+CADF is able to immediately come up with a solution 17% of the time in the first generation; something regular GP and GP+ADF can not. This is the earliest indication that GP+CADF might be the method with the earliest generation of convergence as we increase n. To verify this hypothesis, we can look at the mean generation of convergence for n=3 and n=4 in Table 4. We use the unpaired Welch's T test to determine the statistical significance of the observation. Welch's modification to the t-test is used because we do not have equal variance in the samples. Note that we use 3 and 4 for n because these are the only parity problems where multiple solutions were obtained for the regular GP method.

n	Runs	Regular GP $G_{converge}$	GP+CADF $G_{converge}$	Difference	P Value
3	500	4.18	5.88	1.69	0.0028
4	500	22.80	9.15	13.65	> 0.0001

Table 4 - This table compares the average generation of convergence $G_{converge}$ of regular GP and GP+CADF for even-3 and even-4 parity. The two-tailed P value of Welch's unpaired t test is also given to show statistical significance.

At n=3 the difference between regular GP and GP+CADF mean generation of convergence is 1.69. Although very statistically significant with a two-tailed P value of 0.0028, the difference is not very large indicating that even when regular GP outperforms GP+CADF in computational effort, the difference in average generation of convergence is small.

For n=4, GP+CADF has a mean generation of convergence considerably lower than regular GP with a difference of 13.65 generations. This result is very statistically significant with a two-tailed P value of less than 0.0001. The conclusion is that as n is increased the generation of convergence becomes much lower than for GP+CADF than for regular GP. In addition, for n=4 GP+CADF has the lowest computational effort at 126,000. Regular GP and GP+ADF had 234,000 and 365,400 respectively.

In summary, GP+CADF outperforms regular GP in the even-4-parity problem but regular GP still outperforms GP+ADF. In addition, the mean generation of convergence is lowest for GP+CADF in the even-4-parity problem.

4.4.3 Even-5-parity

In the parity problem, the problem complexity quickly increases as n increases. As we can see from Table 3, the computational effort increased 16 times for regular GP, 4.83 for GP+ADF and 2.8 times for GP+CADF when advancing from n=3 to n=4.

Therefore, it is not surprising that no solutions were found by regular GP after 500 trials for n=5; the computational effort increased at such a rate that not even a single solution was found in a large number of trials. We can also see the overall trend for the computational effort as n increases. On average, E_{min} increases by 2.68 times for every increment of n for GP+ADF and 1.86 times for GP+CADF. The difference in these means is 0.769 and the two-tailed P value from a paired t test is 0.0238 which makes the difference statistically significant. The conclusion is that computational effort increases at a lower rate with GP+CADF than with GP+ADF.

Comparing E_{min} for GP+CADF and GP+ADF at 302,400 and 1,107,000 respectively we see that GP+CADF has a minimum computational effort 3.7 times lower. The performance graphs are given in Figures 5 and 6.

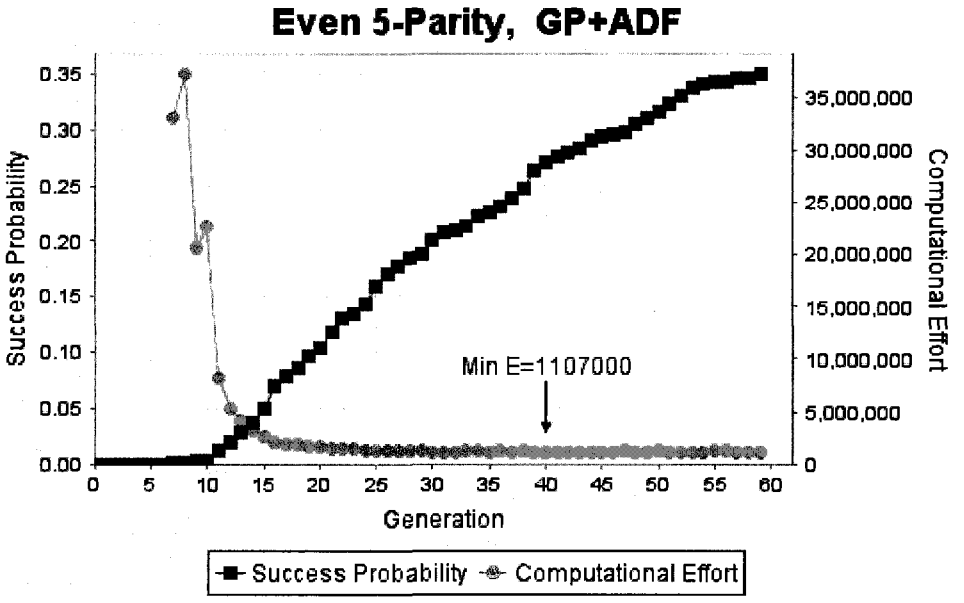

Figure 5 - E_{min} for GP+ADF was 1,107,000 at generation 40 and with success probability 27%.

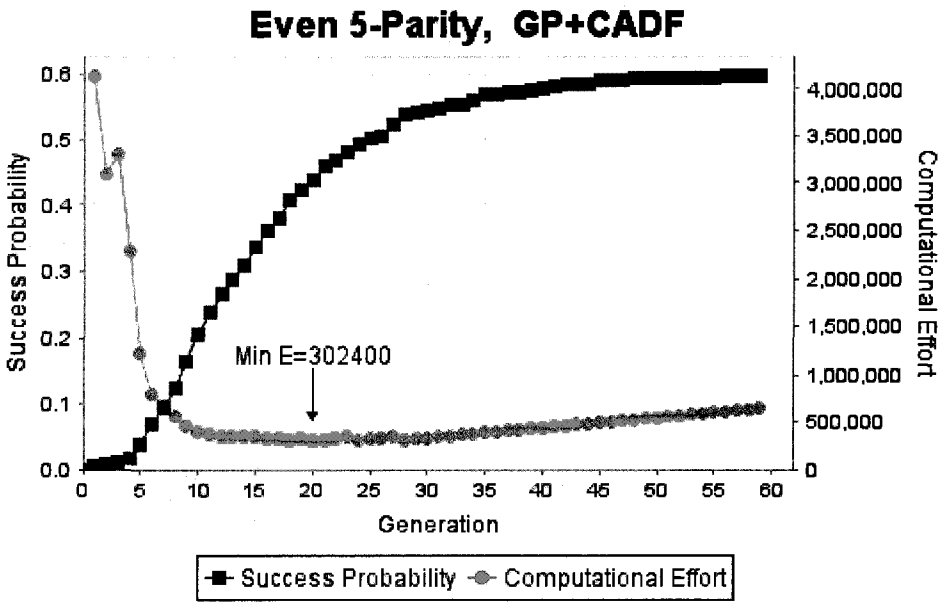

Figure 6 - E_{min} for GP+CADF was 3,024,00 at generation 20 and with success probability 44%.

4.4.4 Even-6-parity Solution with Regular GP

The level of complexity for these parity problems makes the use of ADFs very beneficial. A testament to this fact is that regular GP obtained no solutions for problems greater then even-5-parity. One rare exception is that a single solution was obtained for regular GP within 500 trials for even-6-parity. This success is considered very unlikely to be repeated within 500 trials since a solution was not obtained for the easier even-5-parity problem in the same number of runs. The computational effort is very high at 219,515,400 which is the most obtained in any of the experiments. It would not be feasible to run this many evaluations in practice. However, we must remember that the z value used in determining the computational effort is 0.99. In other words, we find how many evaluations are needed to assure a 99% probability of success. We simply got very lucky and obtained a result where the success probability is at most 0.2 % (1 success

from 500 trials). To the best of our knowledge this is the first time a solution has been obtained with standard GP for the even-6-parity problem under these problem settings.

Koza was unable to obtain a solution for even-6-parity with regular GP [1]. His settings were a much larger population size of 16,000 and 51 generations. He correctly assumed that regular GP would be capable of solving the even-6-parity problem; however, he incorrectly assumed a larger population and many more generations would be required. In our experiments we solved the problem with a population of 1800 and in 52 generations. The only parameter we needed to increase was the number of trials due to the low probability of success.

4.4.5 Even-6 to 10-parity

For parity problems with n=6 to n=10 solutions were obtained by both GP+ADF and GP+CADF. However, in all cases the computational effort was lower for GP+CADF. Table 5 shows the difference in computational effort for different values of n.

n	$E_{min\ GP+ADF}$	$E_{min\ GP+CADF}$	$E_{min\ GP+ADF}$ / $E_{min\ GP+CADF}$
6	2,714,400	446,400	6.1
7	5,508,000	693,000	8.0
8	8,100,000	1,044,000	7.8
9	20,109,600	1,461,600	13.8
10	49,572,000	3,240,000	15.3

Table 5 - The ratio of the minimum computational efforts for GP+ADF and GP+CADF is shown for the even-6-parity to even-10-parity problems.

We see that as n increases, the ratio of the difference in computational effort also increases. The minimum computational effort is 6.1 times smaller for GP+CADF for n=6 and 15.3 times smaller for n=10 which is more than an order of magnitude difference.

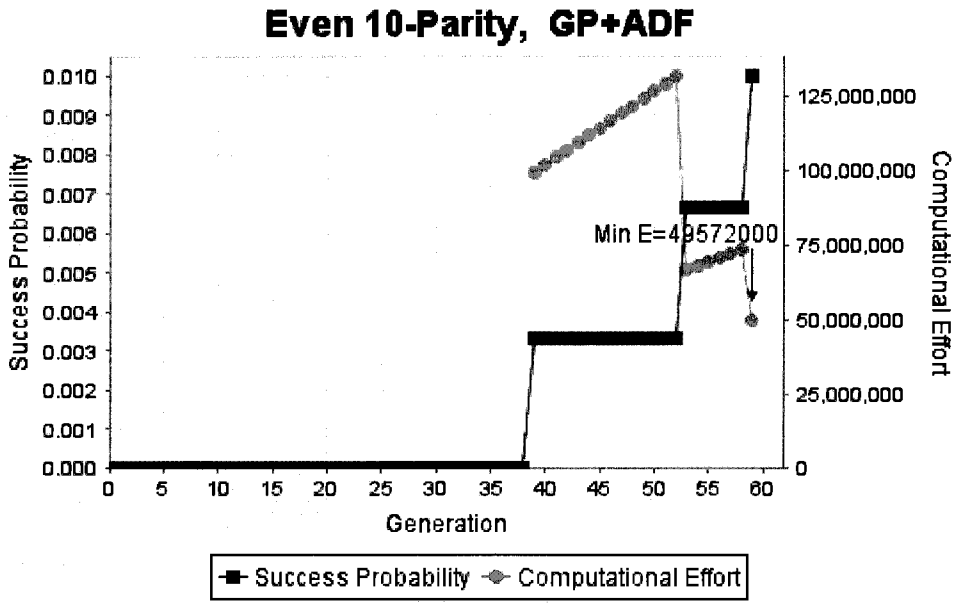

Figure 7 - E_{min} for GP+ADF was 49,572,000 at generation 59 and with success probability 1%.

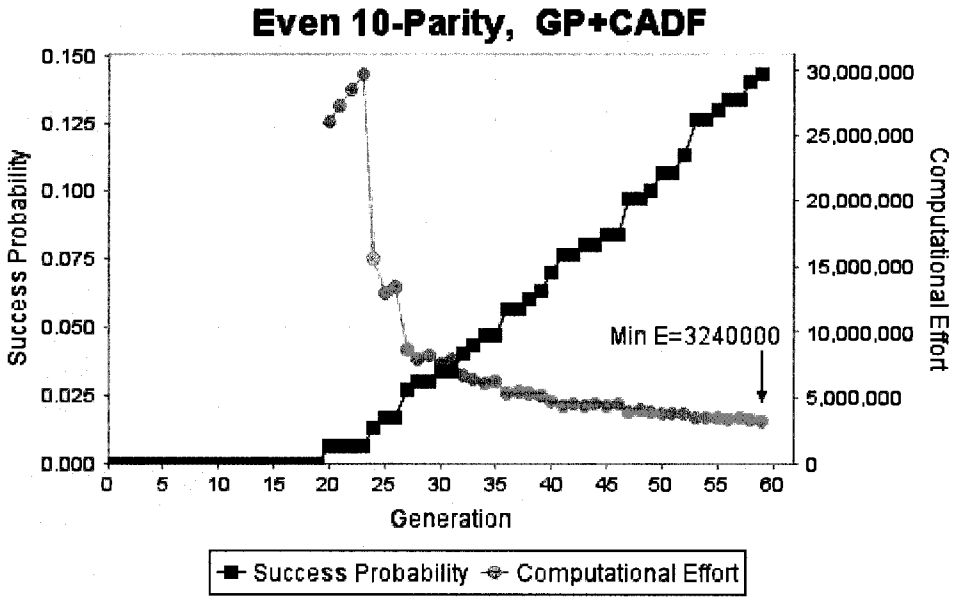

Figure 8 - E_{min} for GP+CADF was 3,240,000 at generation 59 and with success probability 14%.

Figures 7 and 8 show the performance graphs for the even-10-parity problem for the two methods. These results confirm that GP+CADF outperforms GP+ADF in the even-n-parity problem.

Furthermore, we can analyze the solutions to some parity problems with both methods. It is expected that both methods will find useful ADF subroutines that allow a larger problem to be decomposed into smaller sub problems and then assemble the sub problem solutions into the final solution. Let us compare the solutions of GP+ADF and GP+CADF for the even-8-parity problem. Here is the 200 node program evolved by GP+ADF:

```
Main Body:
 (ADF0[1] (ADF1[2] (ADF1[2] (ADF0[1] D0 (and D0 (ADF0[1] (ADF1[2] D6 D0
(ADF0[1] D0 D0)) D0))) (ADF0[1] (ADF0[1] D1 D0) (ADF0[1] (ADF1[2] D5 D4
(nand D4 (ADF0[1] D6 D2))) (ADF0[1] D7 D3))) (nor D4 (ADF0[1] D6 D2)))
(ADF0[1] (ADF0[1] D1 D0) (ADF0[1] (ADF1[2] D5 (or (or D1 D4) D5) (nand
D4 (ADF0[1] D6 D2))) (ADF0[1] D7 D3))) (nor D4 (ADF0[1] D6 D2))) D5)
ADF 0:
 (nor (and ARG[0] (and ARG[0] ARG[1])) (nor (and ARG[1] (or (nand ARG[1]
(or (or (nand (nor ARG[0] (nor ARG[0] ARG[0])) ARG[1]) (and ARG[1]
ARG[1])) ARG[0])) (or ARG[1] (and ARG[1] (or ARG[1] (nand ARG[1] (or
(nand ARG[0] ARG[0]) (nor (nor ARG[0] (nand (nor ARG[1] ARG[1]) (or
ARG[1] ARG[1])))) (nor (or (or ARG[0] ARG[0]) (nand ARG[0] ARG[0]))
(nand (or ARG[1] ARG[1]) ARG[1])))))))))) ARG[0]))
ADF 1:
 (and (or (nand (or ARG[0] (nor (and ARG[2] (and ARG[1] ARG[2]))
ARG[1])) (and (nor ARG[1] ARG[1]) ARG[0])) (nand (or (nor ARG[0] (nor
ARG[2] ARG[1])) (or ARG[2] (nand (nand (and ARG[0] (or ARG[2] (nand
(nand ARG[1] ARG[2]) (or (or (and (or (nor ARG[0] ARG[1]) ARG[0]) (nor
(nor ARG[1] ARG[2]) (nand ARG[1] ARG[1]))) (nand (and ARG[0] ARG[0])
(nand ARG[2] (nand ARG[1] ARG[2])))) ARG[2])))) (nor ARG[0] ARG[1]))
(or (or ARG[0] (nand ARG[0] ARG[2])) ARG[2])))) ARG[2])) (or (nor ARG[1]
ARG[1]) ARG[2]))
```

The first ADF is the Boolean rule 6 or equivalently the XOR or odd-2-parity function. As discovered by Koza in [1], it is frequent for higher order even-n-parity problems with ADFs to come up with parity rules as subroutines. The inclusion of the XOR function makes the parity problem easier to solve for GP and the availability of the XOR function as an ADF makes solving even-8-parity feasible. As Koza stated, the process of solving a higher order parity problem involves decomposing the problem into smaller sub problems and assembling the solutions of the sub problems.

The second ADF is the Boolean rule 203. There is no straightforward explanation for this subroutine except presenting its truth table.

Bit 2	Bit 1	Bit 0	Output
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Table 6 - Truth table for the 3 argument function ADF1.

Even though this 3 argument rule may seem without any practical use, the main body makes use of both ADFs and even calls them hierarchically with each other. ADF0 or the XOR function is called 15 times in the main body and ADF1 is called 5 times. Presumably, since ADF0 is used more often it is a more useful subroutine than ADF1.

The following is the 93 node solution produced by GP+CADF for even-8-parity:

```
Main Body:
 (ADF1[2] (and D6 D6) (ADF0[1] D0 (ADF1[2] D1 D2 (ADF0[1] (nor (nor
(nand D1 D5) (nand D1 D5)) (nor D5 D1)) D3))) (ADF0[1] (ADF0[1] (nor
(and D6 D6) D6) D7) D4))
ADF 0:
 (or (or (and (or (and ARG[1] ARG[1]) (and (and (and ARG[1] ARG[0])
ARG[0]) ARG[1])) ARG[0]) (nor ARG[0] ARG[1])) (or (and ARG[1] (and
ARG[0] (and ARG[1] (and ARG[1] ARG[0])))) (and ARG[1] ARG[0])))
ADF 1:
 (nor (nor ARG[1] ARG[2]) (and (and (and ARG[2] (and (and (and (and
ARG[2] ARG[2]) ARG[2]) ARG[1]) ARG[2])) (and (and (and (and ARG[2]
ARG[2]) ARG[2]) ARG[2]) (and ARG[2] (and ARG[1] ARG[2])))) ARG[2]))
```

The first subroutine, ADF0, can be simplified to the Boolean rule 9, or equivalently, the logical equality (EQ) or even-2-parity function. Recall that this function outputs a 1 when both of its two arguments are equal. Just like XOR, the inclusion of EQ in the function set makes solving the parity problem easier for GP. EQ is also a lower order parity rule that appears as a subroutine in the solution of a higher order parity problem.

The second subroutine, ADF1, is the Boolean rule 60 and can be summarized by the following truth table:

Bit 2	Bit 1	Bit 0	Output
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Table 7 - Truth table for the 3 argument ADF1.

The main body calls both ADFs hierarchically along with the functions in the function set and all input bits in order to obtain a solution. The first ADF is called 4 times, while the second ADF is called twice.

We see that the solutions to even-10-parity are very similar for GP+ADF and GP+CADF. In both cases, the first 2 argument ADF that was found is a parity rule. For GP+ADF it was XOR and for GP+CADF it was EQ. In both cases, the second ADF is a 3-argument function that has no obvious utility but is nevertheless utilized in the final solution; however, it is called fewer times than the first ADF.

4.4.6 Even-11-Parity

For n=11 and higher GP+CADF is the only method that produced a solution after 300 runs with our experimental settings. Figure 9 shows the performance graph of GP+CADF for this problem. A minimum computational effort of 4,248,000 is achieved at generation 58.

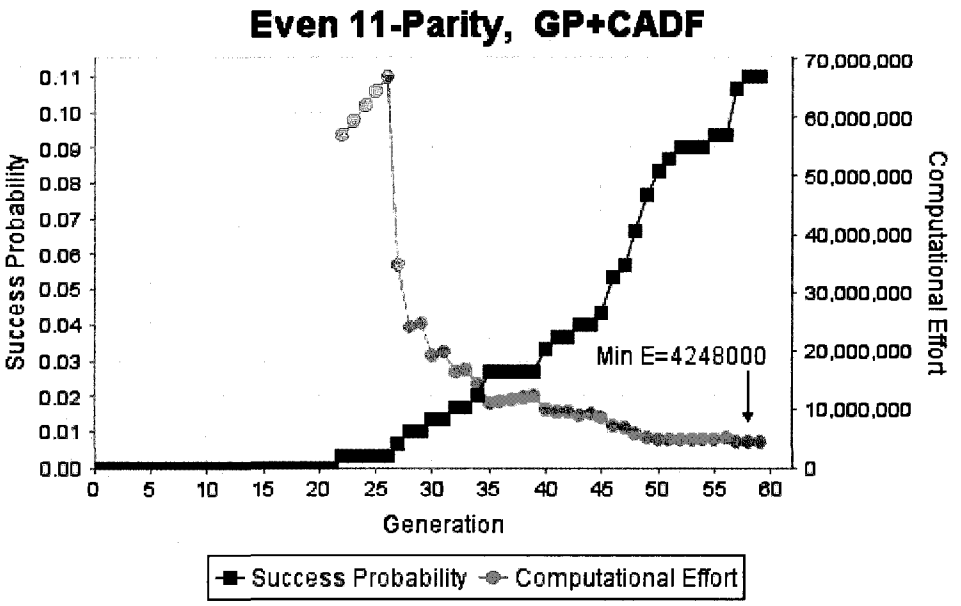

Figure 9 - E_{min} for GP+CADF was 4,248,000 at generation 58 and with success probability 11%.

We may get some insight into how GP+CADF solves high order parity problems by analyzing the solutions. The following is an 83 node solution that was found at generation 58:

```
Main Body:
(ADF0[1] (ADF0[1] (ADF0[1] (ADF1[2] D3 (ADF0[1] (nor D10 D2) D6)
(ADF1[2] (ADF1[2] D4 D6 (and D2 (or D10 D9))) (nand D3 D5) (ADF0[1]
(ADF0[1] D0 (nor (or (or (nor D10 D2) D2) D0) (or D10 D9))) D6)))
(ADF0[1] D5 D1)) (ADF1[2] D7 D1 (ADF0[1] D8 (and D10 D9)))) (nor (and
D10 D9) (nand (nor D10 D2) (nor D9 D9))))
ADF 0:
 (and (nand ARG[0] ARG[1]) (or ARG[0] ARG[1]))
ADF 1:
 (nand (nand ARG[2] ARG[0]) (nand (nand ARG[0] ARG[0]) (nand (nand
(nand (nand ARG[0] ARG[2]) ARG[0]) ARG[0]) ARG[2])))
```

The first ADF is the XOR function or odd-2-parity and is called 8 times. The second ADF is a 3 argument function but the second argument is ignored. It is called 4 times in the main body. When simplified, ADF1 is simply an EQ function for the first and third arguments. That is to say, it outputs a 1 only when the first and third arguments are equal. This time, the solution for even-11-parity incorporated both 2-parity rules XOR and EQ

as subroutines, and the main body assembled them along with the Boolean functions in the function set into the final solution.

4.4.7 Even-12-Parity

This is the last parity problem that could be run 300 trials. Given our practice of evaluating all 2^n fitness cases for even-n-parity, evaluating individuals becomes extremely time-consuming.

Figure 10 shows the performance graph for GP+CADF which tells us that if we run through to generation 59 we would need to process 9,720,000 individuals to obtain a solution with 99% probability for the even-12-parity problem.

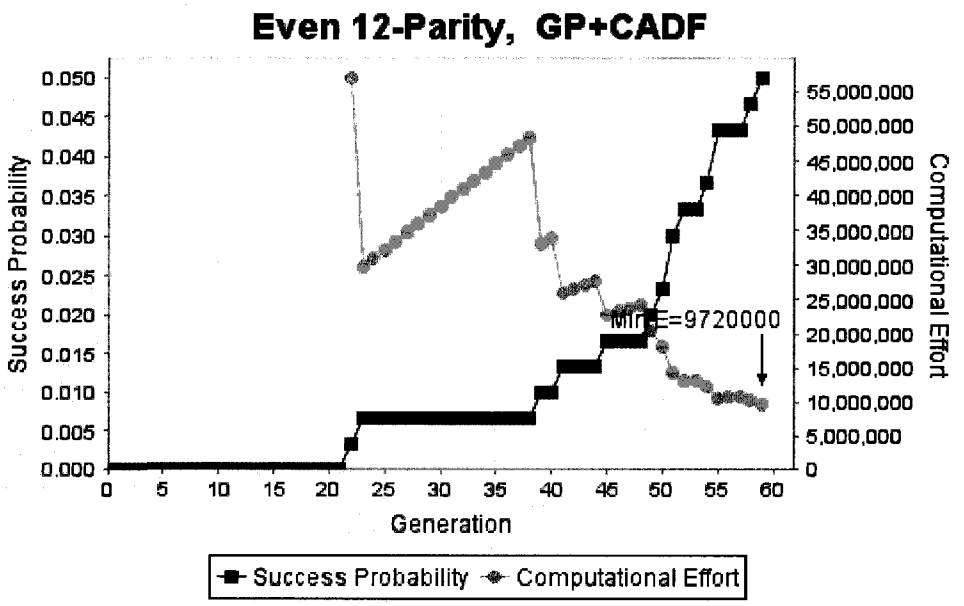

Figure 10 - E_{min} for GP+CADF was 9,720,000 at generation 59 and with success probability 5%.

The following is a 76 node solution that was found at generation 59 that satisfies all 4096 fitness cases:

```
Main Body:
(ADF0[1] (and (ADF1[2] (and D4 (nand D5 D4)) D1 (nand D8 (nor (or D4 D8)
(nor D1 D10)))) (nand D5 D4)) (ADF0[1] (nor (or D7 D8) (nand D2 D7))
(ADF0[1] (ADF1[2] (ADF0[1] (ADF0[1] D6 (ADF0[1] (nor D0 D0) (ADF0[1]
(ADF0[1] D2 (ADF0[1] D8 D3)) (ADF1[2] (ADF0[1] D7 D10) D0 D5)))) D1) D8
```

```
D9) (nand D11 D11))))
ADF 0:
(nand (nand ARG[0] (or (nand ARG[0] ARG[0]) ARG[1])) (or ARG[1] ARG[0]))
ADF 1:
(nor (nor ARG[2] ARG[0]) (and (and ARG[0] ARG[2]) ARG[2]))
```

The first ADF is the EQ function. The second ADF is a 3 argument function but it ignores the second argument. When simplified, it is essentially an XOR function for the first and third arguments.

4.4.8 Even-13-Parity

The total number of fitness cases for even-13-parity is 8192, therefore it was not feasible to continue using 300 trials in an attempt to make the results very statistically significant. Nevertheless, we attempt to analyze the performance with 80 trials. Figure 11 tells us that the minimum computational effort for this problem is 16,380,000 at generation 49. Only 2 solutions were obtained in all the trials.

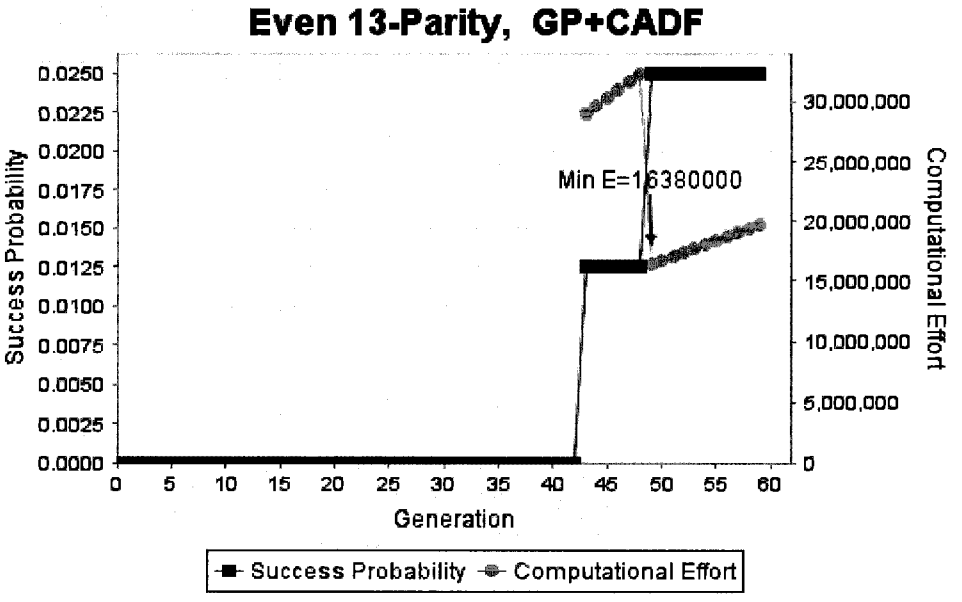

Figure 11 - E_{min} for GP+CADF was 16,380,000 at generation 49 and with success probability 2.5%.

This is the 95 node solution obtained for even-13-parity by GP+CADF:

```
Main Body:
```

```
(ADF0[1] (ADF0[1] (nor D3 D5) (ADF0[1] D9 D2)) (ADF0[1] D0 (ADF0[1]
(and (ADF0[1] D10 (ADF1[2] (ADF1[2] (or D4 D7) (nand D4 (nand D7 D7))
(ADF1[2] (and D3 D7) (ADF0[1] D6 D0) (ADF0[1] D5 D0))) (ADF0[1] (ADF0[1]
D8 (ADF1[2] (ADF1[2] D12 (nor (nand D5 D3) (and (ADF0[1] D12 D9) (nor
D3 D4))) D3) D4 (ADF0[1] D12 (ADF0[1] (nand D5 D3) (ADF0[1] (nor D3 D5)
D11))))) (or D1 (ADF0[1] D1 D1))) (ADF1[2] D10 D2 D12))) (nand (ADF0[1]
D3 D3) (or (ADF1[2] D4 D2 D11) (nor D7 D11)))) (ADF0[1] (ADF0[1] (nand
(ADF0[1] D3 D3) (nor D7 D11)) D6) D11))))
ADF 0:
(and (nand ARG[1] ARG[0]) (or ARG[1] ARG[0]))
ADF 1:
(and (nand (nand ARG[0] ARG[1]) ARG[1]) (nand ARG[0] (nand ARG[0]
ARG[1])))
```

The first ADF is the XOR function. The second ADF is a three argument function. Similar to the second ADF in the even-12 parity problem solution, it ignores one of the 3 arguments and is simply the EQ function for the two arguments that it uses.

4.4.9 Even-14-Parity

This is the last parity problem that will be examined in our analysis. We stop at even-14-parity not because GP+CADF is unable to solve even-15-parity but because

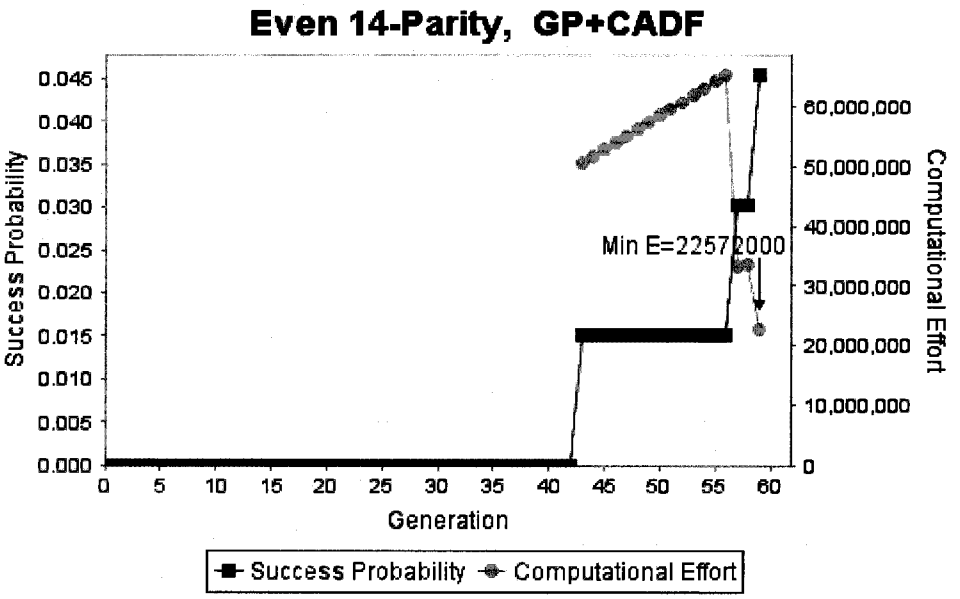

Figure 12 - E_{min} for GP+CADF was 22,572,000 at generation 59 and with success probability 4.5%.

even-15-parity would require 32,768 fitness cases and evaluation would be extremely

time-consuming making it infeasible to accumulate a sufficient number of trials. Figure 12 gives a performance graph for this problem that tells us the minimum computational effort is 22,572,000 at generation 59.

Furthermore, the experimental setup was changed from all the previous problems. We use a larger total population of 3800 with 1400 individuals in the main body population and 1200 in the ADF populations. We also change the second ADF to have a total of 4 arguments. In addition, a total of 66 trials were performed with a total of 3 successes.

The following is a 141 node solution that satisfied all 16,384 fitness cases:

```
Main Body:
 (ADF1[2] (and (nor D4 D10) D4) (ADF0[1] (and D5 D0) (ADF0[1] (ADF0[1]
D4 (nand D11 D6)) (nor D4 D10))) (ADF0[1] D4 D0) (ADF0[1] (ADF0[1] D10
(ADF0[1] D8 (ADF0[1] (ADF0[1] (ADF1[2] (and (nor D10 (and (nor D5 D9)
(and D2 D9))) (nor D4 D10)) (ADF0[1] (and (nor D5 D9) (and D2 D9)) (or
D6 D1)) (nor (nor D8 (or (and D0 D12) (nand D8 D12))) (or (nor D11 D9)
(nand D1 D11))) (ADF0[1] (ADF0[1] D5 D6) (nor D1 D11))) (ADF0[1] D2
(ADF0[1] (ADF0[1] D13 D9) D7))) D12))) D3))
ADF0:
 (and (nor (and ARG[0] ARG[1]) (and ARG[0] ARG[1])) (or ARG[1] ARG[0]))
ADF1:
 (and (or ARG[0] (or ARG[0] (or (or (nor (or ARG[0] ARG[3]) ARG[2]) (or
(or (or (or (and (or (or (or (or ARG[3] ARG[2]) ARG[3]) ARG[3]) ARG[0])
(and ARG[2] ARG[3])) ARG[0]) ARG[0]) ARG[0]) ARG[0])) ARG[0]))) (or
(nor (or ARG[3] ARG[2]) ARG[3]) (and ARG[2] ARG[3])))
```

The first ADF is the Boolean rule 6 or the XOR function. The second ADF is a 4 argument subroutine; however the second argument is ignored. This rule outputs a 1 only when the third and fourth arguments are equal, thus the answer also does not depend on the first argument. Therefore, it is essentially an EQ function for the third and fourth arguments.

We can notice that all solutions for 11, 12, 13 and 14-even-parity developed the XOR and EQ functions as their two ADFs. In addition, the second ADF always had 3

arguments, and in this problem, 4 arguments. In all cases, all but two of the arguments were ignored and the second ADF was simply an XOR or EQ function for those two arguments. Therefore, all of the developed ADFs were 2-parity rules. Furthermore, in all cases the ADFs were useful. There were no identical ADFs, ADFs that recreate an existing function, or intron ADFs that simply return an argument or a 0 or 1 value. Thus, in solving high order parity problems, GP+CADF finds useful ADFs that solve lower order parity problems and discovers the main bodies that assemble those ADFs along with primitive functions into the correct solution.

4.4.10 Conclusion

The GP+CADF method successfully solved the even-n-parity problem up to n=14. We could not proceed further not because GP+CADF was not able to solve even-15-parity, but because increasing evaluation times made it infeasible to accumulate a sufficient number of trials to obtain a successful run. For all values of n, GP+CADF had a lower minimum computational effort than GP+ADF, and for all n>=4 GP+CADF had a lower computational effort than regular GP. The conclusion is that coupling the best main bodies and ADFs with individuals of each population in the GP+CADF method has the benefit of finding a solution with less computational effort than maintaining a single population of complete individuals with GP+ADF. We analyzed solutions and found that GP+CADF is able to discover useful subroutines such as 2-parity rules and couple them with a main body that utilizes their potential and assembles them into the correct solution. Lastly, for n > 10, GP+CADF was the only method able to produce a solution with the settings used in these experiments.

4.5 Effects of Mutation

4.5.1 Introduction

The goal of this section will be to explore the effects of the mutation operator on the GP+CADF method. The motivation is that the effect of mutation with GP+CADF has not been studied before and it is unknown what benefits, if any, we may obtain. Moreover, the effect of mutation has not been dealt with extensively even with the well known GP+ADF method. In the parity and other experiments in [1], Koza does not specify if mutation is used at all, and if it is used, what type and under what settings. In [7], Koza also refers to mutation as a secondary optional operation that is relatively unimportant, while crossover and reproduction are primary. Contrary to this view, we will test empirically what benefits we may obtain with mutation.

In this section, we will explore the effect of combining mutation with GP+CADF and GP+ADF. In particular we will apply the Subtree, Rehang, All Nodes and Swap mutation operations and will determine which operator works best with which method. In addition, we will determine the effect of different mutation probabilities.

4.5.2 Subtree Mutation

Subtree mutation is a mutation operator very similar to the Point mutation operator described in [7]. Essentially, Subtree mutation involves choosing a node within the program tree, and then replacing that sub-tree with a newly grown sub-tree. The node is chosen randomly from the tree with 10% chance it is a terminal and 90% chance it is a non-terminal. The sub-tree is grown using the GROW tree building method described in [1] and [7]. The GROW method essentially picks a random value between 1 and d

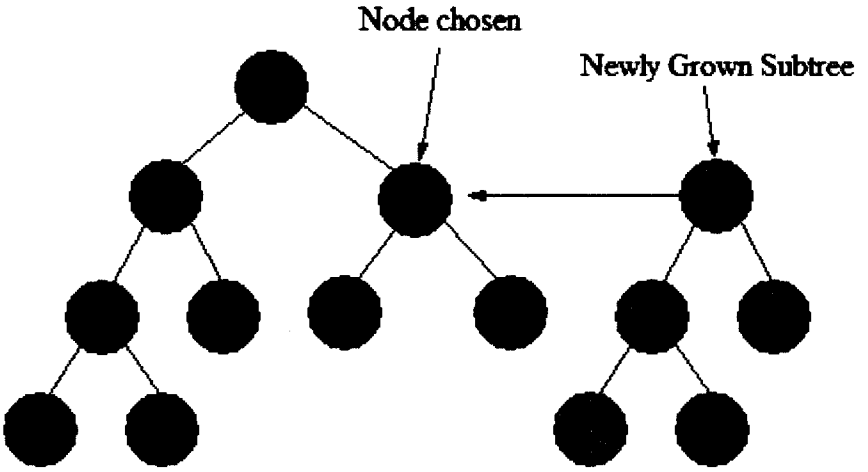

Figure 13 – Subtree mutation involves choosing a node randomly within the tree and then replacing the sub-tree rooted at that node shown in green with a newly grown tree shown in orange.

inclusive and grows a tree to that depth. The standard value for d in [7] is 5. The difference from Koza's implementation is that we have a maximum depth for mutated trees. If after mutation the mutated tree depth exceeds this maximum, we discard the mutated tree and copy the original individual. Figure 13 shows an example of Subtree mutation in action. The newly grown sub-tree will replace the randomly chosen sub-tree.

4.5.3 Rehang Mutation

Rehang mutation is a new mutation operator introduced by Luke [8]. To begin to visualize Rehang mutation, imagine that the program tree is made up of nodes connected by strings all hanging down from the root. We randomly pick a new non-terminal node n and pick it up as the new root letting all the other nodes hang down from it. However, we see that this will not work in such a straightforward way. By picking up a non-terminal node n and setting it as the new root, we are adding its parent p as one of n's children. In addition, n's parent p will have one less child since n becomes p's parent. To solve this problem, Rehang mutation will be performed in a slightly different way.

First, we pick T, a random sub-tree of n and remove it from the program tree. Then, we set n as the new root. The gap left behind by T will be filled by p which is n's parent. There will also be a gap left by n in p's sub-tree. We fill it with p's former parent q. There will also be a gap left by p in q's sub-tree. We fill it with q's former parent r. This process continues until we reach the old root. The root will also have a gap left behind by its child that became its parent. Finally, this gap will be filled with T. Figure 14 is helpful in understanding exactly what happens by showing an example. From Figure 14 we see that the order of nodes in the original tree was q, p, and n when moving toward terminal nodes. In the mutated tree, the order is n, p, and q. However, we also see that only one

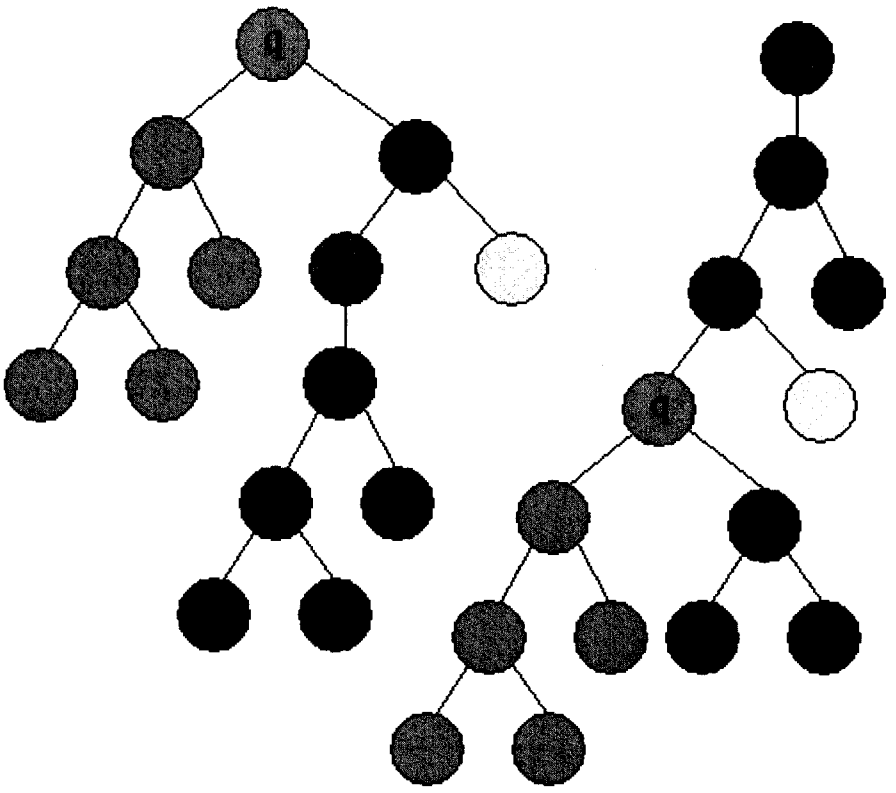

Figure 14 - This is an example of Rehang Mutation. A node n is chosen as the new root along with T, a random sub-tree of n. The gap left behind by T is filled with p, the parent of n. The gap left behind by n in p's sub-tree is filled with q, the parent of p and also the old root. The gap left behind by p in q's sub-tree is finally filled with T.

child is changed in all affected sub-trees. The old root q has its original left child, p has its original right child, and n has all the same descendants except the replaced sub-tree T. Therefore, Rehang mutation has the effect of partially reversing the order of operations and exchanging sub-trees between nodes.

4.5.4 All Nodes Mutation

All Nodes mutation is a mutation operator similar to Subtree mutation and is based on the All Nodes mutation operation described by Kumar and Chellapilla in [9]. A sub-tree is randomly chosen and every node in that sub-tree is replaced by a random node of the same arity. Consequently, the structure of the sub-tree is the same except for the values of the nodes. This is different than the Subtree mutation operator that completely re-grows a sub-tree.

4.5.5 Swap Mutation

Swap mutation as described in [9] is a mutation operator that picks a random swappable node. A node n is swappable if it has at least two children x and y, and the

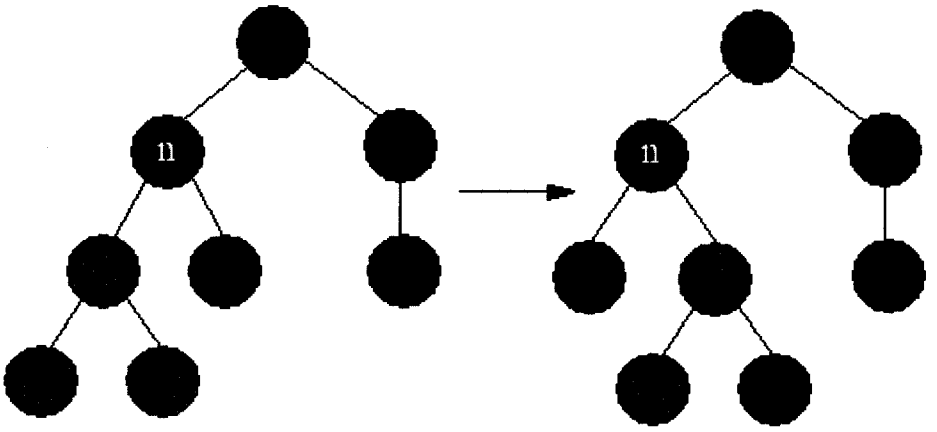

Figure 15 - Swap mutation randomly chooses a swappable node n. Next, it swaps the sub-trees rooted at x and y; two of its children that have compatible return types.

return types of x and y are type-compatible. The sub-trees rooted at x and y are then swapped. Figure 15 shows an example of swap mutation. Note that if n is an operation such as addition or multiplication the mutation will have no effect. However, if n is an operator like division, exponent or an IF, then the mutation can have considerable effect.

4.5.6 Effect of Different Mutation Operators

We use the same settings of Table 2 except we replace the mutation operator by the desired mutation. We performed the experiments on the 8-even-parity problem. The choice of n=8 is because we wanted a higher order parity problem of sufficient complexity. Recall that when we performed the parity experiments for n=3 and n=4, the results were not indicative of the long term trends. However, since we need to perform a lot of experiments for all different methods we do not want to choose n too high because of long evaluation times. In addition, there were no results obtained for n higher than 10 for GP+ADF. Therefore, n=8 or n=7 is a good compromise of problem complexity and evaluation times.

Method	No Mutation	Rehang	Subtree	All Nodes	Swap
GP+ADF	18,018,000	17,690,400	8,100,000	18,468,000	20,295,000
GP+CADF	4,595,400	1,044,000	1,197,000	1,544,400	4,644,000

Table 8 - Experimental results of testing different mutation operators with GP+ADF and GP+CADF. The results given are the minimum computational efforts for the even-8-parity problem.

Primarily, these experiments show that mutation has a very large effect on the computational effort of both GP+ADF and GP+CADF. For GP+ADF, we see that Subtree mutation was the most beneficial mutation and had the largest effect of approximately halving the computational effort. For GP+CADF, Subtree and Rehang mutation had the largest effect reducing the computational effort approximately 4 times.

The only mutation that increased the computational effort for both methods was Swap mutation. Observing the data, we may conclude that mutation is beneficial for genetic diversity because evolution did not stagnate as early as without mutation. This is evidenced by higher success rates and lower computational effort. Lastly, we may conclude that it is recommended to use Rehang or Subtree mutation with GP+CADF when solving parity problems.

These mutation experiments were in fact performed before any of the parity experiments in the previous sections. It was determined that GP+CADF had a lower computational effort than GP+ADF without any mutation as we see in Table 8. Then, once we determined that Rehang mutation was beneficial with GP+CADF, and Subtree mutation with GP+ADF, all the parity experiments were run with their best mutation operators. Note that with each mutation the computational effort for GP+CADF was always lower. To be exact, the computational effort of GP+CADF without any mutation was still lower than any of the GP+ADF runs *with* mutation. Therefore, it is safe to conclude that mutation was not the factor that enabled GP+CADF to have a lower computational effort than GP+ADF. Since it is not feasible to run all the parity experiments under all different settings or a combination of settings, we simply ran with the best setting for each method.

4.5.7 Effect of Mutation Probability

An additional experiment that was performed in analyzing the effect of Rehang mutation on GP+CADF was determining the effect of the mutation probability p_m. The experiments were run with the same settings as Table 2, except that we use the appropriate mutation probability from 0.0 to 1.0 in increments of 0.1. The problem that

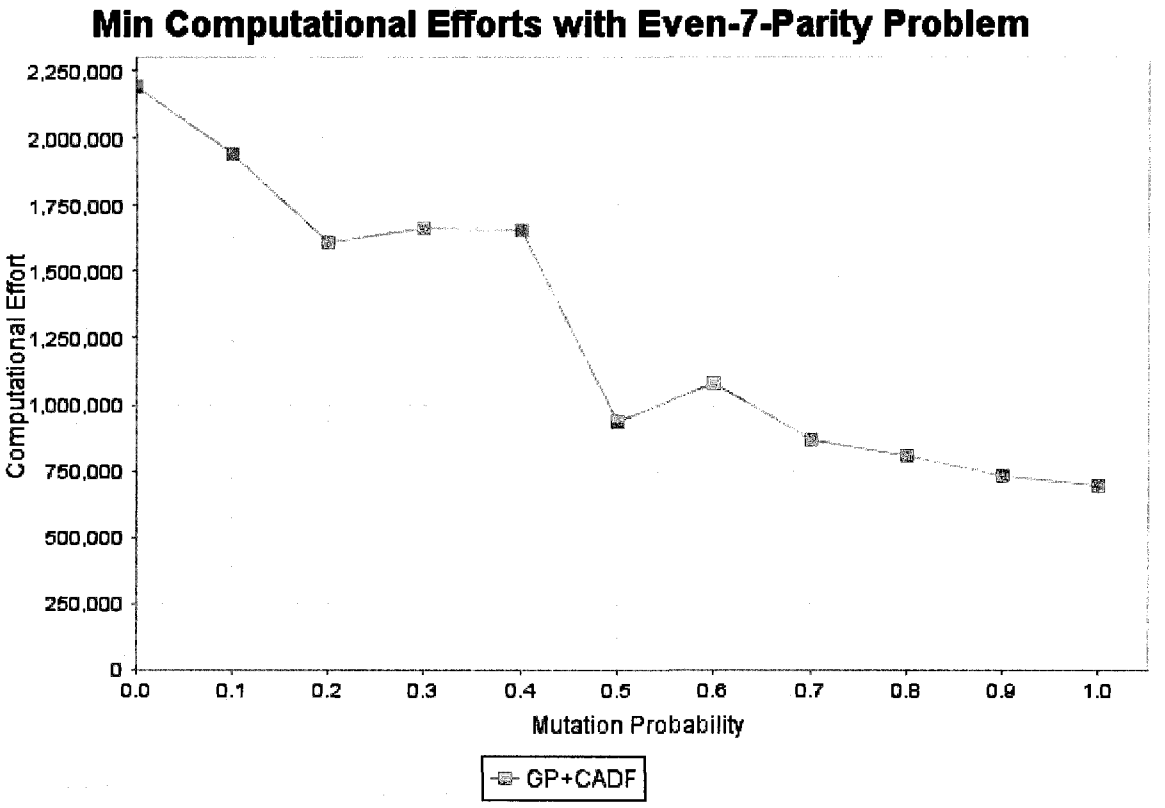

Figure 16 - A graph of mutation probability from 0 to 100% in increments of 10% versus the computational effort. The minimum is with 100% mutation and computational effort of 693,000.

was used to test the rate of mutation was even-7-parity. Figure 16 shows a graph of varying mutation rates and the corresponding computational effort. The graph shows that if Rehang mutation is applied every time an individual is bred (100% mutation), then this leads to the lowest computational effort. The graph drops sharply from E_{min} of 2,194,200 at 0% mutation to 939,600 at 50%. There is a slower drop from 939,600 at 50% to 693,000 at 100%. This suggests that there is an immediate benefit from applying even a low amount of mutation, and this benefit increases at a progressively slower rate as the amount of mutation is increased toward the maximum.

4.5.8 Conclusion

We have been able to determine that Rehang mutation benefited the GP+CADF, and Subtree mutation benefited both GP+CADF and GP+ADF. However, we can not make the assumption that the findings of this section would apply to problems other than parity. Indeed, mutation operators may be problem specific and one technique would not be the best in all cases and there may be other mutation operators optimal for each technique. Nevertheless, we may attempt to improve performance for other problems by adding Rehang or Subtree mutation as breeding operators.

4.6 Improvements for GP+CADF

In section 3.6 we outlined several proposed improvements to GP+CADF. This included using elitism in evaluation, using more than one best individual for evaluation, finding better initial best individuals and generating a better initial population.
In this section we will verify using the even-8 and even-7-parity problem which of these proposed improvements leads to a lower computational effort.

4.6.1 Co-evolutionary Elitism

We will test the even-8-parity problem in two different ways. With elitism we will couple individuals with the best individuals found so far, and without elitism we will

Method	No Elitism	Elitism
E_{min}	11,383,200	1,044,000
$Change_{best}$	168.44	23.23

Table 9 - The results of testing the elitist and non-elitist versions of GP+CADF with the even-8-parity problem. E_{min} is the minimum computational effort and $Change_{best}$ is the mean number of times the best individuals were changed during an evolutionary run. The number of trials was 200 and same evolutionary settings as in Table 2 are used.

couple with the previous generation's best individuals. Results are given in Table 9 which shows that the elitist version by far exceeds the non-elitist version by an order of magnitude. Coupling with the best individuals found in the entire evolutionary run clearly has the advantage. A hypothesis as to why the elitist strategy fares better is that it enables more stability. When we update the best individuals at every generation, these individuals may change more rapidly than when we only use the best of run individuals. We can test this hypothesis by examining how many times the best individuals used for coupling change during an evolutionary run. Table 9 shows the experimental results. The mean number of times the best individuals changed in the elitist version was 23.23 while this value was 168.44 in the non-elitist version. Since the maximum number of times the best individuals in a 3 population, 60 generation system could change is 180 we see that the non-elitist number of changes is close to this value. The difference in means was 145.21. An unpaired t-test tells us that the two-tailed P value is less than 0.0001 which means that the difference is extremely statistically significant.

Since the best individuals in the elitist version change much less frequently than in the non-elitist version, more stability is provided in an evolutionary run. This means that main bodies and ADFs are coupled more frequently with the same best individual and they have more time to evolve to *fit* with these best individuals. A higher success rate and lower computational effort tell us that GP+CADF is finding better fitting individuals with the elitist version. Meanwhile, the frequent changes of the non-elitist version do not give enough time for individuals to evolve to form a good fit.

4.6.2 K-Best Couplings

In the GP+CADF approach as presented so far we evaluate a population by coupling it with only one best individual from all other populations. Instead of just one, we could try coupling it with the k best individuals from the other populations. The benefit is that there would be more diversity in coupling individuals from a population since we are not using just a single best individual. In other words, every individual from every population would be coupled with k best individuals from the other populations, and the fitness would be the best of the k couplings. Effectively, every individual would be given k opportunities to be utilized as a useful main body or ADF. However, the downside is that the number of evaluations increases by the population size for every increment of k. For example, if the sum of all populations is 1800 such as we used in our parity experiments and if we are using single best individuals (k=1), then the total number of evaluations at every generation would be 1800. However, if we are using two best individuals and k=2, then the total number of evaluations at every generation would be 3600. An extreme case would be where k=M, where M is the population size. This means that every individual would be coupled with every other individual from the other populations. Empirically we can determine if increasing k would lower the computation effort. We test with the same settings given in Table 2 for the even-7-parity with 400 trials. The minimum computational efforts for values of k increasing from 1 to 4 are given in Table 10.

There was an increase in the success rate with increasing k. For example, for k=2, the success rate at generation 60 was 67.0% compared with 38.7% for k=1. However, the increase was not enough to lower the computational effort due to the increased number of

k	E_{min}
1	693,000
2	864,000
3	1,382,400
4	1,512,000

Table 10 - Minimum computational efforts obtained when testing the even-7-parity problem with different values of k. k is the number best individuals we use to test all individuals.

evaluations as we can see from Table 10. We may conclude that there is no reduction in computational effort obtained from using more than one best individual for evaluation as far as the parity problem is concerned. The general trend is for E_{min} to increase as k is increased.

4.6.3 Improved Initial Population and Better Initial Best Individuals

In the section 3.6.3 it was suggested that there might be a better way to choose the initial best individuals than just picking them randomly from the initial population. The way that was proposed was to try M (population size) couplings between main body and ADF individuals in the order that they appear in their respective population, and choose those that obtained the highest fitness when coupled together.

In addition, it was suggested in section 3.6.4 that the initial population may be improved by regenerating those main bodies that do not have any ADF calls. The motivation is that since GP+CADF almost entirely relies on finding the best couplings between main bodies and ADFs, it is essential to start off with main bodies that actually utilize ADFs. Initially, it was estimated that only a small percentage of the population would have main bodies with no ADF calls, and that this would not be an improvement

that would have a large impact. However, an investigation of the initial population revealed that the mean percentage of main bodies with zero ADF calls was 48.9 % +- 1.5%. This means that approximately half of the initial main body population will not even reference ADFs! Consequently, with random selection there is approximately a 50% chance of choosing one of these individuals as the best main body. This would imply that in the first generation, the evaluation of all the ADF populations would be completely wasted effort if we select such an individual as the best main body. This is because coupling an ADF with a main body that does not reference an ADF will just return that same main body. Furthermore, the fitness of all the ADFs will be the same, which is the fitness of that best main body with no ADF calls. Due to all individuals having the same fitness we will not be able to perform meaningful fitness proportional selection. It is clear to see that this is a potentially unwanted scenario. Therefore, it is expected that regenerating all main bodies to force them to call ADFs will lead to a considerable improvement.

We will test these two proposed improvements using the even-8-parity problem. The experimental settings used will be the same as those outlined in Table 2. The results of running even-8-parity with and without the better initial best individuals and improved initial population are presented in Table 11. We see that each of the proposed improvements resulted in a lower minimum computational effort for the even-8-parity problem. Together, these two improvements resulted in approximately halving the computational effort.

Initial Best Individuals with M Couplings	Regenerate Main Bodies without ADFs	Even-8-Parity E_{min}
N	N	2,620,800
N	Y	1,539,000
Y	N	1,621,800
Y	Y	1,044,000

Table 11 - Results of running even-8-parity with proposed improvements to the initial population and initial best individuals. When "Initial Best Individuals with M Couplings" is Y that means we choose the initial best individuals after trying M couplings and pick those main bodies and ADFs that obtained the highest fitness when coupled together. When this is set to N, we are choosing the initial best individuals randomly. When "Regenerate Main Bodies without ADFs" is set to Y, we are regenerating all those main bodies in the initial population that do not have ADF calls until they contain at least one. The last column is the minimum computational effort obtained for even-8-parity.

4.6.4 Conclusion

We may conclude based on these experiments that using elitism is beneficial for GP+CADF. Furthermore, based on the parity problem, an evaluation group size of 1 or just a single individual is recommended for the lowest computational effort in GP+CADF. Lastly, we recommend improving the initial population by regenerating those individuals with no ADF calls, and we also recommend choosing the initial best individuals after trying M couplings where M is the population size.

Chapter 5: N-S-Sum of Bits Problem

5.1 Introduction

The n-s-sum of bits problem takes as input a bit string of length n and a desired sum s. There are 2^n possible values for a bit string of length n. The goal is to find all those bit strings where the number of 1's adds up exactly to the sum s. For example, when n is 4, the total number of possible bit strings is 2^4 or 16. Given that s is 2, we are looking for all those bit strings that have exactly two 1's. Therefore, given a bit string of length n and sum s as inputs we want to output a 1 when there are s 1's in the bit string, otherwise 0. Enumerated in Table 1 are all the combinations for bit strings of length 4 with the target bit strings outlined in bold.

Bit	Value	Bit	Value
0	0000	8	1000
1	0001	9	**1001**
2	0010	10	**1010**
3	**0011**	11	1011
4	0100	12	**1100**
5	**0101**	13	1101
6	**0110**	14	1110
7	0111	15	1111

Table 1 – For the 4-2-sum of bits problem, the desired output is 1 for those 4 bit input bit strings with two 1's (outlined in bold).

There are a total of 6 bit strings where the number of 1's is exactly 2. Stated another way there are n choose s (n C s) total bits strings where the number of 1's is s because we are simply choosing s bits to be 1 regardless of order.

We invented this problem specifically for the purpose of comparing the different GP methods with a problem that is even harder than the even-n-parity problem and that is also scalable in the same way. It was claimed in [1] that the Boolean parity problem

was the most difficult Boolean function for GP to learn, however, the results of this chapter will show the sum of bits problem is even harder to learn for GP with equal values of n.

5.2 Problem Difficulty

A useful consequence is that we can potentially adjust the difficulty of the problem by changing s and keeping n constant. The hypothesis is that the difficulty of the problem is directly proportional to (n C s) as we will show in section 5.4. Figure 1 shows how the total number of bit strings with sum s changes with s.

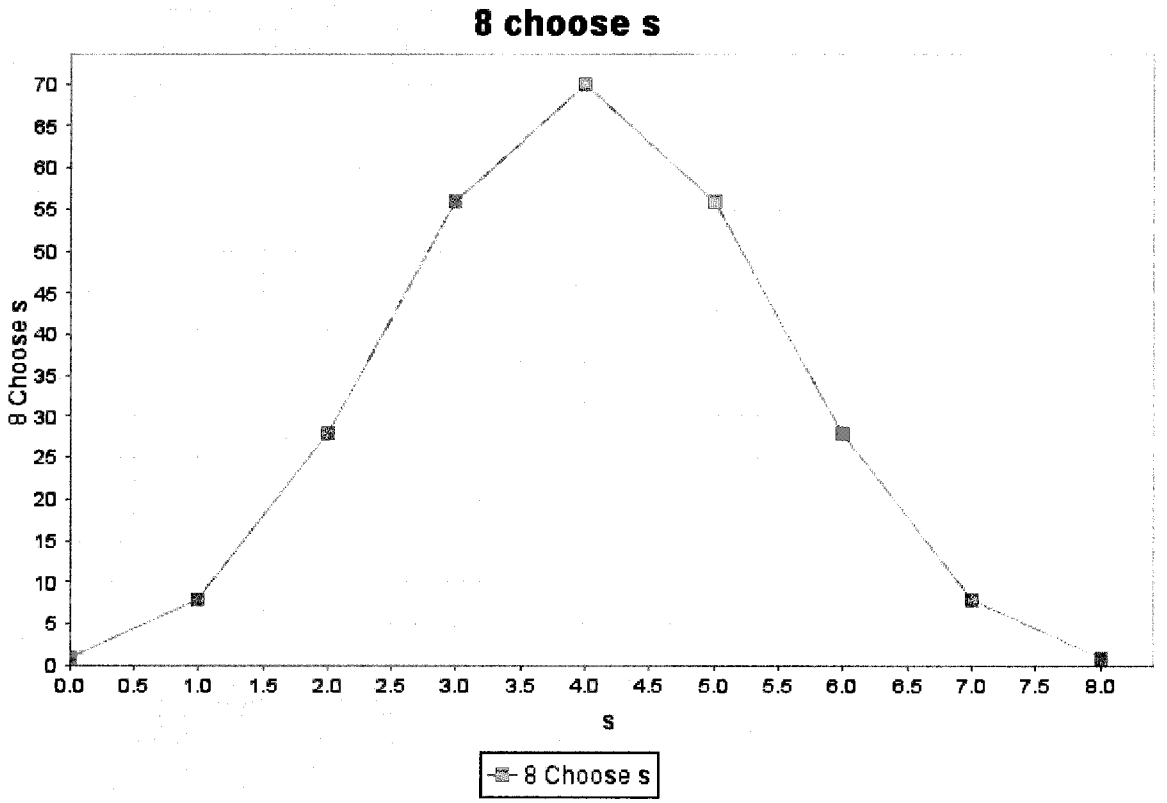

Figure 1 - A graph showing how 8 choose s changes with s.

The peak value is when s is half the value of 8. Intuitively this makes sense because there are a lot more ways to choose 4 bits out of an 8 bit string then 0 out of 8 or

8 out of 8. It is hypothesized that the hardest problem is one where the value of n choose s peaks because we have the most bit strings to differentiate. For example, when we choose k as 0 or 8 there is only one bit string that satisfies this condition 00000000 and 11111111 respectively. Therefore it should be relatively straightforward for GP to come up with a solution that recognizes this unique bit string. Conversely, if k is 4 we have 70 bit strings that satisfy this condition. It should be much harder for GP to single out all 70 of these bit strings.

This would allow us to test if and how the benefits of the ADF and CADF approaches increase with increasing problem difficulty. In addition, we will be able to see at what point using subroutines has an advantage over the regular GP approach. Furthermore, the difficulty can be adjusted without necessarily increasing the number of bits as high as in the parity problem. Recall that raising the number of bits by one doubles the total evaluation time since the total number of fitness cases to be evaluated is 2^n.

5.3 Experimental Setup

Table 2 summarizes the experimental setup for this problem.

Problem	n-s-sum of bits
Objective	Find a program that outputs 1 if the sum of bits in an n bit string is s, 0 otherwise
Number of bits (n)	3 to 6
Sum of bits (s)	1 to n
Population Size (M)	1800, 2700
Generations (G)	60
Runs	200, 300
CADF Populations	3
Mutation	Subtree Mutation (`ec.gp.koza.MutationPipeline`)
Crossover	Subtree Crossover (`ec.gp.koza.CrossoverPipeline`)
Number of ADFs	3
Function Set (no ADF)	D0 to D5, AND, NAND, NOR, OR
Function Set (with ADF) Main Body	D0 to D5, AND, NAND, NOR, OR, ADF0, ADF1
Function Set for ADF0	AND, OR, NAND, NOR, ARG0, ARG1
Function Set for ADF1	AND, OR, NAND, NOR, ARG0, ARG1, ARG2
Fitness Cases	All 2^n possible bit strings
Fitness	2^n – correct outputs
Hits	correct outputs
Success Predicate	0 for n = 3-5 and 2 for n = 6

Table 2 - Experimental setup for the n-s-sum of bits problem.

The bit string size will be varied from 3 to 6 because this range is sufficient to test a relatively straightforward problem of 3 bits to the much harder 6 bit problem where tests indicated that regular GP produced no solutions after 200 runs. The sum of bits s will be varied from 1 to n for a bit string of length n. The reason for this complete testing is to verify if we can adjust the difficulty of the problem by changing s and keeping n constant so we do not have to test more computationally intensive problems with bits strings longer than length 6. The population size chosen was 1800 for problems with n

from 3-5 and 2700 for problems where n is 6 in an effort increase the success probability for the highest difficulty problem.

Furthermore, to make the tests feasible it was decided that 300 runs will be used for n=3 and 200 runs for all tests thereafter since the computation time doubles for every increase in bit string size.

In addition, the same function set as in the parity problem will be used with input bits and the Boolean functions AND, NAND, NOR, OR which are computationally complete. Therefore, the function set for regular GP without ADFs is

F = {D0, D1, D2, D3, D4, D5, AND, NAND, NOR, OR}

with an argument map of

{0,0,0,0,0,0,2,2,2,2}.

In the case with ADFs the function set for the main body is

$F_{MainBody}$ = {D0, D1, D2, D3, D4, D5, AND, NAND, NOR, OR, ADF0, ADF1} with an argument map of

{0,0,0,0,0,0,2,2,2,2,2,3}.

Therefore we have a total of 2 ADFs, one with 2 arguments and one with 3 arguments. It was decided that the ADFs will not have access to the input bits which can only be optionally passed as parameters to the ADFs by the main body.

Therefore the function set for ADF0 is

F_{ADF0} = {AND, NAND, NOR, OR, ARG0, ARG1}

and the function set for ADF1 is

F_{ADF1} = {AND, NAND, NOR, OR, ARG0, ARG1, ARG2}

5.4 Test Results

Table 3 summarizes the raw results obtained from all the tests performed on this problem.

n	M	s	Success Predicate	Runs	Regular GP		GP+ADF		GP+CADF	
					SP	G	SP	G	SP	G
3	1800	1	0	300	100.00	2.45	93.67	7.12	96.67	4.39
3	1800	2	0	300	100.00	2.37	96.33	7.07	96.67	4.47
3	1800	3	0	300	100.00	0.00	100.00	0.00	100.0	0.00
4	1800	1	0	200	98.60	10.62	48.00	23.30	69.00	13.85
4	1800	2	0	200	75.50	21.20	21.00	31.20	42.50	24.20
4	1800	3	0	200	98.50	10.73	39.00	22.05	72.00	13.60
4	1800	4	0	200	100.0	0.00	100.00	0.25	99.50	0.83
5	1800	1	0	200	60.00	27.22	4.00	35.13	20.50	27.07
5	1800	2	0	200	6.00	44.67	No Solution		6.00	33.33
5	1800	3	0	200	5.50	45.10	1.50	46.00	9.00	31.33
5	1800	4	0	200	67.50	24.80	7.50	32.47	22.00	23.30
6	1800	1	2	200	82.50	21.28	18.00	36.00	27.00	23.52
6	1800	2	2	200	No Solution		No Solution		1.00	34.50
6	2700	3	2	200	No Solution		No Solution		1.50	51.67
6	2700	4	2	300	No Solution		No Solution		2.50	35.60

Table 3 – Experimental results. n and s are the values of the n-s-sum of bits problem. M is the population size. Runs is the number of trials performed for each test. SP indicates success probability and G indicates the average generation at which the solution was found. There are columns for each of the three methods regular GP, GP+ADF and GP+CADF.

5.5 3-s-sum of bits (s = 1 to 3)

Tests show that regular GP outperforms both GP+ADF and GP+CADF. The conclusion is that the problem with n=3 is simple enough for regular GP to solve readily without needing to use subroutines. E_{min} for n=3 and s=1 was 10,794 for regular GP, 53,970 for GP+ADF and 37,779 for GP+CADF which indicates how many individuals need to be processed to produce a solution with 99% probability. The results were identical for s=2. The case where the sum is 1 and 2 show worse performance then when

the sum is 3 confirming the assumption that problem complexity peaks when s is half the value of n. Furthermore, since 3 is odd the half value of 3 is either 1 or 2 and the results obtained were identical for s=1 and s=2.

The solution produced by regular GP was the following 67 node program for n=3 and s=1:

```
(and (or (and (nand (or D2 D2) (or D2 D2)) (and (and D0 D2) (and D0
D2))) (or (nor (and D0 D1) (or (and D0 (or D0 (nor D2 D2))) (and D1
D1))) (nand (or D2 D0) (or D1 D2)))) (nand (nor (or (and D0 D0) (or D2
D2)) (or (or D1 D0) (or D1 D1))) (or (nand D0 (and D1 D2)) (or D0 (nor
D2 D2)))))
```

Using the Boolean operators, regular GP was able to find the combinations of the inputs that have only one bit out of D0, D1 and D2 set to 1.

GP+ADF solution was the following 55 node program for n=3 and s=1:

```
Tree 0:
 (nand (nand (ADF0[1] (nand D0 D1) (and D2 D2)) (ADF0[1] (or D1 D2)
(ADF1[2] D1 D2 D1))) (or (or (nand D2 D1) (nand D0 D2)) (ADF0[1]
(ADF1[2] D1 D1 D1) (nor D2 D0))))
Tree 1:
 (and (nor (nor ARG[0] ARG[0]) (nand ARG[0] ARG[0])) (and (nand ARG[1]
ARG[1]) (and ARG[1] ARG[0])))
Tree 2:
 (nand (nand ARG[2] ARG[0]) (and ARG[0] ARG[0]))
```

The first subroutine is a two argument ADF and, interestingly, is just a zero function meaning it outputs zero for all inputs as shown in Figure 2. Even though it is not a useful function, ADF0 is still used in the solution. For example, in the main body,

Input	Output
0 0	0
0 1	0
1 0	0
1 1	0

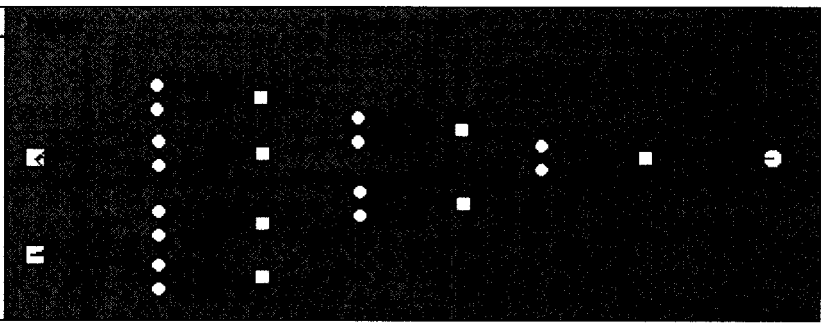

Figure 2 - Circuit for the subroutine ADF0 for n=3 s=1 problem, essentially a zero function.

ADF0 is used as a second argument to an OR function which is equivalent to just the first argument alone. This is an example of an intron [11] that does not do anything useful or change the output in any way.

The second subroutine circuit shown in Figure 3 is a 3 argument ADF that produces false only when the first argument is 1 and the third is 0 (the second argument is not used). Even though this function is not something a human programmer would intuitively use, it is used heavily in finding the solution.

The main body uses both ADFs and even calls them hierarchically one within another. Consequently, even though the number of nodes is less in the ADF solution than regular GP (55 and 67) the ADF solution is more complex when considering the hierarchical calls of ADFs within the main body.

The conclusion is that for a problem at this level of complexity, using subroutines produces a more complex solution and there is less computational effort when subroutines are not used. This case also highlights the fact that the use of ADFs can actually hinder the success rate of finding a solution where the complexity of the problem is not enough to make subroutines useful in reducing the computational effort.

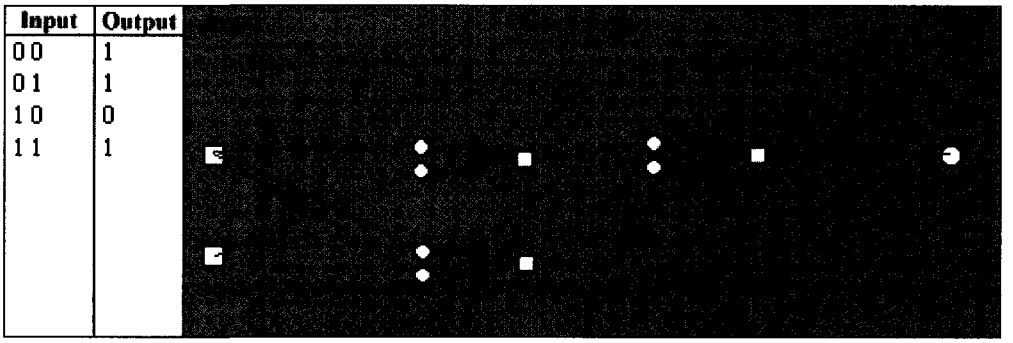

Figure 3 - The circuit for the subroutine ADF1 for the n=3, s=1 problem.

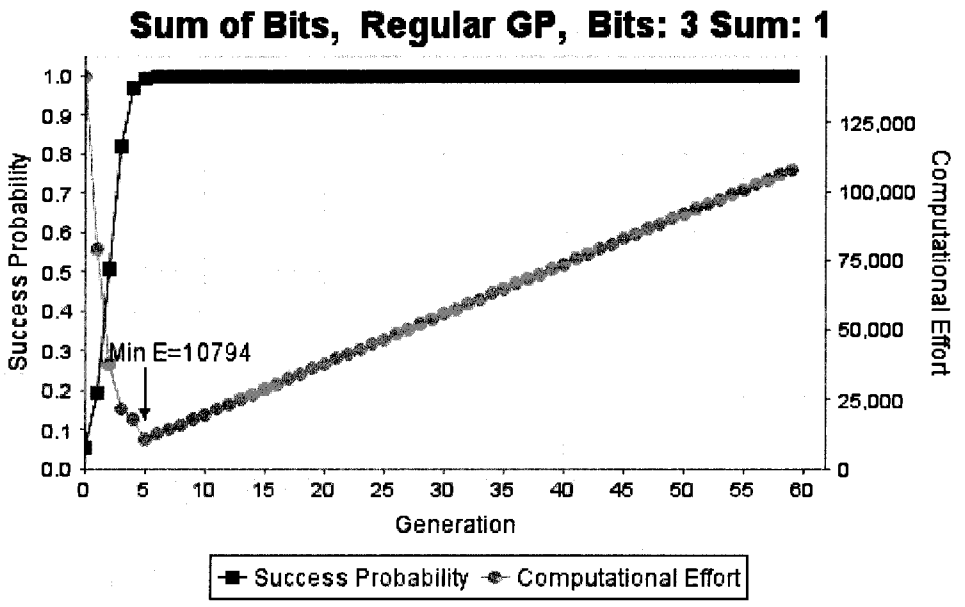

Figure 4 - Performance graph for regular GP shows minimum computational effort is 10,794 at generation 5 with success probability 100%.

However, it is interesting to note that GP+CADF still slightly outperforms GP+ADF.

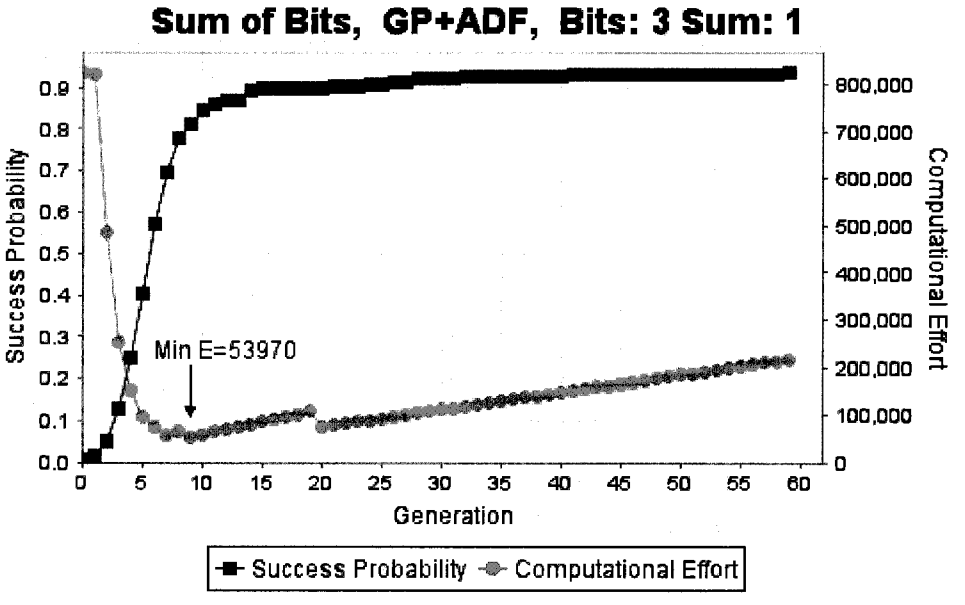

Figure 5 - Performance graph for GP+ADF shows minimum computational effort is 53,970 at generation 9 with success probability 81%.

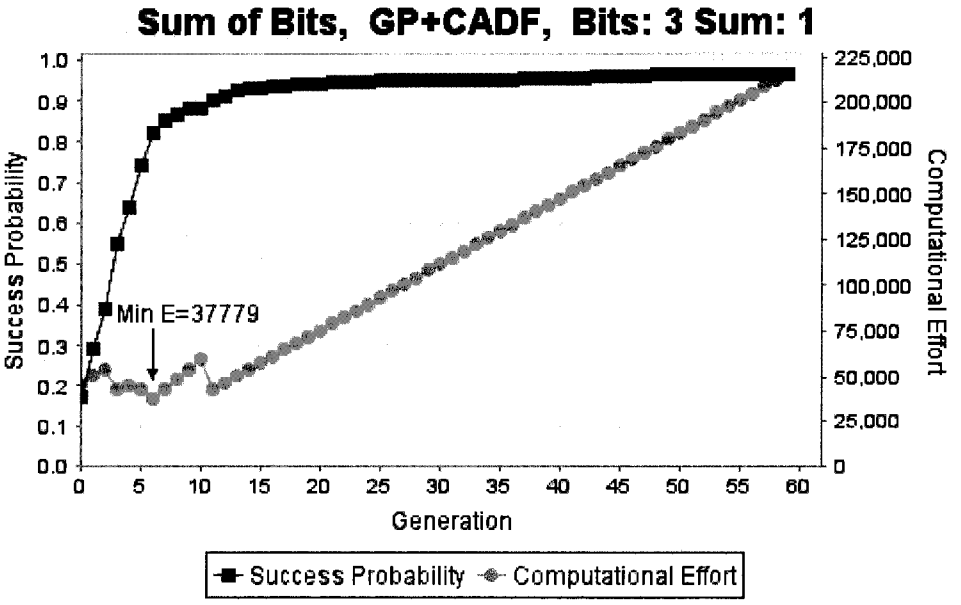

Figure 6 - Performance graph for GP+CADF shows minimum computational effort is 37,779 at generation 6 with success probability 82%.

This is first evidence in this problem that less computational effort is required when matching newly evolved ADFs and Main Bodies to the best ones found so far in different populations with GP+CADF than to search though a single larger population with GP+ADF.

Lastly, the simplest case where s=3 means that we only need to check if all bits are 1. This yielded a 100 % success rate for all three methods.

5.6 4-s-sum of bits (s = 1 to 4)

The case where the bit string length is 4 showed results very similar to the previous section.

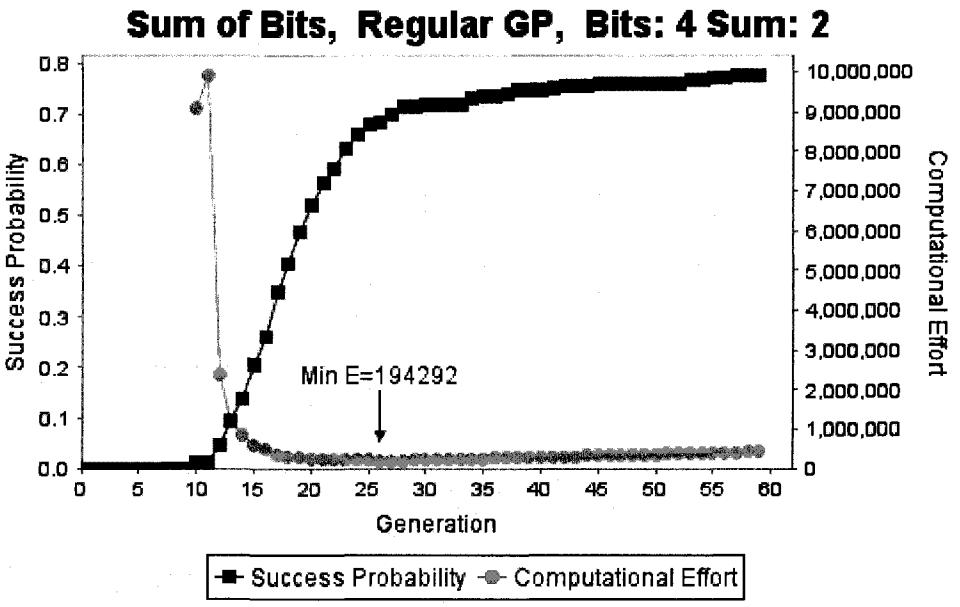

Figure 7 - Performance graph for regular GP shows minimum computational effort is 194,292 at generation 26 with success probability 68%.

The most complex problem was when s was half the value of n which is 2.

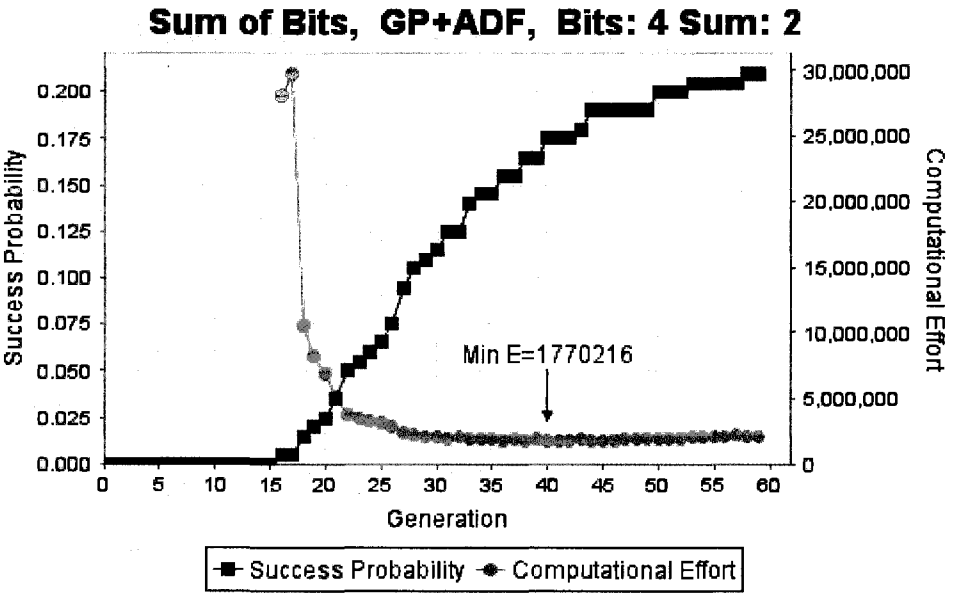

Figure 8 - Performance graph for GP+ADF shows minimum computational effort is 1,770,216 at generation 40 with success probability 18%.

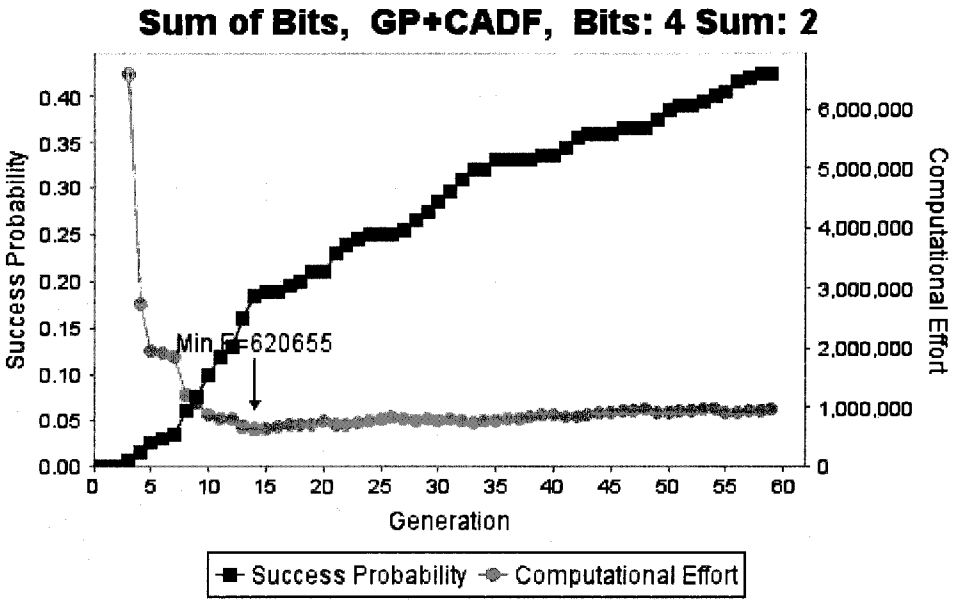

Figure 9 - Performance graph for GP+CADF shows minimum computational effort is 620,655 at generation 14 with success probability 18%.

The problem complexity was such that regular GP still outperformed both GP+ADF and GP+CADF but GP+CADF outperformed GP+ADF. Figures 5, 6 and 7 show that E_{min} is 194,292 for regular GP, 1,770,216 for GP+ADF and 620,655 for GP+CADF respectively.

Therefore, there is empirical evidence that GP+CADF required less computational effort then GP+ADF even in cases where using subroutines does not give an advantage over regular GP. Further sections will explore what happens as the complexity of the problem increases even further with bit strings of length 5 and 6.

5.7 5-s-sum of bits (s = 2 to 3)

This is the first problem where the problem complexity is sufficient such that GP+CADF overtakes regular GP with lower computational effort. Figures 8, 9 and 10 show that the number of individuals to be evaluated to yield a solution with 99% probability is 8,851,080 for regular GP, 29,080,835 for GP+ADF and 3,648,372 for

GP+CADF respectively.

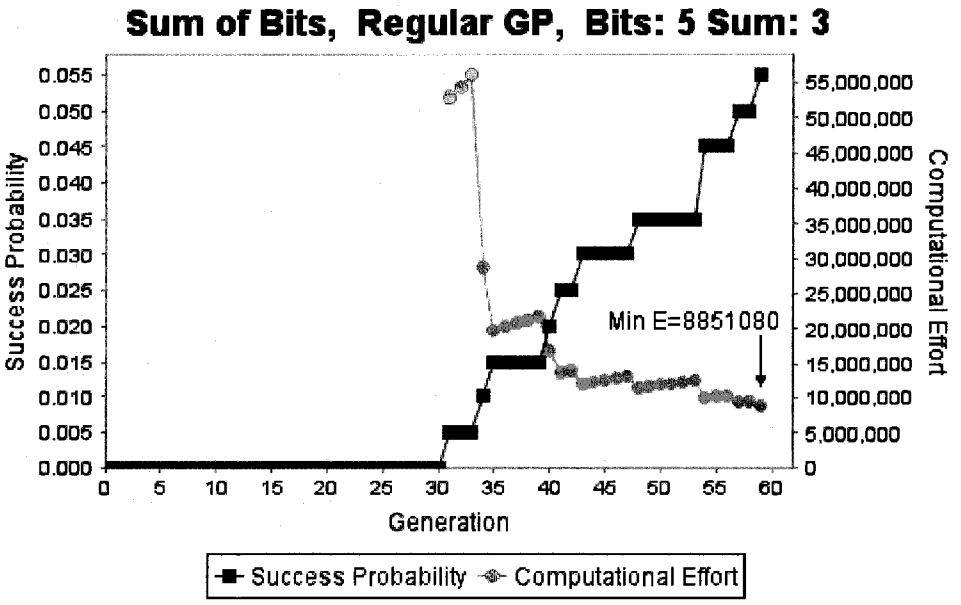

Figure 10 - Performance graph for regular GP shows minimum computational effort is 8,851,080 at generation 59 with success probability 6%.

GP+CADF also outperformed regular GP for n=5 and s=2 with E_{min} 6,911,758 and 7,825,650 respectively. However, regular GP had the best performance for the least complex problems with s=1 and s=4. This is another indication that the co-evolutionary ADF method has an advantage over the regular GP approach as the problem complexity increases.

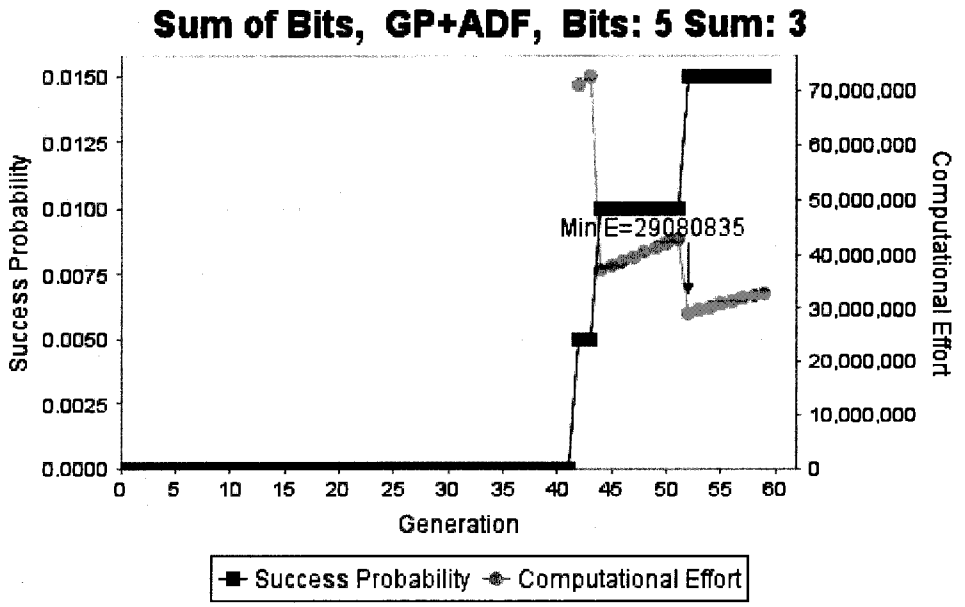

Figure 11 - Performance graph for GP+ADF shows minimum computational effort is 29,080,835 at generation 52 with success probability 2%.

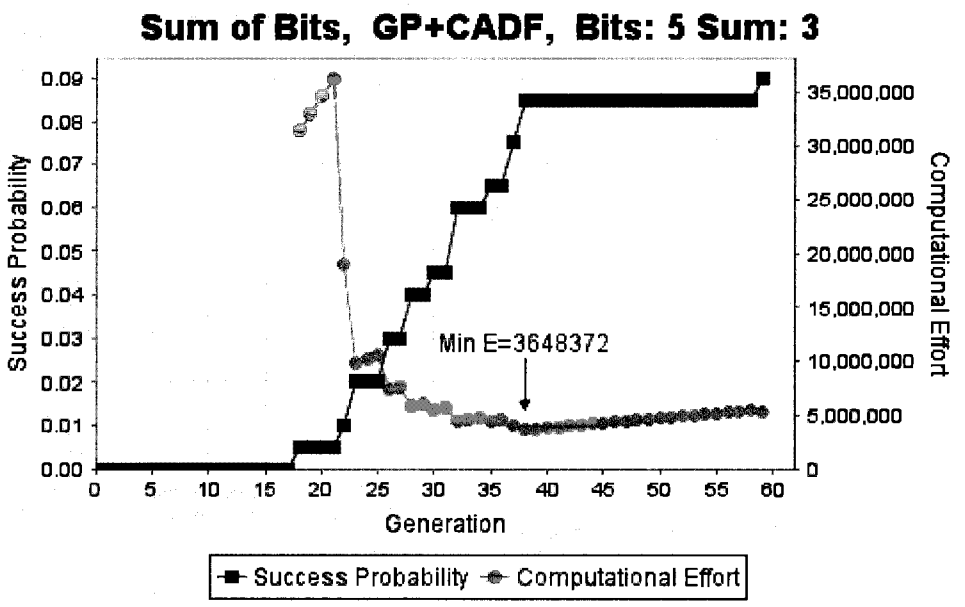

Figure 12 - Performance graph for GP+CADF shows minimum computational effort is 3,648,372 at generation 38 with success probability 8%.

5.8 6-s-sum of bits (s = 1 to 4)

The highest complexity problem was for a bit string of length 6. Increasing the problem complexity further with bit strings of length 7 and higher would make the problem infeasible to run with a large number of trials. Moreover, the complexity of problem increased at such a rate that there were no solutions found with regular GP and GP+ADF after 200 trials for s ranging from 2 to 4 so no solutions would be expected for even harder problems of bit strings 7 and higher. In addition, since the problem complexity increased so suddenly from 5 bits, a success predicate of 2 was allowed for this problem. That means that the final solution is allowed to make two mistakes or misclassifications and still be considered a successful solution. Equivalently, the minimum raw fitness required is 2 for the solution to count as a success.

Figures 11 and 12 show that GP+CADF was the only method to produce a solution for s=3 and s=4 with computational effort 49,391,700 and 26,342,240.

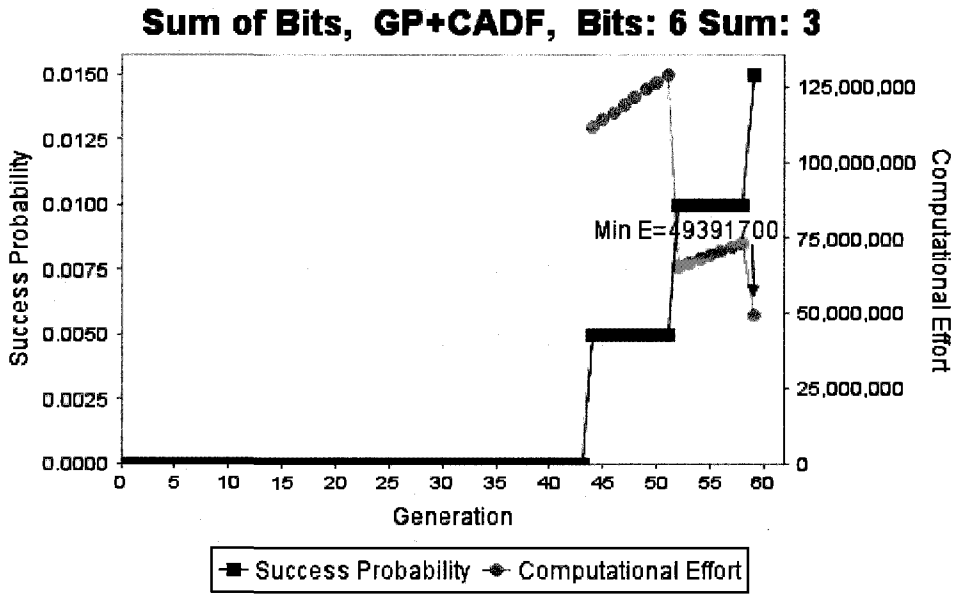

Figure 13 - Performance graph for GP+CADF shows minimum computational effort is 49,391,700 at generation 59 with success probability 2%.

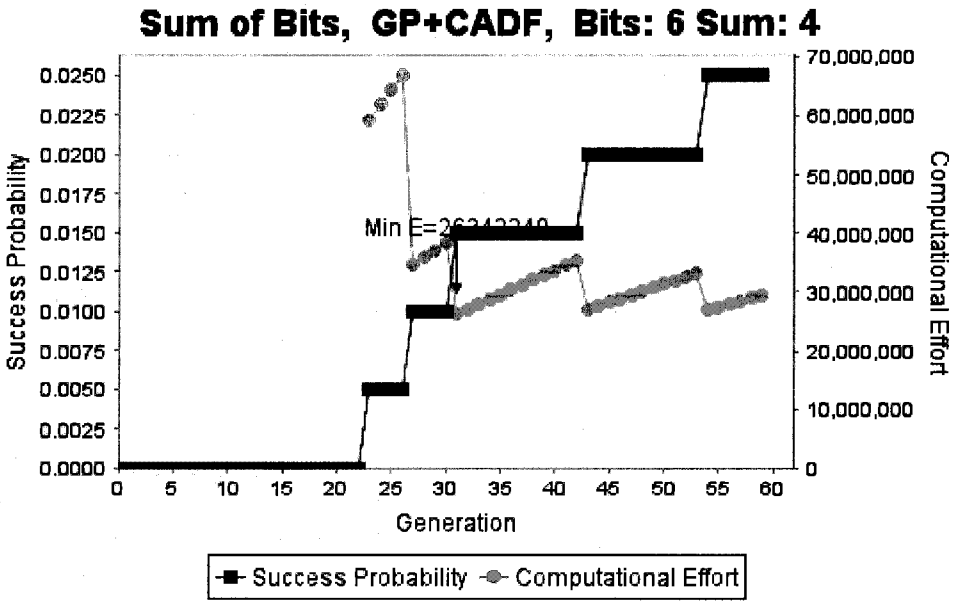

Figure 14 - Performance graph for GP+CADF shows minimum computational effort is 26,342,240 at generation 31 with success probability 2%.

As expected, the highest computational effort required for any of the problems was for n=6 and s=3 where 20 possible bit strings exists out of a total of 64 where the sum of 1's is 3.

The following was one of 3 total successes out of 200 runs for GP+CADF that was found at generation 52 and has 372 nodes for n=6 and s=3:

```
Main Body:
 (and (nand (ADF1[2] (or (ADF1[2] D2 D3 D5) (nor D0 D1)) (ADF1[2]
(ADF1[2] (ADF0[1] D3 D4) D5 D1) D0 D2) D4) (or (ADF1[2] D2 (and D1 D0)
(ADF1[2] D3 D0 D1)) (nor (ADF0[1] D4 (nor (ADF1[2] (ADF1[2] (ADF1[2]
(nor D2 D1) (and D3 (or (nand D1 (ADF1[2] D3 D5 D4)) (ADF0[1] (ADF1[2]
D3 D3 D3) D2))) (ADF1[2] (and D5 D5) (ADF1[2] (ADF1[2] D5 D3 D5) D1
(nor D2 D0)) D4)) D5 D1) D0 D2) D1)) (nand D5 D4)))) (nor (ADF1[2]
(nand (or (and (ADF1[2] (or D5 D0) D3 D5) (or D2 D4)) (ADF0[1] D2 D3))
D2) (nor (ADF1[2] D1 (and (or (ADF1[2] D5 D4 D2) D3) D4) D0) (ADF1[2]
D5 (ADF1[2] D4 D0 (ADF1[2] D4 D3 D3)) (and D1 (ADF1[2] D2 D0 D4))))
(nor D2 (nand D1 D2))) (and (nand (or D2 D4) (or D4 D1)) (nand (ADF1[2]
D3 D0 D5) (nand D1 D2)))))
ADF0:
 (and (nand (and (nor (nand (nand (or ARG[0] (nor (nor (or ARG[1] (or
(and ARG[1] ARG[0]) (nor ARG[1] ARG[0])))) ARG[0]) ARG[0])) (and (nand
(nand ARG[1] ARG[1]) (and ARG[0] ARG[1]))) (nand (nor ARG[1] ARG[1])
(and ARG[1] ARG[0]))))) ARG[1]) (or (or ARG[0] ARG[0]) ARG[1])) (nand
(nand ARG[0] ARG[0]) (and (nand (or ARG[0] ARG[1]) (nand (nor ARG[1]
```

```
ARG[0]) (nand (or ARG[0] ARG[0]) (and ARG[0] ARG[1])))) ARG[0]))) (nand
ARG[0] ARG[0])) (and ARG[0] ARG[1]))
ADF1:
 (or (and (nand (nor (and ARG[2] ARG[0]) (and ARG[0] ARG[1])) (nand
ARG[1] ARG[2])) (or (nor (and (nand (nor ARG[1] ARG[2]) (and ARG[0]
ARG[1])) ARG[2]) ARG[1]) (or (nand ARG[0] (nor ARG[0] (nand ARG[0] (and
(or (nand (nor ARG[2] ARG[2]) (nand (nor (nand ARG[2] (nand (and ARG[2]
ARG[2]) (and ARG[1] ARG[1]))) ARG[0]) ARG[0])) ARG[2]) (and ARG[1]
ARG[0])))))) ARG[0])))) (and (or (nand ARG[1] ARG[2]) (and ARG[2] (nor
(and (and ARG[1] ARG[0]) (nor ARG[1] (and ARG[2] ARG[2])))) ARG[2])))
(and (nand (or ARG[2] (nand (nand ARG[2] (and ARG[0] (and (or ARG[0]
ARG[0]) (and (and (or ARG[1] ARG[2]) (and ARG[2] ARG[1])) (nand (nand
(nor (or ARG[2] ARG[0]) (and ARG[0] ARG[0])) ARG[0]) ARG[1])))))) (or
(or ARG[1] (nand (nor (and ARG[0] ARG[1]) ARG[1]) ARG[1])) (nand (nor
ARG[1] ARG[0]) ARG[1])))) (nand (and (nand ARG[0] ARG[2]) (nor (and (or
ARG[1] ARG[0]) ARG[2]) ARG[0])) (nor (or ARG[1] (or (nor ARG[2] (and
ARG[0] (nor ARG[1] ARG[1]))) ARG[1])) (nor ARG[0] ARG[1])))) ARG[2])))
```

The complete solution is quite complex and one of the subroutines is even larger than the main body. Examining the circuit for the first ADF (ADF0) we see that it is essentially a recreation of the Boolean AND function that was available in the function set. Although this ADF is not useful at all since AND is available in the function set, it is not entirely unexpected for GP+CADF to recreate an available function. Once this ADF is created it can be used interchangeably with AND which is what this solution did, for example when `(ADF0[1] D3 D4)` is called in the main body it is simply an AND of D3 and D4.

The second ADF (ADF1) is a three argument ADF that can be summarized as outputting 1 when two or more of its arguments are 1's, or in other words, it outputs zero when exactly one of its arguments is 1. By combining ADF1 with the other Boolean functions and itself and using ADF1 directly on the inputs, a successful solution was found by GP+CADF. ADF1 was heavily used in finding this GP solution and is called a total of 23 times as opposed to ADF0 which is called only 4 times.

It is not immediately clear how this solution determines if exactly 3 bits are 1 in

the input bit string. However, since the original goal was to find if the bit string contains exactly three 1's we can intuitively see how a 3 argument function that determines if two or more of its arguments are 1 could be useful. By taking the NOT of this ADF we can determine if exactly 1 bit out of 3 is a 1. Then if this ADF looks at different subsets of the input bits it is essentially breaking the bigger 6 bit problem into smaller 3 bit sub problems. This is evidenced by the fact that ADF1 is called multiple times and references all input bits. Then, by combining the solutions to the sub problems it solves the larger 6 bit problem.

The conclusion is that GP+CADF found a solution using a useful subroutine that is not possible with regular GP. In addition, the GP+CADF approach found the correct coupling between ADFs and main bodies that the GP+ADF approach did not discover.

5.9 Conclusion

Results show that as the problem complexity increases in the sum of bits problem, the GP+CADF approach requires less computational effort than regular GP and GP+ADF. In other words, the minimum computational effort increases at a lower rate with GP+CADF than with regular GP and GP+ADF. Figure 15 illustrates this point. We see the problem complexity increasing along the x-axis for every increment of n (the number of bits). We also see a peak of E_{min} when s, the sum of bits, is at a half way point for each n. When n is 3 and 4, regular GP has the lowest E_{min}. When n is 5, GP+CADF overtakes regular GP and when n is 6 GP+CADF is the only approach that produces a result. The y-axis of the graph was made logarithmic to make it possible to show all differences.

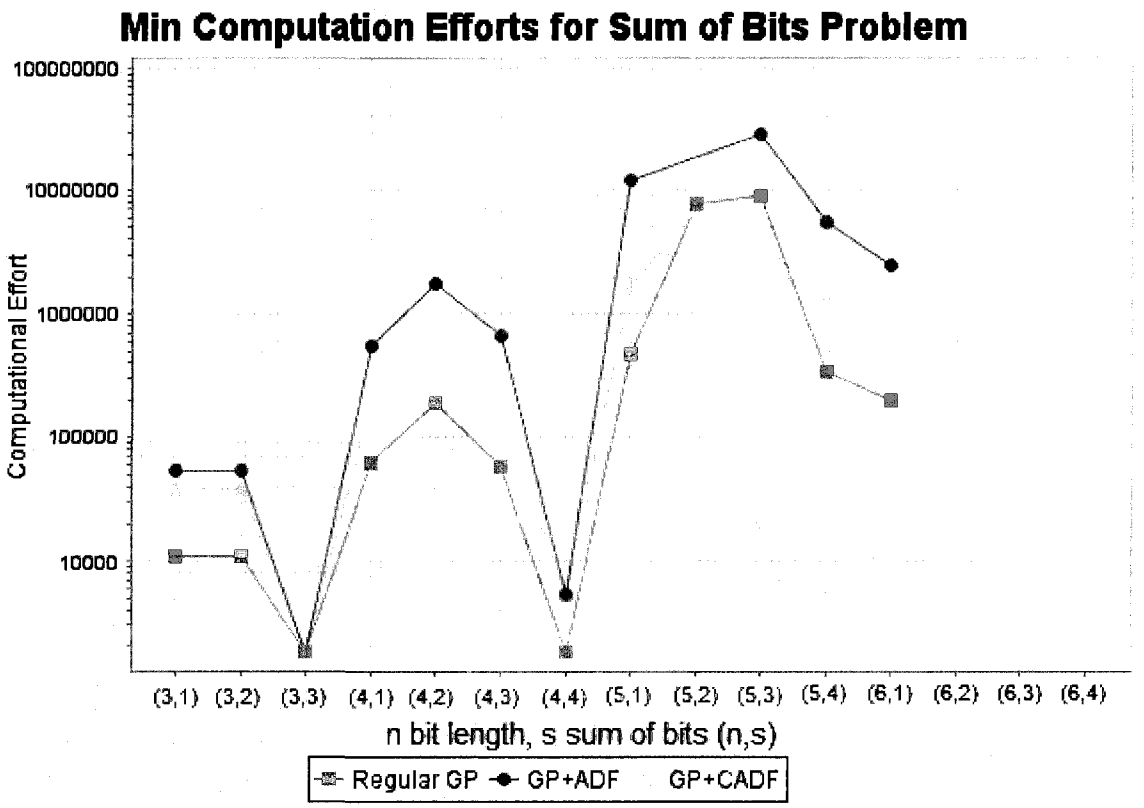

Figure 15 - This graph shows how the minimum computational effort E_{min} changes with problem complexity for all 3 methods for the n-s-sum of bits problem.

Chapter 6: The Image Recognition Problem

6.1 Introduction

The letter recognition problem is based on work done by Koza in chapter 15 of [1]. The inputs to the problem consist of a grid of 6x4 pixels, and on each grid some pixels are filled and some are empty. These grids represent images on which pixels can be filled in such a way to resemble letters. Essentially, the problem consists of taking these images and deciding weather or not they represent a particular letter.

The main motivation for exploring this problem is to examine a problem in which we do not know beforehand what kind of subroutines would be useful. Indeed, part of the solution to the problem will be the discovery of useful feature detectors as subroutines and the combination of the newly discovered detectors. According to [1], in some cases the task of finding exploitable regularities and tackling those using subroutines is the actual problem. In the letter recognition problem, we will attempt to find useful subroutines that work on images without any knowledge of what could be useful in this problem domain. The goal of this chapter will be to determine if GP+CADF is better suited than GP+ADF in discovering and utilizing these previously unknown subroutines or feature detectors.

There are a total of $2^{24} = 16,777,216$ possible images on a 6x4 grid and many letters and symbols we could represent. For the purposes of this problem, we will only use the letters I and L. The reason we only use I, L and various samples that are neither is that we are not interested in a genuine letter recognition task. Rather, we are examining a problem where we want to recognize two things perfectly among many samples. The

main motivation is the discovery of solutions though subroutines that examine local pixel neighbourhoods and the goal of the experiment will be to see if GP+CADF improves on GP+ADF. The inputs will include the letters I and L and 75 other images that represent a

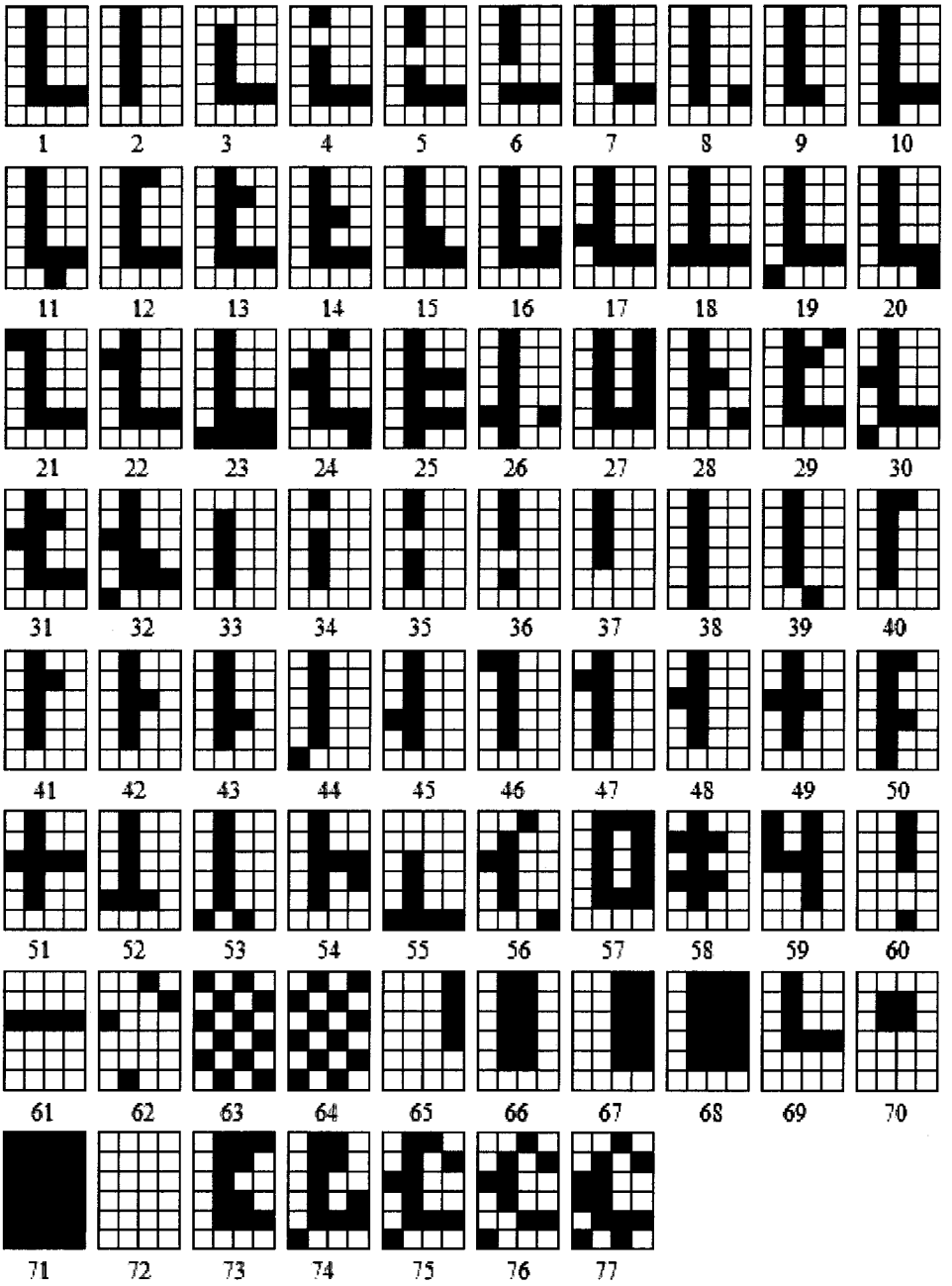

Figure 1 – Fitness cases for the letter recognition problem. The first two images are the letters L and I and the rest are negative cases; some resemble the letters and others are very different.

range of possibilities; some resembling the letters, and some that look very different. In other words, we will have two positive and 75 negative cases. Figure 1 shows the set of images that will be used in this problem. Images 1 and 2 are the letters L and I respectively. Images 3 through 22 are the letter L with one pixel of difference. Images 23 through 32 are the letter L with multiple pixels added or removed. Images 33 to 48 are the letter I with one different pixel. Images 49 to 56 are the letter I modified such that multiple pixels are different. Finally, images 57 through 77 are made to look nothing like the letters. For example, we have a full black, plain white, checker patterns and other random shapes and lines. The input set was chosen in an effort to have a variety of cases with images that are very similar and very different from the two letters I and L. The goal for the program will be to correctly identify the two letters and dismiss all other cases as negatives. Based on what the program outputs we can have 5 different cases.

- True-positive (TP) – the letters are correctly identified
- True-negative (TN) – a non-letter is correctly identified
- False-positive (FP) – a non-letter is identified as a letter
- False-negative (FN) – a letter is identified as a non-letter
- Wrong-positive (WP) – an I is identified as an L or an L is identified as an I

6.2 Experimental Setup

Table 1 summarizes all the parameters in the experimental setup for the letter recognition problem.

Problem	Letter Recognition
Objective	Find a program that determines if an image is an I, an L or none.
Size of Images	6x4
Population Size (M)	8100, 4000, 3000
Generations (G)	60
Runs	300
CADF Populations	4, 2
Mutation	None
Crossover	Subtree Crossover (`ec.gp.koza.CrossoverPipeline`)
Number of ADFs	3, 1
Function Set (no ADF)	IF, AND, OR, NOT, HOMING, I, L, NIL, X, N, NE, E, SE, S, SW, W, NW, GO-N, GO-NE, GO-E, GO-SE, GO-S, GO-SW, GO-W, GO-NW
Function Set (with ADF) Main Body	IF, AND, OR, NOT, HOMING, I, L, NIL, GO-N, GO-NE, GO-E, GO-SE, GO-S, GO-SW, GO-W, GO-NW, ADF0 and optionally ADF1, ADF2, ADF3
Function Set for ADF0-ADF3	AND, OR, NOT, X, N, NE, E, SE, S, SW, W, NW
Fitness Cases	All 77 images from Figure 1
Fitness	FP + FN*77 + WP*77
Hits	TP + TN
Success Predicate	0, 1, 3

Table 1 - Experimental setup for the letter recognition problem.

The solutions to this problem will utilize a turtle that can move across the images. X, N, NE, E, SE, S, SW, W, and NW represent the 9 pixel neighbourhood around a turtle at pixel X. A true value will be returned if a pixel is filled in black and a false value will be returned if a pixel is empty and white. IF is a two argument if statement and AND, OR and NOT are the standard Boolean operators we have already used in previous problems.

HOMING is a one argument operator that evaluates its single argument and then warps the turtle back to the position it was in before the argument was evaluated. I, L, NIL are the three possible classifications of a letter where NIL means that it is neither an I or an L. GO-N, GO-NE, GO-E, GO-SE, GO-S, GO-SW, GO-W and GO-NW are zero argument turtle moving operators. For example, GO-N will move the turtle one pixel upward and GO-E will move the turtle one pixel to the right. The image grid is toroidal which means that if a turtle goes off the edge of an image it reappears on the opposite side.

The settings in Table 1 are quite different from those used by Koza. He used a large population of 8000 and a total of 5 ADFs. In addition, there were constraints imposed on the structure of the program trees. The main body must be a decision tree with IF at the root. The first argument of the IF must be any composition of Boolean operators, ADFs, HOMING, and GO operators. The second argument must be another IF or a terminal I, L or NIL.

We have decided not to follow these constraints. Imposing a prior structure such as this has no justification since this kind of solution structure is not something we can know before obtaining a solution. We can not assume that a solution should be a decision tree as described. In addition, one of our goals was that the solution should automatically find a way of combining initially unknown useful subroutines. The only structure constraint we will allow is to have the ADFs work only with the local neighbourhood sensing operators X, N, NE, E, SE, S, SW, W, and NW and Boolean operators. The main body will have all operators except the local sensors. The reasoning is that we want the subroutines to be local detectors of a 9 pixel neighbourhood while the main body moves the turtle globally and utilizes these subroutines.

In initial trials it was determined that no solutions were obtained by any method with a large population of 8000, 5 ADFs and fitness calculation given by Koza in [1] which is FP + 23*FN + WP. Instead of concluding that we must have the same constraints as Koza in order to obtain a solution, we experimented with different settings. It was concluded that the reasons for no solutions being obtained were the following.

Firstly, the relatively large number of ADFs was actually hindering the discovery of a solution. With 5 and 4 ADFs, no solutions were found by any method. The first solution appeared when using 3 ADFs and the best results were obtained with only a single ADF. The conclusion is that this problem does not require a large variety of different subroutines, and it is easier for the main body to develop a solution with only a single subroutine. With more ADFs, there is more complexity and more combinations for the main body to deal with.

The second reason was the fitness calculation. FP + 23*FN + WP penalizes equally for false positives and wrong positives and penalizes 23 for a false negative. When the unsuccessful solutions were analyzed, it was discovered that all the best solutions had a fitness of 46. This means that the program tree simply said that nothing is a letter and it got penalized twice for not identifying I and L. The problem is that we only have two positive cases and there are a total of 77 fitness cases, so a penalty of 46 would be the local minimum found by GP where the evolution would stagnate. To solve this problem, false negatives are penalized for the total number of fitness cases which is 77. However when doing only this, the solutions switched strategy and identified everything as either an I or an L for a total fitness of 76. The final adjustment was to penalize heavily for a wrong positive with 77. The programs could no longer identify everything as NIL or

everything as either an I or L and not be penalized heavily. The final fitness formula was therefore set to be FP + FN*77 + WP*77. After reducing the number of ADFs and adjusting the fitness calculation, solutions were readily found with a much smaller population of 3000 and without the constraints imposed by Koza.

6.3 Experimental Results

Table 2 summarizes the experimental results obtained with the settings of Table 1 with the population size, number of ADFs and success predicate indicated. As mentioned

Test	M	ADFs	ARGS	SP	Regular GP	GP+ADF	GP+CADF
1	8100	3	N	0	No Solution	No Solution	648,324,000
2	4000	1	N	3	No Solution	8,880,000	5,760,000
3	3000	1	N	1	No Solution	60,888,000	40,356,000
4	3000	1	Y	1	No Solution	40,356,000	30,780,000

Table 2 - Experimental results for the letter recognition problem. M is the population size. ADFs indicates the number of ADFs we use in the solutions. ARGS indicates whether or not the ADFs use arguments that are passed by the main body. SP is the success predicate or how many fitness penalties we allow and still count a solution as a success. The numbers given for all 3 methods are the minimum computational efforts.

in the previous section, no results were obtained with programs attempting to use 4 or more ADFs and no solutions were ever obtained by regular GP. The first solution appeared in test 1 with 3 ADFs, GP+CADF method, and a population of 8100 with an astronomical computational effort of 648 million. There was only a single solution obtained in 300 runs. This shows that meaningful comparisons will not be possible with more than 3 ADFs and a requirement of perfect fitness.

Using only a single ADF, a lower population, and a higher success predicate yielded solutions for both GP+ADF and GP+CADF in tests 2 and 3. Recall that a success

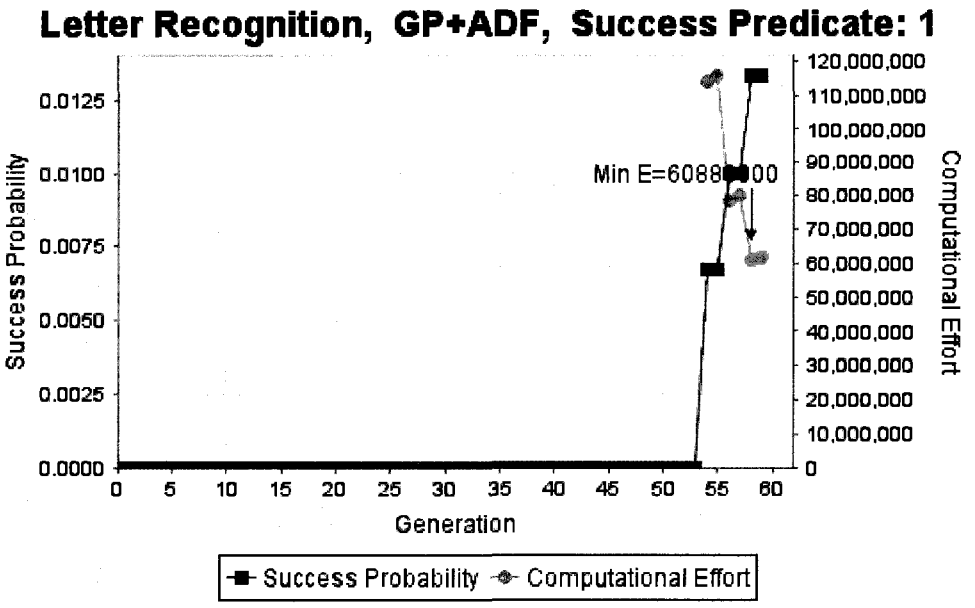

Figure 2 – E_{min} for GP+ADF was 60,888,000 at generation 58 and with success probability 1.33%.

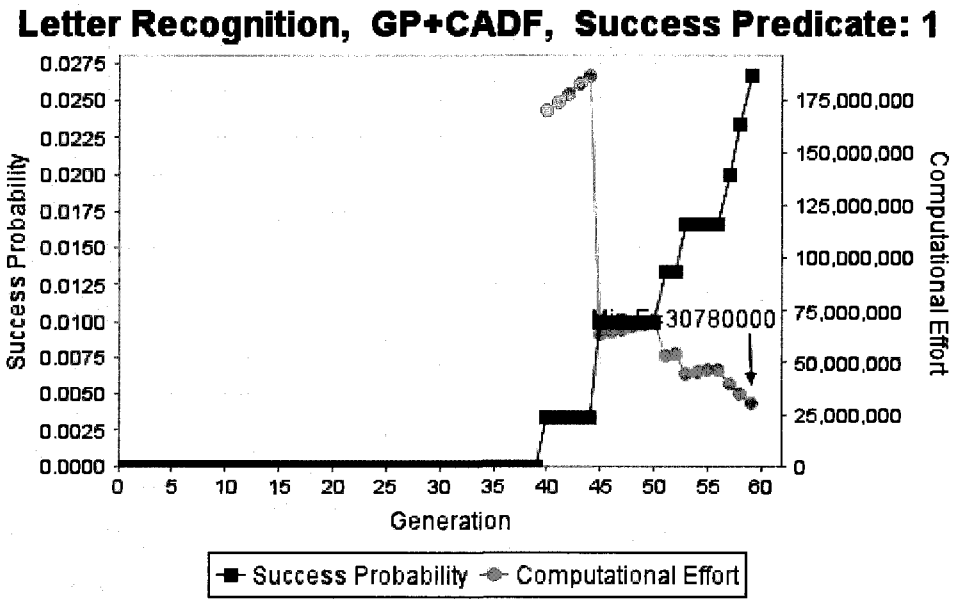

Figure 3 - Emin for GP+CADF was 40,356,000 at generation 58 and with success probability 2.0%.

predicate is how many mistakes or fitness penalties we allow and still consider a solution to be successful. In our case, a success predicate of 1 would mean that we allow 1 false positive which means one non-letter classified as a letter.

Examining the results, we see that GP+CADF yielded a lower computational effort in all tests. For tests 2 and 3, the computational effort difference was the same at a ratio of approximately 1.5, which means that GP+ADF requires 1.5 times more computational effort. The performance charts for test 3 are given for both GP+ADF and GP+CADF in Figures 2 and 3 respectively. Although, the differences were not as large as in the parity problem, these results show that GP+CADF better discovers and utilizes previously unknown subroutines than GP+ADF.

6.4 Solution Analysis

The following was a successful solution for the letter recognition task obtained by GP+CADF in test 2 of Table 2:

```
Main Body:
 (OR (OR (HOMING (IF (IF (IF (IF (NOT Go-SW) (HOMING (IF (IF (IF (NOT
Go-SW) (IF Go-SE Go-S Go-W) (AND Go-NE NIL)) NIL (AND Go-NE NIL)) NIL
(IF (IF (IF (IF Go-SE Go-S Go-W) (IF Go-SE Go-S Go-W) (IF ADF0[1] I Go-
N)) NIL (AND Go-NE NIL)) NIL (IF Go-SE Go-S Go-W)))) (IF ADF0[1] I Go-
N)) (IF (IF ADF0[1] I Go-N) Go-S Go-W) (IF ADF0[1] I Go-N)) NIL (AND
Go-NE NIL)) NIL (IF (IF (IF (NOT Go-SW) (HOMING (IF (IF (IF (NOT Go-SW)
(IF Go-SE Go-S Go-W) (AND Go-NE NIL)) NIL (AND Go-NE NIL)) NIL (IF (IF
(IF (IF Go-SE Go-S Go-W) (IF Go-SE Go-S Go-W) (IF ADF0[1] I Go-N)) NIL
(AND Go-NE NIL)) NIL (AND Go-NE NIL)))) (IF ADF0[1] I Go-N)) NIL (AND
Go-NE NIL)) NIL (AND Go-NE NIL)))) (IF (IF (NOT (OR (IF (IF ADF0[1] I
Go-N) NIL (HOMING (IF (AND Go-NW Go-S) (IF Go-SW I I) (HOMING (IF Go-N
NIL (IF ADF0[1] I Go-N)))))) NIL)) (IF Go-SE Go-S Go-W) (IF ADF0[1] I
Go-N)) (IF ADF0[1] NIL L) (HOMING (OR (IF (IF (NOT (IF Go-N NIL I)) (IF
Go-SE Go-S Go-W) (IF ADF0[1] I Go-N)) (IF ADF0[1] NIL L) (HOMING (OR
(IF (IF ADF0[1] I Go-N) NIL (HOMING (IF (AND Go-NW Go-S) (IF Go-SE Go-S
Go-W) (HOMING (IF Go-N NIL (IF ADF0[1] I Go-N)))))) NIL))) NIL))))
(HOMING NIL))
ADF0:
 (OR (OR (OR (OR SE (OR (OR (OR SE (OR (OR SE (OR (OR SE (OR E (OR (OR
SE SW) NE))) SW)) SW)) (OR SE SW)) SW)) NE) S) NE)
```

The single ADF0 that was developed and used in the solution senses the pixels around a turtle positioned at pixel X, and can be summarized by the following table:

Inputs that produce 0 output	
000000000	100000000
000000001	100000001
001000000	101000000
001000001	101000001
010000000	110000000
010000001	110000001
011000000	111000000
011000001	111000001

Table 3 - These inputs to ADF0 produce an output of 0 and all other inputs produce 1. The inputs are N NE E SE S SW W NW X read from left to right.

If we examine the bit patterns we see that SE, S, SW, W, NW must all be off or 0. The rest of the 4 inputs N, NE, E and X have all 2^4=16 bit combinations included which means that it does not matter what they are, they will not affect the output. Therefore, the subroutine ADF0 essentially checks that SE, S, SW, W, NW are blank pixels or 0. Equivalently, this ADF outputs 1 when at least one of SE, S, SW, W, NW is a black pixels or 1. Note that the main body may use this ADF when it outputs both 1 and 0 by combining it with the NOT operator. This local detector is used a total of 14 times in the solution. We know that the ADF is required for finding a solution because only the ADF has the pixel sensors; it would not be possible for a main body alone to have any idea of what a fitness test is. Therefore, we may conclude that GP+CADF discovered a useful subroutine and utilized it to solve the letter recognition problem.

6.5 ADFs with Arguments

In addition, we also examined what happens when the ADF function set is the following:

F_{adf} = {AND, OR, NOT, ARG0, ARG1, ARG2, ARG3}

and the main body function set is the full function set

$F_{main\ body}$ = { IF, AND, OR, NOT, HOMING, I, L, NIL, X, N, NE, E, SE, S, SW, W, NW, GO-N, GO-NE, GO-E, GO-SE, GO-S, GO-SW, GO-W, GO-NW, ADF0}

This means that the ADFs no longer have the sensing operations of the 9 pixel neighbourhood X, N, NE, E, SE, S, SW, W, NW and instead have 4 arguments. The main body has the full function set and can pass any combination of the functions to the ADF as arguments. Test 4 shows the results with the same settings as in test 3 except we are using arguments. Results show that the computational effort was lowered by a factor of 1.5 for GP+ADF and 1.3 for GP+CADF. In this case, we see that there is an advantage to passing arguments to an ADF as opposed to letting the ADF work on its own functions. A hypothesis that might explain why arguments are advantageous may be the same one that explains why GP+ADF has advantages over regular GP. Consider for example an ADF that works with 2 functions (+, *) and 2 terminals (X, Y). Let us suppose that a useful ADF could be X*Y + X*Y + X*Y. An ADF that works with 2 functions (+, *) and an argument ARG0 may represent the subroutine simply as ARG0 + ARG0 + ARG0 and have the main body call ADF0 with an argument (X*Y). In the non-argument case the ADF is an 8 node tree and in the argument version it was a 3 node tree. Instead of coming up with the same functionality (X*Y) 3 times independently, this is done once by the main body which then passes X*Y to the ADF. We see that the more repetitive regularities are required in the ADF, the more advantage we gain by using arguments. This is similar to why GP+ADF has an advantage over regular GP. Instead of coming up many times with the same functionality in the program tree, we encode this in a single subroutine and invoke it many times.

We can verify this hypothesis by examining if the solution to the letter

recognition problem with ADF arguments has a main body that invokes the ADF with compound arguments. The following is one of successful solutions.

```
Main Body:
 (IF (ADF0[1] (AND (ADF0[1] (AND X NW) (IF (ADF0[1] (AND X (IF Go-SW I
Go-SW)) (IF Go-SW I Go-SW) (OR SW Go-E) (OR SE NW)) I Go-SW) (OR SW Go-
E) (OR SE NW)) NW) Go-SW (OR SW Go-E) (ADF0[1] (AND X (IF Go-SW I Go-
SW)) (IF Go-SW I Go-SW) (OR SW Go-E) (ADF0[1] (AND (IF I Go-S Go-W) NW)
(IF Go-SW I Go-SW) (OR SW Go-E) (ADF0[1] (AND (OR (AND X (IF Go-SW I
Go-SW)) NW) (IF Go-SW I Go-SW)) (IF Go-SW I Go-SW) (OR SW Go-E) (IF
(ADF0[1] (AND (ADF0[1] (AND X NW) (IF (ADF0[1] (AND X (ADF0[1] (AND (IF
I Go-S Go-W) NW) (IF Go-SW I Go-SW) (OR SW Go-E) (ADF0[1] (AND X (IF
Go-SW I Go-SW)) (IF Go-SW I Go-SW) (OR SW Go-E) (OR SE NW)))) (IF Go-SW
I Go-SW) (OR (IF Go-SW I Go-SW) Go-E) (OR (AND X (IF Go-SW I Go-SW))
NW)) I Go-SW) (OR SW Go-E) (OR SE NW)) NW) (IF Go-SW I Go-SW) (OR SW
Go-E) (ADF0[1] (AND X (IF Go-SW I (OR SE NW))) (IF Go-SW I Go-SW) (OR
SW Go-E) (OR SE NW))) I (OR SE NW)))))) I (IF (ADF0[1] (AND (IF I Go-S
Go-W) NW) (IF Go-SW I Go-SW) (OR SW Go-E) (ADF0[1] (IF I Go-S Go-W) (IF
Go-SW I (OR SW Go-E)) (ADF0[1] (AND X (ADF0[1] (AND (IF I Go-S Go-W) NW)
(IF Go-SW I Go-SW) (OR (AND (IF I Go-S Go-W) NW) Go-E) (ADF0[1] (IF Go-
SW I Go-SW) (IF Go-SW I Go-SW) (OR SW Go-E) (OR SE NW)))) (IF Go-SW I
Go-SW) (IF Go-SW I Go-SW) (OR (AND X (IF Go-SW I Go-SW)) NW)) (OR SE
NW))) L NIL))
ADF0:
 (OR (NOT (NOT (NOT (OR (OR (AND ARG[0] ARG[0]) (OR (AND (AND (OR (OR
(NOT ARG[0]) (OR ARG[2] ARG[1])) (AND (OR (NOT ARG[0]) (OR ARG[2]
ARG[1])) (AND (OR ARG[2] ARG[0]) (OR ARG[2] ARG[0]))))) (AND (AND (AND
(AND ARG[2] ARG[2]) ARG[3]) (AND ARG[0] ARG[0])) (OR (AND ARG[1] ARG[2])
(AND ARG[0] ARG[1])))) (AND (OR ARG[0] ARG[0]) (NOT ARG[3])))) (AND (AND
(OR (NOT (AND ARG[0] (NOT (OR (NOT (OR ARG[0] ARG[3])) (OR ARG[2]
ARG[0])))))) (AND (OR (NOT ARG[0]) (OR ARG[2] ARG[1])) (AND (OR ARG[2]
ARG[0]) (OR ARG[1] ARG[3])))) ARG[2]) (OR ARG[1] ARG[0])))) (OR (NOT
(NOT (NOT (AND ARG[1] ARG[3])))) (NOT (OR (OR ARG[3] ARG[2]) (OR ARG[2]
ARG[0])))))))) (AND (AND (OR ARG[2] ARG[0]) ARG[2]) (AND (NOT (OR ARG[3]
ARG[1])) (AND ARG[1] (NOT (NOT (OR ARG[0] ARG[0])))))))
```

We see that every one of the ADF calls in the main body passed a compound argument the smallest of which was `ADF0[1] (AND X NW)` with a 3 node argument. If we imagine the ADF arguments expanded in the ADF we see that it would quickly become very complex when long arguments are passed. What we managed to do with arguments is to avoid having to repeat a whole argument sub-tree many times.

6.6 Conclusion

This chapter explored the letter recognition problem first presented by Koza [1] but modified in substantial ways based on our experimentation. We have determined that GP+CADF has a lower computational effort than GP+ADF in all tests and that regular GP yielded no solutions in any of the tests. Solutions to the problem were also analyzed to show that useful subroutines were discovered by GP+CADF. Finally, we have shown an additional benefit of using ADF arguments in this problem.

Chapter 7: Lawn Coverage Problems

7.1 Introduction

The lawn coverage problems are a set of three problems originally introduced by Koza in [1] to test the benefits of GP+ADF in an environment with increasing complexity. Basically, the problems consist of discovering a program to control a robot moving on a lawn. The robot has a simple set of commands to control its movements and sensors that tell it something about the environment. The objective in all problems is to have the robot cover as much of the total area as possible. The differences in this set of problems arise from what kid of grid we are dealing with.

In the first problem that Koza called lawnmower, the grid is completely empty and the task of the robot is simply to 'mow' the entire area such as would be the task of a real lawnmower. The second problem is called obstacle avoiding robot, and the difference there is that the lawn has a number of obstacles through which the robot can not move. The motivation of putting obstacles on the lawn is to attempt to increase the complexity of the task of coverage. The final variant is called minesweeper, in which the lawn has lethal mines. Whereas the obstacles prevent a robot from moving through it, a mine disables the robot from further operation. This kind of environment was shown to be the most challenging for a robot to cover completely [1].

These problems were used by Koza to show the benefit of adding ADFs to GP. Our task will be to test these problems with regular GP, GP+ADF and GP+CADF under the same settings and perform a comparative analysis of the different methods. In particular, we are interested if the GP+CADF method will yield any advantages.

7.2 Experimental Setup

Table 1 summarizes all the parameters in the experimental setup for the lawn

Problem	Lawn Coverage (Lawnmower, Obstacle Avoiding Robot, Minesweeper)
Objective	Find a program to control a moving robot in a specific grid environment.
Lawn Size	20x20, 16x16, 12x12
Max Movements/Turns	400, 1000
Number of Obstacles / Mines	25 Obstacles, 14 Mines
Population Size (M)	600 for Lawnmower, 2100 for rest
Generations (G)	60 for Lawnmower, 51 for rest
Runs	500 for Lawnmower, 300 for rest
CADF Populations	3
Mutation	None
Crossover	Subtree Crossover (`ec.gp.koza.CrossoverPipeline`)
Number of ADFs	2
Lawnmower Function Set	F-LAWNMOWER = {MOVE-FORWARD, TURN-LEFT, FROG, VECTOR-ADD, PROG-1, PROG-2, PROG-3, ERC}
Obstacle Avoiding Robot Function Set	F-OBSTACLE AVOIDING ROBOT = {F-LAWNMOWER, IF-OBSTACLE}
Minesweeper Function Set	F-MINESWEEPER = {F-LAWNMOWER, IF-MINE}
Function Set (no ADF)	F-LAWNMOWER or F-OBSTACLE AVOIDING ROBOT or F-MINESWEEPER
Function Set (with ADF) Main Body	F-LAWNMOWER or F-OBSTACLE AVOIDING ROBOT or F-MINESWEEPER, ADF0, ADF1
Function Set for ADF0	F-LAWNMOWER or F-OBSTACLE AVOIDING ROBOT or F-MINESWEEPER without FROG
Function Set for ADF1	F-LAWNMOWER or F-OBSTACLE AVOIDING ROBOT or F-MINESWEEPER, ARG0
Fitness Cases	1 lawn for Lawnmower 4 lawns for Obstacle Avoiding Robot 4 lawns for Minesweeper
Fitness	Total number of unexplored grid cells
Hits	Number of visited cells
Success Predicate	0 for Lawnmower, 12 for Obstacle Avoiding Robot, 12 for Minesweeper

Table 1 - Experimental settings for the lawn coverage problem.

coverage problem. We will be using a 20 x 20 toroidal grid of cells for the lawnmower problem, a 16 x 16 grid for obstacle avoiding robot and a 12 x 12 grid for minesweeper. The robot can occupy only one cell on the grid at a time, and can only have four orientations: North, South, East and West. The function set for the robot includes MOVE-FORWARD which moves it forward one cell in the direction it is facing and also marks the newly occupied cell as visited for fitness calculation. If a robot is facing an obstacle, this operation will have no effect. If the robot is facing a mine, this operation will disable the robot. If the robot moves off of the edge of a grid it reappears on the opposite side. The robot can also TURN-LEFT which means that it rotates counter-clockwise to change direction without moving. FROG is an operation that takes a single vector argument and makes the robot jump to a new position specified by the vector from its current position. There are a maximum number of movements and turns allowed in this problem which is set to 400 for lawnmower and 1000 for the rest. VECTOR-ADD is a two argument function that returns the sum of its two vector arguments. PROG-1, PROG-2 and PROG-3 simply take one, two, and three arguments respectively and evaluate them. This is similar to PROG-N used by Koza except we are explicitly specifying the value of N and do not allow more than three arguments. ERC is an Ephemeral Random Constant as described by Koza in [7] except we are using two integers to make a vector. Fundamentally, it is just a numerical constant that gets set to some random value and always returns that value. In our case, we set two random integers between 0 and (lawn size -1) to represent a random vector. IF-OBSTACLE is a two argument function that executes its first argument if there is an obstacle in front of a robot and the second argument if there is no obstacle. IF-MINE works the same as IF-OBSTACLE, except we

check for the presence of a mine.

We decided to have 2 ADFs with the function sets given in Table 1 to stay consistent with Koza's work. The first ADF has zero arguments and all functions except FROG. The second ADF has all functions and one argument. Koza does not give any explanation for this choice of architecture or for the absence of FROG in the first ADF, but since this setup was empirically found to work well we will remain consistent.

There is a single fitness case for lawnmower which is the empty lawn. There are a total of 4 fitness cases for obstacle avoiding robot each a lawn with 6 randomly placed obstacles. There are also 4 fitness cases for minesweeper with 14 randomly placed mines. The success predicate is 0 for lawnmower which means all cells must be visited for perfect zero fitness. The success predicate is 12 for obstacle avoiding robot and minesweeper which means that we can leave up to 12 unexplored cells and still receive a perfect fitness. The success predicate was higher in those cases in order to increase the success rate enough to be able to obtain enough successful runs to perform a meaningful analysis.

7.3 Experimental Results

Experimental results for the lawn coverage problem show that lawnmower was the problem with the least computational effort required, and that no solutions were found with regular GP in any problem. The benefit of using ADFs in this problem is clear since we were able to obtain 100% success rate with both GP+ADF and GP+CADF while the success rate was 0% with regular GP. The computational effort increased approximately 20 fold when we introduced obstacles in the lawn with the obstacle avoiding robot problem. The minesweeper problem was expected to be of the highest difficulty and

Problem	Regular GP	GP+ADF	GP+CADF	GP+ADF / GP+CADF
Lawnmower	No Solution	12,000	6,600	1.8
Obstacle Avoiding Robot	No Solution	264,600	126,000	2.1
Minesweeper	No Solution	4,176,900	2,318,400	1.8

Table 2 - Minimum computational efforts obtained for the lawn coverage problems. The last column is the ratio of the difference between GP+ADF and GP+CADF.

accordingly, the computational effort increased approximately 350 times from the lawnmower problem. Keep in mind as well that we use a smaller lawn and higher success predicate for the more difficult problems which was shown by Koza to reduce the problem complexity [1]. Under exactly the same settings, the increase in computational effort would be even higher than reported here for obstacle avoiding robot and minesweeper.

In addition, we see that the computational effort of GP+CADF was approximately half the value of GP+ADF in all cases. Therefore, the conclusion is that GP+CADF solves the lawn coverage problems with approximately half the computational effort as required by GP+ADF. The performance graphs are given for all problems and for both GP+ADF and GP+CADF in Figures 1 to 6.

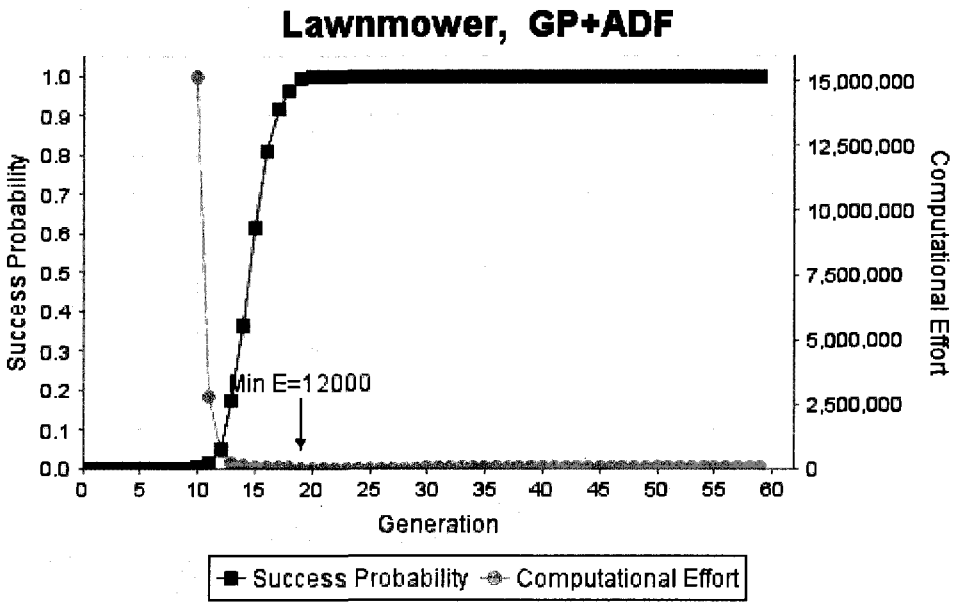

Figure 1 - E_{min} for GP+ADF was 12,000 at generation 19 and with success probability 99.2%.

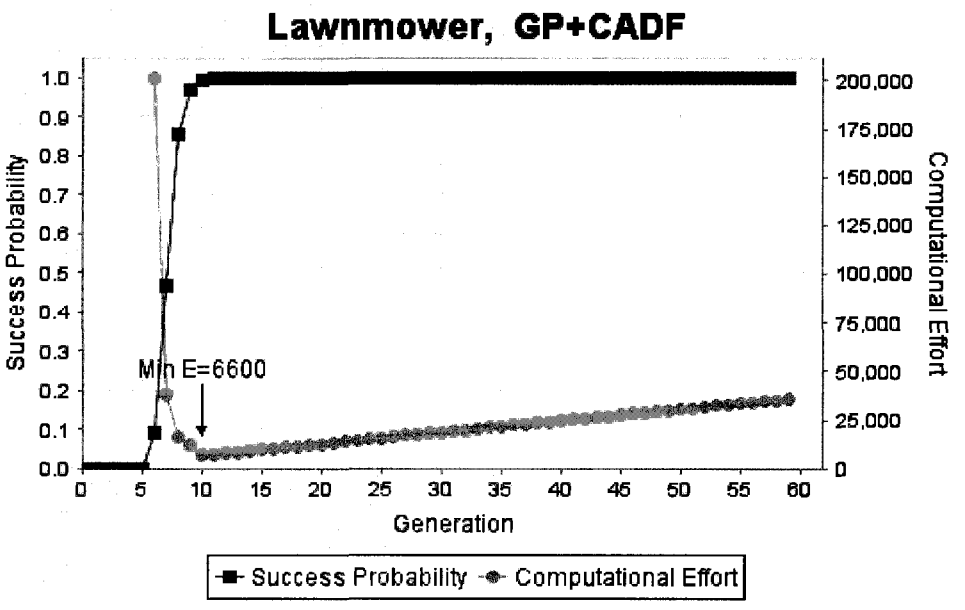

Figure 2 - E_{min} for GP+CADF was 6,600 at generation 10 and with success probability 99.4%.

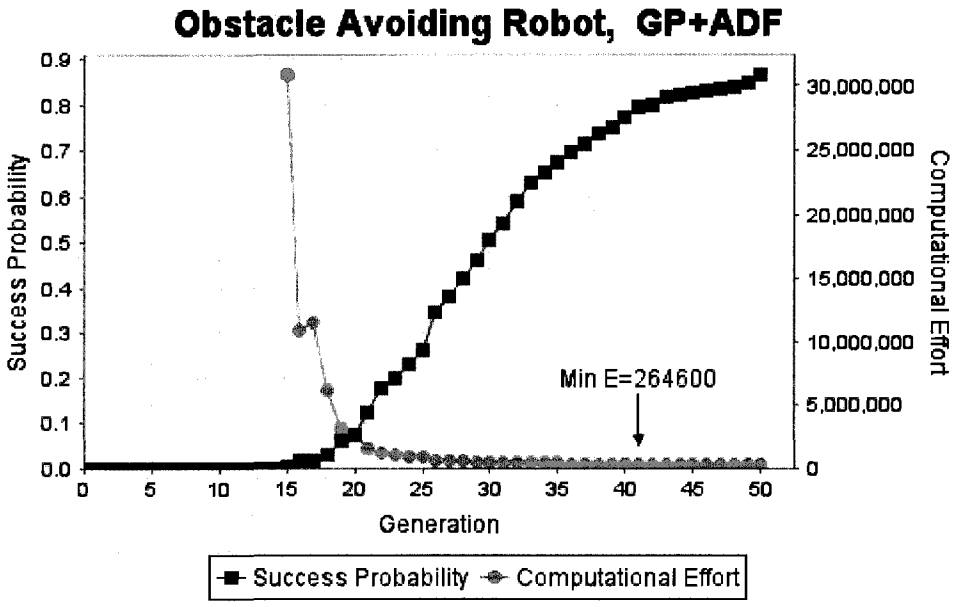

Figure 3 - E_{min} for GP+ADF was 264,600 at generation 41 and with success probability 79.5%.

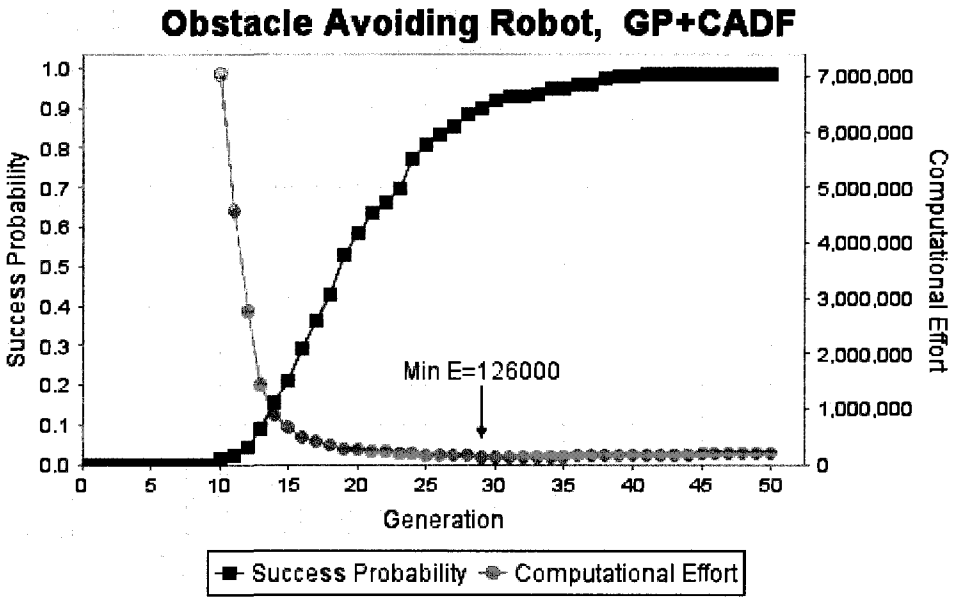

Figure 4 – E_{min} for GP+CADF was 126,000 at generation 29 and with success probability 90.0%.

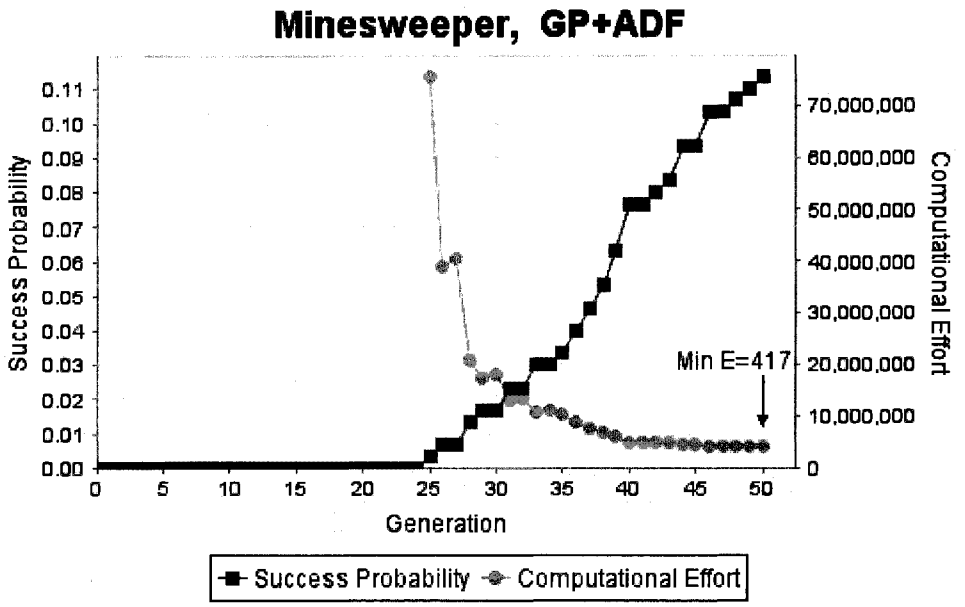

Figure 5 - E_{min} for GP+ADF was 4,176,900 at generation 50 and with success probability 11.3%.

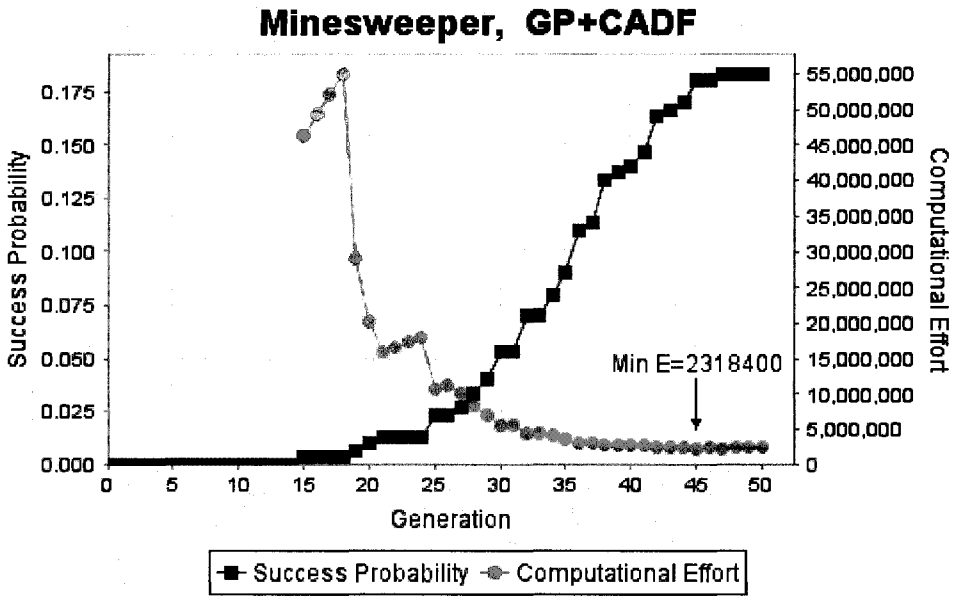

Figure 6 - E_{min} for GP+CADF was 2,318,400 at generation 45 and with success probability 18.0%.

Furthermore, we can demonstrate that GP+CADF has an earlier generation of convergence in the lawn coverage problem. Table 3 gives the mean generation of

convergence for both GP+ADF and GP+CADF. We use Welch's unpaired t test to determine if the difference in means is statistically significant since the variances in the samples are not the same and the data is not paired. There were 500 samples for lawnmower, 173 samples for GP+ADF for obstacle avoiding robot, 197 samples for GP+CADF for obstacle avoiding robot, 34 samples for GP+ADF for minesweeper, and 55 samples for GP+CADF for minesweeper.

Problem	GP+ADF	GP+CADF	Difference Ratio	P Value
Lawnmower	15.10	7.62	1.98	<0.0001
Obstacle Avoiding Robot	29.82	20.44	1.45	<0.0001
Minesweeper	38.24	34.53	1.10	0.0158

Table 3 - The average generation of convergence is given for both GP+ADF and GP+CADF for all problems. Welch's unpaired t test is used to determine statistical significance. The difference between the means and the two-tailed P value is also given.

In the lawnmower problem, the mean generation of convergence for GP+CADF was about half the value of GP+ADF. Welch's unpaired t test gives a P value of less than 0.0001 which is extremely statistically significant. We can see that the difference ratio for the two methods decreases as we move on to the more complex problems. In obstacle avoiding robot, GP+CADF had a mean generation of convergence about 1.5 times lower than GP+ADF, and the difference was again extremely statistically significant. In the minesweeper problem, the difference was the lowest at a ratio of about 1.1 and the P value was 0.0158 which is still statistically significant. Hence, according to the results of Table 3, GP+CADF converges to a solution quicker than GP+ADF in the lawn coverage problems.

Lastly, we will not analyze the discovered program trees. Upon examination, the

structure of the solutions and the types of ADFs were largely similar for GP+ADF and GP+CADF and are comparable to those found by Koza for these problems in [1]. For a detailed description of the solutions and ADFs that are discovered refer to the work in [1].

7.4 Conclusion

We have shown with the lawn coverage problems that GP+CADF once again requires less computational effort than GP+ADF to yield a correct solution with 99% probability. In addition, we have shown that GP+CADF has a lower mean generation of convergence, in other words it converges to a solution sooner than GP+ADF.

Chapter 8: Bumblebee Problem

8.1 Introduction

The bumblebee problem was introduced by Koza in [1] to experiment with a problem that uses floating point numbers as terminals. The problem consists of a two dimensional plane in which a bee is tasked with visiting randomly placed flowers. This is similar to the lawn coverage problems in the previous chapter; however the difference is that we do not have a grid of cells but an entire plane where any position can be specified as a floating point coordinate. In the lawn coverage problems we could only specify a cell location with integers. Furthermore, the bee is not tasked with covering the entire plane but rather to visit a series of flowers which are points on the plane.

8.2 Experimental Setup

Table 1 summarizes all the parameters in the experimental setup for the bumblebee problem. The field is a 10 x 10 area centered on the origin (0, 0) so that it extends to -5 and +5 in each direction. There are 30 randomly placed flowers on the field and a bee is considered to have visited a flower if it gets to within 0.001 of the flower which is called accuracy in Table 1. GO-X takes a vector as an argument and moves the bee in the x direction by the x component of that vector. GO-Y works similarly in the y direction. VECTOR-ADD and VECTOR-SUB are functions for adding and subtracting vectors; consequently they each take two vector arguments. PROG1 and PROG2 take one and two arguments respectively and execute them. BEE is a vector that specifies the current location of the bee. NEXT-FLOWER returns a vector position of a random unvisited flower. RANDOM-VECTOR returns a vector with x and y components

Problem	Bumblebee
Objective	Find a program to control a bee so that it visits flowers on a plane.
Field Dimensions	10 x 10
Max Movements	100
Number of Flowers	30
Accuracy	0.001
Population Size (M)	1200
Generations (G)	51
Runs	300
CADF Populations	2
Mutation	None
Crossover	Subtree Crossover (`ec.gp.koza.CrossoverPipeline`)
Number of ADFs	2
Function Set (no ADF)	GO-X, GO-Y, VECTOR-ADD, VECTOR-SUB, PROG1, PROG2, BEE, NEXT-FLOWER, RANDOM-VECTOR
Function Set (with ADF) Main Body	GO-X, GO-Y, VECTOR-ADD, VECTOR-SUB, PROG1, PROG2, BEE, NEXT-FLOWER, RANDOM-VECTOR, ADF0
Function Set for ADF0	GO-X, GO-Y, VECTOR-ADD, VECTOR-SUB, PROG1, PROG2, BEE, NEXT-FLOWER, RANDOM-VECTOR, ARG0
Fitness Cases	4
Fitness	120 – visited flowers
Hits	Visited flowers
Success Predicate	0

Table 1 - Experimental settings for the bumblebee problem.

between -5.000 and +5.000. There are a total of 4 fitness cases which means there are 4 fields each with 30 randomly placed flowers. The raw fitness of a program is the total number of unvisited flowers among all of the fitness cases. The success predicate is 0 which means that all of the flowers must be visited for the perfect 0 fitness. Following Koza's work it was decided that we will only have a single ADF that contains all of the functions of the main body and a single argument ARG0.

8.2 Experimental Results

Table 2 summarizes the minimum computational efforts obtained in 300 runs of regular GP, GP+ADF and GP+CADF. There were no solutions obtained with regular GP.

Problem	Regular GP	GP+ADF	GP+CADF	GP+ADF / GP+CADF
Bumblebee	No Solution	172,800	148,800	1.2

Table 2 - Minimum computational efforts obtained for the bumblebee problem. The last column is the ratio of the difference between GP+ADF and GP+CADF.

If the success predicate was raised to allow for unvisited flowers or if there were fewer fitness cases or flowers, then it is possible that regular GP might have produced a successful run. However, this does not seem possible with the settings of Table 1 because the problem complexity is such that the use of ADFs becomes necessary to obtain successful runs. Figures 1 and 2 show the performance graphs for GP+ADF and GP+CADF respectively. We see that the difference in computational effort was not large; however GP+CADF was the method with the lower computational effort at a ratio of 1.2.

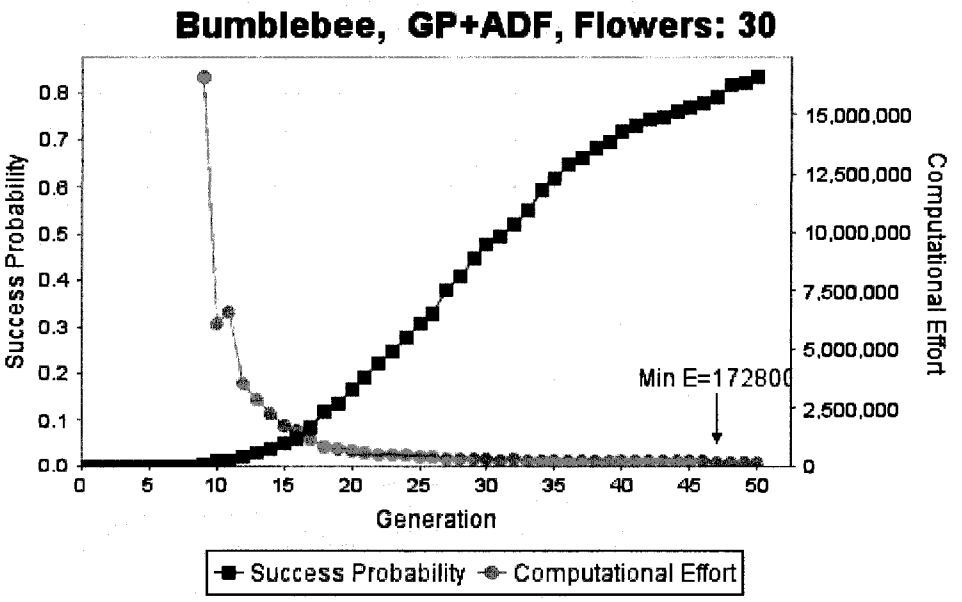

Figure 1 - E_{min} for GP+ADF was 172,800 at generation 47 and with success probability 79%.

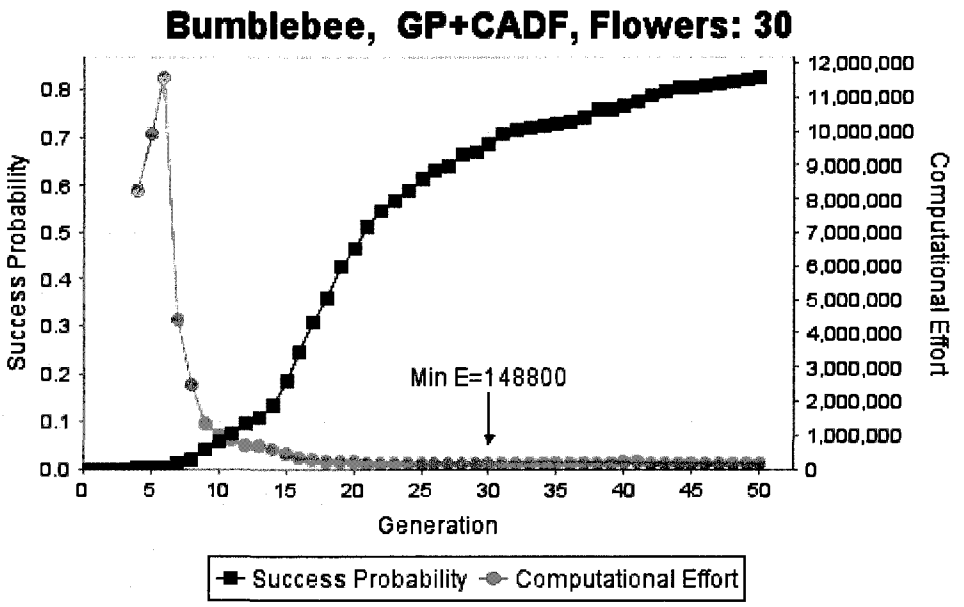

Figure 2 - E_{min} for GP+CADF was 148,800 at generation 30 and with success probability 68.7%.

The bumblebee problem was relatively simple to solve with a method augmented with ADFs, as we can see from the cumulative probabilities of success which was 83% for both methods by generation 51 and 0% for regular GP.

8.3 Conclusion

In this chapter, we have performed a comparative analysis of different GP methods on the bumblebee problem. This problem is unique among those studied since it lies in the domain of programs that utilize floating point numbers. In addition, this problem also highlights the case where ADFs cause a dramatic increase in success probability. The results of our experiments show that under identical settings GP+CADF outperforms GP+ADF by solving the bumblebee problem with a lower computational effort.

Chapter 9: Conclusion and Future Work

9.1 Summary of Contributions

This work began by asking whether the addition of co-evolution to genetic programming would help to overcome the current limitations of the standard GP approach as well as GP augmented with automatically defined functions. We have shown that GP+CADF is a feasible co-evolutionary approach that continues discovery of solutions where regular GP and GP+ADF reach their limits. We have shown that GP+CADF yields solutions for up to 14-even-parity and 6-3 sum of bits problems where other methods find no solutions. We have also shown that GP+CADF requires a lower computational effort to solve the parity, sum of bits, image recognition, lawn coverage and the bumblebee problems. Lastly, we have shown that GP+CADF has a consistently lower generation of convergence than GP+ADF, that is, it finds a solution in fewer generations.

To further improve GP+CADF, we discovered that using elitism and a single best evaluation coupling lowers the computational effort even further. In addition, improving the initial population by regenerating individuals with no ADF calls and choosing best initial individuals after M couplings, where M is the population size, also leads to a lowered computational effort. We have also experimented with different mutation operators and discovered that sub-tree and rehang mutation produced a considerable improvement.

The discovered solutions of GP+CADF were also analyzed to demonstrate that GP+CADF is able to find reusable subroutines as ADFs and use those subroutines to

exploit underlying regularities and repetitions. In addition, we have also shown that GP+CADF uses the ADFs to break the problems into smaller sub-problems and then assembles those parts into the final solution.

9.2 Future Work

9.2.1 Competitive Co-Evolution

The GP+CADF method is a co-operative co-evolution method. While not discussed formally in this thesis, experiments were also performed in an attempt to add a competitive co-evolution component. The competitive part comes from a separate test population where each fitness case is represented as an individual with its own fitness in addition to the candidate solution populations we already have. Then, the fitness of a candidate solution population individual would be determined as usual by the number of fitness cases it passes. The fitness of a test would be how many individuals do not pass that test.

The motivation here is that it is not feasible to evaluate all individuals against every test because this would be too time consuming as we see from problems such as 14-even-parity. When every test has a fitness, we could pick out the *hard* tests that most individuals fail. In this way we would not need to use all tests, but only the hard ones. If an individual passes the hard tests, we continue on to other tests, otherwise we stop the evaluation.

A number of variants of this kind of setup were implemented and tested with the parity problem. However, the results showed that even with 80% of the total tests used as hard tests before continuing on to other tests, the success probability dropped considerably. The parity problem seems to be such that all tests are relatively equal in

importance and we need almost all of them to guide the evolution in the right direction. It might be the case that other types of problems would be more suited to this technique.

9.2.2 Discovering the Function Set through ADFs

Lastly, another idea that came out of this work is to reconsider how restarts are done. In our experiments, when the evolution reaches G generations, we terminate the run and restart. We commonly perform many trials to obtain the success probability. The motivation is that due to premature convergence we might be better off restarting a run than let it continue for a large number of generations. However, there might be a case for a sufficiently hard problem where none of the runs are yielding a solution. We also might be working in an unknown domain where we do not know what functions would be best to include in the function set.

In this case we may begin to include the best discovered ADFs as functions in the function set for future runs. In an unknown domain where we do not know useful subroutines, this may be a viable option. The idea is that if many runs are not producing a solution, then we may need to start discovering the function set which can further be assembled into subroutines by the following runs.

References

[1] J.R. Koza, Genetic programming II: Automatic discovery of reusable programs, MIT Press. 1994.

[2] W. B. Langdon and R. Poli, Foundations Of Genetic Programming, Springer-Verlag New York, Inc., New York, NY, 2002

[3] M. L. Minsky and S. A. Papert, *Perceptrons*. Cambridge: MIT Press, 1969.

[4] R. Poli, J.Page, Solving High Order Boolean Parity Problems with Smooth Uniform Crossover, Sub-Machine Code GP and Demes, Genetic Programming and Evolvable Machines, v.1 n.1-2, p.37-56, April 2000

[5] Nedjah, N., (et al.), Improving Multi Expression Programming: An Ascending Trail from Sea-Level Even-3-Parity Problem to Alpine Even-18-Parity Problem, M. Oltean, Evolvable Machines: Theory and Applications, Springer Verlag, (editors), pp. 229--255, 2005.

[6] C. Gathercole and P. Ross, Tackling the Boolean Even N Parity Problem with Genetic Programming and Limited-Error Fitness, in Genetic Programming 1997: Proceedings of the Second Annual Conference, Stanford University, CA, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (eds.), Morgan Kaufmann, 13-16 July 1997, pp. 119-127.

[7] J. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, The MIT Press, 1992.

[8] Sean Luke, ECJ Evolutionary Computing in Java Framework, Java Documentation, 2006.

[9] Kumar Chellapilla, A Preliminary Investigation into Evolving Modular Programs without Subtree Crossover, GP98: Third Annual Conference on Genetic Programming, Jul 22-25, Univ. of Wisconsin, Madison, 1998.

[10] R. Aler, Immediate transfer of global improvements to all individuals in a population compared to Automatically Defined Functions for the EVEN-5,6-PARITY problems, Springer Berlin Heidelberg, 1998.

[11] P. Nordin and W. Banzhaf, Complexity compression and evolution. In L. Eshelman, editor, *Proceedings* of the Fifth International Conference on Genetic Algorithms, p. 310-317. Morgan Kauffman, 1995.

[12] J. Morrison, F. Oppacher, 'A General Model of Co-Evolution for Genetic Algorithms', in G. Smith, N. Steele, R. Albrecht (Eds.), Artificial Neural Nets and Genetic Algorithms, Springer, Vienna, 401-404, 1999.

[13] W. Daniel Hillis, Co-evolving parasites improve simulated evolution as an optimization procedure, Physica D, v.42 n.1-3, p.228-234, June 1990.

[14] N. Williams , M. Mitchell, Investigating the success of spatial coevolution, Proceedings of the 2005 conference on Genetic and evolutionary computation, June 25-29, 2005.

[15] Edwin D. De Jong, Jordan B. Pollack, Ideal Evaluation from Coevolution, Evolutionary Computation, v.12 n.2, p.159-192, June 2004.

[16] Josh C. Bongard , Hod Lipson, 'Managed challenge' alleviates disengagement in co-evolutionary system identification, Proceedings of the 2005 conference on Genetic and evolutionary computation, June 25-29, 2005, Washington DC, USA.

[17] S.G. Ficici and J.B. Pollack, Pareto optimality in coevolutionary learning, In Advances in Artificial Life: 6th European Conference (ECAL 2001), pages 316–327. Springer Verlag, 2001.

[18] S. Nolfi and D. Floreano, Co-evolving predator and prey robots: Do 'arm races' arise in artificial evolution?, Artificial Life, 4(4), 1998.

[19] Y. Kim, J. Kim, A Tournament-Based Competitive Coevolutionary Algorithm, Applied Intelligence, v.20 i.3, p.267-281, 2004.

[20] Lohn, J., Kraus, W. F., Haith, G. L, Comparing a Coevolutionary Genetic Algorithm for Multiobjective Optimization, In: Fogel, D., et. al. (eds.): Proc. 2002 Congress on Evolutionary Computation (CEC'02). IEEE Press, Piscataway NJ (2002) 1157—1162.

[21] Edwin de Jong, The MaxSolve algorithm for coevolution, Proceedings of the 2005 conference on Genetic and evolutionary computation, June 25-29, 2005, Washington DC, USA.

[22] Gul Muhammad Khan and Julian Francis Miller and David M. Halliday, Coevolution of intelligent agents using cartesian genetic programming, GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, volume 1, pages 269-276, London, 2007. ACM Press.

[23] Kotani, M. Kato, D., Feature extraction using coevolutionary genetic programming, Congress on Evolutionary Computation, CEC2004. v. 1, p. 614- 619, 2004.

[24] Mitchell A. Potter , Kenneth A. De Jong, A Cooperative Coevolutionary Approach to Function Optimization, Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature: Parallel Problem Solving from Nature, p.249-257, 1994.

[25] Mitchell A. Potter , Kenneth A. De Jong, Cooperative Coevolution: An Architecture for Evolving Coadapted Subcomponents, , Evolutionary Computation, v.8 n.1, p.1-29, 2000.

[26] Cai, Z. and Peng, Z., J., Cooperative Coevolutionary Adaptive Genetic Algorithm in Path Planning of Cooperative Multi-Mobile Robot Systems, Intell. Robotics Syst. 33, 1, 2002.

[27] J. C. Bongard, Coevolutionary Dynamics of a Multi-population Genetic Programming System, In Proceedings of the 5th European Conference on Advances in Artificial Life 1999. D. Floreano, J. Nicoud, and F. Mondada, Eds. Lecture Notes In Computer Science, vol. 1674. Springer-Verlag, London, 154-158.

[28] Vanneschi, L., Mauri, G., Valsecchi, A., and Cagnoni, S., Heterogeneous cooperative coevolution: strategies of integration between GP and GA, GECCO '06: In Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation 2006.

[29] Popovici, E. and De Jong, K., The effects of interaction frequency on the optimization performance of cooperative coevolution, GECCO '06: In Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, 2006.

[30] Tulai, A. F. and Oppacher, F., Combining Competitive And Cooperative Coevolution For Training Cascade Neural Networks, In Proceedings of the Genetic and Evolutionary Computation Conference 2002.

[31] Tulai, A. F. and Oppacher, F., Parallel Genetic Algorithm with Strategy Parameters Encoded as Chromosomes, In Proceeding (365) Artificial and Computational Intelligence, 2002.

[32] T. G. Tan, H. K. Laul, J. Teo, Cooperative Versus Competitive Coevolution for Pareto Multiobjective Optimization, Bio-Inspired Computational Intelligence and Applications, Springer Berlin / Heidelberg, v. 4688/2007, p. 63-72, 2007.