Enhancing Scalability in Genetic Programming With Adaptable Constraints, Type Constraints and Automatically Defined Functions

George W. Gerules

M.S., Computer Science, University of Missouri – St. Louis, 2011
B.S./B.A., Accounting, Rockhurst University, 1985

A Dissertation Submitted to The Graduate School at the University of Missouri-St. Louis
in partial fulfillment of the requirements for the degree
Doctor of Philosophy in Mathematical and Computational Sciences with an emphasis in Computer Science

August
2019

Doctoral Committee:
Cezary Z. Janikow, Ph.D.
Chairperson
Uday Chakraborty, Ph.D.
John Aleshunas, Ph.D.
Mark Hauschild, Ph.D.

ABSTRACT

Genetic Programming is a type of biological inspired machine learning. It is composed of a population of stochastic individuals. Those individuals can exchange portions of themselves with others in the population through the crossover operation that draws its inspiration from biology. Other biologically inspired operations include mutation and reproduction. The form an individual takes can be many things. It, however, is represented most of the time as a computer program. Constructing correct efficient programs can be notoriously difficult. Various grammar, typing, function constraint, or counting mechanisms can guide creation and evolution of those individuals. These mechanisms can reduce search space and improve scalability of genetic program solutions. Finding correct combinations of individuals, however, can be extremely challenging when using methods found in GP such as Automatically Defined Functions or other Architecturally Altering Operations.

This work *extends* and combines in a *unique* way previous work on Constrained Genetic Programming, Adaptive Constrained Genetic Programming and Automatically Defined Functions.

This dissertation shows, compared to previous stand alone mechanisms, that a new combination of genetic programming constraint mechanisms and Automatically Defined Functions improve scalability for a number of benchmark problems. The combination of constraint mechanisms include delayed max tree size per evolved generations, typing on the evolved programs, use of automatically defined functions, and use of adaptive heuristics for function and terminals on the evolved programs.

Initial results show that this combination of methodologies create smaller efficient individuals capable of handling larger problems. Moreover, this combined methodology works particularly well for constraints can be applied ahead of time.

ACKNOWLEDGMENTS

There have been numerous people that been a huge influence during this long journey. Here I acknowledge their contributions.

First of all, I'd like to thank my supervisor, Dr. Cezary Janikow. This thesis would not have been able to be completed without his expert input and feedback. I also, would like to acknowledge his guidance during my time as an instructor for the computer science program. Thanks for having me teach and mentor students during my years as an instructor.

I also have valued the many hours of conversation with the other members on my thesis committee, Dr. Uday Chakraborty, Dr. Mark Hauschild and Dr. John Aleshunas. All three of you, in a huge way, helped me navigate various topics related to this research. Thank you.

I'd like to acknowledge Dr. Sanjiv Bhatia, my masters adviser. His feedback on my first publication was invaluable. I'd also like to thank him for introducing me to teaching computer science.

During my time as an instructor, I never held my office hours in my actual office. I held those office hours in the computer lab. That's where the computer science students are after all. So here I'd like to put a thanks to all of the students I met and mentored there. It was gratifying to see their skills and confidence grow during time in the computer lab. I wish all of you the best.

Thanks go out to James Parr and Sarah Jennings of NASA's Frontier Development Lab. The summer of 2017 was amazing. That research experience helped guide portions of my research. I'd also like to thank the mentors for our space weather challenge, Dr. Mark Cheung, Dr. Andrés Muñoz-Jarmillo, Dr. Troy Hernadez and Dr. Daniel Angerhausen. Each of one of you helped in a deep meaningful way. Yes, fast paced research can be done and science communication is as important as science product.

The person who had the biggest impact on seeing this dissertation finished is my wife Dr. Ann Steffen. She was there every step of the way with her encouragement, love and support.

Contents

List of Acronyms

AAO Architecturally Altering Operation. ii, 5, 78, 94, 98, 102, 105, 108

ABGP Abstraction Based Genetic Programming. 7, 12, 13, 78

ACGP Adaptable Constraint Based Genetic Programming. x, 6, 7, 10–15, 26, 29–31, 36, 37, 49, 59, 63, 73, 74, 77, 80, 86

ACGPF Adaptable Constraint Based Genetic Programming with Automatically Defined Functions. x, 14, 15, 34, 36, 37, 44, 49–51, 54, 56, 59, 63, 73–80

ADF Automatically Defined Function. ii, x, xv, 3–7, 9, 12–19, 25–29, 31, 32, 37, 40, 42, 44, 45, 49–51, 54, 56, 59, 63, 73–75, 77, 79, 80, 86–88, 92, 94, 95, 98, 101, 102, 104–106, 109

ADGP Adaptive Genetic Programming. 88, 98

ADI Automatically Defined Iteration. xv, 13, 104, 106, 108

ADIS Automatically Defined Internal Storage. 104, 108

ADL Automatically Defined Loop. xv, 13, 104, 106–108

ADM Automatically Defined Macro. 101, 102

ADR Automatically Defined Recursion. xv, 13, 104, 107, 108

ADS Automatically Defined Subroutine. 5, 13, 104, 108

AI Artificial Intelligence. 1, 94

AR Adaptive Representation. 13, 88, 89, 91, 92, 94

ARL Adaptive Representation Through Learning. 13, 77, 78, 88, 92, 97, 98, 100, 102, 103

BB Building Block Hypothesis. 89

BB Building Block. 88–90, 92

BNF Bakus-Naur Form. 12

BOA Bayesian Optimization Network. 11

CGP Canonical Genetic Programming. 1

CGP Constraint Based Genetic Programming. x, 6, 7, 9, 10, 13–17, 26, 27, 29–31, 36, 49, 56, 59, 63, 73–75, 77, 80, 86

CGP2.1 Constraint Based Genetic Programming version 2.1. 10

CGPF Constraint Based Genetic Programming with Automatically Defined Functions. 14–16, 34, 36

CNN Convolutional Neural Network. 1

CO Combinatorial Optimization. 94, 95

CPU Central Processing Unit. 79

CSL Constraint Specification Language. 27

CSS Constrained Syntatic Structures. 8

DBB Defined Building Block. 5

EC Evaluation Complexity. 90

EDA Estimation of Distribution Algorithm. 11

EDF Evolutionary Defined Function. 87

ERC ephemeral random constant. 76

ES Evolutionary Strategies. 1

FOLDL The LISP language fold left function. 12

FOLDR The LISP language fold right function. 12

GA Genetic Algorithm. 1, 12, 89, 94–96

GP Genetic Programming. ii, xv, 1, 2, 4–13, 15, 18, 22–27, 31, 70, 73–76, 78–80, 86–94, 97, 100–102, 104–108

GPU Graphics Processing Unit. 79, 80

HGP Hierarchical Genetic Programming. 88, 89, 92, 93, 95, 97, 102

HLDM Hierarchical Locally Defined Modules. xv, 13, 102–104

HLDM_minor Hierarchical Locally Defined Modules Minor. 103, 104

HOF Higher Order Functions. 12, 78

IPB Iteration Performing Branch. 106

LA Lambda Abstractions. 12

LBB Loop Body Branch. 106

LCB Loop Control Branch. 106

LIB Loop Initialization Branch. 106

LISP List Processing Language. 8, 12, 32, 101

LUB Loop Update Branch. 106

MA Module Acquisition. 13, 86–89, 92, 98, 100–102

MAP The LISP language MAP function. 12

MDL Minimum Description Length. 77, 79, 88, 90–93

ML Machine Learning. 1

NTH The LISP language NTH function. 12

OAR Obstacle Avoiding Robot. 102

PBIL Population Based Incremental Learning. 12

PIPE Probabilistic Incremental Program Evolution. 12

POLYGP Polymorphic Genetic Programming. 7, 12, 13

POSC Principle of Strong Causality. 93

RBB Recursion Body Branch. 107

RCB Recursion Condition Branch. 107

RGB Recursion Grounding Branch. 107

RL Reinforcement Learning. 94, 95

RPB Result Producing Branch . 6, 7, 16, 17, 24, 32, 37, 45, 49, 74, 75, 86, 104

RUB Recursion Update Branch. 107

SA Simulated Annealing. 95

SC Structure Complexity. 90

SGP Standard Genetic Programming. x, 1, 6, 8, 13, 15, 37, 50, 51, 54, 56, 63, 73, 74, 88, 94, 104, 109

STGP Strongly Typed Genetic Programming. 6–8, 12, 13

System F The Girard-Reynolds polymorphic lambda calculus. 12, 13, 76

List of Terms

F−specification *semantic* constraint, one function that is disallowed. Here *F* means **false**. The set of things disallowed.. x, 9, 16–26

T−extensive F−*intensive*F−specification is the normal form.. 21

T−specification *syntactic* constraints are based on domains for function arguments and on function ranges. They are similar to function prototypes. Here *T* means **true**. The set of things allowed.. x, 9, 16–26

`"acgp1p1p12 nadf"` Used in in the key for figures in ch5. It means the ACGP framework, which does not have ADFs.. 63

`"acgpf2p1 yadf gr"` Used in in the key for figures in ch5. It means the ACGPF framework with ACGPF and using the generation ramp feature.. 63

`"acgpf2p1 yadf"` Used in in the key for figures in ch5. It means the ACGPF framework with ADFs and not using the generation ramp feature.. 63

`"cgp2p1 nadf"` Used in in the key for figures in ch5. It means the CGP framework, which does not ADFs.. 63

`"orig nadf"` Used in in the key for figures in ch5. It means the original SGP framework not using ADFs.. 63

`"orig yadf"` Used in in the key for figures in ch5. It means the original SGP framework using ADFs.. 63

F−intensive F−specification , are *F−specifications* that do not include any F_* constraints.. 20, 21, 25

T−extensive F−specification are semantics-based constraints extended by syntactic constraints on function calls.. 19

T−intensive F−specification list only some additional constraints – which cannot be derived from *T−specifications*. 19–21, 25

ACGP1.1.2 Adaptive Constrained Genetic Programming version 1.1.2. Adaptive heuristics, no types, no ADFs.. 26, 27, 29, 109

ACGPF2.1 Adaptive Constrained Genetic Programming with ADFs version 2.1. Adaptive heuristics, has types, has ADFs.. 26, 27, 30, 49, 109

CGP2.1 Constrained Genetic Programming version 2.1. It has types but no ADFs.. xv, 26–29, 109

CGPF2.1 Constrained Genetic Programming with ADFs version 2.1. It has types and ADFs.. xv, 26–30, 109

crossover The process of creating a new individual with material from 2 parents. Select a subtree in parent 1. Select a subtree in parent 2. Swap the subtrees thereby creating a new individual that has donor material from the two parents.. 24, 26

first-order counting heuristic. Counts parent node and one child relationship combination.. 11

function are routines which can appear in function nodes. e.g. The function $sin(a)$ can appear in a function node.. xi

function argument A function can have no parameters or can have one or more parameters. These 1 or more parameters are referred to as function arguments.. xi, 16, 19

function call the act of calling another function with possible function argument.. 19–21

function node the node in a program tree where a function can be placed.. 21–24, 26

internal node Function nodes are internal nodes.. 24

lilgp1.02 lilgp1.02 Original standard genetic programming framework. 79, 109

lilgp1.03 lilgp1.03 Original standard genetic programming framework with modifications to display statistics correctly for large scale gp runs.. 109

methodology A set of actions to investigate or research a problem. 26

MPICH Message Passing Interface. Coordinates programs and results from those programs on many of today's supercomputers. The original implementation of mpi was based on the Chameleon portability system. More information can be found at www.mpich.org.. 79

mutation The process of randomly changing a subtree.. 23, 25, 26

mutation set Whether it be through crossover or mutation, a tree is changing or mutating into a new form.. 23

normal form The minimal form that express Tspecification and Fspecification constraints. Normal form is the minimum set of constraints.. 23

off-line Heuristics extracted after a number of generations or after the evolution converges. Could be done to collect domain heuristics into a database or used in complex problem solving instances where one set of heuristics could be used as a starting point for another evolution process.. 11

on-line Heuristics extracted possibly every generation, often as possible. Heuristics could be used to guide evolution process during a particular run.. 11

search space The population of all possible combinations of functions and terminals that can be combined to formed programs.. 5, 7, 8, 16, 87, 88, 92, 101

second-order counting heuristic. Counts parent node child relationship combinations.. 11

strong constraint A type of constraint that will not allow an action or operation to occur.. 11

sufficiency principle which states that in order to solve a problem there must be enough functions and terminals.. 16

terminal node the node in a program tree where a terminal can be placed.. 16, 21–25

weak constraint placing a restriction on something or some action part of the time.. 11

what2 ACGPF or ACGP feature where we are using the configuration that extracts and uses heuristics produced by each generation . 50, 54, 56, 59

what3 ACGPF or ACGP feature where we are using the configuration that extracts and uses heuristics produced by each generation and then regrows the population . 50, 54, 56, 59, 63

zero-order counting heuristic. Just counts parent node.. xv, 11

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

In Machine Learning and Artificial Intelligence there are many biologically inspired techniques that help find solutions for problems. Some examples like Convolutional Neural Networks are modeled after the visual cortex of the brain and are particularly adept at image recognition.[1] Other techniques like the Genetic Algorithm by Goldberg (1989); Holland (1962) and Evolutionary Strategies by Beyer and Schwefel (2002) are inspired by operations performed in genetics. In GA and ES, potential solutions are encoded as binary strings of 0's an 1's in the case of GAs and as real numbers in the case of ESs. Still other techniques, like Genetic Programming, by Koza (1992) are also inspired by genetic operations; but for these, potential solutions are encoded as a population of programs represented as parse trees. Whether it be the 0's or 1's of a GA, the real numbers of ES or the individual parse trees of a GP, some individuals are more fit than others when measured against a desired goal. We will focus on GP for the remainder of this work.

An example, in GP, of a correct individual program fragment representing the trigonometric sin function

$$a \times sin(p \times x + s)$$

can be seen in figure 1.1.

In earlier work by Koza (1992) on GP, which is referred to as Standard Genetic Programming for purposes of this proposal[2], there was a requirement called *closure*. Closure is a requirement that a function can take as its argument any other combination of arguments from a set of functions and

[1] See section 9.10 in *Deep Learning* by Goodfellow et al. (2016).

[2] In some research they refer to Standard GP as Canonical Genetic Programming. The use of the acronym (CGP) is going to conflict with the acronym for another framework that we are going to highlight, Constraint Based Genetic Programming. So, we'll use the acronym SGP.

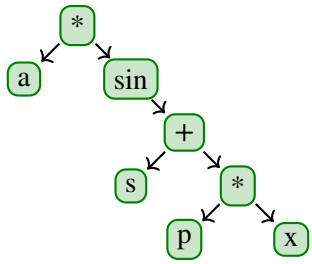


Figure 1.1: Correct Example Individual GP Program

terminals. A problem with this approach is that undesirable function and terminal combinations can take place. For example there is no restriction for a program fragment being created as the following.

$$sin(\pi \times sin(\sqrt{\pi}))$$

The parse tree which can be seen in 1.2. Including these incorrect program fragments only makes the search space unnecessarily large.

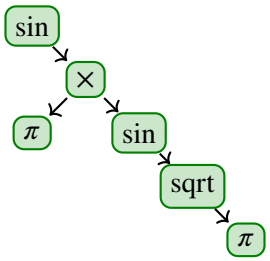


Figure 1.2: Incorrect Example Individual GP Program

There are a number of mechanisms that can help in forming correct program fragments. Some of these include work by Montana (1993), Janikow (1996b), Yu and Clack (1998) and Binard (2009). In addition to these methods of construction, there are heuristic methods where the program tries to learn which fragments make better combinations than others. Examples of these efforts can be found in Janikow (2004a) and in Rosca and Ballard (1996).

The research contained in this dissertation reduces this large search space, as generated by the closure principle, in a rather dramatic way. This is accomplished through a combination of previous methodologies, described in chapter 4. Positive results can be seen on scaled up problems in chapter 5.

Before we get to methodology and improved results for this research, we need to take a look at previous efforts in this area. We'll take a look at research that is a basis for this improved method in chapter 3. Other related and important material is found in appendix A.

The next section contains an outline of what is covered in this dissertation.

1.2 Outline of Dissertation

This dissertation includes the following.

Chapter 2: Thesis Statement – Goals of this research effort.

Chapter 3: Literature Review – Efforts related to this topic have been explored by previous researchers.

Chapter 4: Methodology – Theoretical and practical considerations on completing this body of research.

Chapter 5: Results – A description of experiments and an analysis of results.

Chapter 6: Results Discussion and Future Directions – Observations are made based on results, and possible future directions of research.

Appendix A: Additional Background Literature – Important related material on ADFs and constraints. This section can be skipped on first reading. It is, however, included if more detail is needed on related bodies of research.

Appendix B: Computer Environment for Experiments – The computing environment where experiments were run.

Chapter 2

Thesis Statement

Many research efforts in Genetic Programming, as reviewed in Chapter 3 and Appendix A., either use a typing mechanism *or* constraint mechanisms *or* heuristic methods *or* ADFs. Each one of these have been used to investigate an aspect of reducing the search space of optimal programs. What if we combine all of these? Would this help with scalability of GP problems? Would it improve efficiency in GP? Would it make for smaller GP solutions?

To show the benefit of the combined power of each of these mechanisms, this work extends in a *unique* way previous work on Constrained Genetic Programming and Adaptive Constrained Genetic Programming.

This dissertation shows, compared to previous mechanisms, that a combination of genetic programming constraint mechanisms improve scalability for a number of benchmark problems. The combination of constraint mechanisms include delayed max tree size per evolved generation, typing on the evolved programs, use of automatically defined functions, and use of adaptive heuristics for functions and terminals on the evolved programs.

Chapter 3

Literature Review

3.1 Introduction

This part of the dissertation reviews previous related research efforts. The scope of this effort includes how subroutines and various constraints are used to reduce the complexity of the search space. The review follows the chronological flow of when ideas were introduced.

In a published survey of modularity in Genetic Programming, Gerules and Janikow (2016), reviewed a number of research efforts. This dissertation revisits many of those efforts outlined in that publication, but it also reviews studies that address scalability.[1] All of this serves as a background for this dissertation.

During early research on Automatically Defined Functions in Genetic Programming different terms were used to mean a function body or a subroutine. Those terms changed a lot during the course of research but, roughly, meant the same thing. For example, just focusing on Koza's body of research, the terminology for subroutines has changed since his early work on ADFs. Originally a subroutine was called a Defined Building Block, in Koza's (1992) first book. In Koza (1994b) the terminology for a subroutine changed to Automatically Defined Function. By his third book, Koza et al. (1999), the term Automatically Defined Function included the meaning for the term Automatically Defined Subroutine. That last example also includes his terminology in AAO. Other researchers use the terms module and macro to mean a subroutine. Outside of this literature review chapter we aim for consistency. We will use the term subroutine and ADF to mean a body of code that is called 0 or more arguments and returns 0 or more values. We will use the term "framework" to mean the software that expresses a researcher's implementation of their particular methodology. But,

[1] At the time of the compilation of the survey in 2015 – 2016, this author had not yet explored scalability. After participation in a summer 2017 research workshop by NASA's Frontier Development Lab, during which we were tasked with using AI techniques paired with extremely large data sets. As a result of a failure to apply GP to those large data sets, scalability was added as a part of this dissertation.

inside this chapter when reviewing a particular framework, we will use the author's own terminology when referring to a subroutine.

Figure 3.1 has a representation of frameworks discussed in this dissertation. This graph represents a hierarchy of the technology on which a particular framework was based. This includes constraint based mechanisms and use of subroutines. A few frameworks that do not have subroutines are included as a focus to address some aspect of scalability. For purposes of this dissertation, we are focusing on the following branches of research SGP, ADF, STGP, CGP and ACGP. Other branches of research are included in Appendix A and be safely skipped on first reading.

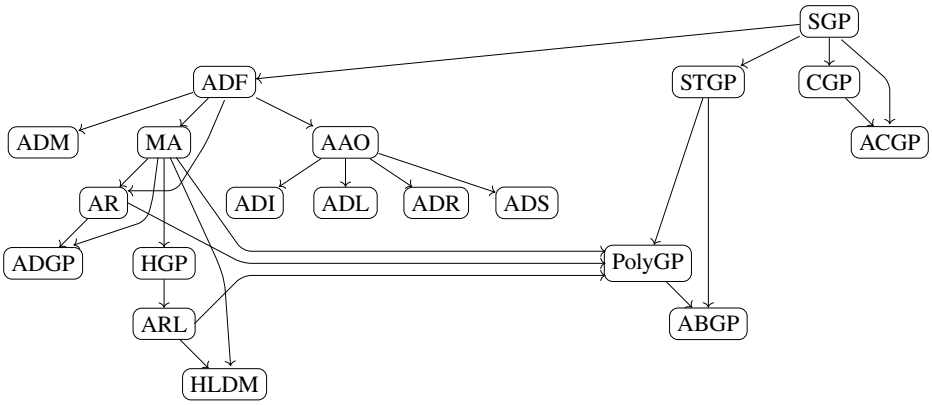


Figure 3.1: Frameworks

3.2 Automatically Defined Functions - (ADF)

This section introduces the Automatically Defined Function concept and how it is used in GP. We get a flavor of how ADFs are used for a particular program in a GP population in the next figure 3.2. In Koza (1994b), he indicates there can be 2 or more program trees making up an individual program. One tree contains the Result Producing Branch and the other tree contains the ADF. The RPB can or can not contain a call to an ADF. The ADF can or can not make a recursive call to itself. In figure 3.2a, the RPB makes a call to the ADF in figure 3.2b with three parameters. Notice that the ADF tree contains terminals as arguments to the ADF. This may be done to control side effects. Side effects happen when the ADF makes changes to a shared global variable used by one or more other functions.

For Koza's early work on ADFs, the function's name, returning value and parameters were formed prior to program being evolved during a GP run. During a GP run, the body of an ADF could undergo evolutionary changes. If the RPB found a particular ADF useful it would place more calls to that ADF. Crossover and or mutation could play a role in the successful ADFs being formed.

In Koza (1994b), 8 claims are made on how ADFs help improve GP. Many of the claims involve scalability in using ADFs to efficiently solve problems.

(a) Result Producing Branch (RPB)

(b) Automatically Defined Function (ADF)

Figure 3.2: ADF Example Individual

Since Koza's introduction of ADFs, others have explored subroutines and how they can be constructed. As noted previously, terminology defining a subroutine has changed, but there is some commonality regarding the nature of a subroutine.

Accross Koza's body of work, relating to ADFs, many benchmarks are used to showcase a particular aspect of GP. These range from symbolic regression, path finding, video game play, electronic circuit synthesis and constructed problems.[2]

3.3 Constraints in Genetic Programming

In the following sections, we take a look at areas of research where constraints on the search space are the primary focus. Montana's work on Strongly Typed Genetic Programming is followed by Janikow's on CGP. This includes constraints using the constraint language found in Constraint Based Genetic Programming, (CGP), and adaptive constraints found in Adaptable Constraint Based Genetic Programming, (ACGP). This is followed by Yu and Clack's work with Polymorphic Genetic Programming, (POLYGP). We finish out this section with Binard and Felty's work on Abstraction Based Genetic Programming, (ABGP).

3.3.1 Strongly Typed Genetic Programming - (STGP)

In Koza's original design of GP, there was no restriction on how various elements could be combined which could lead to an unnecessarily large search space. This was because of the closure property for GP. Montana proposed a restriction on the search space using a typed constraint system.Montana (1995)[3]

Montana (1995) introduces data typing by describing how it is used in several programming languages. For example in the languages Ada and Pascal data types are checked at compile time.

[2]Constructed problems are a category of problems that exercise a part of what GP can accomplish; for example the two box volume difference problem, which is used to test ADFs.

[3]There are 3 papers by Montana entitled Strongly Typed Genetic Programming. The first was published in Montana (1993). Followed by Montana (1994). The next paper published in Montana (1995). The later papers are updates for the original STGP algorithm. We focus on the last paper.Montana (1995)

This is called static typing. Other languages like LISP use dynamic typing. For this language, the data type is checked when the program is run. Montana (1995) points out that in Koza's SGP, data types are not strict and can cause inefficiencies because of the closure property and lead to incorrect mixing of operations on functions and terminals. For example, in SGP there is no support for vector and matrix data types so there could be incorrect operations on these data types if they were present in SGP.

Montana's (1995) implementation for strong typing is similar to Koza's Constrained Syntatic Structures as found in chapter 19 of Koza (1992). Both, mechanisms put syntactic constraints on the kinds of valid programs that can be constructed within a GP population. Where STGP differs from CSS is in the construction and use of generic functions and generic data types. A main motivation for his method is to have GP correctly handle vector and matrix calculations.

An outline of the modifications to SGP to implement strong typing is provided here in detail. In STGP, each variable and constant has an assigned type ahead of time. Each function argument is also assigned a type. There are additional requirements that program parse trees be correctly formed. These include that first, the root node must return a correct type for the problem. And second, there must be agreement of types for parent and child nodes. In the initialization process, if a tree can not be recursively constructed correctly, it is thrown away and a new attempt at tree construction is started. The genetic operations for crossover and mutation occur only if the types match. Generic functions are implemented to further carry out correct semantic behavior of evaluations. For example, performing a dot product operation for matrix calculations can have one generic function for both $2D$ and $3D$ matrices while excluding the vector data type from evaluation. Another modification for STGP was to have a special data type, called void, that is used to indicate that a function is a procedure and returns no data. STGP allows for local variables in a function. The drawback of this is that they have to be specified ahead of time by the user and must require some domain knowledge of the problem ahead of time. A global variable that signifies an error code is used by a run time error system. The calling function checks and takes appropriate action if the error code is set. For example, one of the bench mark problems tries to evolve the LISP NTH function. An error code could be set if it goes beyond the legal boundary. Another example that is used is a time limit error code. This error code is set if a function takes too long to run.

Results for Montana's (1995) research show that STGP scales well when compared to random search methods. Four benchmark problems were used: the multidimensional least squares problem; evolving a portion of the multidimensional Kalman filter; the lisp NTH function; and the lisp MAPCAR function.

3.3.2 Constraint Based Genetic Programming - (CGP)

Independently of Montana's (1995) work and during roughly the same time period, Janikow's (1996a) describes in a series of papers, how to place constraints on the search space. For this section, we outline some of the themes of this body of this work which is relevant to this dissertation. We leave

some details, where noted, to a later chapter on methodology.

Six papers, Janikow (1996a), Janikow (1996b), Janikow and DeWeese (1997a) Janikow and DeWeese (1997b), Janikow and Mann (2005) and Janikow and DeWeese (1998) outline the core of CGP. The main themes for CGP can be found in Janikow (1996a) and Janikow (1996b). The generalized methodology can be found in Janikow (1996a). A weaker version of this was used for implementation purposes in Janikow (1996b). These are generally reviewed here and explored in detail as a part of this dissertation's methodology section. Most of these papers mention ADFs for possible future research opportunities.

In order to place constraints on the search space Janikow and colleagues build a generalized constraint methodology, using mathematical notation. Set theory language is used heavily in their methodology. Major themes are summarized next.

The first theme involves building sets of what functions and terminals are allowed and disallowed. The first type of set that holds allowed objects is referred to as *syntactic constraints* and is represented by the notation $T-specifications$. The second type of set that holds disallowed objects is referred to as *semantic constraints* and is represented by the notation $F-specifications$. Here T and F do not refer to terminals and functions. The T and F are used as monikers for allowed specifications and disallowed specifications respectively. This lays a foundation for the next theme.

The second theme involves compatibility of functions and values. Here a function uses the standard notion that a function has a domain and a range. The specification of the function's arguments are its domain of usable objects. The specification of objects that a function can return are its range.

Building of these allowed and disallowed sets based on compatibilities brings us to the third theme. Building can be done in an intensive way or an extensive way. Terminology here is borrowed from set theory and is described next. This description is based heavily on a definition found in Cook (2009).

An intensive set definition has the necessary and sufficient conditions for set definition. The classic definition as seen in Cook (2009) goes as follows. The intensive definition of a bachelor is an unmarried man. It is necessary because a person that is a man cannot be a bachelor without being an unmarried man. It is a sufficient condition because "any" unmarried man is a bachelor. An extensive set definition just lists everything in the set.

Successively smaller and smaller sets are built through specification of function's compatible domain and range. This is done through careful building of combinations of what is allowed or disallowed combined with whether it was built intensively or extensively. There are many combinations. For each node, there is a set of what can be placed at that node without invalidating the program tree. In the next theme, we look at how an individual program tree is initialized.

When initializing individuals, we need to discuss what is called a *mutation set*. A mutation set helps construction of a minimal set of rules that govern whether an individual is valid or not. Here mutation is not related to the GP mutation operator. It is related to the set of allowable items that can be placed at a node. Next we move to CGP crossover and mutation.

For mutation we randomly pick a node and regenerate the tree that is below that node. The

mutation set is consulted for proper parent child combinations as nodes are generated for this sub tree. For crossover, a random node is chosen in parent trees and two sub trees are swapped. Here too, the mutation set is consulted for what is an allowable replacement candidate based on parent child rules stored in the mutation set.

Overloaded functions and their constraints were added to CGP methodology in Janikow and DeWeese (1997a). Also, an in depth exploration of constraining the search space is also found in Janikow and Mann (2005).

For this methodology, the constraint language just restricts the search space to valid programs. There was no mechanism to track heuristics to help guide the evolution process. That work, as we shall see in the next section, is taken up by Adaptable Constraint Based Genetic Programming.

A new unpublished but documented operator was added in CGP. In Janikow (2007b) it is called the collapse operator and documented in the user manual for CGP2.1. This operator is a variant of a mutation operator. A random node is chosen in a sub tree and is placed at a higher location in the overall tree. Since the sub tree has a smaller tree height and is placed at a higher location in the overall tree, it collapses the tree at the higher location. Care is taken to ensure proper parent child relationships.

Problems used for experiments were the 11-multiplexer, inverse kinematics related to control of a robotic arm, a classifier for a concept learning system and the Santa Fe Trail.

A goal in CGP is to create a constraint language that a GP program can use to help assemble correct programs from the set of all programs. The two papers discussed, so far, form a methodology for implementation papers. This implementation of the constraint language can be seen in technical reports Janikow and DeWeese (1997b) and Janikow and DeWeese (1997a) a publications Janikow (1996b) and Janikow and DeWeese (1998). As noted previously, we revisit some details in the next chapter on methodology, but for now we are moving onto introduction of ACGP

3.3.3 Adaptive Constraint Based Programming - (ACGP)

Another kind of constrained GP was developed after CGP. This constraint system uses an adaptive heuristics.

Papers by Janikow and Deshpande (2003), Janikow (2004b), Janikow (2004a), Janikow (2005a), Janikow (2005b), Janikow (2007c), Janikow et al. (2011), Aleshunas and Janikow (2011), Janikow and Aleshunas (2013) and Aleshunas (2013) make up the body of work on investigating ACGP. A few are introduced here and others revisited later in the methodology section of this dissertation.

ACGP is a progression from work on CGP. In CGP, the constraint language reduces the search space to one of correctly constructed programs. This is good, as it reduces incorrect combinations of functions and terminals. In CGP we want to evolve solutions from correctly assembled programs. There might be a way of having GP learn from previous solutions during the course of a run. This leads to the methodology on adaptation for CGP. Next we introduce introduction some concepts that are used in this methodology.

The first concept is that ACGP uses, as inspiration, some EDA techniques similar to those found in the work of Pelikan et al. (1999). An estimation of the distribution of labels, for nodes, in the GP tree are calculated, then combinations of statistics are calculated for parent-child relationships for nodes. In Bayesian Optimization Network, an individual is a binary string of $0's$ and $1's$. In GP, an individual is made up of a tree or trees of functions and terminals. More information on the parent child relationships in GP are addressed later in this section.

The second concept is that of *strong* and *weak constraints*. A strong constraint disallows some combination of functions and/or terminals all of the time. A weak constraint disallows a combination of functions and terminals some of the time. Weak constraints involves probabilities while strong constraints do not involve probabilities. Both kinds of constraint are discussed in papers cited above.

The third concept is that of on-line and off-line heuristics. This kind of heuristic deals with how information is generated and used. In on-line heuristics, information is generated and potentially used at the end of every generation to help guide evolution. In off-line heuristics information generated at the end of a GP run and potentially used for the beginning of another GP to guide the start of another GP run or runs. Combinations of both of these can be used. Two useful papers related to this topic are Janikow (2004a) and Janikow (2007c).

A fourth concept is that of *global* and *local heuristics*. If one wants to place a particular kind of node at the root, that would be a global heuristic. If one wants to have certain kinds of parent child relationships regardless of the specific position in the tree, that would be a local heuristic. A good description of this can be found in Janikow et al. (2011).

A fifth concept is that of *zero*, *first* and *second order heuristics*. This kind of heuristic involves parent child relationships between nodes. Figure 3.3 give a visual depiction of these relations. The importance of identifying these relationships is in helping identify and evolve good program structure. A good description of this can be found in Janikow et al. (2011). Collecting statistics on these relationships is referred to as the local distribution technique, which helps guide evolution and also identifies good parent-child relationships. More information can be found in Janikow (2004b). This raises the question of what happens with higher order heuristics. That was investigated by Aleshunas and Janikow (2011). The main conclusion in that paper is that increased processing time and memory usage of higher order heuristics are prohibitively expensive to use higher order heuristics.

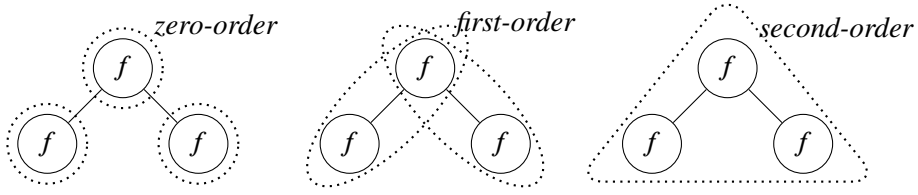


Figure 3.3: The three different levels of heuristics. Note that zero-order heuristics are meaningless if the only node information is the node's label.

A sixth concept can be found in a new operator for ACGP called "regrow" and first appears in

the publication Janikow (2004b). Regrow works by regenerating the population from statistics kept in the mutation set, and was found beneficial for longer term runs in ACGP. This idea is similar to PIPE in Ondas et al. (2005) for GPs and PBIL in Baluja (1994) for GA.

There are a few other items to note for this first look at ACGP. First, for most of the research on ACGP, ADFs were not mentioned. Second, most of the research was on untyped GP programs. There was one technical report, Janikow (2005a), where types were integrated into the ACGP methodology.

Experiments were conducted using the 11-multiplexer and the Santa Fe Trail problems.

The next sections address how functions and typing are continuing themes for reducing the search space.

3.3.4 PolyGP

Techniques from functional programming are used for GP research by Yu (1999). These include function polymorphism, implicit recursion and Higher Order Functions.

Constraints are handled by using two types of grammars. These grammars help produce type correct programs and help with polymorphism. The first grammar is used to help define correct expressions. The second grammar is used to help define correct types those expressions can use. Programs in this system are similar to Bakus-Naur Form. These grammars helps constrain search space making for efficient evaluation of individuals in the population of programs. She points out that the type system is different than Montana's (1993) STGP in key way. In STGP type information is stored in a lookup table while in POLYGP type information is part of the grammar that helps create type correct programs.

Implicit recursion is explored as part of this work. This type of recursion always terminates based on input. Examples of implicit recursion can be found in the LISP functions MAP, FOLDR, FOLDL and NTH.

Higher Order Functions can be evolved in the POLYGP system. A HOF is a function that can accept and return other functions. Subroutines in the POLYGP system are Lambda Abstractions. And these abstractions can handle multiple types.

Polymorphic functions in POLYGP are handled through a typing system.

The unique insight for this research is that these functional programming techniques help set the stage for using GP for automatic theorem proving which we see in the next section.

3.3.5 Abstract Based Genetic Programming - (ABGP)

In work by Binard and Felty (2007) and Binard and Felty (2008) a system called ABGP combines ideas from mathematical logic and computer science. His work uses a typed lambda calculus developed by Girard and Reynolds called System F. Genetic programming is applied to a framework that has System F as it's foundation.

System F is defined by two grammars, one for the data type of the term, and the other for how terms are combined. Binard uses System F properties to constrain search space thereby making for efficient searches for solutions. Now, because System F uses the Curry–Howard isomorphism, second order logic statements can be used in System F syntax. For example, second order logic statements that involve $\exists x$ and $\forall x$ can be used and transformed using second order proof methods. An individual in ABGP is a statement to be proved using second order logic. It is interesting to note that it is one of the first uses GP for automatic theorem proving. For a more in depth review of his work see Gerules and Janikow (2016).

3.4 Observations and Summary

We have seen many of the frameworks discussed in previous sections contain elements of using ADFs that have their function bodies modified. Some allow the function arguments to be added or deleted as we have seen for ADS. As described in Appendix A, some allow ADFs themselves to be added or deleted dynamically from the list of functions as we have seen in MA, AR, ARL, ADS, ADI, ADL, ADR and HLDM. Other researchers have focused on heuristics and constraint based mechanisms as in STGP, CGP, POLYGP and ABGP. All of these efforts address some aspect of scalability in regards to finding solutions to problems quicker. Table 3.1 shows a comparison for these frameworks. This includes the frameworks as described in the Appendix A. The table shows the following. One, for functions, used was there a typing mechanism used for correct pairings of functions to functions and functions to terminals. And two, were ADFs used. If ADFs were used, was the name, arguments and number of ADFs set prior to a GP run ; or were the ADF constructed on the fly during a GP run. In the table, if an ADF is set prior to a GP run it is called *static*. If an ADF is created on the fly it is called *dynamic*.

		SGP	ADF	STGP	CGP	ACGP	MA	AR	ARL	ADS	ADI	ADL	ADR	HLDM	POLYGP	ABGP
Functions	UnTyped	x	x			x	x	x	x	x	x	x	x	x		
	Typed			x	x										x	x
Subroutines	Used		x	x			x	x	x	x	x	x	x	x	x	
	Static	x	x	x												
	Dynamic						x	x	x	x	x	x	x	x		

Table 3.1: Attributes of Frameworks

In the next chapter two frameworks are chosen, CGP and ACGP, as a foundation for work in this dissertation.

Chapter 4

Methdology

4.1 Introduction

This chapter outlines enhancements for current CGP and ACGP methodologies. This body of work extends current CGP and ACGP methodologies. How these two new methodologies fit into the overall body of research can be seen as the gray shaded nodes in Figure 4.1. The generalized, *CGP notation found in Janikow (1996a) and in Janikow (2007a) is developed and* ***extended*** *to handle ADFs*. We'll call this enhanced methodology CGPF, constraint based genetic programming with automatically defined functions. This enhanced CGP methodology is used, in turn, to enhance the ACGP methodology. We will call that methodology ACGPF, adaptive constraint based genetic programming with automatically defined functions. A new global constraint is introduced also to help combat bloat. This new constraint places a cap on tree growth for a number of generations. It delays the max tree size for trees in individuals and gives trees that make up an individual more time to work with given functions and terminals. Benchmark problems are described. The method section concludes with a description of statistical analyses for comparing one framework to another.

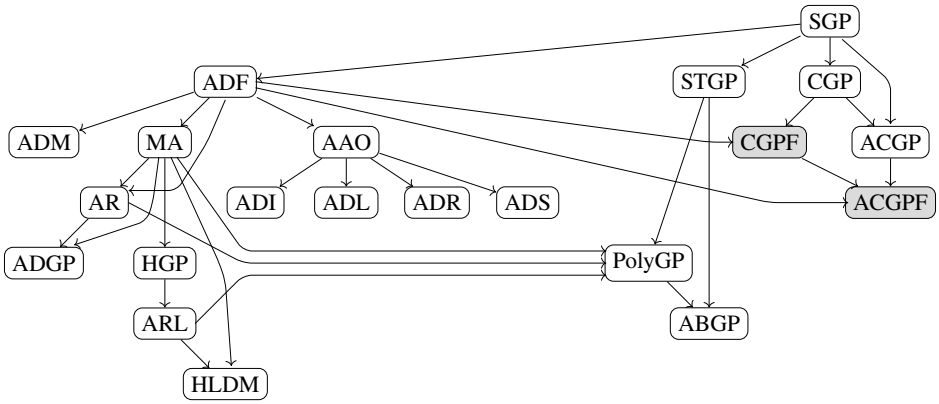


Figure 4.1: New Frameworks Relating to Overall Area of Research

The aim of this work is to demonstrate that enhancing CGP with ADFs is superior to original CGP which only has types, and enhancing ACGP with ADFs and CGP types is superior to ACGP that only has heuristic discovery.

Table 4.1 shows relevant features of frameworks under investigation. "Current" means what is current state of the art. "New" refers to new methodologies and frameworks created for this dissertation. "SGP" means Standard GP, "CGP" means Constrained GP, and "ACGP" means Adaptable Constrained Genetic Programming. ACGPF combines features of all previous frameworks.

Framework	ADF	Strong Constraints (Typing)	Adaptive Heuristics
(Current) SGP	x		
(Current) CGP		x	
(Current) ACGP			x
(New) CGPF	x	x	
(New) ACGPF	x	x	x

Table 4.1: Comparison of Current and New/Proposed Features

Enhancements to CGP giving rise to ACGP is reviewed next.

4.2 Enhancements to CGP and ACGP Foundations

This work begins with the original CGP methodology from Janikow (1996a) and then introduces a slight modification in the notation to allow multiple trees and constraints to be defined. Much of the notation looks very familiar. This extended terminology is developed in the same order as the original work in Janikow (1996a). Wording is similar if not the same as original CGP so that comparisons can be made on the extension.

The objective is to extend the constraint specification notation as seen in Janikow (1996a). The

goal is to make sure that only valid programs are evolved that use ADFs. ***This involves extending, not replacing notation.*** Addng ADFs makes the search space larger than in Janikow (1996a). Our goal is the same though. We want the effective search space constrained to the space of valid programs that include valid ADFs.

We now proceed in a similar fashion, as in Janikow (1996a), to take a look at the search space of possible program structures.

Let T be terminals. Let F be functions. Each $f_i \in F$ has fixed arity a_i.

As in the original paper Janikow (1996a), function compositions help create the space of possible program structures. Function compositions also include ADFs. This is where we start ***extending the original CGP methodology*** by introducing the notion of program tree. In original CGP, each individual program was isolated and was a single parse tree. In CGPF, however, there are multiple parse trees for an individual. This is similar to Koza's ADF method, but here we are constraining, in a general way, using ADFs and types. There is the RPB and one or more ADF trees all using types. For our purpose, branch and tree are synonymous.

The number of nodes in any tree for any individual can be infinite, so we also restrict the search space the same as in the original paper. But here, we have multiple trees that make up an individual. So we can: restrict the number of nodes per tree, and/or restrict the maximal depth per tree for the program. We are hoping to find the correct combination of nodes in trees for an individual that contains the correct solution. But, there may be many such solutions. This is Koza's sufficiency principle.

Terminals, T, are values that can appear in a terminal nodes and are at the edges of the program tree. They do not determine program structure.

We now ***extend*** the original definition for $T-$*specifications* and $F-$*specifications*.

4.2.1 T- and F- Constraint Specifications

As with the original CGP constraint specifications, there are two different kinds of specifications. Here the constraint is a specification on what function or terminal is allowed or disallowed. These constraint specifications are not completely separate from each other.

The first constraint specification is denoted as $T-$*specification* and is a *syntactic constraint*. The second constraint specification is denoted as $F-$*specification* and is a *semantic constraint*. Because we are working with multiple trees per individual, we need to be careful on applying the original transformations found in original CGP. Here we perform them in a recursive way on a per tree basis. Like original CGP, because there is overlap between allowed and disallowed constraints, only a few constraints per tree are needed. Our constraint language will grow in size, because we are working with multiple trees, but it is more powerful because it is working with a program structure that includes ADFs.

$T-$*specifications* are based on domains for function arguments and on function ranges.

A note on terminology is needed. When we talk about $T-$*specifications* or $F-$*specifications*,

the T doesn't refer to terminal and the F doesn't refer to function. Here $T-specifications$ refers to the set of *allowed* things that *can* be fed to or returned from a function. And correspondingly, $F-specifications$ is a set of *disallowed* things that *can not* be fed to or returned from a function. In a loose sense the T here stands for "true" and allowed. F stands for "false" and disallowed. The goal is to keep our notation consistent with the original CGP notation conventions.

Before we start extending CGP, we need to say a few words about subscript and superscript notations for what follows. In original CGP notation, a function's range was expressed as T_i where i would be the i^{th} function in a particular set of functions. The domain of the function would be expressed as T_i^j where i would be the i^{th} function and j would be the j^{th} argument of the i^{th} function. The subscript was reserved for functions and the superscript was reserved for arguments. If we add multiple trees for an individual we need to find place for the index of the tree number. This is important because when we introduce ADFs an individual can have multiple trees. One RPB tree and possible multiple ADFs trees. Now, where to place this index? For illustration purposes let us say m is tree number. We could place the tree number in the upper left as in ${}^mT_i^j$. This seems clean, but we run into a problem when we place two objects of this representation next to each other like ${}^mT_i^j, {}^nT_i^j$. This representation starts to give us trouble. If they are too close together it is easy to miss read where a super or sub script belongs. A better representation would be if we eliminate superscripts and just have positional subscripts where the position in the subscript produces a context for the kind of object being dealt with. We adopt the following notation for tree ranges. This takes the form $T_{m,i}$ where m is the tree number and i is the function number. And, for tree domains, it takes the form $T_{m,i}^j$ where m is the tree number, i is the function number and j is the argument number. In the section after this when we need to consider the kind of functions and terminals for implementation purposes, we rely on the subscript positioning to help with that. *All of this effort helps* ***extend*** *the original CGP notation and methodology. We will see this in the next portion of this dissertation.*

Definition 1. *Let us define $\Rightarrow$ to stand for domain compatibility. That is, $X \Rightarrow Y$ means that X can replace Y, where both X and Y stand for sets of values (finite or countably infinite) allowed for domains or returned as function ranges.*

Definition 2. *Define the following* $T-$specifications *(syntactic constraints):*

1. T_*^{root} *– the set of values allowed at a tree root. T_*^{root} actually specifies both a domain (for a tree root node) and a range (for a tree).*
2. $T_* - T_{m,i}$ *is the range of f_i, that is the set of values returned by a function f_i in a tree m.*
3. $T_{*,*}^* - T_{m,i}^j$ *is the domain for the j^{th} argument of a f_i in a tree m, that is the set of values allowed there (which may be returned by functions used as this argument).*
4. $T_{*,*} \overset{?}{\Rightarrow} T_{*,*}^*$ *– compatibilities between ranges and domains for a tree*
5. $T_{*,*} \overset{?}{\Rightarrow} T_{*,*}^{root}$ *– compatibilities between function ranges for a tree and the tree range*

$\overset{?}{\Rightarrow}$ *indicates whether there is a compatibility or not*

Definition 3. *Define the following* $F-$specifications *(semantic constraints):*

1. $F^{root}_{*,*}$ *– the set of functions disallowed at a tree root.*
2. $F_{*,*} - F_{m,i}$ *is the set of functions disallowed as direct callers to* $f_{m,i}$ *in a tree (generally, a function is unaware of the caller; however, GP constructs one or more trees, which represents the dynamic structure of the program). There are multiple trees because we may have ADFs in addition to the calling program tree.*
3. $F^{*}_{*,*} - F^{j}_{m,i}$ *is the set of functions disallowed as* arg_j *to* $f_{m,i}$ *in tree m.*

Example 1. *Assume a function (*$if\ arg_0\ arg_1\ arg_2$*), interpreted as: if* arg_0 *evaluates to true, return the evaluation of* arg_1*. This happens to call a zero argument ADF named* $ADF0$ *which returns a real number value, else return the evaluation of* arg_2 *which happens to also be a call to a zero argument ADF named* $ADF1$ *which also returns a real number value. The function* if *is tree number 0.* $ADF0$ *is in tree number 1.* $ADF1$ *is in tree number 2. Now let us specify* arg_0 *such that all terminals that are boolean values, or only functions that return boolean values. Now let us assume that before hand we specified* $T^{1}_{0,if} = \{T, F\}$ *and* $T^{2}_{0,if} = \{T, F\}$*. That is for* arg_1 *and* arg_2 *the* if *function in tree* 0 *can only handle boolean valued objects. That would mean that neither call to* $ADF0$ *or* $ADF1$*, which both return real numbers, is not compatible.*

Proposition 1. $X \Rightarrow Y \leftrightarrow X \subseteq Y$
$X \Rightarrow Y$ *means that in places where values from* Y *are valid, one may place any value from* X*, or any function returning a value from* X*. To guarantee that no out-of-domain values are used for the original* Y*,* X *may not contain values not found in* Y*. Therefore, it must be a subset of* Y*, or it must equal* Y*.*

Using properties of $\subseteq$, domain compatibilities could be automatically computed (giving compatibility $T-$*specifications* Definition 2 item 4, and 5, as long as these are restricted to syntactic constraints.

Example 2. *Assume two sets. In the first set,* $T_{0,2} = \{1, 2, 3\}$ *represents masses of physical objects in kilograms in tree 0.* $T_{0,1}$ *is tree* 0 *and object* 1 *which is 1 kilogram. In the second set,* $T_{1,2} = \{1, 2\}$ *representing times in seconds in tree 1.* $T_{1,2}$ *is tree 1 and object* 2 *which is* 1 *second. One may mistakenly conclude that* $T_{0,2} \Rightarrow T_{1,2}$ *because the numbers* $\{1, 2\} \subseteq \{1, 2, 3\}$ *for tree 0 and tree 1. But by looking at the type interpretations for these objects, an obvious conclusion is that* $T_{1,2} \not\Rightarrow T_{0,2}$*.*

4.2.2 Rules on T-Specifications and F-Specifications

Because of the definitions for the above to express constraints as $T-$*specifications* and $F-$*specifications*, there is the question of possibly redundant rules, or the existence of sufficiently minimal specifications.

We propose that after some initial preprocessing, that there is a sufficient minimal set of constraint rules. In other words, there is a set of specifications that are more easily expressed with original $T-$*specifications* and $F-$*specifications*.

We first need is to extend $F-$*specifications*.

Definition 4. *Define 'complete'* $T-$specifications *as those that list all elements of Definition 2, including ranges and domains for all functions and their arguments and compatibilities between all pairs range–domain and range–program range.*

Proposition 2. *The following* $F-$specification *constraints are implied by complete* $T-$specifications*:*

1. $\forall f_{m,n} \in F_m (T_m \nRightarrow T^j_{m,n} \rightarrow f_{m,n} \in F^j_m)$
2. $\forall f_{m,n} \in F_m (T_m \nRightarrow T^{root}_{m,n} \rightarrow f_{m,n} \in F^{root}_m)$

If $f_{m,n}$ *in a particular tree* m *returns a range which is not compatible with the domain for a specific function argument, then* $f_{m,n}$ *cannot be used to provide values for the argument. The same applies to values returned from the program.*

Proposition 2 is very important because the compatibility $T-$*specifications* from Definition 2 items (4, and 5) can be automatically generated from other $T-$*specifications*, according to the rule, they can be automatically translated to $F-$*specifications*. The later, as we see, are easier to handle.

Note that the opposite of these implications is not true because some $F-$*specifications* are based solely on interpretations. In other words, it is not true that $f_{m,n} \in F^j_{m,n} \rightarrow T_{m,n} \nRightarrow T^j_{m,n}$.

Note that the following is also not true either: $\forall f_{m,n} \in F_m (T_{m,n} \Rightarrow T^j_{m,n} \rightarrow f_{m,n} \in F^j_{m,n})$. See example 2).

Fortunately, the first implication is sufficient for us as it tells us that properly extended $F^*_{*,*}$, $F_{*,*}$ and F^{root}_* specifications subsume the $T_{*,*} \nRightarrow T^*_{*,*}$ and T_* $T-$*specifications*.

Example 3. *Suppose there are two functions in two trees. One is a primitive function in tree* 0 *and is function number* 1*,* $f_{0,1}$*. It returns a real valued number (*$T_{0,1} = \mathbb{R}$*). A second function,* $f_{0,3}$ *is an ADF used in in tree* 0 *and is function number* 3*. It can only accept boolean arguments (*$T^1_{0,3} = \{T, F\}$*). Because* $T_{0,1} \nRightarrow T_{0,3}$ *we can conclude that* $f_{0,1}$ *cannot be placed as the arguments to* $f_{0,3} : f_{0,1} \in F^1_{0,3}$*. Note: that the body of the ADF is evolved in another tree, but only takes boolean arguments.*

Definition 5. *If* $F-$specifications *explicitly satisfy Proposition 2 then call them* $T-$extensive $F-$specifications*. If* $F-$specifications *do not explicitly satisfy Proposition 2 for any function* $f_{m,n} \in F_m$*, then call them* $T-$intensive $F-$specifications*.*

In other words, $T-$*intensive*$F-$*specifications* list only some additional constraints – which cannot be derived from $T-$*specifications*. $T-$*intensive*$F-$*specifications*, on the other hand, are those semantics-based constraints extended by syntactic constraints on function calls. Note we are using

set theory definitions for extensive and intensive. An extensive definition lists all of the members of a set. An intensive definition give the necessary and sufficient conditions for objects to belong to a set Cook (2009).

Now, we look at redundancies among $F-$*specifications*.

Proposition 3. *Suppose* $f_{m,k} \in F_m$ *and* $F-$specifications *are T-extensive. Then*

$$\forall f_{m,i} \in F_m(f_{m,k} \in F_{m,i} \leftrightarrow \forall j \in [1, a_k] f_{m,i} \in F^j_{m,k})$$

If a function $f_{m,i}$ *cannot call* $f_{m,k}$*, then* $f_{m,k}$ *will never be called by* $f_{m,i}$*. Also, if* $f_{m,k}$ *is never called from* $f_{m,i}$*, it must not be called from any of* $f_{m,i}$*'s arguments for all trees m.*

With Proposition 3 one may wonder whether we need both $F^*_{*,*}$ and $F_{*,*}$ constraints – they seem equivalent. The next rule says they are not.

Proposition 4. *Suppose* $f_{m,k} \in F_m$ *and* $F-$specifications *are T-extensive. Then*

$$\forall f_{m,i} \in F_m(\exists j \in [1, a_i] F_{m,k} \in F^j_{m,i} \nrightarrow f_{m,i} \in F_{m,k})$$

The implication is true only when Proposition 3 applies.
If $f_{m,k}$ *cannot be called from* $f_{m,i}$ *by its* j^{th} *argument, it may possibly be allowed as another argument (unless, according to Proposition 3, it cannot be called from any of the arguments).*

Even though these constraints are not equivalent, both are not needed. It turns out that $F^*_{*,*}$ $F-$*specifications* are stronger.

Definition 6. *If* $F-$specifications *explicitly satisfy Proposition 3, call them* $F-$intensive$F-$specifications*. If* $F-$specifications *do not include any* $F_{*,*}$ *constraints, call them* $F-$intensive$F-$specifications*.*

Proposition 5. $F-$intensive$F-$specifications *are sufficient to express all possible* $F-$specifications*. According to Proposition 3,* $f_{m,k} \in F_{m,i}$ *can be deduced when* $f_{m,i}$ *is excluded from all arguments of* $f_{m,k}$*. According to Proposition 4, it can happen only when Proposition 3 applies. Therefore,* $F-$intensive$F-$specifications *provide sufficient information to produce* $F-$intensive$F-$specifications*.*

We now return to the question of $T-$*specifications* vs. $F-$*specifications*. We have seen that $T-$*intensive*$F-$*specifications* provide restrictions on function calls based on interpretations, and that they can be extended to $T-$*intensive*$F-$*specifications*, which also take syntax into account.

One question that comes to mind is: do we still need $T-$*specifications* after they have been used to produce $T-$*intensive*$F-$*specifications*? In other words, is there any constraint in $T-$*specifications* which is not expressed with $T-$*intensive*$F-$*specifications*? The answer is 'no' for certain $T-$*specifications*.

Proposition 6. *$T-$*intensive*$F-$*specifications *are sufficient to express constraints imposed by compatibility (#4 and #5) and $T_{*,*}$ (#2) $T-$*specifications*. Let us look at compatibilities of the form $T_{m,k} \stackrel{?}{\Rightarrow} T^j_{m,i}$. Proposition 2 says that the negated forms ($\stackrel{?}{\Rightarrow}$) are all expressed in $T-$*intensive*$F-$*specifications*. However, the straight form ($\Rightarrow$) can be superseded by $F-$*specifications*, which provide additional constraints based on interpretations. Thus, if $f_{m,k} \in F^j_{m,i}$, then the corresponding $T-$*specification *is irrelevant. On the other hand, if $f_{m,k} \notin F^j_{m,i}$ (in the T-intensive form), then we have two cases:*

- *if $T_{m,k} \not\Rightarrow T^j_{m,i}$, then according to Proposition 2 we put $f_{m,k}$ into $F^j_{m,i}$: $f_{m,k} \in F^j_{m,i}$ in T-extensive forms.*
- *if $T_{m,k} \Rightarrow T^j_{m,i}$, then we have no reason to extend $F-$*specifications *– thus, $f_{m,k} \notin F^j_{m,i}$*

*The same can be argued for $T_{m,k} \stackrel{?}{\Rightarrow} T^{root}_m$. As to $T_{*,*}$ $T-$*specifications*, they are sets of values returned by functions. Therefore, they place restrictions on function calls. But, $F-$*intensive*$F-$*specifications *express all possible restrictions on function calls. Said differently, $T_{*,*}$ is only used for other specifications.*

Definition 7. *Define $T-$*extensive*$F-$intensive$F-$*specifications *as the 'normal' form.*

Theorem 1. *(Fundamentals of T- and $F-$*specification *constraints) Even if the user provides only T-$F-$*intensive*$F-$*specifications*, T-$F-$*intensive*$F-$*specifications *can be computed, and along with domains and the program range they are sufficient to express all T- and F- specification constraints. Moreover, just the normal $F-$*specifications *along with domains and the program range are sufficient as well.*
This follows from Propositions 5 and 6.

Based on Theorem 1, we may now restrict our discussion to *$F-$specifications* only, assuming that these are in the *normal* form. To make sure they are, a simple preprocessing mechanism suffices.

4.2.3 Exploration of Constraint Handling Methods

We propose to implement the specified constraints into "smart" operators. To do so, we must define operators "closed" in the valid program structure – from valid parents always generate valid offspring. This also requires an initialization procedure with valid programs.

Definition 8. *In the program tree, we call 'function nodes' all nodes which correspond to a function. In this case, we say that the function labels the node. All other nodes are called 'terminal nodes'.*

Note the font change for T and F. Here, $\mathcal{T}$ are values and $\mathcal{F}$ are functions. This is different than the T and F specifications.

Definition 9. *Define $\mathcal{T}_{M,N}$ to be the set of values which can replace node N in tree M. That is, $\mathcal{T}_{M,N}$ is the set of values that the node in tree M can assume without invalidating the tree or the program as a whole, with respect to* $T-$specifications *and* $F-$specifications*, the program tree containing that node.*

Definition 10. *Define $\mathcal{F}_{M,N}$ to be the set of functions which can replace node N in tree M. That is, $\mathcal{F}_{M,N}$ is the set of functions which can label that node in tree M without invalidating the tree or the program as a whole, with respect to* $T-$specifications *and* $F-$specifications*, the program tree containing that node.*

For a terminal node, we cannot determine what other possible values can it contain by just looking at the node. We must look at the parent of the node (unless it is the *root* of a tree). For function nodes, we could either use the set of values returned by the function labeling that node ($T_{m,i}$ for $f_{m,i}$) in tree m. However, after replacing the function node with a terminal node, we would have to look at the context where the node appears. Therefore, we decide to use the context information even for function nodes.

As the subsequent rules state, the above sets not only can be efficiently computed, but some can also be guaranteed to be non-empty under certain conditions, which hold for GP. Moreover, in the next section we see that these sets can be precomputed for all possible node types, and that functions to extract random elements of these sets can be precomputed as well. This leads to a very efficient enforcement of these constraints.

Proposition 7. *Assume a node N is the j^{th} argument of $f_{m,i}$ in tree M and* $F-$specifications *are normal. Then,*

$$\mathcal{T}_{M,N} = T^j_{m,i}$$

$$\mathcal{F}_{M,N} = \{f_{m,k} | (f_{m,k} \in F_M) \wedge (f_{m,k} \notin F^j_{m,i})\}$$

Any value that does not invalidate the domain $T^j_{m,i}$ is OK. Any function that is not explicitly excluded form $F^j_{m,i}$ is OK. This is so because if $f_{m,i} \in F_{m,k}$, that is if $f_{m,i}$ cannot be accepted as a caller to $f_{m,k}$, then according to Proposition 3 $f_{m,k} \in F^j_{m,i}$, but it is not.

Proposition 8. *Assume a node N is the root for tree M and* $F-$specifications *are normal. Then,*

$$\mathcal{T}_{M,N} = T^{root}_m$$

$$\mathcal{F}_{M,N} = \{f_{m,k} | (f_{m,k} \in F_M) \wedge (f_{m,k} \notin F^{root}_m)\}$$

Arguments are analogous to those for Proposition 7, except that the root provides the constraints.

Proposition 9. *$\mathcal{T}_{M,N} = \emptyset$ for any terminal node in any valid tree M in a program. The valid program does not change when the terminal node is replaced with itself.*

Proposition 10. *As long as any function returns a value (as it is in GP), $\mathcal{T}_{M,N} = \emptyset$ for any function node in any valid program. If the function node is labeled with $f_{m,i}$, then it can be replaced with any terminal form $T_{m,i}$. This set is not empty as long as each function returns at least one value.*

Proposition 11. *For any function node N of any valid tree in a program, $\mathcal{F}_{M,N} = \emptyset$. If the node is labeled with $f_{m,i}$, then $f_{m,i} \in \mathcal{F}_{M,N}$.*

Note that $\mathcal{F}_{M,N}$ is not guaranteed to be non-empty for terminal nodes. That is, some terminals may only be used for computations, but are never be computed.

Note, that the symbol $\square$ means a function that returns the closest integer to its real-valued argument.

Example 4. *Suppose $F = \{f_{m,\square}\}$, and $f_{m,\square}$ returns the closest integer to its real-valued argument for tree m. Then, $T_{m,\square} = I$, $T^1_{m,\square} = R^1$, and $T_{m,root} = I$. Also, $(f_{m,\square} \in F^1_{m,\square}) \wedge (f_{m,\square} \in F_{m,\square})$ (n the T-extensive F-extensive form), but only $(f_{m,\square} \in F^1_{m,\square})$ is sufficient (in the normal form). For the program $(f_{m,\square}3.27)$, the terminal node 3.27 has $\mathcal{T} = I$ and $\mathcal{F} = \emptyset$.*

We can now define closed operators. We assume that all random numbers are taken from a uniform distribution. For any node N in tree M, denote

- $r^{\mathcal{T}}_{M,N}$ to be a random element from $\mathcal{T}_{M,N}$.
- $r^{\mathcal{F}}_{M,N}$ to be a random element from $\mathcal{F}_{M,N}$ (assuming that it is non-empty).
  For any terminal node N in tree M, denote
- $v_{M,N}$ to be the current value from that node in the tree

4.2.4 Mutation Operator

Assume that node N is chosen in a tree M for mutation. This selection can be based on a fixed probability of mutating any *allele* in all chromosomes (often called *post mutation* in GP), or on selecting a random *allele* in a given selected parent (*normal mutation* in GP).

Operator 1. *(mutation) If a node N is selected for mutation, then replace it with $r^{\mathcal{T}}_{M,N}$ with probability p^1_q, or with $r^{\mathcal{F}}_{M,N}$ with $1-p^1_q$. If $r^{\mathcal{F}}_{M,N}$ is used, then recursively repeat exactly the same Operator 1 on all arguments of the selected function for a particular tree M. If $r^{\mathcal{F}}_{M,N}$ is needed for the node N for tree M and the $\mathcal{F}_{M,N}$ set is empty, try another random node from the same parent (in normal mutation) or abandon the operation (in post mutation). If $r^{\mathcal{F}}_{M,N}$ is needed for descendent of N, tree M, and the $\mathcal{F}_{M,N}$ set is empty, use $r^{\mathcal{F}}_{M,N}$ instead.*

Proposition 12. *For any valid parent program, mutation Operator 1 is guaranteed to take place as long as $p^1_q > 0$. For a function node, Operator 1 is guaranteed to take place immediately. Moreover, all $T-$*specification *and $F-$*specification *constraints are guaranteed to be preserved. The parent is valid for a tree. According to Propositions 9 and 10, the set $\mathcal{T}_{M,N}$ is never empty. Therefore, as long as this set is allowed in mutation ($p^1_q > 0$), mutation eventually takes place on any node for a particular tree M. However, if N is a function node, then according to Propositions 10 and 11, both $\mathcal{T}_{M,N}$ and $\mathcal{F}_{M,N}$ are non-empty, so mutation immediately takes place regardless of p^1_q. The mutation sets are computed based on normal $F-$*specifications*, which are sufficient according to Theorem 1. Mutation set are calculated for each tree in a program.*

4.2.5 Crossover Operator

Because crossover with two offspring can be accomplished with two crossover operations, each with one offspring, we define crossover with one offspring. However, there can be multiple trees for an individual. Each kind of a tree is called a class. For crossover, the RPB class can only crossover with another RPB class. An ADF0 class of an individual can only crossover with another ADF0 class for another individual. An ADF1 class of an individual can only crossover with another ADF1 class of another individual and so on. In unconstrained GP, there are no specific constraints for a particular class. But if we impose constraints, we do not permit crossover between classes. Therefore, crossover is reduced to finding two random crossover points between trees that are of a particular class. In constrained "smart" crossover, the choices of plausible crossover points for a class can be highly reduced. Requiring that two offspring can be generated from the same two crossover points of a class further reduces chances of finding such points, but can be done if necessary. This brings up the question of whether there is more than one crossover operation if there are more than one tree per individual. For now we focus on one crossover operation between a tree in an individual. It would be a simple extension to allow a crossover operation for individuals containing more than one tree.

Definition 11. *Define $S_{(M,N,x)}$ to be the set of nodes from tree M from $parent_2$ which can replace a given node N selected for crossover. Classes of trees have to be the same.*

Proposition 13. *For crossover at node N_1 in $parent_1$ of tree M, and another $parent_2$ of the same tree class,*

$$S_{M,N_1,2} = \begin{cases} M, N_2 | v_{N_2} \Rightarrow \mathcal{T}_{M,N_1} & \text{if} M, N_2 \text{ is a terminal node in } parent_2 \\ M, N_2 | f_i \in \mathcal{F}_{M,N_1} & \text{if } M, N_2 \text{ is a function node labeled } f_i \text{in } parent_2 \end{cases}$$

Operator 2. *(crossover) If $parent_1$ and $parent_2$ are two crossover parents of the same tree class, select a random crossover point M, N_1 in $parent_1$, except that internal nodes have collective probability of p_c^1 and leaves have collective probability $1 - p_c^1$ (following standard GP practice of directing crossover into internal nodes). Based on whether M, N_1 is the root, apply Proposition 13 to compute $S_{M,N_1,2}$. If the set is not empty, then select a random node N_2 (leaves and internal nodes may be given distinct probabilities with p_c^1), and replace the subtree starting with M, N_1 with that staring with M, N_2. If the set is empty, try another crossover point M, N_1 from $parent_1$. Trees classes have to be the same.*

We want the RPB to crossover with another RPB. We want $ADF0$ to only crossover with another $ADF0$ and so on.

Proposition 14. *For any two valid parents, Operator 2 is guaranteed to find valid crossover, and the operation satisfies $T-$*specifications *and $F-$*specifications*. Both parents are valid. Therefore, replacing them wholly produces a valid offspring. Moreover, the offspring is created by replacing a*

subtree with another subtree of the same tree class from a set computed according to $T-$specifications *and* $F-$specifications*. Therefore, any offspring satisfy these constraints.*

4.2.6 Feasible Initialization Procedure

Operator 3. *(initialization) Initialize the population by growing chromosomes starting each with a random terminal node* N *for tree* M *such that* $v_M, N \in \mathcal{T}^{root}$*, and then applying Operator 1 to that node.*

Proposition 15. *The above initialization routine Operator 3 only generates individuals which are valid with respect* $T-$specification *and* $F-$specification *constraints, for each tree that makes up the program. Operator 1 guarantees a valid offspring from a valid parent (Propositions 7 and 8). The initial terminal node is valid as the root of the program as well as the roots of the ADFs.*

4.2.7 Constraint Preprocessing

We do not need terminals T to be explicitly given as in GP; $T_{*,*}$, $T^*_{*,*}$, T^{Root}_* determine individual sets. The preprocessing need to ensure that $F-$*specifications* are *normal* and that our operators can apply, can be described as follows:

1. T^{Root}_*, $T_{*,*}$ ranges from functions of $F^*_{*,*}$ and $T^*_{*,*}$ domains for their arguments
2. Read $T-$*specification* compatibility constraints of the form $T_{*,*} \stackrel{?}{\Rightarrow} T^*_{*,*}$ and $T_{*,*} \stackrel{?}{\Rightarrow} T^{Root}_{*,*}$ (not necessary if computed automatically)
3. Read (at least $T-$*intensive*$F-$*specifications* and $F-$*intensive*$F-$*specifications*)
4. Compute *normal* $F-$*specifications*
5. Produce functions for $r^{\mathcal{T}}_M$ and $r^{\mathcal{F}}_M$ for all necessary sets for all trees.

Given this preprocessing mechanism, the defined operators can be used in any GP.

Proposition 16. $r^{\mathcal{T}}_M$ *and* $r^{\mathcal{F}}_M$ *can be precomputed, as part of the preprocessing mechanism, into functions returning random elements of those sets. For mutation, we directly need* $r^{\mathcal{T}}_{M,N}$ *and* $r^{\mathcal{F}}_{M,N}$*. For any mutation, which of these two is needed can be determined in one step. Based on whether N is the Root or not for a particular tree, Proposition 8 or 7 gives us exactly the sets from which the random element is selected. There is a fixed number of these sets: there are exactly* $1 + \prod_{m_j \in M}(\prod_{f_i \in F}(a_i))$ *of each* $\mathcal{T}$ *and* $\mathcal{F}$ *sets. All* $\mathcal{F}$ *sets are always finite with up to* $|F|$ *elements, and* $\mathcal{T}$ *is either finite, or infinite when domains such as reals are used. Moreover, these sets never change as GP operates. For the* $\mathcal{F}$ *sets, the elements can be enumerated for each tree in the program and* $r^{\mathcal{F}}_{M,N}$ *can be compiled into a function returning a random function from each enumerated set. For the* $\mathcal{T}$ *sets*

which are infinite, $r_M^{\mathcal{T}}$ can be precompiled to returning a random entry from the domain for a particular tree. For finite sets, the elements can be enumerated again and $r_M^{\mathcal{T}}$ can be compiled into a function returning a random element from each enumerated set for a particular tree. For sets which are unions of finite or infinite subsets, one may first determine which class of subsets to choose from (assuming that we provide some measures comparing cardinalities of finite and infinite sets, or the user provides such information), and then apply one of the two above techniques. For crossover, we need to use the $S_{M,N,a}$ sets for the trees that make up an individual. At each item, however, we know whether the node N_a is a terminal or a function node and the class of the tree at which moment the problem reduces to the same as in mutation – selecting random entries from the appropriate $r_M^{\mathcal{T}}$ or $r_M^{\mathcal{F}}$ set. Moreover, if P_c^1 is used, the elements may be divided into two groups from which to select the random entry – p_c^1 would determine which group to use for a particular tree.

Theorem 2. *(Implementation Theorem for GP) The defined mutation and crossover operations not observing size constraints are as efficient as the standard operators in GP, when implemented with the preprocessing mechanism. In GP, mutation generates a random function from F or a random element of T. Crossover selects a random subtree. It follows directly from Proposition 16 that in our approach any mutation or crossover can be accomplished by selecting a random entry from a fixed set, even through the sets are more plentiful. For any node it is deterministic, in a fixed time, which set should be used.*

Conjecture 1. *Provided* $T-$specifications *and* $F-$specifications *are the maximal constraints that that can be implemented into a generic constrain processing methodology in GP without invalidating Theorem 2. Other constraints require information about a node position in a tree – processing complexity would be a function of tree depth.*

The above forms a foundation for the enhanced CGP methodology. Constraints are specified by the user and this has to be done ahead of the GP run. If we want to have the program learn during a GP run we move to enhanced ACGP. Enhanced ACGP uses the same methodology as CGP but with multiple trees per individual. Nothing changes as far as the notation above all we are doing is tracking the distribution of functions and terminals for multiple trees per individual.

4.3 Implementation Enhancements for CGP and ACGP

This portion of the dissertation describes enhancements to the lil-gp based frameworks implemented for CGP and ACGP. lil-gp 1.02 is a public domain GP framework developed by Zongker and Punch (1996). CGP2.1, CGP1.1 and ACGP1.1.2 are based on lil-gp 1.02. CGP was originally outlined in Janikow (1996b) and Janikow (2007a). ACGP was outlined in Janikow (2004b) and Janikow (2004a). Our goal is two fold for this section. First, we outline implementation issues to add ADFs to CGP2.1. This new version is called CGPF2.1. And second, we outline implementation issues to add CGPF2.1 to ACGP1.1.2. This new version is called ACGPF2.1. It is the hope that CGPF2.1

performs better than the previous version CGP2.1. In addition it is hoped that ACGPF2.1 performs better than CGPF2.1 and ACGP1.1.2. CGPF2.1 is used to make ACGPF2.1. Later experiments will not be testing CGPF2.1 as it is embedded in ACGPF2.1.

The next section describes the CGP application environment.

4.3.1 CGPF2.1 Application Environment

The application environment for enhanced CGP, CGPF2.1, remains largely the same except for modifications to permit multiple trees per individual in order to allow ADFs. In a previous section of this dissertation, we extended the basis for the Constraint Specification Language specified in Janikow (1996b) to include multiple trees. Those changes to allow for multiple trees to ripple through CGP2.1, as outlined below to permit the multiple tree version of CGPF2.1. This includes mutation sets from CGP2.1 that are extended to handle multiple trees. Type constraints from CGP2.1 are augmented to allow specification of these constraints on a per tree basis. As with CGP2.1, specification of these constraints for CGPF2.1 take place prior to a GP run.

4.3.2 lilgp and CGPF2.1

The original high level architecture of lil-gp with ADFs can be seen in Figure 4.2. Note that we are using the enhanced CGP notation. In the first set of trees, $f_{1,1}$ is function number 1 in tree 1 which calls an ADF that is function number 1 in tree 2. Both of those trees make an individual that is in population P. The individual, as seen in the second set of trees, tree 1 and / or tree 2 can change through crossover and or mutation. Collapse and uniform mutate were added in CGP2.1 and are available in CGPF2.1.

In CGP2.1, minimal changes were made to lil-gp to implement constraints and typing. The architecture for these changes can be seen in 4.3. This version introduced mutation sets and fsets. Because ADFs were not a part of CGP2.1, the `MS_czj` and `fset` data structures only worked with one tree per individual. Adding ADFs to CGP2.1 to make CGPF2.1 entails adding an extra dimension to the `MS_czj` and `fset` data structures. Adding this extra dimension allows for multiple trees per individual. Figure 4.4 gives a high level view for the architecture of CGPF2.1.

The next set of numbered items list at a high level changes needed by CGPF2.1.

1. In CGP2.1, fset sets are read and stored by lil-gp (and after the functions are ordered in lil-gp 1.02 and later), the function $create\_MS\_czj$ is called. This function accesses the global fset structure to get information about functions and terminals. In CGPF2.1 modifications to add ADFs are needed to propagate throughout functions and data structures.

2. As in CGP2.1, constraints are transformed into the normal form, and expressed as mutation sets MS_czj. The information contained here is global and available to lil-gp. In CGP2.1 the MS_czj structure is only aware of one tree per individual. Information is accessed through

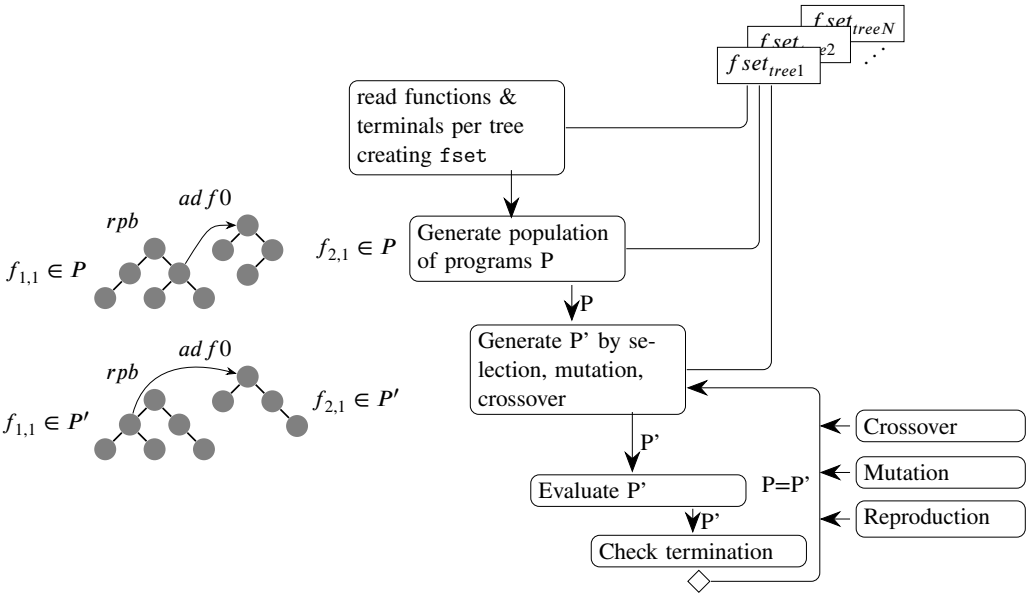


Figure 4.2: lil-gp's original architecture (highly abstracted)

a combination of "C" array index combinations and pointer referencing and dereferencing. In order to add ADFs, for CGPF2.1, the array structure that contains MS_czj information increases in size by one. In CGP2.1 MS_czj contained functions, arguments and types. In CGPF2.1 MS_czj contains trees, functions, arguments and types.

3. In CGP2.1, population initialization, mutation, and crossover of lil-gp were modified to interact with MS_czj. The changes were to accommodate the constraint and typing system.

This next list of items lists two parameters added for CGP2.1. The functionality for these two parameters do not change in CGPF2.1.

1. In CGP2.1, two parameters added in addition to those of lil-gp:[1]

   Initialization parameters `init.depth_abs, =true,false, default=false`
   In the default mode, this parameter has no effect over lil-gp 1.02
   When set, `init.depth_ramp=m-n` causes rejection of initial trees shallower than m

   Mutation parameter `depth_abs, =true,false, default=false`
   In the default mode, this parameter has no effect over lil-gp 1.02
   When set, `depth_ramp=m-n` causes rejection of mutation offspring shallower than m
   (it is a good idea to have `keep_trying` set to true as well)

2. Function's index member is adjusted after functions are sorted by lil-gp (if at all). lil-gp 1.02 does not use or set this index correctly. CGP2.1 uses this information.

[1]This part comes from the cgp/acgp technical manual.Janikow (2007a)

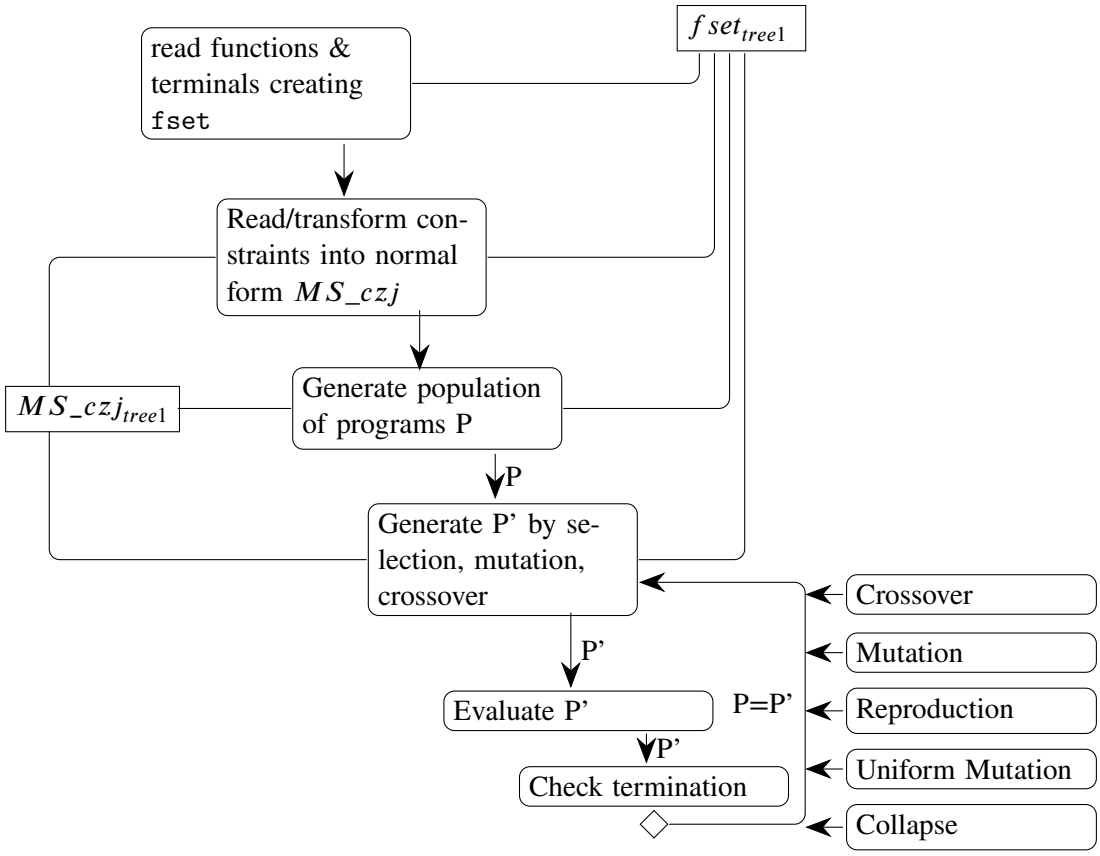


Figure 4.3: CGP2.1 lil-gp's architecture (highly abstracted)

A detailed list of changes to CGP2.1 is available, but is not included in this dissertation. That list of changes will be the basis of a technical manual on CGPF2.1.

4.3.3 Using CGPF2.1 and ACGP1.1.2 to Create ACGPF2.1

Our approach to add multiple trees to ACGP will be to add features described in the previously described enhanced CGP. The procedure to do this will be, at a high level, a little different than that of the procedure to add ADFs. Our approach to make ADFs a part of ACGP will be to follow how ACGP is used in the original version of ACGP1.1.2 and start copying relevant functions. We will strive as always for a clean compile at various times when we are merging code. When we are merging ACGP code into the enhanced version of CGP, we may need to alter some of the original ACGP code to handle multiple trees. Also, some of the output files, introduced in ACGP, will need to change to reflect ADFs being added. The files affected are the `.cnt` and `.wgt` files.

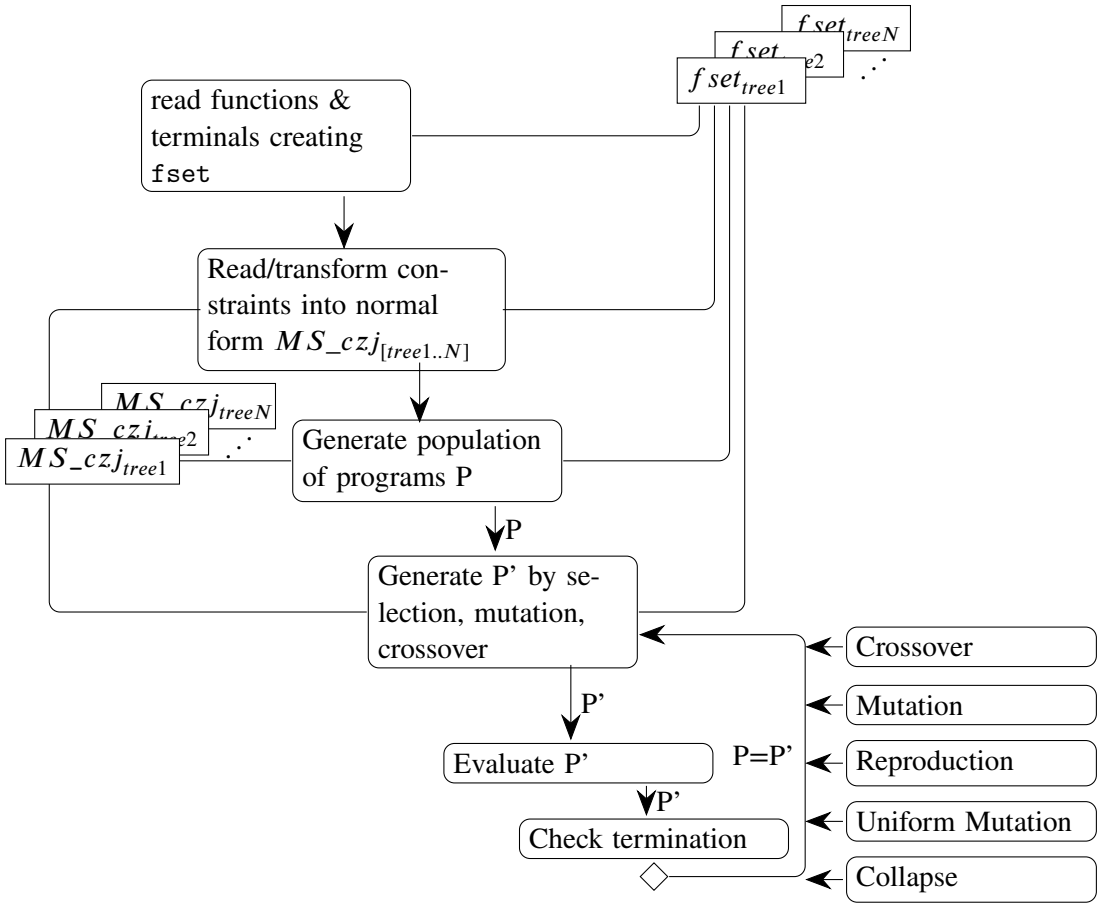


Figure 4.4: CGPF2.1 lil-gps architecture (highly abstracted)

4.3.4 Detailed List and Plan for Modifications to CGPF2.1 needed by ACGPF2.1

We will effectively move the enhanced CGP code base to an enhanced ACGP code base. This enhanced version of ACGP will be called ACGPF2.1. As with the original version of CGP, most of the modifications in the original version of CGP were made in the files `cgp_czj.h` and `cgp_czj.c`. We will start by finding where ACGP first shows up in the original ACGP version and then merge that functionality the enhanced version of CGPF2.1 to create an enhanced ACGP.

A detailed list of changes to CGPF2.1 is available, but is not included in this dissertation. That list of changes will be the basis of a technical manual on ACGPF2.1.

4.4 Rationale for and Implementation of Delayed Tree Growth (Generation Ramp)

In lil-gp there are two functions in the `tree.c` module, `generate_random_full_tree` and `generate_random_grow_tree`. These functions are passed a random integer which represents the

maximum depth of a given tree for an individual. There is the possibility that maximum depth trees could be generated for individuals that take over the population. These maximal depth trees might not be optimal small trees if we believe that a smaller solution exists when using ADFs. Placing a cap on maximal tree size for a number of generations, and gradually relaxing the maximal tree size could both help give the GP run a chance to evolve smaller programs if they exist in the smaller solution space. This could also help with finding smaller scallable GPs programs. This will be an easy change to implement to both enhanced CGP and ACGP. The changes are the same for both CGP and ACGP.

This takes us the end of implementation changes changes for enhancing ACGP. We now move on to the results portion of this dissertation.

4.5 Overall Experimental Setup

Three problems are explored in this research, the lawnmower problem, the bumble bee problem and the two box problem. The details for these problems are in the next sections.

These problems were chosen for a number of important reasons. First, these problems are difficult and extremely hard for a GP to solve as seen in Koza (1994b). Second, these problems are easily tunable. We can increase the difficulty of the problem by increasing some aspect of the problem. For example we can exponentially increase the difficulty of the lawnmower problem by increasing the grid size dramatically. We can increase the difficulty of the bumble bee problem by increasing the number of dimensions a bee has to travel. Additionally, we can increase the difficulty of the bumble be problem increasing the number of flowers a bee has to find. In the case of the two box problem previous methods did not performed very well at all. We use the two box problem as extreme case to showcase the superiority of the combined methods.

The next sections will outline and describe parameters specific each experiment run.

4.6 Lawnmower Problem

In chapter 8 of Koza (1994b) the Lawnmower program is investigated with and without ADFs. The goal of this problem is to evolve the path for a lawnmower to visit each cell in a rectangular grid. Different grid configurations were investigated. Configurations include $8x4$, $8x8$, $8x10$, and $8x12$. Originally these were a benchmark to showcase ADFs. Here we explore scalability of this problem with configurations that include $25x25$ and $50x50$.

The original Lawnmower problem used ADFs but no typing or constraints were used. Here we investigate the usage ADF and constraints for experiments involving the five frameworks. Types are not used for experiments on the lawnmower problem. There is only one type, which is the x y location of the lawnmower.

4.6.1 Description of Functions and Terminals

Here we describe the functions and terminals used in Lawnmower problem . Tables 4.2 and 4.3 list functions and terminals for the Lawnmower problem with out and with ADFs. RPB is the Result Producing Branch . ADF0 is a zero argument ADF. ADF1 is a 1 argument ADF. Next is a description of the functions and terminals them selves.

There are two ways for a mower to move on the grid. One is with the `frog` function. The other is with a `mow` function. Koza gave the problem more than one way to move to a solution.

The `frog` function takes in a 2 dimensional vector. It first checks to see if the abort flag is set. If the abort flag is set, the function returns the passed in vector. If the abort flag is not set we move into the body of the `frog` function. Based on a global variable that holds the current direction the mower is facing, it will jump to that new location. The grid edges are connected. Both the x and y values of the passed in vector are used for calculation of the new mower location. If the jump location is beyond the end of the grid, the target location wraps around to the opposite edge. Modulo arithmetic based on grid width and height is used in the calculation of the new location. Once the mower arrives at the new location that grid cell is marked as mowed. The move counter is incremented. If the move counter is beyond the maximum move count an abort flag is set. The `frog` function at this point returns the passed in vector to be potentially used again by another function.

The `mow` function is a zero argument terminal that has the following functionality. Upon entry into this function a zero vector is declared. A check of the abort flag is made. If the flag is set the `mow` function returns the zero vector. If the flag is not set we move into the body of the `mow` function. A check of the direction global variable is made and we use the width or height of the grid to move the mower along the x or y direction to place the mower at the new location. Like the `frog` function modulo arithmetic based on the width and height of the grid is used to ensure that the new location is wrapped around the edge to the new destination. The new destination is marked as mowed. The move counter is incremented by one. A check is made to see if the move counter is beyond our maximum move counter. If it is, we set the abort flag to signify we need to abort future move operations. At this point we return the zero vector to the calling routine.

The `vma` function takes in two arguments that are vectors and adds them. Modulo arithmetic based on the width and length of the grid ensures that all values stay in the boundary of the grid. A vector holding the sum of the addition is returned from this function.

The `prog2` function is a LISP inspired function. This function takes in the second argument and just returns it to the function that called `prog2`. For example, if the call looked like `(prog2 3 4)`, the value 4 would be returned.

The `left` function changes the direction of the mower to rotate 90_0 counter clockwise. For example, if we are facing north the global direction variable is set to west. The left counter is incremented by 1. We check the left counter to see if it is past the maximal left count limit. If it is past it we set the global abort flag abort all future move operations.

Ephemeral random vectors are used for the Lawnmower experiments. The x value of the vector is a random integer between the range of 0 and the width minus 1. The y value of the vector is a

random integer between 0 and the height minus 1.

Table 4.2: Functions and Terminals for Lawnmower Problem No ADFs

body	name	Fun/Term	Argument Count
rpb	frog	fun	1
	vadd	fun	2
	prog2	fun	2
	left	term	0
	mow	term	0
	Rvm	term	0

Table 4.3: Functions and Terminals for Lawnmower Problem With ADFs

body	name	Fun/Term/Arg	Argument Count
rpb	frog	fun	1
	vadd	fun	2
	prog2	fun	2
	left	term	0
	mow	term	0
	Rvm	term	0
	adf0	fun	0
	adf1	fun	1
adf0	vadd	fun	2
	prog2	fun	2
	left	term	0
	mow	term	0
	Rvm	term	0
adf1	frog	fun	1
	vadd	fun	2
	prog2	fun	2
	left	term	0
	mow	term	0
	Rvm	term	0
	adf0	fun	0
	a0	arg	0

4.6.2 Run Parameters

There are many run time parameters for the five frameworks under investigation here. Some of the parameters are shared among the five frameworks and others are specific to the successful run of a particular framework. A detailed explanation of the run parameters can be found at Janikow (2007b). A full user manual and technical manual for CGPF and ACGPF will be created after the work on this dissertation. But in the mean time we give an overview of the run time parameters used for experiments in this problem.

Table 4.4 lists parameters shared by all various lawnmower experiments. In all experiments the lilgp *half and half* init method was used. The initial depth of the population was set at between 2 and 6 nodes. Crossover rate was 90%. The selection method was a tournament selection of the best individual from 7 randomly selected individuals based on fitness. A reproduction rate was 10%. The selection method was the same as the crossover selection method. The maximum number of generations was set at 52. This number was used so that there would be the same ending generation number when the generation ramp feature is used. The parameters for that will be described shortly. The maximum tree depth was set at 17. 1000 individuals are generated for the population. 50 independent runs are conducted. There is only one fitness case there is only one lawn to be mowed. Other framework dependent parameters are discussed next.

Table 4.4: Global Run Parameters for Lawnmower Problem

Parameter	Value
init method	half and half
init depth	2-7
crossover rate	%90
crossover select	tournament, size 7
reproduction rate	0.10
reproduction select	tournament, size 7
MaxGen	52
MaxDepth	17
Population	1000
NumIndRuns	50
NumFitnessCases	1

Table 4.5 has generation ramp specific features. When the generation ramp feature was used all maximum tree depths are set at 5. The ending maximum tree depth was set at 17. The generation ramp interval was set to 4. This means that at generation 0 the max tree depth is 5. After 4 generations the max tree depth was allowed to increase to 6. After 52 generations the max tree depth would be 17. This is a bloat control scheme to give the smaller depth solutions more time to evolve and explore

possible solutions for that given depth.

Table 4.5: Generation Ramp Global Run Parameters for Lawnmower Problem

Parameter	Value
generation ramp interval	4
generation ramp max tree depth	17
RPB initial max tree depth	5
ADF0 initial max tree depth	5
ADF1 initial max tree depth	5

Table 4.6 contains the maximum number of hits for our single fitness case. So for a $8x4$ lawn size, 32 cells would need to be visited for a 100% success rate. Koza investigated grid sizes of up to $8x12$. We explore grid sizes up to $50x50$ with 2500 cells.

Table 4.6: Maximum Hits Per Fitness Case for Lawnmower Problems

Problem	MaxHits
Lawn Mower8x4	32
Lawn Mower8x8	64
Lawn Mower8x10	80
Lawn Mower8x12	96
Lawn Mower25x25	625
Lawn Mower50x50	2500

Because of the increased scale of problems explored counters are employed to exit a evaluation of an individual if too many moves occur. If counters are not used, the mower could encounter a situation where it gets stuck in a loop of movement where not all of the lawn has been mowed. Table 4.7 shows those counter limits.

Table 4.7: Maximum Move Counters for Lawnmower Problems

Problem	MoveCount	LeftCount
Lawn Mower8x4	100	100
Lawn Mower8x8	100	100
Lawn Mower8x10	100	100
Lawn Mower8x12	100	100
Lawn Mower25x25	10000	10000
Lawn Mower50x50	20000	20000

Now we move to framework specific parameters. Table 4.8 has two parameters. The first is initialization depth absolute and it is set to true. This will force trees to be of a minimal depth. Trees of a smaller depth are rejected. Random attempts of 200 is used to keep trying to create trees that do not meet constraints. In this problem there are two ways for the mower to move. One is through the `mow` function and the other is the `frog` function. The problem can be solved with either of these functions. So in order to restrict the search space of functions for the CGP, CGPF, ACGP and ACGPF frameworks we don't allow the `frog` function to be used.

Table 4.8: Shared CGP CGPF ACGP and ACGPF Parameters for Lawnmower Problems

Parameter	Value
init depth abs	true
init random attempts	200
constraint on	frog

Table 4.9 contains specific ACGP and ACGPF parameters. The use trees percent was set at 2%. This means that the best 2% of the trees are extracted. A select all option of 0 was used. This parameter affects the number of extracted trees. The gen start pct parameter is used when calculating when to start extracting best individuals for ACGP evaluation. It is calculated as max number of generations times the gen start pct number. The gen step parameter is set at 20. This means that we extract the best every 20 generations once the starting extraction generation is reached. Gen slope is a flag that is set to 1. If this flag is set to 1 old heuristics are updated with new heuristics. Also, if this flag is set the rate of change increases as the number of generations increases. In ACGP and ACGPF has a parameter to help determine "what" kind of run we are having. If a "what" parameter of 2 is used we extract and adjust heuristics for this type of run. If a "what" parameter of 3 is set the population is regrown after heuristics have been extracted and adjusted. The population is regrown through the use of a special operator. For our purposes, we are only "what" parameters 2 and 3. The regrow rate is 5%. Selection is achieved by tournament selection with a size of 7. The stop on term

flag is set to 1. This says that once our success condition is met we stop the ACGP run early.

Table 4.9: Shared ACGP and ACGPF Parameters for Lawnmower Problems

Parameter	Value
use trees prct	0.02
select all	0
gen start pct	0.0125
gen step	20
gen slope	1
regrow rate	5%
regrow select	worst
acgp what	2,3
stop on term	1

4.7 2D Bumble Bee Problem

Chapter 9 of Koza (1994b) the Bumble Bee program is investigated next. The problem is one of finding a path from a starting location on a $2d$ grid and visiting all of the flowers. In the Lawnmower problem grid locations were integer coordinates on a grid. Here, there is a rectangular are and the flower locations are real valued $2d$ coordinates. The problem is scaled with flowers added in increments of 5. 10, 15, 20 and 25 flowers were investigated. Comparisons were made with and without ADFs.

As with the original Lawnmower Problem, the Bumble Bee problem used ADFs. Here we investigate the usage ADF for experiments involving the five frameworks. Types and constraints are not investigated for this problem. We would like to have some comparability to the SGP implementation when investigating ACGPF functionality.

4.7.1 Description of Functions and Terminals

In this section we describe the functions and terminals used in $2d$ Bumble Bee problem. Tables 4.10 and 4.11 list functions and terminals for the problem with out and with ADFs. As with the Lawnmower problem the RPB is the Result Producing Branch . ADF0 is a one argument ADF. Next is a description of the functions and terminals them selves.

`vadd` is a function that takes 2 real valued vectors and returns the sum of the two vectors as a new vector.

`vsub` is a function that takes 2 real valued vectors and returns the difference of the two vectors as a new vector.

`gox` is a function that takes in 1 real valued vector. The x component of the vector is added to the x location of the bumble bee. A check is made to see if the global x and y bee location is near anyone of the 10 flowers for this particular fitness cases. There are 10 fitness cases for all versions of this problem. If a bee is near a flower, then the flower's location is marked as a hit. The value returned by the `gox` is a $2d$ zero vector.

`goy` is a function that takes in 1 real valued vector. The y component of the vector is added to the y location of the bumble bee. As with `gox`, a check is made to see if the global x and y bee location is near anyone of the 10 flowers for this particular fitness cases. If a bee is near a flower, then the flower's location is marked as a hit. The value returned by the `goy` is a $2d$ zero vector.

The functionality for the `prog2` is the same as the Lawnmower problem. This function takes in the second argument and just returns it to the function that called `prog2`.

`bee` is a function that returns the Bee's current x and y location.

`nv` is a terminal that randomly selects a flower and returns it's x and y location as a vector.

`Rv` is an ephemeral random constant. It's value is a random real vector. The value of the vector is within the length and with of the area of where the flowers are contained.

For this experiment there are no checks to see if the bee is outside of the predefined area where flowers are placed.

Table 4.10: Functions and Terminals for 2d Bumble Bee Problem No ADFs

body	name	Fun/Term	Argument Count
rpb	vadd	fun	2
	vsub	fun	2
	gox	fun	1
	goy	fun	1
	prog2	fun	2
	bee	term	0
	nf	term	0
	Rv	term	0

Table 4.11: Functions and Terminals for 2D Bumble Bee Problem With ADFs

body	name	Fun/Term/Arg	Argument Count
rpb	vadd	fun	2
	vsub	fun	2
	gox	fun	1
	goy	fun	1
	prog2	fun	2
	bee	term	0
	nf	term	0
	adf0	term	0
	Rv	term	0
adf0	vadd	fun	2
	vsub	fun	2
	gox	fun	1
	goy	fun	1
	prog2	fun	2
	bee	term	0
	a0	arg	0
	Rv	term	0

4.7.2 Run Parameters

Run parameters for this problem are outlined in the tables below. Like the Lawnmower problem, many parameters are common across all instances of the problem. Although, some frameworks have specific parameters.

Table 4.12 contains parameters that all of the frameworks share. The only difference, when compared to the Lawnmower problem, is in the number of individuals in the population, 4000. This matches Koza's original implementation of the problem.

Table 4.12: Global Run Parameters for 2D Bumble Bee Problem

Parameter	Value
init method	half and half
init depth	2-7
crossover rate	%85
crossover select	tournament, size 7
mutation rate	%10
mutation select	tournament, size 7, depth 3-5
reproduction rate	%5
reproduction select	tournament, size 7
MaxDepth	17
Population	4000
NumIndRuns	50

The parameters for the generation ramp feature are almost the same as in the Lawnmower problem. There difference is that there is one less ADF to set up. The parameters for this feature can be seen in 4.13.

Table 4.13: Generation Ramp Global Run Parameters for 2D Bumble Bee Problem

Parameter	Value
generation ramp interval	4
generation ramp max tree depth	17
RPB initial max tree depth	5
ADF0 initial max tree depth	5

The maximum number of hits for this problem type are dependent on the number of flowers. There are 10 fitness cases. The flower locations are generated randomly within an area defined by a length and width parameter which is set ahead of time. Those number of hits can be seen in Table 4.14.

Table 4.14: Maximum Hits for 10 Fitness Cases for the 2D Bumble Bee Problems

Problem	MaxHits
10 flowers	100
15 flowers	150
20 flowers	200
25 flowers	250

Like the Lawnmower problem there are parameters specific to a framework. There is no difference between these paramters and those for the Lawnmower problem. These can be seen in 4.15 and 4.16 respectively.

Table 4.15: Shared CGP CGPF ACGP and ACGPF Parameters for 2D Bumble Bee Problems

Parameter	Value
init depth abs	true
init random attempts	200

Table 4.16: Shared ACGP and ACGPF Parameters for 2D Bumble Bee Problems

Parameter	Value
use trees prct	0.02
select all	0
gen start pct	0.0125
gen step	20
gen slope	1
regrow rate	5%
regrow select	worst
acgp what	2,3
stop on term	1

4.8 3D Bumble Bee Problem

The Bumble Bee problem was originally $2D$. Here, we scale up to $3D$. Here the Bee travels inside a bounding box that has has flowers placed randomly. As we'll see in the results section, this makes

the problem much harder for the bee to find the flowers. One could envision that this is analogous to modern day path finding for areal drones.

Much of the functionality of the $2D$ version of this problem is the same as the $3D$ version of this problem. The exceptions are adding functions and variables to deal with the z dimension. Also, the run parameters are the same as the $2D$ version of this problem.

We'll take a look at a description of functions and terminals next.

4.8.1 Description of Functions and Terminals

Tables 4.17 and 4.18 list functions and terminals for the non ADF and ADF versions of the Bumble Bee problem. As with the $2D$ version of this problem there is a one argument ADF called `adf0`.

`vadd` is a function that takes 2 real valued vectors and returns the sum of the two vectors as a new vector.

`vsub` is a function that takes 2 real valued vectors and returns the difference of the two vectors as a new vector.

`gox` is a function that takes in 1 real valued vector. The x component of the vector is added to the x location of the bumble bee. A check is made to see if the global x and y bee location is near anyone of the 10 flowers for this particular fitness cases. There are 10 fitness cases for all versions of this problem. If a bee is near a flower, then the flower's location is marked as a hit. The value returned by the `gox` is a $2d$ zero vector.

`goy` is a function that takes in 1 real valued vector. The y component of the vector is added to the y location of the bumble bee. As with `gox`, a check is made to see if the global x and y bee location is near anyone of the 10 flowers for this particular fitness cases. If a bee is near a flower, then the flower's location is marked as a hit. The value returned by the `goy` is a $3d$ zero vector.

`goz` is a function that takes in 1 real valued vector. The z component of the vector is added to the z location of the bumble bee. A check is made to see if the global x, y and z bee location is near anyone of the flowers for this particular fitness cases. If a bee is near a flower, then the flower's location is marked as a hit. The value returned by the `goz` is a $3d$ zero vector.

The functionality for the `prog2` is the same as the Lawnmower problem. This function takes in the second argument and just returns it to the function that called `prog2`.

`bee` is a function that returns the Bee's current x and y location.

`nv` is a terminal that randomly selects a flower and returns it's x and y location as a vector.

`Rv` is an ephemeral random constant. It's value is a random real vector. The value of the vector is within the length and with of the area of where the flowers are contained.

For this experiment there are also no checks to see if the bee is outside of the predefined area where flowers are placed.

Table 4.17: Functions and Terminals for 3d Bumble Bee Problem No ADFs

body	name	Fun/Term	Argument Count
rpb	vadd	fun	2
	vsub	fun	2
	gox	fun	1
	goy	fun	1
	goz	fun	1
	prog2	fun	2
	bee	term	0
	nf	term	0
	Rv	term	0

Table 4.18: Functions and Terminals for 3d Bumble Bee Problem With ADFs

body	name	Fun/Term/Arg	Argument Count
rpb	vadd	fun	2
	vsub	fun	2
	gox	fun	1
	goy	fun	1
	goz	fun	1
	prog2	fun	2
	bee	term	0
	nf	term	0
	adf0	term	0
	Rv	term	0
adf0	vadd	fun	2
	vsub	fun	2
	gox	fun	1
	goy	fun	1
	goz	fun	1
	prog2	fun	2
	bee	term	0
	a0	arg	0
	Rv	term	0

Like the $2D$ version of the Bumble Bee problem the run parameters for the $3D$ version are the

same and are not duplicated. So there is no separate section for run parameters.

We now move onto a description of Two Box problem and its run parameters.

4.9 3D Two Box Problem

The Two Box problem was an early problem Koza (1994b) where ADFs were explored. It is a symbolic regression problem where structure is imposed on the solution space. The goal of this problem is to calculate the volume of two intersecting overlapping boxes. Each box is defined by length, width and height. The calculation to find the intersecting box is as follows.

$$length0 \times width0 \times height0 - length1 \times width1 \times height1.$$

This problem is good for exploration in that it is sufficiently complex enough to showcase combined features found in ACGPF. Types, constraints, ADFs and ACGPF heuristic methods are all used, with success. A description of functions and terminals can be found in the next section. After that section there is a description of run time parameters.

4.9.1 Description of Functions and Terminals

The Two Box problem has 4 mathematical operations that are functions taking 2 arguments. The problem also has 6 terminals that hold real numbers that help define the length, width and height of each box.

`fmul` is the multiplication operation. `fdiv` is the protected divide operation. `fsub` is the subtraction operation. `fadd` is the addition operation. Each one of these takes a real number, performs the operation and returns a real number.

`l0` and `l1` are length numbers. `w0` and `w1` are width numbers. `h0` and `h0` are height numbers. These numbers are set prior to the start of an experiment as part of a fitness case.

Functions and terminals used when running this experiment not using ADFs can be found in Table 4.19.

Table 4.19: Functions and Terminals for 3D Two Box Problem No ADFs

body	name	Fun/Term	Argument Count
rpb	fmul	fun	2
	fdiv	fun	2
	fsub	fun	2
	fadd	fun	2
	l0	term	0
	w0	term	0
	h0	term	0
	l1	term	0
	w1	term	0
	h1	term	0

When using ADFs for the Two Box problem the RPB contains an extra function, as seen in Table 4.20. This extra function is a 3 argument ADF that takes three real numbers returns a real number. The name of this ADF is adf0. A second program tree which contains the body of the ADF has three argument terminals $a0$, $a1$ and $a2$. The argument terminals contain the values of passed in ADF function arguments.

Table 4.20: Functions and Terminals for 3d Two Box Problem With ADFs

body	name	Fun/Term/Arg	Argument Count
rpb	fmul	fun	2
	fdiv	fun	2
	fsub	fun	2
	fadd	fun	2
	l0	term	0
	w0	term	0
	h0	term	0
	l1	term	0
	w1	term	0
	h1	term	0
	adf0	fun	2
adf0	fmul	fun	2
	fdiv	fun	2
	fsub	fun	2
	fadd	fun	2
	a0	arg	0
	a1	arg	0
	a2	arg	0

The run time parameters for the Two Box problem are in the next section.

4.9.2 Run Parameters

Table 4.21: Global Run Parameters for 3D Two Box Problem

Parameter	Value
init method	half and half
init depth	2-8
crossover rate	%85
crossover select	tournament, size 7
mutation rate	%10
mutation select	tournament, size 7, depth 3-5
reproduction rate	%5
reproduction select	tournament, size 7
MaxGen	52
MaxDepth	17
Population	4000
NumIndRuns	50
NumFitnessCases	10

Table 4.22: Generation Ramp Global Run Parameters for 3D Two Box Problem

Parameter	Value
generation ramp interval	4
generation ramp max tree depth	17
RPB initial max tree depth	5
ADF0 initial max tree depth	5

The maximum number of hits for the two box problem is 10.

Table 4.23: Shared CGP CGPF ACGP and ACGPF Parameters for 3D Two Box Problem

Parameter	Value
init depth abs	true
init random attempts	300

Table 4.24: Shared ACGP and ACGPF Parameters for 3D Two Box Problem

Parameter	Value
use trees prct	0.02
select all	0
gen start pct	0.0125
gen step	20
gen slope	1
regrow rate	5%
regrow select	worst
acgp what	2,3
stop on term	1

The next section contains a description of data collected and statistical methods performed.

4.10 Measures

For our experiments we will be accumulating a number of measures for comparison. For hypothesis we are looking at 5 measures for best of run individuals across 50 independent runs . These measures are mean hits, mean generation where the best individual first appeared, mean number of nodes, mean tree depth and summation of the evaluation times for all individuals for problems and frameworks. Evaluation time is the cumulative number of seconds or minutes to evaluate individuals across 50 independent runs. It does not include any problem setup times or the times to write statistics to the disk.

A short note on terminology. Hits is the number of times the evolved genetic program has matched the target test case data. Earlier in this chapter there are tables for the maximum number of hits for each problem. The maximum number of hits for the lawnmower problem it is in Table 4.6. For the $2D$ and $3D$ bumble bee problem it is in Table 4.14. For the two box problem the maximum number of hits is 10. Like the Lawnmower problem, the bigger problems hold more interesting results. When investigating the Bumble Bee problem we show results using 25 as the maximum number of flowers.

For statistical significance testing the non-parametric Mann–Whitney U test is used. Inspecting the histograms confirmed that the data did not conform the shape of a normal distribution. A key is placed at the bottom of each table to help show the significance of p-values for measures. "$***$" signifies a p-value is below 0.001. "$**$" signifies a p-value that is greater than 0.001 but less than 0.01. "$*$" signifies a p-value that is greater than 0.01 but less than 0.05. Anything that is below 0.05 is considered significant for this test.

As a reminder test problems are run highlighting some combination of types, constraints, ADFs or ACGPF heuristic extraction adjustment parameters. For example it is pointless to generate statistics for types in the lawnmower problem. There is only one type in the lawnmower problem, the location vector. A description of these what combinations are used is described next.

For the Lawnmower problem we are using ADFs and constraints. A constraint is placed on not having the `frog` function used. For the Bumble Bee problem we are using ADFs. There are no constraints or types used. For the Two Box problem we are using ADFs, types and constraints. Types are used to constrain what functions and terminals are allowed. Specifically, if a `fmul` function is used a function's argument can only contain `l0`, `w0` or `h0` arguments. Mixing arguments of differing types is not allowed in the RPB. For example, `l0` and `w1` can not be in the same function argument. There is only one type for the ADF. We'll call that type `float`, to reflect the real valued numbers being used for this problem. Constraints are placed on not using the `fdiv`, `fadd` and `fmul` functions in the RPB. `fsub` is only allowed at the root of the RPB. Constraints in the ADF tree are placed on not using the `fadd`, `fsub` and `fdiv` functions. And an additional constraint is place on not allowing these functions to be at the root of the ADF.

In the next chapter when we reference ACGPF, we are using these features when comparing frameworks. If the framework can used types or constraints we'll use them. For example we'll use types for CGP and constraints in ACGP for comparisons with ACGPF. The use of a type or constraint depending on the framework is implicit in our use of the term when comparing CGP to ACGPF or ACGP to ACGPF.

And a further note, times reported in the tables in the next chapter are total evaluation time of all best of run individuals for that configuration. Not wall clock time.

4.11 Hypothesis

The dissertation aims to combine the best of previous methodologies across frameworks, leading up to ACGPF2.1. The hypothesis for our experiments is in Table 4.25.

Hyp1:	$ACGPF$ shows improved performance compared to SGP with no ADFs
Hyp2:	$ACGPF$ shows improved performance compared to SGP using ADFs
Hyp3:	$ACGPF$ shows improved performance compared to CGP which has no ADFs but has Types
Hyp4:	$ACGPF$ shows improved performance compared to $ACGP$ which has no ADFs or Types

Table 4.25: Dissertation Hypothesis

The next chapter contains the results of our experiments.

Chapter 5

Results

5.1 Introduction

In this chapter we outline experiment results for the best of run individuals for three problems. These three problems are the Lawnmower problem, the Bumble Bee problem and the Two Box problem. Outcomes are reported through a series of tables based on hypothesis posed at the end of the last chapter on methodology. There are additional figures where needed to illustrate the evolving fitness of best of run individuals.

A note on terminology for this chapter. In the context of ACGPF when we see what2, this means we are using the feature that extracts and uses heuristics produced by each generation. When we see what3, this means we are using the ACGPF feature that extracts and uses heuristics and then regrows the population.

We now move onto the results and additional analysis for each specific hypothesis.

5.2 Hypothesis 1: ACGPF vs SGP No ADFs Results

For this hypothesis, we are comparing ACGPF using ADFs, types and constraints to SGP not using ADFs, types or constraints. Results for this hypothesis did not include using the generation ramp feature. In the next section we'll add the generation ramp feature to ACGPF and look at those comparison results for SGP not using ADFs. In section 5.6, we'll compare ACGPF not using the generation ramp feature to ACGPF using the generation ramp feature.

Table 5.1 has the results for hypothesis 1. All of the p-values, where marked with "$*$" are significant. ACGPF outperformed SGP on most measures. We'll discuss the outcomes for each problem.

For all sizes and configurations of the Lawnmower problem and the Two Box problem, ACGPF

outperformed SGP. Using ACGPF less nodes were used to find a solutions; tree depth for solutions was less and finding the first occurrence of the best of run individual happened earlier in a run. Also, the total run time was less than SGP when using ACGPF.

For the $2D$ and $3D$ Bumble Bee problem the results are mixed. More hits were found with ACGPF than with SGP without ADFs, however, the mean number of nodes and the mean tree depth were both larger. When using ACGPF it took longer to find the first best of run individual. The total evaluation time was larger when using ACGPF. The maximum number of hits based on 10 fitness cases and 25 is 250 for both the $2D$ and $3D$ versions of the Bumble Bee problem. Almost all of the hits were found for the $2D$ version of the problem. A fraction of the maximum hits was found for the $3D$ ACGPF version of the problem, which still outperformed the $3D$ version of SGP not using ADFs.

Next, we investigate the generation ramp feature for ACGPF.

5.2.1 Additional Analysis and Results ACGPF With Generation Ramp vs SGP No ADFS

In this section we look at adding the generation ramp feature to ACGPF. First, we'll look at ACGPF compared with SGP which is not using ADFs. The results can be seen in table 5.2.

Under both sizes of the Lawnmower problem and the Two Box problem, ACGPF with the generation ramp feature outperformed SGP not using ADFs. The maximum number of hits was found for both problems. This was more than experimental setup not using ADFs with SGP. In addition, best of run individuals had less nodes. Their tree depth was less. And, the generation of the first occurrence happened earlier.

Like results for hypothesis 1 in the previous section, the results were mixed for the $2D$ and $3D$ Bumble Bee problem. The number of hits was larger than SGP for ACGPF using the generation ramp feature. However, ACGPF using the generation ramp feature had larger number of nodes and tree depth. The average generation where the best of run individual was found occurred later. In addition, the total run time was larger than the SGP framework for these configurations.

Later in this chapter we perform additional analysis on the Bumble Bee problem. The additional analysis and results include the following. First, we look at the role of mutation for the Bumble Bee problem. In the Lawnmower problem mutation was not used and favorable results were obtained. We also look at increasing the $3D$ Bumble Bee problem maximum generation to 104. Then, we look at increasing the maximum generation from 52 to 104, 156 and 208 for the $2D$ Bumble Bee problem.

We now move onto a comparison of results using ADFs with SGP and ACGPF.

5.3 Hypothesis 2: ACGPF vs SGP With ADFs Results

For this hypothesis, we are comparing the ACGPF using ADFs, types and constraints to SGP using ADFs. We'll report results in the next section on the generation ramp feature for this experimental

Table 5.1: Hypothesis 1: Compared to SGP (with no ADFs), ACGPF Shows Improved Performance

Outcome Variable	Lawn Mower Problem						Bumble Bee Problem						Two Box Problem		
	25x25			50x50			2D			3D			3D		
	SGP	ACGPF (what=2)	ACGPF (what=3)	SGP	ACGPF (what=2)	ACGPF (what=3)	SGP	ACGPF (what=2)	ACGPF (what=3)	SGP	ACGPF (what=2)	ACGPF (what=3)	SGP	ACGPF (what=2)	ACGPF (what=3)
Mean # Hits	464.56	625.00 ***	625.00 ***	713.04	2500.00 ***	2500.00 ***	48.00	246.50 ***	194.00 ***	8.50	95.50 ***	32.50 ***	1.06	10.00 ***	9.44 ***
Mean Generation Where Best Individual Appeared	51.78	0.14 ***	0.40 ***	51.90	4.38 ***	6.90 ***	27.90	28.96	41.74	6.50	23.92	21.04	48.58	2.10 ***	4.76 ***
Mean # Nodes	2911.00	500.52 ***	501.42 ***	3248.78	581.78 ***	587.48 ***	84.06	250.08	174.08	17.04	101.58	40.84	329.36	14.42 ***	19.00 ***
Mean Tree Depth (entire run)	17.00	7.02 ***	7.12 ***	17.00	10.20 ***	11.54 ***	11.66	16.28	15.12	4.76	11.42	7.26	15.42	2.08 ***	3.12 ***
Total Execution Time (all runs)	17M 58S	6S	3S	18M 48S	5M 35S	4M 45S	7M 6S	21M 0S	17M 8S	4M 15S	15M 39S	9M 54S	14M 44S	11M 4S	12M 58S

$* \; p < 0.05$, $** \; p < 0.01$, $*** \; p < 0.001$

Table 5.2: Hypothesis 1: Compared to SGP (with no ADFs), ACGPF Shows Improved Performance with Generation Ramp Feature

Outcome Variable	Lawn Mower Problem						Bumble Bee Problem						Two Box Problem		
	25x25			50x50			2D			3D			3D		
	SGP	ACGPF (what=2)	ACGPF (what=3)	SGP	ACGPF (what=2)	ACGPF (what=3)	SGP	ACGPF (what=2)	ACGPF (what=3)	SGP	ACGPF (what=2)	ACGPF (what=3)	SGP	ACGPF (what=2)	ACGPF (what=3)
Mean # Hits	464.56	625.00 ***	625.00 ***	713.04	2500.00 ***	2500.00 ***	48.00	233.00 ***	179.00 ***	8.50	58.00 ***	27.00 ***	1.06	10.00 ***	10.00 ***
Mean Generation Where Best Individual Appeared	51.78	11.20 ***	11.60 ***	51.90	19.62 ***	21.02 ***	27.90	38.62	37.76	6.50	25.56	18.08	48.58	1.82 ***	2.94 ***
Mean # Nodes	2911.00	180.20 ***	180.22 ***	3248.78	303.94 ***	291.66 ***	84.06	151.00	113.38	17.04	51.12	29.48	329.36	14.16 ***	16.00 ***
Mean Tree Depth (entire run)	17.00	6.76 ***	6.80 ***	17.00	8.62 ***	9.10 ***	11.66	12.54	10.60 **	4.76	7.82	5.72	15.42	2.04 ***	2.42 ***
Total Execution Time (all runs)	17M 58S	1M 0S	51S	18M 48S	4M 40S	4M 9S	7M 6S	14M 11S	11M 3S	4M 15S	10M 9S	8M 31S	14M 44S	56S	1M 4S

$* \; p < 0.05$, $** \; p < 0.01$, $*** \; p < 0.001$

setup.

Table 5.3 has the results for hypothesis 2. All of the p-values, where marked with "$*$ are significant. ACGPF outperformed SGP on most measures which is similar to the results in the last hypothesis. We next discuss the outcomes each problem.

For all sizes of the Lawnmower problem, ACGPF outperformed SGP on most measures. The maximum number of hits was found for both SGP and ACGPF. ADFs help exploit structural regularity in the search space. Also, when using ACGPF with ADFs and constraints the mean number of nodes was less and the mean tree depth was less. The best of run individuals were found faster using ACGPF. Results were mixed on evaluation time. Total evaluation time was slightly longer for the $25x25$ version ACGPF versus SGP when the ACGPF what2 feature.

For all sizes of the Two Box problem, ACGPF outperformed SGP. Using ACGPF solutions found more hits. Less nodes were used to find a solutions. Tree depth for solutions was less. Finding the first occurrence of the best of run individual happened earlier in a run. And, the total run time was less when using ACGPF.

The results are are also mixed for the Bumble Bee problems. For the $2D$ Two Box problem ACGPF had mixed results when using the ACGPF what2 feature. The number of hits was slightly better than SGP using ADFs. However, the mean number of nodes increased. The tree depth was substantially larger. Also, it took longer for all of the runs to finish. Results were equally mixed for the $2D$ Bumble Bee problem when the ACGPF what3 feature. The number of hits was were worse than SGP using ADFs. The mean generation of where the best individual was found occurred later. The mean tree depth for ACGPF was larger than SGP. But, the mean number of nodes was less for ACGPF than SGP. Also, the total execution time was less for ACGPF than for SGP.

The results were also mixed for the $3D$ Bumble Bee problem. When using the ACGPF what2 feature, the mean number of nodes, the mean tree depth and total execution time were all larger by a factor of two. On average, the best individual was also discovered later during a run. When using the ACGPF what3 feature, the mean number of hits was worse and the mean tree depth was worse. Also, the total execution run time was worse than SGP using ADFs. But, best individuals on average found earlier than in SGP. Also, the mean number of nodes for the best individuals was less.

Results for the generation ramp feature used with ACGPF are next reported and compared to SGP using ADFs.

5.3.1 Additional Analysis and Results ACGPF With Generation Ramp vs SGP With ADFS

Here we compare SGP using ADFs to ACGPF using the generation ramp feature. Those results can be seen in table 5.4.

For the Lawnmower problem and Two Box problem maximum number of hits was achieved when using the generation ramp feature. Best of run individuals on average had a smaller number of nodes and the tree depth was smaller. It took longer to find the first generation of the best of run

Table 5.3: Hypothesis 2: Compared to SGP (with ADFs), ACGPF Shows Improved Performance

Outcome Variable	Lawn Mower Problem						Bumble Bee Problem						Two Box Problem		
	25x25			50x50			2D			3D			3D		
	SGP	ACGPF (what=2)	ACGPF (what=3)	SGP	ACGPF (what=2)	ACGPF (what=3)	SGP	ACGPF (what=2)	ACGPF (what=3)	SGP	ACGPF (what=2)	ACGPF (what=3)	SGP	ACGPF (what=2)	ACGPF (what=3)
Mean # Hits	625.00	625.00	625.00	2500.00	2500.00	2500.00	241.00	246.50	194.00	56.00	95.50 **	32.50	0.88	10.00 ***	9.44 ***
Mean Generation Where Best Individual Appeared	2.28	0.14 ***	0.40 ***	7.48	4.38 ***	6.90 **	31.92	28.96 **	41.74	25.30	23.92	21.04	47.76	2.10 ***	4.76 ***
Mean # Nodes	423.50	500.52	501.42	623.04	581.78 **	587.48 **	261.32	250.08	174.08 ***	61.70	101.58	40.84 **	293.06	14.42 ***	19.00 ***
Mean Tree Depth (entire run)	17.00	7.02 ***	7.12 ***	17.00	10.20 ***	11.54 ***	11.66	16.28	15.12	4.76	11.42	7.26	15.42	2.08 ***	3.12 ***
Total Execution Time (all runs)	24S	6S	3S	4M 58S	5M 35S	4M 45S	19M 36S	21M 0S	17M 8S	7M 11S	15M 39S	9M 54S	24M 6S	11M 4S	12M 58S

$* \; p < 0.05$, $** \; p < 0.01$, $*** \; p < 0.001$

individual. It also took longer for a run to complete. This is to be expected since we are giving best individuals time to emerge by governing the maximum tree depth in a stepped fashion for a number of generations.

The results for the Bumble Bee problem were mixed. The ACGPF what2 feature gave a worse number of hits for the $2D$ setup. The average number of hits was only better than SGP using ADFs for the ACGPF $3D$ what3 setup. For the most part, solutions found with the generation ramp ACGPF combination were smaller and had less nodes. The only exception was the tree depth for the ACGPF $2D$ setup, which was slightly larger than that for SGP using ADFs. For all instances of this experimental setup ACGPF completed a run faster, on average than SGP using ADFs.

We now move to compare Constraint Based Genetic Programming with ACGPF.

5.4 Hypothesis 3: ACGPF vs CGP With No ADFs Results

For this hypothesis, we are comparing the ACGPF using ADFs, types and constraints to CGP not using ADFs and using types or constraints depending on the problem. The generation ramp ACGPF results are in the next section.

Table 5.5 has the results for hypothesis 3. All of the p-values, where marked with "$*$ are significant. ACGPF outperformed SGP on most measures. We'll discuss the outcomes for each problem.

Similarly to hypothesis 2 and depending on the problem, ACGPF performed very well when using ADFs, types and constraints when compared to CGP.

On all measures for the Lawnmower problem, ACGPF outperformed CGP. ACGPF had more hits. The best of run individual was found earlier when using ACGPF. The mean number of nodes were less. The mean tree depth was less. The time for completing a run were all less then CGP.

For the Two Box problem, ACGPF outperformed CGP for all but the evaluation time. When using ACGPF more hits were found. The best of generation individual was found earlier with ACGPF than with CGP. The mean number of nodes and mean tree depth were all lower when using ACGPF, however, the evaluation time was substantially larger when using ACGPF than when using CGP .

As with hypothesis 2, results were mixed when considering using ACGPF for the Bumble Bee problem. For both configurations of ACGPF outperformed CGP, finding more hits. However, the best of run individuals tended to have more nodes and a deeper tree depth. Also, it took longer for a run to complete than using CGP.

Next, we add the generation ramp feature to ACGPF and compare performance to CGP.

5.4.1 Additional Analysis and Results ACGPF With Generation Ramp vs CGP

In this section we compare CGP which does not have ADFs to ACGPF using the generation ramp feature. These results can be seen in table 5.6.

For the Lawnmower problem and Two Box problem maximum number of hits was achieved

Table 5.4: Hypothesis 2: Compared to SGP (with ADFs), ACGPF Shows Improved Performance with Generation Ramp Feature

Outcome Variable	Lawn Mower Problem						Bumble Bee Problem						Two Box Problem		
	25x25			50x50			2D			3D			3D		
	SGP	ACGPF (what=2)	ACGPF (what=3)	SGP	ACGPF (what=2)	ACGPF (what=3)	SGP	ACGPF (what=2)	ACGPF (what=3)	SGP	ACGPF (what=2)	ACGPF (what=3)	SGP	ACGPF (what=2)	ACGPF (what=3)
Mean # Hits	625.00	625.00	625.00	2500.00	2500.00	2500.00	241.00	233.00	179.00	56.00	58.00	27.00	0.88	10.00 ***	10.00 ***
Mean Generation Where Best Individual Appeared	2.28	11.20	11.60	7.48	19.62	21.02	31.92	38.62	37.76	25.30	25.56	18.08 **	47.76	1.82 ***	2.94 ***
Mean # Nodes	423.50	180.20 ***	180.22 ***	623.04	303.94 ***	291.66 ***	261.32	151.00 ***	113.38 ***	61.70	51.12	29.48 ***	293.06	14.16 ***	16.00 ***
Mean Tree Depth (entire run)	17.00	6.76 ***	6.80 ***	17.00	8.62 ***	9.10 ***	11.66	12.54	10.60 **	4.76	7.82	5.72	15.42	2.04 ***	2.42 ***
Total Execution Time (all runs)	24S	1M 0S	51S	4M 58S	4M 40S	4M 9S	19M 36S	14M 11S	11M 3S	7M 11S	10M 9S	8M 31S	24M 6S	56S	1M 4S

$*\ p < 0.05$, $**\ p < 0.01$, $***\ p < 0.001$

Table 5.5: Hypothesis 3: Compared to CGP (with Types, with Constraints, with no ADFs), ACGPF Shows Improved Performance

Outcome Variable	Lawn Mower Problem						Bumble Bee Problem						Two Box Problem		
	25x25			50x50			2D			3D			3D		
	CGP	ACGPF (what=2)	ACGPF (what=3)	CGP	ACGPF (what=2)	ACGPF (what=3)	CGP	ACGPF (what=2)	ACGPF (what=3)	CGP	ACGPF (what=2)	ACGPF (what=3)	CGP	ACGPF (what=2)	ACGPF (what=3)
Mean # Hits	494.46	625.00 ***	625.00 ***	804.48	2500.00 ***	2500.00 ***	60.00	246.50 ***	194.00 ***	16.00	95.50 ***	32.50 ***	0.28	10.00 ***	9.44 ***
Mean Generation Where Best Individual Appeared	51.28	0.14 ***	0.40 ***	51.86	4.38 ***	6.90 ***	33.16	28.96 ***	41.74	12.64	23.92	21.04	5.68	2.10 ***	4.76 ***
Mean # Nodes	3497.04	500.52 ***	501.42 ***	3966.16	581.78 ***	587.48 ***	140.90	250.08	174.08	36.16	101.58	40.84	32.88	14.42 ***	19.00 ***
Mean Tree Depth (entire run)	17.00	7.02 ***	7.12 ***	17.00	10.20 ***	11.54 ***	14.36	16.28	15.12	7.24	11.42	7.26	15.94	2.08 ***	3.12 ***
Total Execution Time (all runs)	27M 34S	6S	3S	28M 59S	5M 35S	4M 45S	11M 30S	21M 0S	17M 8S	6M 1S	15M 39S	9M 54S	2M 8S	11M 4S	12M 58S

$* \; p < 0.05$, $** \; p < 0.01$, $*** \; p < 0.001$

when using the generation ramp feature. Best of run individuals on average had a smaller number of nodes and tree depth. It took longer to find the first generation of the best of run individual. It also took longer for a run to complete.

Results for the Bumble Bee problem were very similar to previous hypothesis with the average number of hits being greater than those of CGP. For the $2D$ ACGPF setup average tree depth was less than those of CGP. However, for the $3D$ ACGPF setup tree depth was greater than CGP for the ACGPF with the what3 configuration.

5.5 Hypothesis 4: ACGPF vs ACGP Results

For this hypothesis, we are comparing ACGPF using ADFs, types and constraints to ACGP that has constraints and no types. ADFs and the type mechanism was not designed into ACGP. Constraints are used for both comparisons. The generation ramp feature is not used here.

Tables 5.7 and 5.7 have the results for hypothesis 4. All of the p-values, where marked with "$*$" are significant. ACGPF outperformed ACGP on most measures. As with previous hypothesis, we'll discuss the outcomes each problem. Since we are comparing ACGP based frameworks we are comparing like heuristic settings. So when comparing ACGP to ACGPF frameworks, we compare what2 to what2 and what3 to what3 configurations.

For all measures, the Lawnmower problem ACGPF outperformed ACGP. Like previous hypotheses, ACGPF had more hits. The best of run individual were found earlier when using ACGPF. The mean number of nodes and tree depth were less when using ACGPF. Also, the time for completing a run were all less then ACGP.

For the Two Box problem ACGPF also outperformed ACGP for all measures but the evaluation time. When using ACGPF found more hits. The best of generation individual also was found earlier with ACGPF than with CGP. The mean number of nodes and mean tree depth were all lower when using ACGPF. However, the evaluation time was larger than ACGP when using ACGPF.

As with previous hypothesis, results were mixed when considering using ACGPF for the Bumble Bee problem. For both configurations of ACGPF also outperformed CGP by finding more hits. However, like previous hypothesis, best of run individuals tended to have more nodes and a deeper tree depth. Also, it took longer for a run to complete than using ACGP.

Now, we move to adding the generation ramp feature to ACGPF and compare it to ACGP.

5.5.1 Additional Analysis and Results ACGPF With Generation Ramp vs ACGP

In this section we compare CGP which does not have ADFs to ACGPF using the generation ramp feature. These results can be seen in tables 5.9 and 5.10.

Similarly to previous hypothesis, for the Lawnmower problem and Two Box problem, maximum number of hits was achieved when using the generation ramp feature. Best of run individuals on

Table 5.6: Hypothesis 3: Compared to CGP (with Types, with Constraints, with no ADFs), ACGPF Shows Improved Performance with Generation Ramp Feature

Outcome Variable	Lawn Mower Problem						Bumble Bee Problem						Two Box Problem		
	25x25			50x50			2D			3D			3D		
	CGP	ACGPF (what=2)	ACGPF (what=3)	CGP	ACGPF (what=2)	ACGPF (what=3)	CGP	ACGPF (what=2)	ACGPF (what=3)	CGP	ACGPF (what=2)	ACGPF (what=3)	CGP	ACGPF (what=2)	ACGPF (what=3)
Mean # Hits	494.46	625.00 ***	625.00 ***	804.48	2500.00 ***	2500.00 ***	60.00	233.00 ***	179.00 ***	16.00	58.00 ***	27.00 **	0.28	10.00 ***	10.00 ***
Mean Generation Where Best Individual Appeared	51.28	11.20 ***	11.60 ***	51.86	19.62 ***	21.02 ***	33.16	38.62	37.76	12.64	25.56	18.08	5.68	1.82 ***	2.94 ***
Mean # Nodes	3497.04	180.20 ***	180.22 ***	3966.16	303.94 ***	291.66 ***	140.90	151.00	113.38 **	36.16	51.12	29.48	32.88	14.16 ***	16.00 ***
Mean Tree Depth (entire run)	17.00	6.76 ***	6.80 ***	17.00	8.62 ***	9.10 ***	14.36	12.54 ***	10.60 ***	7.24	7.82	5.72 **	15.94	2.04 ***	2.42 ***
Total Execution Time (all runs)	27M 34S	1M 0S	51S	28M 59S	4M 40S	4M 9S	11M 30S	14M 11S	11M 3S	6M 1S	10M 9S	8M 31S	2M 8S	56S	1M 4S

$* \; p < 0.05$, $** \; p < 0.01$, $*** \; p < 0.001$

Table 5.7: Hypothesis 4: Part A: Compared to ACGP (with Constraints, with no ADFs), ACGPF Shows Improved Performance

Outcome Variable	Lawn Mower Problem								Bumble Bee Problem							
	25x25				50x50				2D				3D			
	what=2		what=3		what=2		what=3		what=2		what=3		what=2		what=3	
	ACGP	ACGPF	ACGP	ACGPF	ACGP	ACGPF	ACGP	ACGPF	ACGP	ACGPF	ACGP	ACGPF	ACGP	ACGPF	ACGP	ACGPF
Mean # Hits	484.18	625.00 ***	269.58	625.00 ***	746.98	2500.00 ***	321.20	2500.00 ***	51.00	246.50 ***	25.00	194.00 ***	10.50	95.50 ***	7.50	32.50 ***
Mean Generation Where Best Individual Appeared	51.52	0.14 ***	38.02	0.40 ***	51.92	4.38 ***	37.98	6.90 ***	26.02	28.96	2.74	41.74	10.76	23.92	5.70	21.04
Mean # Nodes	4092.28	500.52 ***	1043.44	501.42 ***	4912.08	581.78 ***	1084.56	587.48 ***	90.98	250.08	24.74	174.08	20.00	101.58	20.66	40.84
Mean Tree Depth (entire run)	17.00	7.02 ***	17.00	7.12 ***	17.00	10.20 ***	17.00	11.54 ***	11.84	16.28	5.14	15.12	5.94	11.42	5.24	7.26
Total Execution Time (all runs)	36M 24S	6S	4M 45S	3S	39M 13S	5M 35S	5M 13S	4M 45S	11M 47S	21M 0S	5M 18S	17M 8S	7M 33S	15M 39S	5M 17S	9M 54S

$* \; p < 0.05, \;\; ** \; p < 0.01, \;\; *** \; p < 0.001$

Table 5.8: Hypothesis 4: Part B: Compared to ACGP (with Constraints, with no ADFs), ACGPF Shows Improved Performance

Outcome Variable	Two Box Problem			
	3D			
	what=2		what=3	
	ACGP	ACGPF	ACGP	ACGPF
Mean # Hits	5.02	10.00 ***	3.98	9.44 ***
Mean Generation Where Best Individual Appeared	31.18	2.10 ***	29.60	4.76 ***
Mean # Nodes	97.56	14.42 ***	83.48	19.00 ***
Mean Tree Depth (entire run)	10.34	2.08 ***	10.00	3.12 ***
Total Execution Time (all runs)	6M 55S	11M 4S	4M 44S	12M 58S

$* \; p < 0.05, \;\; ** \; p < 0.01, \;\; *** \; p < 0.001$

average had a smaller number of nodes and tree depth. It took longer to find the first generation of the best of run individual. It also took longer for a run to complete.

Results for this configuration of the Bumble Bee problem were very similar to previous hypothesis. Average number of hits were greater than those of ACGP. For the $2D$ ACGPF setup average tree depth was less than those of ACGP. However, for the $3D$ ACGPF setup tree depth was greater than ACGP for the ACGPF with the what3 configuration.

Next, we focus on the ACGPF framework and the generation ramp feature.

5.6 ACGPF No Generation Ramp vs ACGPF With Generation Ramp Results

Here, we compare performance of ACGPF framework with and without the generation ramp feature. We will use tables 5.1 and 5.2 for comparisons. Figures 5.1,5.2,5.3,5.4 and 5.5 show best of run individual evolving fitness for the various configurations of problems.

A short note non nomenclature is needed for the figures. In the figures, frameworks refers to the actual frameworks where experiments were conducted. `""acgp1p1p12 nadf""` means the ACGP framework, which does not have ADFs. `""acgpf2p1 yadf""` means the ACGPF framework not using the generation ramp feature. `""acgpf2p1 yadf gr""` means the ACGPF framework using the generation ramp feature. `""cgp2p1 nadf""` means the CGP framework, which does not ADFs. `""orig nadf""` means the original SGP framework not using ADFs. `""orig yadf""` means the original SGP framework using ADFs.

It can be seen in these figures and tables that the generation ramp feature delays the best of run individuals' improvement in fitness. When using or not using the generation ramp feature, the number of hits was the maximum for the Lawnmower problem and $3D$ Two Box problem.

It is interesting to note that for the Lawnmower problem and Two Box problem that maximum hits were achieved for both. The version of those problems that use the generation ramp feature produced smaller solutions, but at the cost of increased time of completion for a run. The Bumble Bee problem performed worse with the addition of the generation ramp feature.

Table 5.9: Hypothesis 4: Part A: Compared to ACGP (with Constraints, with no ADFs), ACGPF Shows Improved Performance with Generation Ramp Feature

Outcome Variable	Lawn Mower Problem								Bumble Bee Problem							
	25x25				50x50				2D				3D			
	what=2		what=3		what=2		what=3		what=2		what=3		what=2		what=3	
	ACGP	ACGPF	ACGP	ACGPF	ACGP	ACGPF	ACGP	ACGPF	ACGP	ACGPF	ACGP	ACGPF	ACGP	ACGPF	ACGP	ACGPF
Mean # Hits	484.18	625.00 ***	269.58	625.00 ***	746.98	2500.00 ***	321.20	2500.00 ***	51.00	233.00 ***	25.00	179.00 ***	10.50	58.00 ***	7.50	27.00 ***
Mean Generation Where Best Individual Appeared	51.52	11.20 ***	38.02	0.62 ***	51.92	19.62 ***	37.98	21.02 ***	26.02	38.62	2.74	37.76	10.76	25.56	5.70	18.08
Mean # Nodes	4092.28	180.20 ***	1043.44	61.18 ***	4912.08	303.94 ***	1084.56	291.66 ***	90.98	151.00	24.74	113.38	20.00	51.12	20.66	29.48
Mean Tree Depth (entire run)	17.00	6.76 ***	17.00	3.98 ***	17.00	8.62 ***	17.00	9.10 ***	11.84	12.54	5.14	10.60	5.94	7.82	5.24	5.72
Total Execution Time (all runs)	36M 24S	1M 0S	4M 45S	51S	39M 13S	4M 40S	5M 13S	4M 9S	11M 47S	14M 11S	5M 18S	11M 3S	7M 33S	10M 9S	5M 17S	8M 31S

$* \; p < 0.05, \;\; ** \; p < 0.01, \;\; *** \; p < 0.001$

Table 5.10: Hypothesis 4: Part B: Compared to ACGP (with Constraints, with no ADFs), ACGPF Shows Improved Performance with Generation Ramp Feature

Outcome Variable	Two Box Problem			
	3D			
	what=2		what=3	
	ACGP	ACGPF	ACGP	ACGPF
Mean # Hits	5.02	10.00 ***	3.98	10.00 ***
Mean Generation Where Best Individual Appeared	31.18	1.82 ***	29.60	2.94 ***
Mean # Nodes	97.56	14.16 ***	83.48	16.00 ***
Mean Tree Depth (entire run)	10.34	2.04 ***	10.00	2.42 ***
Total Execution Time (all runs)	6M 55S	56S	4M 44S	1M 4S

$* \; p < 0.05$, $** \; p < 0.01$, $*** \; p < 0.001$

Figure 5.1: Problem: Lawn Mower 25x25
Best of Run Individuals
Generations 52
No Mutation

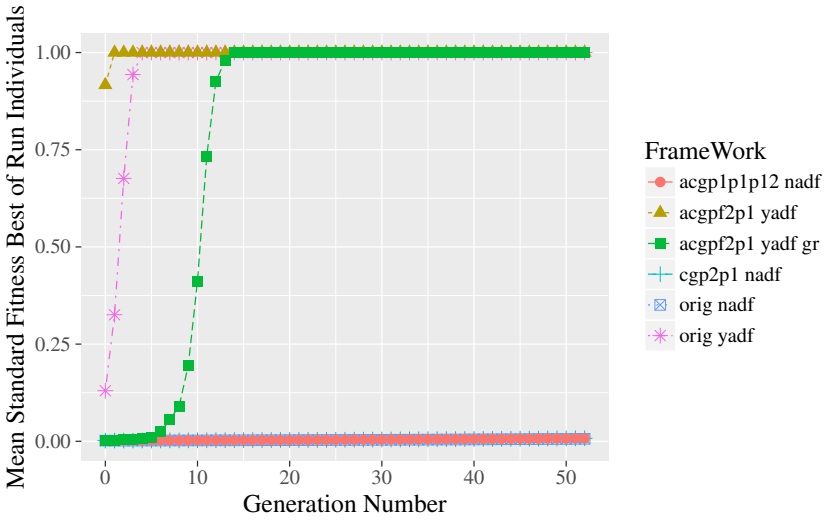


Figure 5.2: Problem: Lawn Mower 50x50
Best of Run Individuals
Generations 52
No Mutation

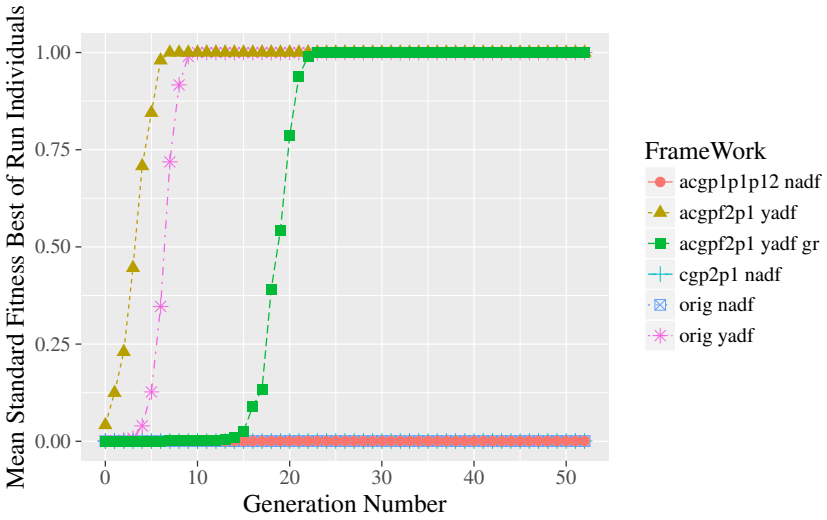


Figure 5.3: Problem: Bumble Bee 2d Flowers 25
Best of Run Individuals
Max Depth 17 Max Generations 52

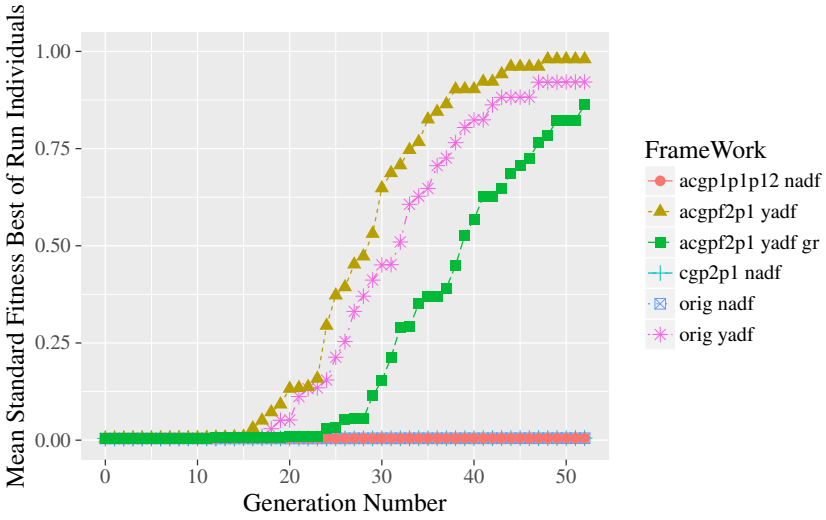


Figure 5.4: Problem: Bumble Bee 3d Flowers 25
Best of Run Individuals
Max Depth 17 Max Generations 52

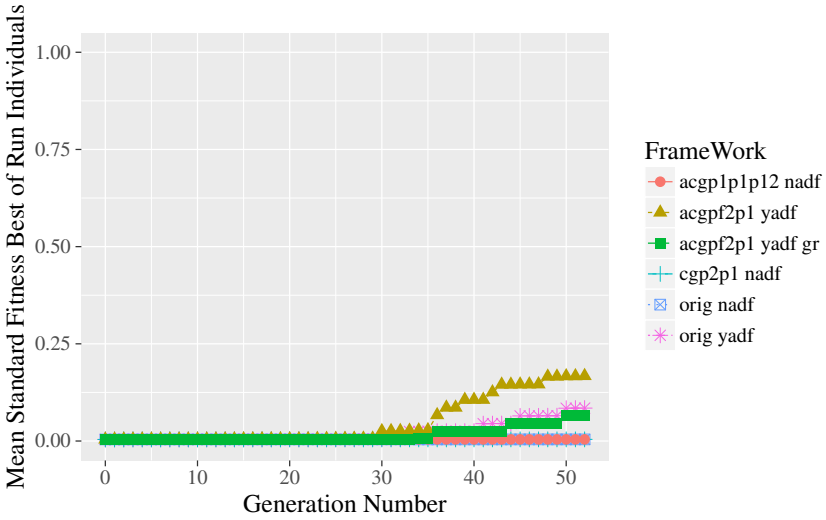
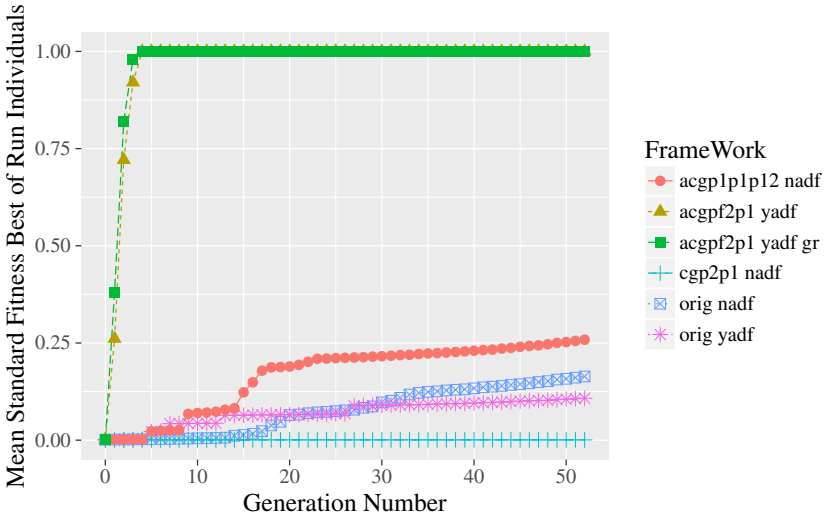


Figure 5.5: Problem: 3D Two Box
Best of Run Individuals
Max Depth 17 Max Generations 52



Because of the performance of the Bumble Bee problem, we perform additional analysis. First, we look at the role of mutation for the Bumble Bee problem. Then, we look at increasing the maximum generation from 52 to 104, 156 and 208 for the $2D$ Bumble Bee problem. We also look at increasing the $3D$ Bumble Bee problem maximum generation to 104.

5.7 Bumble Bee Additional Analysis and Results No Mutation

In this section we report the results of additional experiments where mutation was not used. In the Lawnmower problem favorable results were found not using the mutation operator. So, maybe mutation is hurting performance for the Bumble Bee problem.

It can be seen in figures 5.6 and 5.7 for the $2D$ Bumble Bee problem mutation didn't harm the fitness of the best of run individuals when compared to figures 5.3 and 5.8 respectively.

Figure 5.6: Problem: Bumble Bee 2d Flowers 25
Best of Run Individuals
Max Depth 17 Max Generations 52
No Mutation

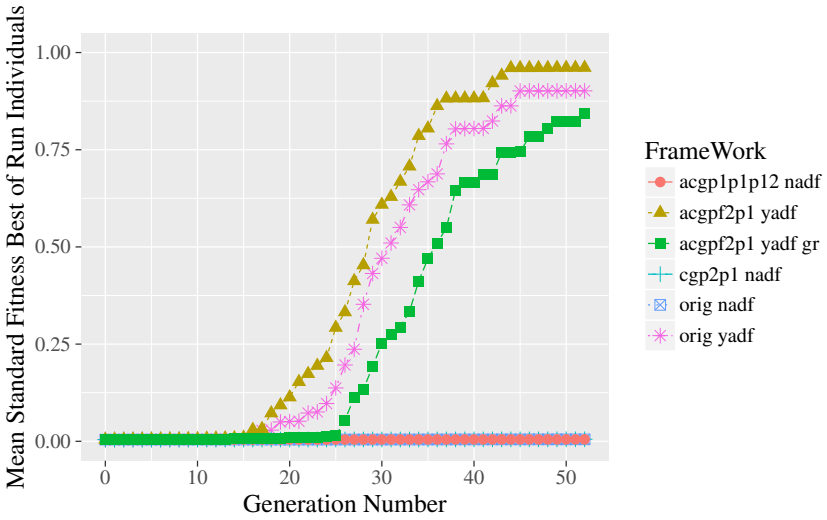
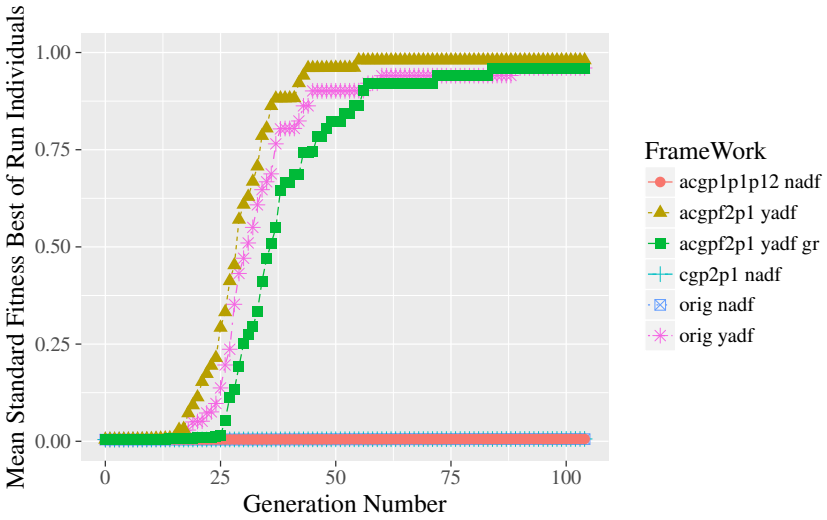


Figure 5.7: Problem: Bumble Bee 2d Flowers 25
Best of Run Individuals
Max Depth 17 Max Generations 104
No Mutation



Next we in experiment with increasing the maximum generation including mutation and the generation ramp feature.

5.8 Bumble Bee Additional Analysis and Results Increasing the Maximum Generation With Mutation

In this section we increase the maximum number of generations for the Bumble Bee problem. In the version of the Bumble Bee problem where maximum generation is set to 52, a run would start at generation 0 where the tree depth was capped at 5. That cap would remain in place for 4 generations before being allowed to increase to a maximum tree depth of 6. The maximum tree depth would be allowed to increase every 4 generations until the maximum tree depth of 17 is reached by generation 48.

In the version of Bumble Bee problem where the maximum generation is 104 a GP run would start out where the maximum depth is capped to 5 on generation 0. The cap would remain at 8 generations. Every 8 generations the maximum depth would be allowed to increase by one until a maximum tree depth of 17 is allowed to happen at generation 96.

For the Bumble Bee problem where the maximum generation is 156, the cap would change every 12 generations.

And, for the Bumble Bee problem where the maximum generation is 208, the cap would change every 16 generations.

It can be seen in figures 5.8, 5.9 and 5.10 that increasing the maximum number of generations from 104 to 156 and to 208 has helped in the $2D$ version of the Bumble Bee problem. It has also helped in the $3d$ version of the problem. That can be seen in figure 5.11.

Figure 5.8: Problem: Bumble Bee 2d Flowers 25
Best of Run Individuals
Max Depth 17 Max Generations 104

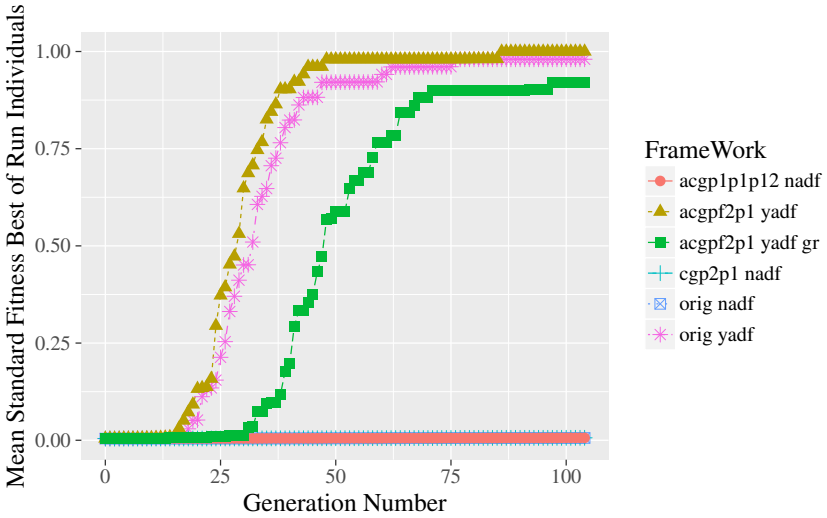


Figure 5.9: Problem: Bumble Bee 2d Flowers 25
Best of Run Individuals
Max Depth 17 Max Generations 156

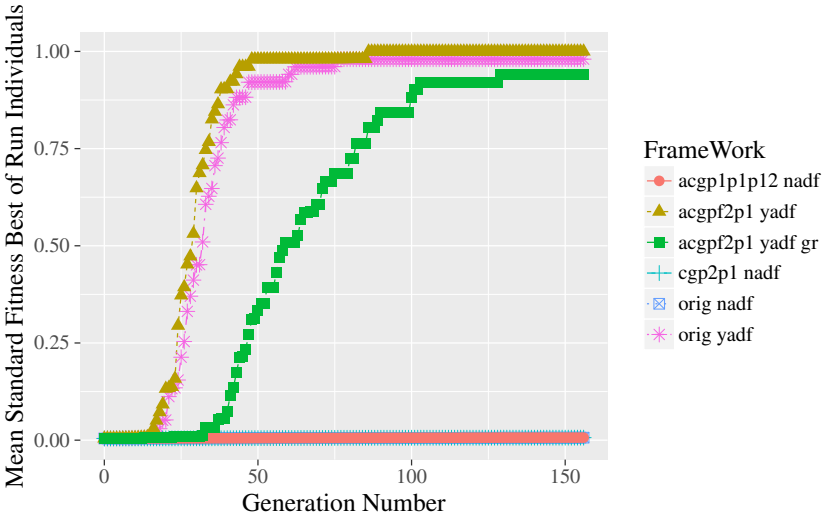


Figure 5.10: Problem: Bumble Bee 2d Flowers 25
Best of Run Individuals
Max Depth 17 Max Generations 208

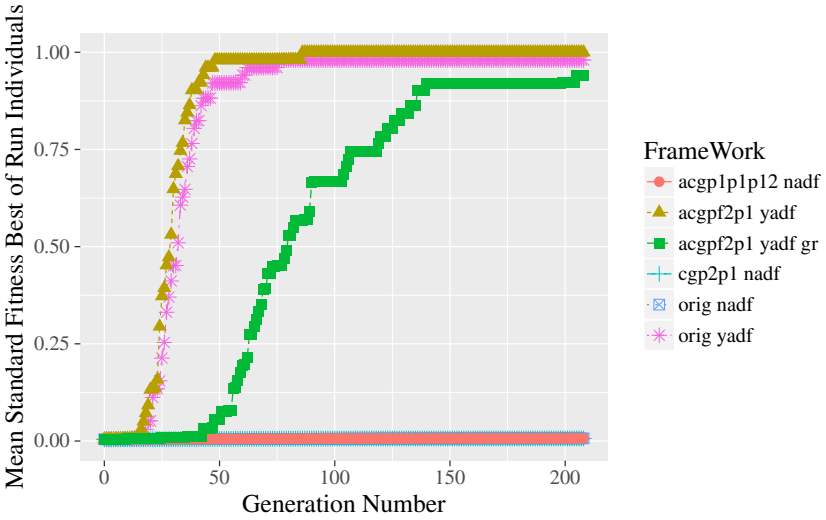
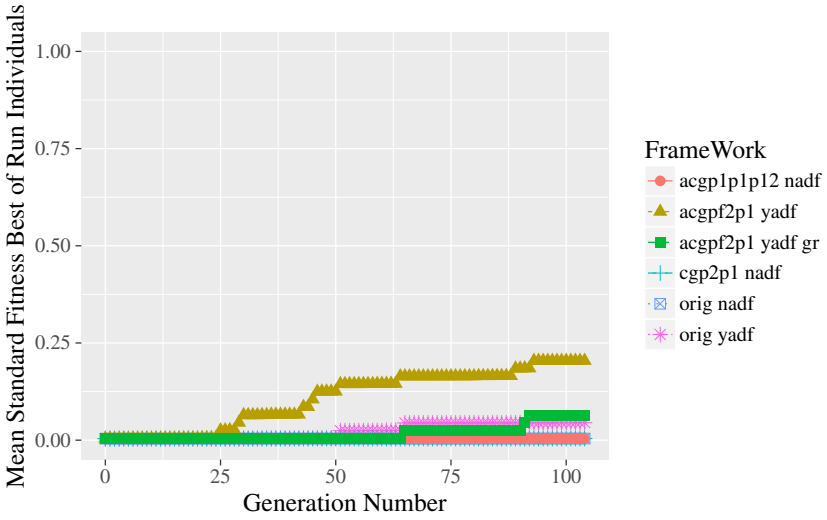


Figure 5.11: Problem: Bumble Bee 3d Flowers 25
Best of Run Individuals
Max Depth 17 Max Generations 104



We now move onto a discussion of the results found in this dissertation.

Chapter 6

Results Discussion and Future Directions

6.1 Results Discussion

We have shown that a combination of previous constraint methods as well as a new bloat control method has improved performance of Genetic Programming on larger problems. A number of benchmark problems were used to achieve those results as described in the previous chapter. We give a brief summary of those results next, followed by a discussion of implications and directions for future research.

When looking at the Lawnmower problem, ACGPF outperformed SGP, CGP and ACGP. The goal of the problem is to create a path for a lawnmower to mow; the mowing operation is performed on a grid of tiles. Koza created the Lawnmower problem to be a problem which could be scaled as seen in Koza (1994b). The problem in its original incarnation was a hard problem to solve mainly due to the computing power of that decade. When the problem grid size was increased from his maximum size of $8x12$ to our size of $25x25$ and $50x50$, the problem was exponentially harder to solve thus making for a good benchmarking problem. Original SGP did not include a typing or constraint mechanism so there was no way to constrain the search space, hence SGP with or without ADFs did not perform very well. In CGP, we use prior knowledge to use the typing mechanism and a constraint mechanism to reduce search space. The Lawnmower problem was designed as a vehicle to showcase the power of ADFs, but ADFs are absent from CGP. Thus, CGP did not perform as well as ACGPF when configured to use ADFs, a strong constraint on the `frog` function and using ACGP features. This was also true of ACGP, which has no typing mechanism or ADFs. ACGPF outperformed them by combining features from each one of these frameworks. It also found small correct solutions when the delayed tree growth feature was used to combat bloat.

For the Bumble Bee problem we started out with Koza's (1994b) $2D$ version and later created a much harder $3D$ version of the problem. Types were not used for this problem and there were no

strong constraints. The goal of this problem is for a bee to find a number of preset flowers. This too is a scalable problem with 10, 15, 20 and 25 flowers located on $2D$ plane and later in a $3D$ box. For the $2D$ version, the bee's (x, y) position is given as 2 real numbers. For the $3D$ version, the bee's (x, y, z) position is given as 3 real numbers. This problem was more about the interaction of ADFs with ACGP heuristic features. Results were mixed but promising. We saw, in general, that ACGPF outperformed SGP, CGP and ACGP. The solutions, however, tended to be bigger. It is interesting to note that average fitness of individuals were slightly worse when using the delayed tree growth feature when the maximum generation count was set at 52. Additional experiments were done to increase the maximum generation to 104, 152 and 208 to see if the delayed tree growth feature would perform better. It did, but at the cost of increased GP evaluation times. Care will have to be taken when drawing conclusions using this feature for problems where only ADFs and ACGP features are used.

Where ACGPF really shows its power is for problems where ADFs, types, strong constraints and ACGP features can all be used. Prior domain knowledge is needed for this feature to work as planned. The two box problem was used to showcase this configuration. In this problem, there were strong constraints on not using the mathematical division and addition operations. We further put type constraints on how functions and terminals were combined to help find the solution. ACGPF was the only framework to find solutions for all ten fitness cases.

We now turn to a discussion of specific aspects of the inner workings of the ACGPF framework. We also discuss several issues encountered when using the ACGPF framework.

When ADFs are used on their own for a GP run, individuals tend to have a large tree depth and node count for both the RPB and any ADFs. This was seen in the SGP results when looking at the average maximum tree depth of individuals. This bloating of individuals happened early in the first generation of a GP run. This was one of many inspirations for usage of the delayed tree growth feature in conjunction with the usage of ADFs, types and adaptive discovery of constraints. Bloat occurs as a result of crossover. If a random location is chosen for cross over at a high enough location in a large tree individual and swapped to a random location in a smaller tree individual, that smaller tree will become larger as a result, unless a cap is placed on maximum tree size for that generation. If this happens enough times, the population will contain larger individuals. Smaller solution individuals will have a higher chance of getting overlooked because of the shear number of large tree individuals being created due to crossover. Intuitively, this makes sense and was proven by the experiments and results.

One might think that bloat would happen with the mutation operation, because a location chosen for mutation might get larger as a result of a newer longer subtree being created at the mutation location. On the other hand, there could be a corresponding chance that the randomly chosen location for creation of a subtree might be shorter. We saw that mutation did not affect the fitness nor the size of the individuals for the Bumble Bee problems with and without mutation.

Types used in ACGPF are a form of strong constraint, ensuring that functions call correct functions and functions call or use correct terminals. The search space of correct programs is drastically

reduced because we are not evaluating incorrect combinations. Using types is even more powerful when used with ADFs. We saw this in the results for the Two Box problem. ACGPF outperformed all other methodologies when types were used. If types were not used there was no restriction on function to function and function to terminal pairings. Because of bloat being caused by crossover, there will be little chance that highly fit small solutions are created and discovered.

The type and constraint system, when used, can affect how functions and or terminals are placed in the upper layers of an individual. This includes combinations for the RPB and any ADFs. If types were used, for example, in the Two Box problem, the upper levels of the RPB and the ADF where identical for most of the best of run individuals. This was true when also using the generation ramp feature. In addition, when placing strong constraint on using `sub` operation at the root of the RPB, it too, helped make the upper levels similar if not identical. It is interesting to note that when using the generation ramp feature, best of run individuals were discovered earlier. This is due to placing a cap on the max tree depth to a low number allowing smaller individuals to be created. The smaller trees contain the solutions for this problem. The upper levels for all of the best of run individuals for the Bumble Bee problem and the Lawnmower problem were different combinations of functions or ADFs. Neither of those problems used types. For the Lawnmower problem there was a strong constraint on not using the `frog` function, but there were no constraints on what could be placed at the root of the trees that make up the individual.

A downside in using a type system is that we have to have prior knowledge of the problem domain to use types correctly. In fact, types might actually cause more harm than good. For example, suppose we are trying to constrain the search space using types, ahead of time, and we incorrectly assume that functions and terminals are to be combined in a unique but incorrect way. Correct individuals would be discarded because they do not meet the requirements of the incorrect type definition. In creating a type system, care needs to be taken. Currently, ACGPF ensures correct individuals are created through definition of correct pairings of function calls to functions or terminals. If the type system is set up incorrectly it may never create correct individuals. It is important to point out that the type system in ACGPF has some limitations. When an individual is created, it is checked for accuracy against the definitions in the type system. There is a slight chance that a GP run will halt early due to the attempt account reaching a preset number. The number of attempts was set at 200. In practice, the attempt count was rarely reached. There were, however, rare instances when a run would end early because the attempt count was reached.

A similar issue to the attempt count being reached happened in the Lawnmower problem for the move count parameter. The move count was a limit on how many moves a lawnmower could move. There were rare times, on the larger problem sizes, when the move count hit the maximum because it would evolve a solution that involved all left turns for only a portion of the lawn. There was no move count implemented for the Bumble Bee problem; eventually, the bee would find a path to all flowers.

Because ACGPF contains typing functionality of CGP, function overloading is available. This was used in the Two Box problem for math operations. A word of caution when using types in

ACGPF: one may think, because of the problem definition, that terminals can be overloaded. Reasoning could lead to incorrectly thinking that terminals are zero-argument functions. Currently, the ACGPF type system does not support overloaded terminals; this is undefined behavior and the ACGPF system will likely have a runtime crash. The typing system needs improvement in this area.

When comparing the typing system in ACGPF to frameworks, there are other frameworks with much more advanced GP typing systems. Examples described in chapter 3 include the work of Yu (1999) and Binard and Felty (2007). The first one uses the expressiveness of a typed lambda calculus. The second is a polymorphic lambda calculus called System F. ACGPF should be extended to handle these more expressive and powerful type systems.

Caution needs to be inserted at this point in the discussion as it relates to usage of the choice of benchmarks and conclusions being drawn. Symbolic regression benchmarks and the Bumble Bee benchmark will be used as a example to illustrate this caution.

There is the success of using ACGPF for the symbolic regression two box problem. Many other benchmark problems were explored during the course of this work. Early in this research there were implementations of all 53 symbolic regression benchmark as found in the work of McDermott et al. (2012). Included where all 3 of Koza's symbolic regression benchmarks as found in Koza (1992). There were, however, much more difficult benchmark problems included this original symbolic regression suite. Originally this dissertation was going to include all of these results. They were not included, at the advice of committee members, to keep the page count down on this dissertation. We can, however, use a portion of this work as an cautionary illustrative explanation for the choice of which benchmark to use when doing scalability GP research. An example from one of the benchmarks described above is entitled Korns10 which can be seen on the next line.

$$z = 2.0 - (2.1 * (cos(9.8 * x0) * sin(1.3 * x1)))$$

For a fitness case $x0$ and $x1$ are initialized to a random real values number between -50 and 50. Preliminary results were not good. One might observe that it has the mathematical minus operation and incorrectly conclude that the solution is to constrain the search space similar to the constraints in the Two Box problem. The real problem can be found in use of the real valued constants. Search space is extremely large if we include all 99 terminal combinations like $0.0, 0.1 \ldots 9.8, 9.9$. Current GP frameworks, including ACGPF are not able to handle problems of this nature with the computer hardware available for this research. One could think that ephemeral random constants could be of use. Instead of having 99 terminal combinations, a ERC could be used. An ERC is a special function that generates a random number, in this case, between 0.0 and 9.9. Usage of ERCs only disguises the problem of a large search space. We still need to have GP correctly construct a program whose portion is drawn from $99^2 = 9801$ constant terminal combinations. To get a feel for the enormity of this search space let's say we magically are able to generate the correct formula for every generation we wanted, but we were unlucky enough to know that the correct choice of constant terminal combination was to happen on the last generation. We would need 9801 generations to come to that correct evolution. In reality with that many terminals many more generations would be needed

to converge on a solution.

This leads to a research problem. How to do scalability research on current hardware? You have to construct tunable benchmark problems to test a methodology and that requires prior knowledge. Generalizations based on this research have to be made very cautiously.

With the above cautionary tail in mind the Two Box problem was a difficult symbolic regression problem that used prior knowledge in constraining. We needed that kind of problem to compare to other research efforts that used prior knowledge, such as the CGP. It also illustrates the tug and pull between what can fit on current computer hardware and yet be difficult enough to compare the ACGPF methodology to previous methodologies.

In addition, with this cautionary tail in mind, the Bumble Bee problem shows what happens when we don't use prior knowledge to constrain the search space. There were no types or constraints placed on the search. The problem forced us to only use the ADFs and the ACGP portion of the ACGPF framework. It was a much more difficult problem to solve and the combination ACGP with ADFs were beneficial to finding correct paths.

An additional caution needs to be discussed here. When we just focus on the very low evaluation times reported in the results tables, these problems were constructed to explore some aspect of ADFs. We see this in the low evaluation times of the Lawnmower problem and the Two Box problem. In the Lawnmower problem ADFs, ACGPF and strong constraints were used effectively. Also, we see this even to a higher degree in the Two Box problem where ADFs, ACGP, typing and strong constraints were used. When typing and strong constraints were not used for the 2 versions of the Bumble Bee problem, evaluation times were in some cases increased when using a combined ADFs and ACGP. The reason for this is that we are comparing runs where we are using ADFs to frameworks that do not have ADFs in their implementation. For example, CGP and ACGP do not have ADFs in their implementation. Those implementations have less nodes to evaluate than ACGPF which is setup to use ADFs.

Next, we move to a remark about the uniqueness of the ACGPF methodology.

The closest cousin of ACGPF is Rosca's work on ARL, described in a number of papers in the mid to late $1990's$, and reviewed in the appendix to this dissertation. In their method and implementation they evolve function bodies and use a frequency counting mechanism similar to ACGP. In their research, however, ADFs can be added and deleted during the course of a ARL run. In ACGPF, an ADF function signature is predefined; it is never deleted. It is created at compile time and the function body evolves.

One of the inspirations for delayed tree growth was Rosca's concept of an "epoch" in ARL and their work with Minimum Description Length. Delayed tree growth, however, has a different function than what an epoch does in ARL and parsimony of individuals in ARL. In ARL, a new epoch is recorded at the discovery of a new kind of individual and ends when no new kinds of individuals are discovered. When no new individuals are discovered, extinction is triggered and fit individuals are retained to seed the population for the next epoch. Their cross generational use of an epoch is tied to population diversity. In ACGPF, our cross generation operation is not linked to diversity in

the population, but is used to find smaller fit individuals. For ACGPF to do the kind of work done in ARL, individuals would have to be encoded to track the number of unique individuals. Implementing a prefix tree that holds all of the individuals generated during a ACGPF run could help with this effort.

We now move to possible future directions for ACGPF research, as listed in the next sections.

6.2 Future Directions

6.2.1 Expand to 2nd Order Heuristics

We currently use first order heuristics in ACGPF. The concept of 1^{st} and 2^{nd} order heuristics was discussed in section 3.3.3. The upgrade would be to take ACGPF and enhance it to use second order heuristics, with the advantage that 2^{nd} order heuristics could improve the search for fit individuals. The new version will be called ACGPF2.2.

6.2.2 Type System Enhanced to Handle Higher Order Logic

As noted earlier, it is desirable to expand the type system to handle higher order logic. The work on Abstraction Based Genetic Programming by Binard and Felty's has many examples of using GP with HOF. The advantage of this expanded type system is that evolved solutions are proofs in 2^{nd} order logic.

6.2.3 Add Architecture Altering Operations

Architecturally Altering Operation were introduced by Koza (1994a) and are more fully explored in appendix A section A.6. One part of AAOs is to have parameters to functions be dynamically added or deleted. In ACGPF at the moment, functions, terminals and their parameters are hard coded into an experiment before it is compiled to an executable and run. The `fset` structure holds these functions and terminals as well as a count of arguments to each of these. Functions and terminals are setup in the `app_build_function_sets` lilgp funciton. The `fset` memory structure could be allocated dynamically. A separate mechanism would need to be developed to keep track of how many parameters a function is using. More work would need to be done on keeping programs valid as the number and type of function arguments are added and deleted. It would be interesting to see if the adaptive features of ACGPF could help find the right number of parameters for functions.

6.2.4 Explore Minimum Description Length

Rosca and Ballard (1994c) explore using Rissanen's (1978) Minimum Description Length in their work on adaptive representations in GP. See Appendix A Section A.3.1 for a fuller review of that topic with examples. The ideas found in MDL are simple; if there is a way to describe a complex problem through a short program, that program becomes itself a short version of the data. Much of how data compression works is based on these ideas. Even if Rosca did not find decent results using this technique, it still is intriguing. This could be a way to track and use higher order heuristics. In many ways, ACGPF using ADFs, types and constraints is dynamically producing that short program that is computed when using the MDL.

6.3 Implementation Specific

6.3.1 Library

The current implementation is written in the "C" language and based on the lilgp1.02 framework. This framework could be changed into a library that could be callable by other languages. Many of the foreign function interfaces of other languages such as Python, R or Ocaml could make calls to the functionality provided by this implementation of ACGPF. An advantage of this is to make the functionality of ACGPF available to users familiar with other programming languages. In Python and R many packages are written in the "C" language. For example an implementation of GPU tensorflow neural network package is implemented in C and is callable by Python. In R, the stats library is implemented in C and callable by R.

6.3.2 Parallelism

The current implementation of ACGPF is limited to running on a core of a CPU. With the GNU command line tool parallel we can exploit parallelism found on modern day CPU processors. For this dissertation there was a limit on 8 processes running at a time on the CPU. Langdon and Banzhaf (2008) and Langdon and Harman (2010) first explored using GP on a GPU. More recent efforts have taken place by da Silva et al. (2015); in their work, they could handle larger problems and arrive at solutions faster. None of these methods include types or ADFs. Porting ACGPF to take advantage modern GPUs technology could allow it to handle larger problems. On modern GPUs, instead of 8 or 16 cores found on CPUs, there are $1000's$ of cores.

Another possible direction on parallelism is to port this implementation to the MPICH methodology and library. This tool is used by many of today's fastest super computers. Using this programming methodology and set of libraries, computers can be clustered. These clustered computers each have many cores and many GPUs. As of the date of this dissertation, the number of cores found on the fastest super computers is in the millions.

6.4 Summary

In summary, this dissertation successfully explored that a combination of GP methods is better than each method used on its own. These methods include constraints and types found in CGP, adaptive constraints and heuristics found in ACGP, ADFs found in standard GP and the new generation ramp feature. Larger problems could be tackled, and in some cases, time to find best individual solutions took substantially less time and were substantially smaller in size. There were, however, situations in which ACGPF did not perform as well. For example, in the performance of ACGPF on the Bumble Bee problem, types and strong constraints were not applied. A number of future directions are possible in this area of research, including use of 2^{nd} order heuristics, applying minimum description length techniques, expanding the type system to handle higher order logic and making use of the massive parallelism found on today's modern GPUs.

References

Ahluwalia, M. and Bull, L. (2001). Coevolving functions in genetic programming. *Journal of Systems Architecture*, 47(7):573–585.

Aleshunas, J. and Janikow, C. (2011). Cost-benefit analysis of using heuristics in acgp. In Smith, A. E., editor, *Proceedings of the 2011 IEEE Congress on Evolutionary Computation*, pages 1177–1183, New Orleans, USA. IEEE Computational Intelligence Society, IEEE Press.

Aleshunas, J. J. (2013). *GP Representation Space Reduction Using a Tiered Search Scheme*. PhD thesis, University of Missouri, St. Louis.

Angeline, P. J. and Pollack, J. (1993). Evolutionary module acquisition. In Fogel, D. and Atmar, W., editors, *Proceedings of the Second Annual Conference on Evolutionary Programming*, pages 154–163, La Jolla, CA, USA.

Angeline, P. J. and Pollack, J. B. (1992). The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pages 236–241, Bloomington, Indiana, USA. Lawrence Erlbaum.

Baluja, S. (1994). Population–based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report CMU–CS–94–163, School of Computer Science Carnegie Mellon University, Pitsburgh, Pennsylvania, 15213.

Banzhaf, W., Banscherus, D., and Dittrich, P. (2000). Hierarchical genetic programming using local modules. *InterJournal Complex Systems*, 228.

Beyer, H.-G. and Schwefel, H.-P. (2002). Evolution strategies–a comprehensive introduction. *Natural computing*, 1(1):3–52.

Binard, F. and Felty, A. (2007). An abstraction-based genetic programming system. In Bosman, P. A. N., editor, *Late breaking paper at Genetic and Evolutionary Computation Conference (GECCO'2007)*, pages 2415–2422, London, United Kingdom. ACM Press.

Binard, F. and Felty, A. (2008). Genetic programming with polymorphic types and higher-order functions. In Keijzer, M., Antoniol, G., Congdon, C. B., Deb, K., Doerr, B., Hansen, N., Holmes, J. H., Hornby, G. S., Howard, D., Kennedy, J., Kumar, S., Lobo, F. G., Miller, J. F., Moore, J., Neumann, F., Pelikan, M., Pollack, J., Sastry, K., Stanley, K., Stoica, A., Talbi, E.-G., and Wegener, I., editors, *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1187–1194, Atlanta, GA, USA. ACM.

Binard, F. J. L. (2009). *Abstraction-Based Genetic Programming*. PhD thesis, Ottawa-Carleton Institute for Computer Science, School of Information Technology and Engineering, Faculty of Engineering, University of Ottawa, Ottawa, Canada.

Cook, R. T. (2009). *A Dictionary of Philosophical Logic*. Edinburgh University Press Series. Edinburgh University Press.

da Silva, C. P., Dias, D. M., Bentes, C., Pacheco, M. A. C., and Cupertino, L. F. (2015). Evolving

gpu machine code. *Journal of Machine Learning Research*, 16:673–712.

Dessi, A. (1998). Scoperta automatica di subroutine in programmazione genetica. Master's thesis, Department of Computer Science, University of Pisa, Italy.

Dessi, A., Giani, A., and Starita, A. (1999). An analysis of automatic subroutine discovery in genetic programming. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 996–1001, Orlando, Florida, USA. Morgan Kaufmann.

Gerules, G. and Janikow, C. (2016). A survey of modularity in genetic programming. In Ong, Y.-S., editor, *Proceedings of 2016 IEEE Congress on Evolutionary Computation (CEC 2016)*, pages 5034–5043, Vancouver. IEEE Press.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. The MIT Press. `http://www.deeplearningbook.org`.

Graham, P. (1994). *On LISP: Advanced Techniques for Common Lisp*. Prentice Hall.

Grünwald, P. D. (2007). *The Minimum Description Length Principle (Adaptive Computation and Machine Learning)*. The MIT Press.

Holland, J. H. (1962). Outline for a logical theory of adaptive systems. *J. ACM*, 9(3):297–314.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI. second edition, 1992.

Iba, H. and de Garis, H. (1996). Extending genetic programming with recombinative guidance. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 4, pages 69–88. MIT Press, Cambridge, MA, USA.

Iba, H., de Garis, H., and Sato, T. (1994). Genetic programming using a minimum description length principle. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, chapter 12, pages 265–284. MIT Press.

Janikow, C. and Aleshunas, J. (2013). Impact of commutative and non-commutative functions on symbolic regression with acgp. In de la Fraga, L. G., editor, *2013 IEEE Conference on Evolutionary Computation*, volume 1, pages 2290–2297, Cancun, Mexico.

Janikow, C. Z. (1996a). Constraints in genetic programming. Summer Faculty Fellowship Program 1995 Volumes 1 and 2, Page: 11-1 - 11-15, NASA.

Janikow, C. Z. (1996b). A methodology for processing problem constraints in genetic programming. *Computers and Mathematics with Applications*, 32(8):97–113.

Janikow, C. Z. (1997–2007a). CGP and ACGP technical manuals. `http://www.cs.umsl.edu/~janikow/cs6340/docs/`.

Janikow, C. Z. (1997–2007b). CGP and ACGP user's manuals. `http://www.cs.umsl.edu/~janikow/cs6340/docs/`.

Janikow, C. Z. (2004a). ACGP: Adaptable constrained genetic programming. In O'Reilly, U.-M., Yu, T., Riolo, R. L., and Worzel, B., editors, *Genetic Programming Theory and Practice II*, chapter 12, pages 191–206. Springer, Ann Arbor.

Janikow, C. Z. (2004b). Adapting representation in genetic programming. In Deb, K., editor, *Genetic and Evolutionary Computation – GECCO 2004*, pages 507–518, Berlin, Heidelberg. Springer Berlin Heidelberg.

Janikow, C. Z. (2005a). Adaptable constrained genetic programming: Extensions and applications. Summer Faculty Fellowship Program 2004 Volumes 1 and 2, Page: 11-1 - 11-7, NASA.

Janikow, C. Z. (2005b). Adaptable representation in GP. In Rothlauf, F., Blowers, M., Branke, J., Cagnoni, S., Garibay, I. I., Garibay, O., Grahl, J., Hornby, G., de Jong, E. D., Kovacs, T., Kumar, S., Lima, C. F., Llorà, X., Lobo, F., Merkle, L. D., Miller, J., Moore, J. H., O'Neill, M.,

Pelikan, M., Riopka, T. P., Ritchie, M. D., Sastry, K., Smith, S. L., Stringer, H., Takadama, K., Toussaint, M., Upton, S. C., and Wright, A. H., editors, *Genetic and Evolutionary Computation Conference (GECCO2005) workshop program*, pages 327–331, Washington, D.C., USA. ACM Press.

Janikow, C. Z. (2007c). Evolving problem heuristics with on-line acgp. In Bosman, P. A. N., editor, *Late breaking paper at Genetic and Evolutionary Computation Conference (GECCO'2007)*, pages 2503–2508, London, United Kingdom. ACM Press.

Janikow, C. Z., Aleshunas, J., and Hauschild, M. W. (2011). Second order heuristics in acgp. In Hauschild, M. and Pelikan, M., editors, *Optimization by building and using probabilistic models (OBUPM-2011)*, pages 671–678, Dublin, Ireland. ACM.

Janikow, C. Z. and Deshpande, R. A. (2003). Adaptation of representation in genetic programming. In Dagli, C. H., Buczak, A. L., Ghosh, J., Embrechts, M. J., and Ersoy, O., editors, *Smart Engineering System Design: Neural Networks, Fuzzy Logic, Evolutionary Programming, Complex Systems, and Artificial Life (ANNIE'2003)*, pages 45–50. ASME Press.

Janikow, C. Z. and DeWeese, S. (1997a). Improving search properties in genetic programming. Summer Faculty Fellowship Program 1996 Volumes 1 and 2, Page: 15-1 - 15-9, NASA.

Janikow, C. Z. and DeWeese, S. (1998). Processing constraints in genetic programming with CGP2.1. In Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H., and Riolo, R., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 173–180, University of Wisconsin, Madison, Wisconsin, USA. Morgan Kaufmann.

Janikow, C. Z. and DeWeese, S. W. (1997b). Cgp lil-gp 2.1;1.02 user's manual. Technical report, NASA.

Janikow, C. Z. and Mann, C. J. (2005). Cgp visits the santa fe trail: effects of heuristics on gp. In Beyer, H.-G., O'Reilly, U.-M., Arnold, D. V., Banzhaf, W., Blum, C., Bonabeau, E. W., Cantu-Paz, E., Dasgupta, D., Deb, K., Foster, J. A., de Jong, E. D., Lipson, H., Llora, X., Mancoridis, S., Pelikan, M., Raidl, G. R., Soule, T., Tyrrell, A. M., Watson, J.-P., and Zitzler, E., editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1697–1704, Washington DC, USA. ACM Press.

Johnson, L. (1981). The thermodynamic origin of ecosystems. *Canadian Journal of Fisheries and Aquatic Sciences*, 38(5):571–590.

Kernighan, B. W. (1988). *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition.

Kinnear, Jr., K. E. (1994). Alternatives in automatic function definition: A comparison of performance. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, chapter 6, pages 119–141. MIT Press.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

Koza, J. R. (1994a). Architecture-altering operations for evolving the architecture of a multipart program in genetic programming. Technical Report STAN-CS-94-1528, Dept. of Computer Science, Stanford University, Stanford, California 94305, USA.

Koza, J. R. (1994b). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts.

Koza, J. R., Andre, D., Bennett III, F. H., and Keane, M. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufman.

Langdon, W. B. and Banzhaf, W. (2008). A simd interpreter for genetic programming onăgpuăgraphicsăcards. In O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A. I., De Falco, I., Della Cioppa, A., and Tarantino, E., editors, *Genetic Programming*, pages 73–85, Berlin, Hei-

delberg. Springer Berlin Heidelberg.

Langdon, W. B. and Harman, M. (2010). Evolving a CUDA kernel from an nVidia template. In Sobrevilla, P., editor, *2010 IEEE World Congress on Computational Intelligence*, pages 2376–2383, Barcelona. IEEE.

Li, M. and Vitnyi, P. M. (2008). *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Publishing Company, Incorporated, 3 edition.

McDermott, J., White, D. R., Luke, S., Manzoni, L., Castelli, M., Vanneschi, L., Jaskowski, W., Krawiec, K., Harper, R., De Jong, K., and O'Reilly, U.-M. (2012). Genetic programming needs better benchmarks. In Soule, T., Auger, A., Moore, J., Pelta, D., Solnon, C., Preuss, M., Dorin, A., Ong, Y.-S., Blum, C., Silva, D. L., Neumann, F., Yu, T., Ekart, A., Browne, W., Kovacs, T., Wong, M.-L., Pizzuti, C., Rowe, J., Friedrich, T., Squillero, G., Bredeche, N., Smith, S. L., Motsinger-Reif, A., Lozano, J., Pelikan, M., Meyer-Nienberg, S., Igel, C., Hornby, G., Doursat, R., Gustafson, S., Olague, G., Yoo, S., Clark, J., Ochoa, G., Pappa, G., Lobo, F., Tauritz, D., Branke, J., and Deb, K., editors, *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 791–798, Philadelphia, Pennsylvania, USA. ACM.

Montana, D. J. (1993). Strongly typed genetic programming. BBN Technical Report #7866, Bolt Beranek and Newman, Inc., 10 Moulton Street, Cambridge, MA 02138, USA.

Montana, D. J. (1994). Strongly typed genetic programming. BBN Technical Report #7866, Bolt Beranek and Newman, Inc., 10 Moulton Street, Cambridge, MA 02138, USA.

Montana, D. J. (1995). Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230.

Ondas, R., Pelikan, M., and Sastry, K. (2005). Genetic programming, probabilistic incremental program evolution, and scalability. In Knowles, J., editor, *WSC10: 10th Online World Conference on Soft Computing in Industrial Applications*, pages 363–372, On the World Wide Web.

Pelikan, M., Goldberg, D. E., and Cantú-Paz, E. (1999). Boa: The bayesian optimization algorithm. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1*, GECCO'99, pages 525–532, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Rissanen, J. (1978). Paper: Modeling by shortest data description. *Automatica*, 14(5):465–471.

Rosca, J. (1995a). Towards automatic discovery of building blocks in genetic programming. In Siegel, E. V. and Koza, J. R., editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 78–85, MIT, Cambridge, MA, USA. AAAI.

Rosca, J. and Ballard, D. H. (1995). Causality in genetic programming. In Eshelman, L. J., editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 256–263, Pittsburgh, PA, USA. Morgan Kaufmann.

Rosca, J. P. (1995b). Entropy-driven adaptive representation. In Rosca, J. P., editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 23–32, Tahoe City, California, USA.

Rosca, J. P. (1995c). Genetic programming exploratory power and the discovery of functions. In McDonnell, J. R., Reynolds, R. G., and Fogel, D. B., editors, *Evolutionary Programming IV Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pages 719–736, San Diego, CA, USA. MIT Press.

Rosca, J. P. (1997). *Hierarchical Learning with Procedural Abstraction Mechanisms*. PhD thesis, Department of Computer Science, The College of Arts and Sciences, University of Rochester, Rochester, NY 14627, USA.

Rosca, J. P. and Ballard, D. H. (1994a). Genetic programming with adaptive representations. Technical Report TR 489, University of Rochester, Computer Science Department, Rochester, NY,

USA.

Rosca, J. P. and Ballard, D. H. (1994b). Hierarchical self-organization in genetic programming. In *Proceedings of the Eleventh International Conference on Machine Learning*. Morgan Kaufmann.

Rosca, J. P. and Ballard, D. H. (1994c). Learning by adapting representations in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 407–412, Orlando, Florida, USA. IEEE Press.

Rosca, J. P. and Ballard, D. H. (1996). Discovery of subroutines in genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 9, pages 177–201. MIT Press, Cambridge, MA, USA.

Schrödinger, E. (1944). *What is life? : the physical aspect of the living cell / by Erwin Schrödinger*. Cambridge University Press Cambridge [England].

Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423.

Spector, L. (1995). Evolving control structures with automatically defined macros. In Siegel, E. V. and Koza, J. R., editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 99–105, MIT, Cambridge, MA, USA. AAAI.

Steele, G. L. (1990). *Common LISP: the language, 2nd Edition*. Digital Pr.

Tackett, W. A. (1994). *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California, Department of Electrical Engineering Systems, USA.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK.

Wicken, J. (1988). Thermodynamics, evolution, and emergence: Ingredients for a new synthesis. In Depew, D. J., Weber, B. H., Smith, J. D., and California State University, F., editors, *Entropy, information, and evolution : new perspectives on physical and biological evolution / edited by Bruce H. Weber, David J. Depew, and James D. Smith*, book 7. MIT Press Cambridge, Mass.

Yu, G. T. (1999). *An Analysis of the Impact of Functional Programming Techniques on Genetic Programming*. PhD thesis, University College, London, Gower Street, London, WC1E 6BT.

Yu, T. and Clack, C. (1998). Polygp: A polymorphic genetic programming system in haskell. In Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H., and Riolo, R., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 416–421, University of Wisconsin, Madison, Wisconsin, USA. Morgan Kaufmann.

Zongker, D. and Punch, B. (1996). lilgp 1.01 user's manual. Technical report, Michigan State University, USA.

Appendix A

Additional Background Literature

A.1 Introduction

The notion of encapsulating a function has been explored under the guise of many names in GP. This section give an overview of these bodies of research. They are included here in that, although related in spirit, they are not direct ancestors of the CGP and ACGP body of research. These bodies of research could serve as a inspiration for future research for CGP and ACGP.

A.2 Module Acquisition - (MA)

Angeline and Pollack (1993) approach how a subroutine is formed differently than Koza's ADFs. In their approach, called MA, a portion of the RPB is clipped and removed. The clipped portion, the newly created module, is named and inserted into a library for further potential use. A depth limit is placed on how much is clipped from the portion of the RPB. The name of the newly created module is inserted at the location where code was clipped. If there are connections below the depth of the clipped tree, those connections become parameters to the newly formed module. Once added to the library, modules themselves are frozen and do not evolve. A reference count is kept on the most used modules which is used as a measure of fitness.

The clipping portion of the overall tree is referred to as a compression operation by Angeline and Pollack (1993). A representation of this operation can be seen in figure A.1. The process of creating this new module is similar to Koza's encapsulation operation as described in his first book.[1]

In their work, another operation is called the expansion operation. This is the opposite of the compression operation. During the course of a GP run, an individual may have references to one or

[1] Koza's first book, p110, section 6.5.4.

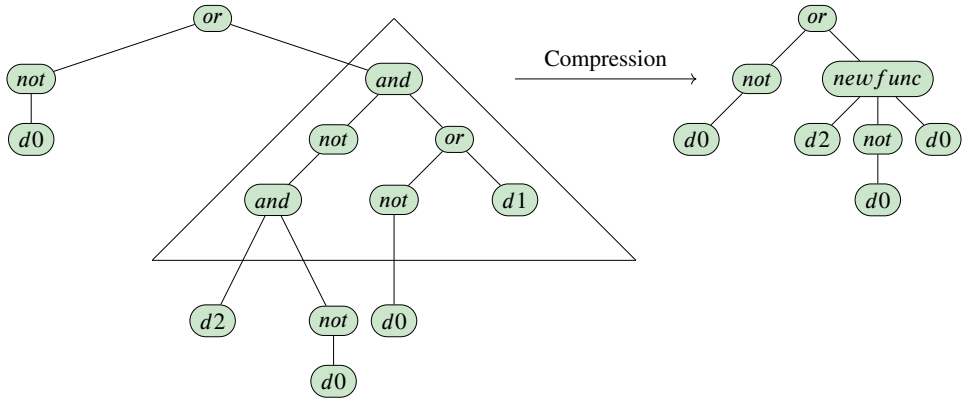


Figure A.1: MA compression operation

more modules in the library. The expansion operation replaces the named reference of that module with the actual code that makes up that module. If the program tree makes a call to a named module in the library, code that is stored in the library is inserted at the location of the name in the tree. It is similar to a macro expansion in the ”C” programming language. As the authors point out, this allows genetic material from the function body to participate in crossover and mutation.

The artificial ant problem was used as a benchmark problem for their work.

In a paper comparing of Koza's ADF to Angeline and Pollack's MA, using the 4 bit even parity problem as their benchmark, Kinnear, Jr. (1994) notices that MA does perform very well. ADF outperforms MA because of the *structural regularity* of individuals in a search space. Kinnear, Jr. (1994) notes that, if one were to expand all of the ADFs used by a GP, there would be large number of similar sub trees in that individual. ADFs exploit this regularity to its success. Those similar sub trees might not be necessarily be identical because the formal parameters to those ADFs might be different. MA does not create this kind of structural regularity.

Ahluwalia and Bull (2001), extend the MA concepts to the realm of multiple sub populations and call their method Evolutionary Defined Function. Original MA only used one population for source material for individuals. Their method enhances MA, but uses terminology borrowed from Koza's ADF work. In this approach, each module has its own population. This population coevolves independently from other populations. They use compression and expansion operators found in original, Angeline and Pollack (1993) MA, but on their enhanced method. Modules are stored and used in a library similar to original MA.

Two classification problems are used by Ahluwalia and Bull (2001) for their approach. The first problem is to evolve a credit worthiness indicator. The second problem is to evolve code to recognize images of letters. For the credit worthiness indicator, their approach did better than standard GP and about as well as a standard ADF implementation. On the letter recognition problem, the standard ADF approach did better than their EDF approach.

A.3 Adaptive Representations

This section investigates the theme of Adaptive Representations in GP. Two research groups' work will be highlighted. First, the work of Rosca and Ballard (1994b) is explored. These frameworks are original AR, HGP and ARL. Those frameworks will be covered in the following sub sections.[2] After this, the work of Dessi et al. is explored. Their multi paradigm framework is called ADGP. Their work includes many of the previous topics covered in this chapter such as MA and ARL. Because their focus is on ARL, it is included in this section. Information on these frameworks will be covered in the following sections.

A.3.1 Adaptive Representation - (AR)

As we discussed in the introduction of this proposal, a SGP individual is made up of a tree of nodes and terminals. These nodes form a collective set of operations on the leaves of the tree which are the terminals. This combination of nodes and terminals make up the *representation* of an individual in the population of programs. This representation is fixed and static. In a technical report by Rosca and Ballard (1994a), the representation is dynamic.[3] This dynamic representation, which they call AR, is created by finding substructures in the evolving program. These substructures are called Building Blocks. What makes AR different from Koza's ADF of this time period is how subroutines are dynamically created. As covered later in in Section A.6 Koza's representation also becomes dynamic. The next section will investigate the advantages of AR over ADFs and MA.

For the ADF framework, Rosca and Ballard (1994a) note that GP extended with ADFs makes GP more efficient. If ADFs were not used, the individuals that make programs in the population population would increase in size dramatically. Another interesting observation they note is that a GP individual using ADFs will make larger jumps to a solution in the GP search space. The reason for this is the potential hierarchies of calls on how ADFs are used. A change, either by mutation or crossover, in the body of one ADF may radically alter the architecture of the overall individual. They also state that there are many advantages of using ADFs because of their flexibility and performance enhancements.

For the MA framework, the authors highlight that there are a few potential problems with this approach. First, because of the compression and expansion operations, the individual's *representation* is extended with too many modules, thereby reducing the individual's usefulness. They also point out that a module's intrinsic value is defined by the number of times it is called. If a module is not used very frequently, it must not be very valuable. Next, they point out the increased memory overhead of keeping a library of modules around. For Rosca and Ballard this is not a good thing. This might be due to the computing resources available in the mid $1990's$. They also cite Kinnear,

[2] There is overlap of information in the papers cited in the next three sections. Much of the work cited becomes Rosca's (1997) dissertation. Relevant themes will be cited along the way.

[3] An abbreviated from of this technical report without discussion of Minimum Description Length and complexity measures was published as Rosca and Ballard (1994c).

Jr. (1994) as a motivation for their improved method.

The inspiration for AR comes from the work of Holland (1975) and Goldberg (1989) on GAs. In GAs a schema is a pattern of 0s and 1s embedded with a wild card symbol. An example might be seen as the string in reference A.1.

$$110 * 010 \tag{A.1}$$

Here the $*$ is a wild card and can be either a 1 or 0 at that location. The BB states that through evolutionary pressure of a GA run, fit schema, or BBs will be created. Rosca and Ballard point out that the reason GAs are successful is that the internal representation of BBs in a GA change representation during the course of evolution.

By way of analogy Rosca and Ballard (1994a) argue that a portion of a GP tree can be a BB. Here they define a BB as an entire subtree of a given height. That BB is chosen from the lower depths of the tree. The reason for this is that BBs chosen from the lower depths of a tree hold more promise for being correctly identified as valuable material for subroutines. In contrast, MA selects a random location in the GP tree and selects a subtree of a given height. That subtree might have calls to other trees and introduce complexity where it is not needed or break apart good BBs.

When identified the subroutine, in AR, it is stored in a library. The function set is extended with the new subroutine. The routine is not deleted, however, in later versions of AR, there is a mechanism for deletion.

A BB can be generalized in their framework by replacing leaves of the subtree with variables thus making it a function. This process of selecting BBs of a max height from lower levels of the tree and turning them into functions is analogous to what a schemata does in GA when it is forming BBs out of $1's$ and $0's$. This process is referred to as a "bottom up" approach. [4] Entire BB hierarchies are built up from the genetic material found at the lower depth of the tree. This process of identifying, measuring and potentially using BB material from the lower depths is actively changing the representation of a GP individual on the fly. This is how they arrived at calling their framework adaptive representation.

The authors contend that good BBs can be identified as either frequent blocks or fit blocks. As noted earlier in this section, frequent blocks are not necessarily fit blocks. They cite work where they do a histogram of blocks according to fitness versus frequency. They found that many of the highly fit blocks were not frequently used, but played an important part in the overall fitness of a particular individual. They also found that highly fit blocks have appeared quite late in the evolution process. Conversely, low fitness blocks that were highly used in earlier generations became less so in later generations.

Because of the unsuitability of the frequent block metric, they reason that a better metric is needed for fit blocks. They outline a few potential methods to help. One method is to use the fitness function used by the entire system on a BB. There might be a problem with this approach in that

[4] In a separate paper Rosca and Ballard (1995) give more detail, which will be discussed in the HGP section that follows.

preexisting domain knowledge might be needed in correct formation of this fitness function. A second method could be to create a fitness function that is simpler and operates on a smaller set of combinations of functions and terminals. This would scale the problem down to come up with a solution more efficiently. But, this too poses the problem that some preexisting knowledge of the problem space is needed. And a third method would be to give separate fitness functions to each building block based on preexisting knowledge.

As stated previously, their goal is to identify BBs from the lower depths of the tree. They record the path from the root to that BB which is called the pivot. The path to the pivot is called the *pivot path*. That pivot path, the generation of BB discovery and the fitness of that BB are stored at the pivot location. Their hypothesis is that the pivot location is the place to search for new building blocks. Pivot location is found as part of the crossover operation, as described later in this chapter in Rosca and Ballard (1995).

To put evolutionary pressure on keeping tree size manageable, they develop two complexity measures. These complexity measures modify the fitness of the individual. One is the Structure Complexity and the other is Evaluation Complexity. Their notation will be used when explaining these measures in the next few paragraphs.

The first complexity measure, SC, N is the number of nodes used for a given program F and is denoted as N = Size(F). F might make 1 or more calls to BB or functions. This is represented by the following formula A.2

$$SC(F_0) = \sum_{0 \leq j \leq m} Size(F_j) \tag{A.2}$$

where F_0 is made up of direct or indirect calls to the functions $F_1, F_2 \ldots F_m$. In the formula SC stands for structural complexity. m is the number of BBs or subroutines discovered during evolution. j is used to denote a particular BB or subroutine.

For the second complexity measure, EC is a way to generically evaluate how long a function takes to complete. Their original formula is as follows

$$EC(F_i) = Size(F_i) + \sum_{j \in J} EC(F_j) \cdot |Calls(F_i, F_j)| \tag{A.3}$$

where $Calls(F_i, F_j)$ is the number of times F_i calls F_j. Some explanation is needed for this formula. EC stands for evaluation complexity. $Size(F_i)$ is the size of program F_i. Note the recursive nature of EC. $Calls(F_i, F_j)$ is the number of times F_i calls F_j which is summed up.

In standard GP, if there are no subroutines defined, the SC and EC will be the same assuming that each function node takes 1 unit of time.

After the introduction of these two complexity measures, the authors introduce Rissanen's (1978) Minimum Description Length.[5] They advocate that MDL helps encode the complexity of the tree into fitness functions. In this part of the paper there is little detail on the nature and importance of

[5]Rosca and Ballard's original work did not cite Rissanen's MDL directly. It is included here for sake of completeness. They cite Rissenen's work on MDL by a way of Li and Vitnyi (2008)

MDL. There is an appendix at the end of the report where they discuss what is called *descriptional complexity* as it relates to hierarchical organization of the function set. Some background not found in the original report is needed on the idea of MDL before we can proceed to Rosca and Ballard use of MDL.

MDL is a principle that states that if we have some data set, there might be a way to compress the data so that no data is lost when uncompressed.[6] If data can be compressed, it is the description of that compression that is important. If there is any regularity in the data, we can exploit that regularity and come up with a short way to describe the data. That description could be the program used to do the compression and the compressed data resides in an encoded form readable by that program. Using an example similar to Grünwald's (2007), suppose we have a string of $100,000$, $1's$ and $0's$. And suppose it looks like the following in A.4.

$$1000010000 \cdots 1000010000 \tag{A.4}$$

A "C" like program fragment could be written to print out those data and then halt as in figure A.2. The "for" loop is itself turned into a pattern of $1's$ and $0's$ that can be run on a compiler. We chose "C" to express the data string, but it could have been written in any computer language and it would still get compiled or translated into a pattern of $1's$ and $0's$ that could be run on a computer. The length of $1's$ and $0's$ for that program fragment is the *minimum description length*.

```
for(int i = 0; i < 100000; i++)
{
 if(i % 5)
   printf("%d", 0);
 else
   printf("%d", 1);
}
```

Figure A.2: "C" code fragment to print out pattern of $1's$ and $0's$

What Rosca and Ballard are saying is that if the tree coding is chosen carefully and included in the fitness function, we can encourage parsimony as part of the evolution of an individual in GP. We now take a closer look at their use of MDL in AR.

The description complexity of a given binary tree T, is defined in formula A.5.[7] The first part of the formula is devoted to the number of bits needed to encode information about nodes in the tree. The second part of the formula is devoted to the number of bits needed to encode the overall fitness of the tree based on the number of fitness cases. Each of these parts will be described in turn.

$$DC(T) = \overbrace{n \cdot (\log_2 A + 2 \cdot \log_2 n)}^{\text{node encoding}} + \overbrace{Misses(T) \cdot \log_2 k}^{\text{fitness case encoding}} \tag{A.5}$$

[6] A detailed look at the ongoing active research in MDL can be found in Grünwald (2007).

[7] In the original paper, they use log which could be misinterpreted as $\log_{10}$. We are going to add the subscript to clarify that it is $\log_2$ and correct from an information theory view point.

For node encoding, T is a given tree. A is described in more detail below. n is the size of tree. k is the number of fitness cases. The tree can be encoded into an array of size n. Each element of the array is a node in T. That element, or node, has an encoded label from A and indices two children.

For fitness case encoding, $Misses(T)$ is the number of misses for a particular tree T. If there are a number of fitness cases used on a particular tree, the number of misses are tallied up. Here a *hit* is where the evaluation of test data matches the test outcome. A *miss* is the opposite; test data used with the tree do not match the test outcome. So, $\log_2 k$ is the number of bits needed to store the number of fitness cases.

A is defined in formula A.6. $\mathcal{F}$ is the set of functions. $\mathcal{T}$ is the set of terminals. m is defined as the highest number of functions that could be discovered.

$$A = \mathcal{F} + \mathcal{T} + m \tag{A.6}$$

There are other papers that explore MDL in a more indepth way during the time period of this research. In a separate paper, Rosca and Ballard (1994b), he cites Iba et al.'s (1994) MDL paper. That paper has more detail on how MDL could be used in GP. As a side note, MDL research in of itself, separate from GP, is very active area of research. See Grünwald's (2007) excellent comprehensive book on the subject.

One of the key insights of Rosca and Ballard's (1994a) work is in that placing restrictions on the height of BBs, a hierarchy of BBs make up the overall structure and solution to a particular problem. In this case they used the even 8 parity problem. In Figure 12 of their paper there is a nice graphic showing that this particular problem is made up of smaller parity problems. Lower in the tree are even representations 2 and 1. Higher up in the tree there calls to even representations 5 and 4. And part of their road to success is how they encoded the complexity of the tree into the fitness function for their framework.

Rosca and Ballard will emphasize this hierarchical nature of subroutines, HGP, in a number of papers, to be covered in the next section. This will lay the foundation for the section after that on ARL. All of this published work is summarized in his dissertation not covered here.Rosca (1997)[8]

A.3.2 Adaptive Representation - Hierarchical Genetic Programming - (HGP)

An outgrowth of AR by Rosca and Ballard can be seen in the following papers on HGP[9] As stated previously there is overlap of material in these papers, so we will pull out unique points that make up the HGP will be emphasized.

Rosca and Ballard (1994b) discuss how BBs self organize into hierarchies of functions through adapting representation of the search space. Three quarters of the paper is material derived from their previous work in Rosca and Ballard (1994c) and Rosca and Ballard (1994a). ADFs, MA, MDL

[8]His dissertation is essentially a collection of his published work.

[9]Here HGP is different than work by Banzhaf et al. (2000). They also use the term HGP. Their implementation is not based on AR and will be described in a section later in this chapter.

as well as the complexity measures described in the previous section were covered. The even parity problem for 3, 4 5 and 8 bits was used for their benchmark problems and reported in the paper. The claim that parity problems of up to order 11 were solved using their method.

There are a few items to point out in this paper that were not mentioned in the previous section. In numerous places in the paper they emphasize that a hierarcy of functions make GP more efficient and scalable. Second, they show the fitness function used. This fitness function, seen below, plays a dual role as the overall fitness function and as the block fitness function for the problem studied in this paper.

$$C_{standardized}(i) = [2^n - Hits(i)] \cdot C1 + Size(i) \cdot C2 \tag{A.7}$$

For the even parity problem of n input variables, there can be 2^n combinations of those input variables. All of those fitness combinations are fed into the boolean function for the even parity problem under consideration to create desired output. A hit for an individual i is considered a match between the input and output. The sum of individual hits is $Hits(i)$ in the above equation. $Size(i)$ is the size of program i. $C1$ and $C2$ are constants. The values of which are changed if they are evaluating the entire individual, as opposed to particular block. They make reference to their previous work on descriptional complexity as inspiration for this standardized fitness. They note that if $C1 = n$ it would account for misses, but not account for program complexity. No specific values are given for what $C1$ and $C2$ are for an entire GP run. They do state that $C2$ is set to 0 for building block fitness. If a variable is not used in a building block, that variable is set randomly set to a value. They reported that they developed a separate formula derived directly from MDL literature that did not perform as well. They speculated that their MDL formula might have aggressively pruned dead regions of code that might been a future source of genetic material. None the less, MDL could possibly be a source of inspiration for further research on bloat issues in GP research.

Alluded to in the previous section, Rosca and Ballard (1995) next investigate causality in HGP. Here causality is related to how structure and behavior change for an individual in GP. They state that many optimization problems use the POSC. It states that the cause of small changes to the structure of an individual will correspondingly cause small changes to the behavior of the individual. They use this as an analogy that if there are large changes in the structure there will be large changes in the behavior of the individual. Two explanations for causality are described. First, crossover can be a source of structural change in GP. Second, subroutines can amplify this change. That amplification can be either good or bad. It is interesting to note that POSC might be problem dependent. Rosca and Ballard (1995) cite as an example that a small change may alter an individual in a small way for boolean problems. However, for symbolic regression problems a small change could induce a large change in an individual. The authors do not explain why a small change for individual for a symbolic regression problem could lead to large changes for that individual. A possible reason for this is the non commutative nature of many operations found in many mathematical operations. For example, $4 - 3 \neq 3 - 4$. Non commutative operations could cause large abrupt fitness changes in an

individual.[10]

Rosca and Ballard (1995) track the number of generations where no new subroutines are discovered. This comes into play when a new subroutine is discovered; it signals the end of an epoch. When a new subroutine is discovered, a new epoch unfolds and triggers an extinction event in the population. A portion of individuals in the population are replaced with new individuals that use the extended function set.

The term,"pivot", is used in their introduction on how crossover works in SGP. Crossover, in SGP, is first performed by selecting two individuals from the population. For each individual, a random location is selected and then their subtrees are swapped. The locations of the swap are called *pivot* nodes. When a new subtree is introduced as part of crossover, that node is given a birth certificate.

The pivot location is also used as a justification of the "bottom up" approach. When crossover happens, the pivot locatation is given a birth certificate. The birth certificate has information on its originating parents. Using this information, Rosca and Ballard (1995) track the changing structure of how trees and the use of functions change over time. They noticed that the use of subroutines started to naturally form hierarchies when creating solutions for the boolean parity problem.

There are a few other items to note in this research. The authors note that AR functions once added to the library are not allowed to evolve further. Koza's (1994a) AAOs are described, which will be addressed later in this review. Rosca and Ballard (1995) mention that a future version of AR will have functions that can evolve like ADF GP.

In the third and final paper in this section, Rosca (1995b) discuss the use of information theory as a tool to help guide the search for creating and modifying functions in AR. Topics that are covered in this paper, include, approaches to search effort allocation, computational effort in GP, comparison of GP population dynamics to those of a physics based dynamical system, interpretations of entropy and information measures, discussion of population diversity in GA and GP, examination of their use of entropy as a measure for the boolean parity problem and pac–man problem and a discussion of their conclusions on entropy as a measure. Each of these will be discussed next.

Rosca (1995b) starts out by describing different approaches for search effort found in a number of fields in AI. *Exploitation* versus *exploration* is at the heart of this effort. Exploitation can mean that if we have found a promising solution we exploit it heavily even though it might not be the perfect solution. In exploration, we devote a lot of computing power searching for the perfect solution at the expense of devoting fewer computer resources to exploitation. This is a trade of how to allocate computer resources. The authors outline approaches from the AI fields of Genetic Algorithms, Reinforcement Learning, Combinatorial Optimization, Automatically Defined Functions, and Adaptive Representations.

First, from the field of GA the two armed bandit problem is described. It was designed by Holland (1975) as a problem to illustrate the trade off between exploitation versus exploration. In this problem, each arm represents a random variable. It is not known ahead of time which random vari-

[10]Research on this topic was done by Janikow and Aleshunas (2013).

able will payout the most. So how do we allocate a number of trials to find the highest payout? This problem is one of the foundational problems for the schema theorem and building block hypothesis that helps describe how GAs work.

Next Rosca (1995b) illustrate the exploration versus exploitation problem from the field of RL. One paragraph cites work of Watkins's (1989) thesis. The authors state that a main idea from the thesis is that an agent in an environment will change its current behavior based on future discounted rewards. If there is a need to search the algorithm will switch from exploitation to exploration.[11]

After this Rosca (1995b) talk about how CO attempt to efficiently search for solutions. In this type of optimization, there is a distinction between local and global search. Heuristics are used to guide the search from local neighborhood to local neighborhood.

Next, Rosca (1995b) present a few relevant ideas from Simulated Annealing. In SA there is focus on finding a global optimum by changing temperature parameters. The temperature parameter is an analogy that comes from the field of metallurgy. The authors indicate that SA has some nice advantages for finding a global optimum but is too slow to converge in practice.

Rosca (1995b) transition to pointing out that HGP should have an adaptive search policy balancing exploitation versus exploration. Citing previous work in Rosca and Ballard (1994c) and Rosca (1995c), argues that HGP needs to make *informed choices* when selecting subroutines. Crossover can drastically change an individual's fitness as shown by their work in Rosca and Ballard (1995).

Koza's (1994b) computational effort is also discussed by Rosca (1995b), which was covered in depth in the ADF part of this chapter.[12] They note that Koza's measure cannot be used in an adaptive search. The reason for this is that a parameter for the probability of success after the i^{th} generation is experimentally determined after a number of runs.

Rosca (1995b) also argues that natural section strongly linked to the concept of energy in a dynamical system. In this interpretation an individual is competing for energy in the system. The authors introduce some concepts from Ludwig Boltzmann where, micro state and macro state play a role in the statistical development of thermodynamics. Rosca (1995b)reason that there are micro and macro states that can be measured in the dynamic evolution of individuals in a population. The measures which can be observed are things like average fitness (global) and best of generation individual fitness (local). Mathematically they build a case for entropy of a dynamic physical system using the partition function:

$$Z = \sum_i e^{-\frac{H(i)}{T}} \tag{A.8}$$

Which Rosca (1995b) point out can be used in the Bolztmann Gibbs distribution:

$$Prob(i) = p_i = Z^{-1} e^{-\frac{H(i)}{T}} \tag{A.9}$$

[11] An interesting thing not mentioned this part of the description of different approaches can be found on p130 of Watkins (1989). In that part Watkins talks about hierarchical control policies. The overall control policy is optimal if all of the lower hierarchies of controls are also optimal

[12] Section 4.11 of Koza (1994b).

Z is a constant that helps turn the calculation into a distribution. T is a temperature.[13]

The above equation can be used in the equation for free energy which can be defined as:

$$F = -T \cdot \log Z \tag{A.10}$$

That equation is used in combination with the next formula to help derive the entropy equation.

$$F = \langle H \rangle - T \cdot S \tag{A.11}$$

$\langle H \rangle$ is the mean value of random variable H. S is the *entropy* of the system and can be calculated as:

$$S = -\sum_i p_i \cdot \log p_i \tag{A.12}$$

Free energy is the probability estimation of discovering the system in a certain set of states.

Rosca (1995b) use the entropy equation as a way to introduce Shannon's information theory. Entropy for information theory uses the same equation. Shannon (1948), cites the work of Bolztmann in statistical mechanics. In the original work, a discrete information source and recipient is modeled as a Markov process. Each step along the way is determined by a probability; it is a measure of choice. Or, to put it another way, it is a measure of uncertainty.

Rosca (1995b) note that several researchers have led efforts to link thermodynamics to biology. It is important to cite their work here, as this ties together thermodynamics, biology and information theroy. Schrödinger (1944) states that there is a paradox, namely that an increase in entropy causes a system to be less organized. On the other hand, if a system has less entropy, it is more organized. Wicken (1988), states that population *diversity* could make for a good measure. Johnson (1981) uses Shannon's entropy to measure diversity and points out that this is not a perfect analogy for diversity in a population.

Diversity is a theme for the next section of Rosca (1995b).

Rosca (1995b) state that diversity for a GA diversity limits early convergence, citing their previous work on evauational and structural complexity. Also highlighted is Koza's (1992) use of histograms for tracking the fitness of a population as it evolves.

Bringing this all together Rosca (1995b) state that entropy could be used a measure of diversity for a population. The following equation illustrates this.

$$E(P) = -\sum_k p_k \cdot \log p_k \tag{A.13}$$

p_k is a proportion of the population P that is divided into k classes. The division is done by behavior or phenotype. Population entropy is done by calculating the number of individuals that belong to each class. There was no discussion on how many classes there were or what consti-

[13] $H(i)$ wasn't defined in the paper, but it helps as a mathematical construct. Here i is a label or an index. It helps in differentiating, let's say, $H(x)$ from $H(y)$.

tuted any particular class. A deeper discussion by Rosca (1995b) in this area would have been very enlightening.

Two problems were used; the even 5 parity problem and the evolution of a controller for a version of Koza's (1992) pac man game.

Rosca (1995b) observed a number of overall patterns when using entropy as a measure of population diversity. First, there were plateaus or decreases in population entropy while the implementation was running. The authors state a possible cause for this is that a local search optimum was found. Second, entropy decreases were related to decresses in population diversity, but population fitness did not necessarily decrease. Thirdly, if there was an average fitness improvement for the population, it might be atttributed to more fit individuals in the population. Fourth, comparing entropy and average fitness suggests when computational effort might be wasted.

Rosca (1995b) conclude that entropy is a good measure for to discovering new functions. Since crossover potentially disrupts the fitness of an individual adaptive techniques could help GP does not waste search effort.

In summary, for HGP, Rosca's and colleagues' papers serve as a spring board for how learning is achieved in ARL. This topic is covered in the next section.

A.3.3 Adaptive Representation with Learning - (ARL)

ARL first appears in Rosca (1995a).[14] Much of the paper is a compilation of previous results in Rosca and Ballard (1994a), Rosca and Ballard (1994b), Rosca and Ballard (1994c), Rosca and Ballard (1995) and Rosca (1995b). However, he extends previous work in several ways. First, ARL can delete subroutines. In previous work subroutines were not deleted. And second, he introduces a typed representation for functions based inspired by the work of Montana (1995). He is keeping track of the function signatures and typing for function argument and function return. For this work, he uses a version of Koza's (1992) pac man problem. In this implementation, there are multiple types that help in keeping track of the function signature, including boolean types for "if" conditionals and boolean types for relation operations like greater than or less than. Also, there are types related to distance as it relates to food distance. He reported favorable results and reported that ARL does not need specific block fitness functions as reported previously.

We now move onto other researchers' work that compares a number of heuristics added to ARL.

A.3.4 Modified ARL - (ADGP)

Dessi et al. (1999) refute many of the claims within Rosca's work in ARL. Based on a master's thesis by Dessi (1998) which combines and compares most of Rosca's some of Koza's previous work into one framework. Through this framework the authors offer an extension to the ARL framework. The results of this effort were not very encouraging for the ARL framework, but there are many positive

[14]This work also appears in book chapter form in, Rosca and Ballard (1996)

benefits that were an out growth of this effort. In the following paragraphs we will take a look at their research and offer some possible explanations for their results.

Neither Dessi et al. (1999) or Dessi (1998) offer an official moniker for their work. However, in the readme file for the source code to the author's work, the acronym ADGP appears. There is no mention in the source code documentation or the thesis on what this acronym means. They were studying ARL, which about "adaptive representations", so for the purposes of this section of the review, we will take ADGP to mean Adaptive Genetic Programming.

In the introduction section of the paper, Dessi et al. (1999) introduce the three main subroutine discovery mechanisms of the time. These mechanisms are Koza's ADF and AAO framework, Angeline and Pollack's MA mechanism[15] and Rosca's ARL framework. These approaches were given names that are respectively called *evolutionary selection*, *random selection* and *heuristic selection*. This classification naming becomes important when they are comparing different approaches later in their research effort.

In *evolutionary selection* classification, ADF and AAO, subroutines co–evolve with the main routine. Dessi et al. (1999) cite a few drawbacks for this kind of classification, including that subroutines as they evolve require synchronization with the main body of code. Also another problem under this framework is the lack of flexibility of genetic operators. This concern is unclear and is not discussed further.

For *random selection*, MA code is randomly chosen and frozen and stored in a library for future use. The authors indicate that this type of selection is not efficient. It is unclear however, what they meant because they cited the wrong researcher; Rosca's work was cited when they should have cited Angeline and Pollack's work.

And in *heuristic selection*, ARL which is the focus of this research effort, uses heuristics to find pieces of code. They state that ARL is superior because it does not require new genetic operators to be created and that it can be viewed as a "general framework" because of its ability to find "building blocks". This is not an entirely accurate statement. Koza (1992) first introduced the concept of building blocks. Also, one could view this as a definition problem. We have already seen many researchers' changing definitions what are subroutines and building blocks.

Even with these problems, it is still useful to have this classification for future use.

After the introductory statements, Dessi et al. (1999) move onto a analysis of ARL. The analysis for this part of the dissertation is part comment and part description of their framework. This section is divided up into 5 sections: when to create subroutines; selection of useful building blocks; when to use or create function arguments; diffusion of subroutines; deletion of subroutines. Each will be examined in turn. There is some introduction of new methods and heuristics not found in previous researchers work and worth attention.

In the section on when to create new routines in ARL Dessi et al. (1999) propose changing a population according to population diversity. This was explored earlier in Rosca's (1995b) own work. Dessi et al. (1999) point out the complexities of not knowing ahead of time how many classes

[15] In their paper they refer to MA as "GLib" approach. We're going to stick with MA for consistency in this review.

to partition the fitness values. The authors also cite that entropy as a measure is noisy and it is hard to determine when a population is stuck at a local minimum. The authors advocate a strategy called *maxfit* to replace entropy as a measure of population stagnation. In this strategy, the fitness of the best individual is tracked. Using that measure instead of entropy could be used as an indicator of when to add new subroutines.

For the section on selecting useful blocks, the authors review methods covered in Rosca's previous work. In addition to these, they discuss two additional methods for finding building blocks. In the first method, they cite Tackett's (1994) use of average fitness of a category of blocks to help match individuals. This method serves as a schema for the block. A potential drawback with this approach is linking a large number of individuals to a particular category. In the second method, they cite from Iba and de Garis (1996) use of a statistical correlation between the computed return value of a block versus the overall return value of the program as a whole. They state that this method is not attractive because building blocks tend to look like the overall program.

At the end of the section on selecting useful blocks, Dessi et al. (1999) introduce three new heuristic selection methods collectively called *saliency*. The idea behind saliency is identifying blocks that actively contribute to the success of the overall individual. This is accomplished by taking the value that is returned by a potential and altering it by a small amount. If it has a big impact on the fitness of the individual, then that block is probably making a contribution to the overall individual. If there is no change in the individuals fitness after a small change, then the block is likely not having an impact on the overall fitness and should not be used.[16]

Three different heuristic calculations are explored for saliency. Using their terminology, the first one is called *SalOut* and is defined by the equation.

$$SalOut = \frac{1}{N} \cdot \sum_{i}^{N} \begin{cases} 0 & \text{if out(i) == out'(i)} \\ 1 & \text{otherwise} \end{cases} \tag{A.14}$$

N is the number of fitness cases. *out* is the output of the non modified building block return value. *out′* is the output of the modified build block value. This calculation is calculating percentage of the fitness cases where a change made an impact.

The second one is called *SalFit* and is defined by the equation.

$$SalFit = ABS(f - f') \tag{A.15}$$

ABS is the absolute value of fitness for unmodified block value return, f and the fitness of the modified block value return, f'.

The third one is called *Sal*, which is a combination of the previous two equations and is defined by the equation:

[16] Since one of their benchmark problems involves symbolic regression, this approach will suffer the same arithmetic problem as seen in non commutative operations

$$Sal = SalOut \cdot SalFit \tag{A.16}$$

Dessi et al. (1999) raise doubt about the utility of function arguments in ARL. Taking a piece of code and replacing the leaves of the piece of code with variables changes the semantics of that piece of code. Even though the authors do not cite experimental results or give any examples for this doubt there could be an intuitive explanation. If the GP system does not have a typing mechanism for functions, a function that worked well in one setting may not work well in another situation. Montana's (1995) work in strong typing is an example where a function would be generalized to work with only vector or matrix calculations. Others have worked in this area and show utility of typing for functions. We will take a deeper look into this in section 3.3.

Diffusion of subroutines is the topic of the next section. Here instead of subroutines passively being introduced, they are actively placed into the population. Dessi et al. (1999) reason that if a good block happens to be in a low fitness individual, that block might not survive when it should survive. They further reason that the longer the individual is in the population, lower fit individuals that contain the good block will be starved out. A good block has to be used early in the reproductive process rather than later. They propose a solution for this problem with a new kind of mutation operator. A set of random individuals is chosen and a branch is chosen and replaced with the newly discovered subroutine.

In the last part of this section of the paper, deletion of subroutines is discussed. Citing their own research efforts, deletion of subroutines entails tracking which programs each program belongs. The tracking and deleting subroutines is wasteful for computer resources. Dessi et al. (1999) state that if one found a useful heuristic to find useful subroutines it would be a duplication of effort and one wouldn't need to go to the extra effort of having a heuristic to delete a subroutine. They state that if a heuristic is not reliable that a random selection should occur with fitness monitoring. From a review standpoint this is a concern. In original ARL routines are deleted to help with computer resources; deletion of unused or ineffective subroutines intuitively would save the overhead involved of repeatedly evaluating subroutines that would accumulate in the library.

The next major section in the paper is labeled as experimental analysis. The results, however, are not published in Dessi et al. (1999) and can be found in Dessi (1998).

Three benchmark problems were chosen for experiments: 6 bit multiplexer; symbolic regression; and sorting a vector of numbers. Dessi (1998) provides more details on the last two problems. The following function was used for the symbolic regression problem.

$$f(x, y) = 2x^2 - 3y^2 + 5xy - 7x + 11y - 13 \tag{A.17}$$

The range of the function $f(x, y)$ from -4 to 4 for both the x and y values.

The goal of the sorting problem was to sort, in global memory, a list of numbers. Operators like swapping memory locations, FOR loops and conditionals were used.

For their efforts, the MA approach yielded better results than ARL.

There are a few comments that can be made about the methodology. First, it would have been optimal to implement the parity problem to see if they could reproduce the results of research they were critiquing. Second, for the symbolic regression problem, they had only the number 1 as a numeric terminal. Since their problem has constants, their implementation would have to evolve many combinations of fundamental arithmetic operations with the number 1 to arrive at those constants. Perhaps a better way would be to add the constants $0 \cdots 9$ to cut down on the search space size. And lastly, the fixed epochs are very short in their implementation, because they were using max tree size of 2 to 4 for subroutine discovery, perhaps it would have performed better if given more of a chance.

A.4 Automatically Defined Macros - (ADM)

This section takes a look at a method used by Spector (1995) that is similar to an ADF or the expansion operation of MA. We will focus on how he used ADM in GP. His goal was to explore how and when to use macros in GP. First we will take a look at the concept of macros in a high level language and how they apply this to GP. Second, we will take a look at their bench mark problem. And last, results will be discussed.

Spector (1995) introduces us to the function and macro concept. Many computer languages like C and LISP have a mechanism to expand code at a particular location in a program.[17] This is different than a function call. It is a special mechanism in the computer language itself where a name and a possible argument or arguments are shorthand for a longer piece of code. When invoked, the entire body of the macro is copied to the call invocation site. In the macro body, a variable is actually substituted by the call parameter. If a macro is called twice, code is copied to each call site. The overall program would grow by a small amount. This is different than a function call. For a function, control is passed to the body of the function. That body will reside somewhere else. When the function has completed, control will be passed back to the call site. Various other mechanisms are used to help shuttle back and forth data to the function body.

Leading up to the implementation of ADM, Spector (1995) discusses how they might be used. One of the benefits of macros is that they can implement control structures. Control structures could be code fragments that are evaluated later.[18] This type of behavior could be exploited if there are side effects in the program.[19]

For the implementation, Spector (1995) chose not to mix usage of ADFs and ADMs. Each paradigm ran on its own for benchmark purposes.

Two benchmark problems were used: the lawnmower problem as defined in chapter 8 of Koza

[17]Detailed information on this topic can be found in Kernighan (1988) and in Steele (1990). They draw a distinction between C and LISP. In C macro is text substitution and preprocessed before it is sent for code generation. In lisp the code is transformed into new code. It is called macro expansion.

[18]They refer to Graham (1994) for further details. In chapter 8 of that book they state that macros, in LISP, "are more like instructions to the compiler". See p110 of that book.

[19]Side effects are global variables that can cause the body of the program or function to behave differently. Something that does not have side effects would be local variables or pass–by–value arguments.

(1994b); a variant of the Obstacle Avoiding Robot found in chapter 13 of the same book. For the lawnmower problem, he reported negative results in which ADFs performed better. For the OAR problem, ADMs preformed better than ADFs.

Spector (1995) speculates on when to use an ADM. If all of the operators in the function set are in the functional programming paradigm, there will be not difference between use of ADFs and ADMs. However, if side effects can be exploited and are part of the solution, then ADMs may help. A downside discussed is that ADMs may incur additional computational effort. The reason for this is the extra computational effort of reevaluating and regenerating redundant code fragments.

A.5 Hierarchical Genetic Programming using Local Modules - (HLDM)

Banzhaf et al. (2000) explore hierarchies of locally defined modules. In the introduction section of paper, they stress that unlike other methods, their method emphasizes the context of the call. We will see what is meant by this later in this section.

In the second section, they introduce us to why subroutines are important to GP.[20] A major problem with GP is that GP has a hard time to scale up. Complexity of real world problems have a hard time translating to current GP frameworks. An analogy is drawn between how programs are created from a programmer in a hierarchical nature. Complex tasks are modularized and structural hierarchies are formed by these tasks.

Previous methods are presented, including Koza's (1994b) on ADF, Angeline and Pollack's (1992) work on MA and Rosca and Ballard's (1994c) work on ARL. Observations are made on each of these methods.

First, Banzhaf et al. (2000) state that the fixed structure of an ADF is a "mixed blessing". A downside is that this approach forces the user to specify how many ADFs are needed ahead of time. An upside is that the user could code existing domain specific knowledge into the representation to encourage a positive outcome. There is mention in the paper of Koza's (1994a) AAOs.

For MA, reference is made to Kinnear, Jr.'s (1994) work, commenting that Kinnear, Jr. bases his results on only one benchmark problem. More work needs to be done to see if efficiency can be improved.

When discussing the improvements of the ARL approach, the authors note that, as seen in earlier in Section A.3, ARL works with epochs. Also, evolution speed in ARL is improved, but it is unclear how ARL helps improve an individual in GP.

Banzhaf et al.'s (2000) hierarchical genetic programming method is described next.[21]

HLDM has a different architecture for generating and using modules. As with standard GP, a population is made up of individuals, but this is where it diverges from previous methods. The

[20] They actually use the term modularity instead of subroutines. We're going to use subroutine to keep terminology consistent for purposes of this review.

[21] In their paper they refer to their hierarchical method as hGP. Since Rosca and Ballard (1994b) explored HGP in an earlier version of ARL, we're going to use HLDM to denote the version of HGP found in Banzhaf et al. (2000). This will help in keeping acronyms and concepts straight with regard to hierarchies and GP.

nodes at each level belong to a hierarchy of actions each with their own level population. Functions are evolved on the higher levels. The higher levels all consist of primitive functions, discovered modules or variables. In addition, higher levels discover new modules. The lowest level consist of primitive functions and terminals. Each module is local to its call site. Overall, the population of individuals converge. Each of the local modules converge in a similar fashion. Each individual in the population has to discover good modules for itself. Crossover only happens at among hierarchy levels. A representation of HLDM can be seen in Figure A.3.

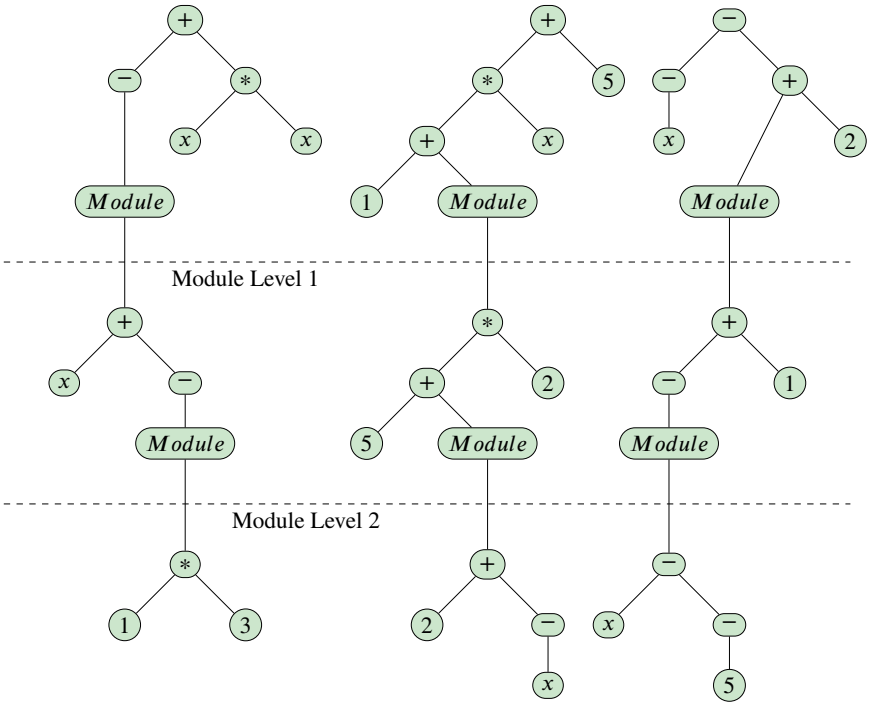


Figure A.3: HLDM Example

The speed of evolution is altered at each hierarchy level. Higher levels are allowed to evolve more rapidly than the lower levels. This gives the lowest levels time to find solutions used in higher levels. Similar to Rosca and Ballard's (1996) ARL, new subroutines are found through differential fitness. When good modules are discovered, they would not change as much. A good module is discovered by changing a module with a neutral module and comparing it to the localized fitness. So, if an individual has a large jump in fitness by swapping out a module for a neurtal module, the that individual may have modules and those modules are selected for futher use. A performance metric for time is used on the population as a whole, which is a count of the number of nodes evaluated.

Banzhaf et al. (2000) explore two versions of HLDM.[22] There are a number of restrictions on both modules, including that there is only one module level and an individual can only make 1 call to a module. Also, mutation is only allowed on the highest level. If we were to take a look at figure

[22]We will modify the acronyms here as not to cause confusion with other frameworks that use the same letters. In the original paper the first framework was referred to as *hGPminor*. We will call this one HLDM_minor. The second framework was referred to as *hGP*. For that one we will use HLDM.

A.3, only the first and last level were present for their tests. Evolution is not allowed to occur on the top most level for HLDM_minor. And for HLDM, evolution is allowed to occur on the top level. In addition a few modifications of HLDM were explored. In the first one, crossover was implemented on the module level by selecting a bad tree and replacing it randomly with another subtree. In the second one, evolution speed was set per level. Unfortunately, in the paper the evolution speed for the last modification was truncated due to formatting issues. They reported negative results for the first version.

For their tests, they used 4 symbolic regression problems and 2 even parity problems. A summary of these can be seen in figure A.4. HLDM_minor outperformed HLDM. Both outperformed SGP. It was unclear whether SGP was set up to run ADFs or not. If ADFs were used there would have been a description of how many ADFs were used along with how many parameters were used.

Problem	Type	
1	continuous	randomly selected values
2	continuous	steps
3	continuous	$x^6 - 4x^5 - 3x^4 + 4x^3 - 2x^2 - x + 4$
4	continuous	$\frac{x^3-x^2-x+3}{x+\frac{5}{9}}$
5	discrete	even–5–parity
6	discrete	even–7–parity

Figure A.4: hGP Test Problems

A.6 Architectural Altering Operations - (AAO)

In his third book, Koza et al. (1999) expands on earlier work by investigating how the architecture of a program can change and evolve. In this earlier work, a program would be represented, without ADFs, as seen previously in 1.1. With the introduction of ADFs in the second book, the architecture of a GP is changed. While it is being evaluated, a GP run make calls from the RPB to the ADF for evaluation and return to the RPB where the ADF was called. This was seen earlier in this chapter in figure 3.2. With this we are seeing the architecture of a particular GP beginning to take shape. In his third book Koza investigates how the architecture of a GP can change and evolve. In this work, ADFs play an ensemble role with other architectural operations to help evolve solutions to problems. The other members of this ensemble, in addition to ADSs, are ADIs ADLs ADRs and ADISs. These will be investigated, in the next few sections.

A.6.1 Automatically Defined Subroutines - (ADS)

As stated in Koza (1994b) where scalability is a focus, the parameters for an ADF and the number of ADFs are fixed before an experiment. However in Koza et al. (1999), this requirement is relaxed.[23] There are two parts to this relaxation. In the first part, the entire ADF can be created *or* deleted *or* duplicated. In the second part, parameters for the ADF itself can be created *or* deleted *or* duplicated.

During a run, a GP's evolution takes place across generations. Evolutionary pressures, may emphasize the need for particular ADFs to form. Individual ADFs and the parameters for those ADFs will either be useful and survive or parish and need to be removed. Specifics for an ADF are as follows:

1. *ADF creation* – When a new individual is created information from a preexisting individual's ADF, an additional new ADF with some of that additional ADF information will be created.
2. *ADF duplication* – When a new individual is created if there is one or more ADFs, one will be chosen and duplicated.
3. *ADF deletion* – When a new individual is created if there is one or more ADFs, one will be chosen and deleted.

And, specifics for ADF parameters are as follows:

1. *ADF parameter creation* – When a new individual is created that has an ADF with a number of arguments, an additional argument is created and references to that ADF are updated with that new argument.
2. *ADF parameter duplication* – When a new individual is created that has a reference to an ADF, an argument is chosen and duplicated. References to that ADF are updated.
3. *ADF paramter deletion* – When a new individual is created that has a reference to an ADF with arguments, an argument is deleted and references to that ADF are updated.

For purposes of this dissertation, these dynamic procedures are not investigated, but are considered for future work. For this dissertation, ADFs and the parameters for those ADFs are fixed ahead of a GP run.

The next three sections contain AAOs that concern repeated behavior. When ADF is called multiple times, the act of calling the ADF could be considered repeated behavior for the body of that function. The same could be said for *looping* operations. There is a beginning of the loop body, code is evaluated and at the end of the loop body control is returned to a location where the rest of the program is continued.

[23] See Koza et al. (1999), chapter 5

A.6.2 Automatically Defined Iterations - (ADI)

By way of introduction, an ADI and an ADL are very similar.[24] An ADI has the following characteristics: an initialization procedure; a termination condition; an update of a loop control parameter and the body of the loop itself. A ”for” loop could be expressed in a high level language ”C” as seen in figure A.5.

```
//    |------ loop initialization
//    |       |------ loop termination condition
//    |       |       | ------ loop control variable
for(i = 0; i < 5; i++)
{ // beginning of loop body

  sum = sum + 1;

} // end of loop body
```

Figure A.5: For loop example for ADI

For Koza’s GP, ADI are similar to ADFs called multiple times. For this purpose, the ADF is replaced with what is the body of the ADI which is its own branch. There is a IPB where a control variable is updated with new iteration values. The variable is shared with the ADI branch.

A.6.3 Automatically Defined Loops - (ADL)

Koza et al. (1999) take the ADI and make a more generalized version called an ADL. There are four branches that make up an ADL. First is the LIB; operations that initialize the loop are evolved in this branch. Second is the LCB; loop termination code is evolved for this branch. And third is the LBB; this would be the body of the loop. Like an ADF, this would be where code would be evolved that is to be repeatedly run a number of times. And fourth is the LUB; in this branch, any variables that are used by the LCB are updated to help control the number of repetitions of the LBB.

Using the ”C” example from above in A.5, each operation for a loop becomes its own branch. This can be visualized as follows in figure A.6.[25]

Each branch in the ADL can be allowed to evolve it’s own code. However, as Koza comments, a restricted form of the loop is used due to computation resources available at the time.

A.6.4 Automatically Defined Recursion - (ADR)

Recursion is a very similar looping construct to what has been seen with ”for” loops. In recursion, a function is defined and it is allowed to call itself repeatedly until termination condition is met, where

[24] Koza:gp3 Chapter 6 and Chapter 7 respectively.

[25] This figure is essentially the same as p136 Koza et al. (1999). There aren’t too many ways to rewrite a ”for” loop in ”C”.

```
int i = 0; // global var
float sum = 0;
int LIB(void)
{ return i = 0;
}

int LCB(void)
{ return i < 5;
}

int LUB(void)
{ i++;
}

float LBB()
{ // beginning of loop body
  sum = sum + 1;
} // end of loop body

float ADL(void)
{
  float val = 0;
  //    |------ loop initialization
  //    |      |------ loop termination condition
  //    |      |      | ------ loop control variable
  for(LIB(); LCB(); LUB())
  {
    LBB();
  }

  return val;
}
```

Figure A.6: For loop example for ADL

it returns the result of each of the calls to itself. These are called Automatically Defined Recursion in Koza's GP. In "C" recursion can be seen in figure A.7.[26]

Similar to ADL, each portion of an ADR has its own branch. There are four branches that make up an ADR. First is the RCB; this branch returns a value if recursion is to be continued. Second is the RBB; if the RCB returns a positive condition, then the RBB is run. The RBB is the body of the ADR. Third is the RUB; this branch is run after the body of the RBB is run. Any state information governing recursion is updated in this branch. And fourth is the RGB; this branch is run only one time, and could be considered the base case for recursion. A "C" language version of an ADR can be seen in figure A.8.[27]

[26] Koza book 3, chapter 8.

[27] This is similar to Koza's "C" version on Koza et al. (1999) p148. There aren't too many ways to write this differently in "C".

```
int sum = 0;

int fun(int i)
{ if(i == 0)
    return 1;
  fun(i-1);
  return sum = sum + 1;
}

int main()
{ int i = 5;
  printf("%d\n", fun(i));
}
```

Figure A.7: Recursion example for ADR in ”C”

```
float ADR0(float ARG0)
{ float val;
  if(RCB(ARG0) > 0)
  { val = RBB(ARG0);
    RUB(ARG0)
  } else {
    val = RGB(ARG0);
  }
  return val;
}
```

Figure A.8: Example of an ADR in ”C”

A.6.5 Summary of Architectural Altering Operations

A final part of Koza et al.’s (1999) AAO is mentioned for sake of completeness is Automatically Defined Internal Storage. We have seen in previous sections, memory variables being used in the ”C” like code. For ADISs, evolved code could make use of temporary variables to help achieve some goal. In ADIS a value can be read or written to a dynamically created memory location during program evolution. An example of this might be an index variable used when accessing an array of memory locations.[28]

For all of the AAOs, each of ADIs ADLs, ADRs and ADISs can be dynamically created, deleted or duplicated during a GP run.

Koza does not place restrictions, design wise, on whether one AAO can call another. An ADS can call an ADI an ADL an ADR or ADIS and vis–a–versa. One might place restrictions ahead of time if warranted by the problem’s computational resources and computational time to complete.

A variety of problems investigating AAOs are used to illustrate the power of AAOs. These include a number of problems from the boolean, robotic control, transmembrane segment identification and electrical circuit design domains.

[28] Koza et al. (1999), Chapter 9.

Appendix B

Computer Environment for Experiments

All experiments were conducted on a Dell Alienware 15 R3. Details of the environment are listed in Table B.1. Execution times reported in Chapter 5 is based on this hardware and operating system setup.

CPU	Intel 8 Core i7-7700HQ CPU @ 2.80GHz
Operating System	Ubuntu 16.04
RAM	16GB
Disk Type	Samsung 970 EVO+ 1TB V-NAND M.2 2280 PCIe NVMe 3.0
Compiler	gcc
Compiler Flags	CFLAGS = -O2 -fsanitize=bounds -fsanitize=undefined

Table B.1: Experiment Environment

As described in the Chapters 5 and 6 the following frameworks were compared; original SGP lilgp1.02, CGP2.1, CGPF2.1, ACGP1.1.2 and ACGPF2.1.

Since we are investigating scalability issues for all frameworks a few modifications were needed to all of the frameworks for correct display and capture of run statistics. Original lilgp1.02 and all derivative frameworks were modified in a minor way to handle correct generation of memory statistics and run statistics. The lilgp framework has a facility for tacking how much memory was allocated and freed. The data type for that was an 64 bit integer which was the "C" "int" data type. Code that contained any counters on memory allocations was changed from an "int" to a "long" to ensure proper display of those statistics. In addition other counters were changed from an "int" to a long" data type for all lilgp popstats counters to ensure proper display. These were changed to a bigger data type because there were integer overflows for non ADF problems. Changing these variables from an "int" to a "long", got rid of these overflows. The new version of lilgp is called lilgp1.03. Changes to CGP2.1 and ACGP1.1.2 are available to those original framework authors.

Because of the setup requirements of each of the frameworks taking into account all of the run-time and compile time attributes required, a code generation tool, gnu autogen, was used to help generate correct "C" code for compilation and run time parameters files for each one of the experiments described later in this chapter. In addition the gnu parallel tool was used to make full utilization of all 8 cores of the Intel cpu.

Code was compiled using gnu's gcc compiler with the compiler flags set at the O2 level. To ensure proper dynamic memory handling the ubsan instrumentation library was used. In particular there were run time checks on out of bounds memory access and checks on undefined behavior for an "C" code compiled.