Algorithmic Simulation in System Design and Innovation

by

Timothy Harsh

Bachelor of Science in Business Administration, Bryant University, 2001

Submitted to the System Design and Management Program
in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Engineering and Management

at the

Massachusetts Institute of Technology

February 2011

© 2011 Timothy Harsh
All rights reserved

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Signature of Author ______________________________

Timothy Harsh
System Design and Management Program
December 16, 2010

Certified by ______________________________

Christopher L. Magee
Thesis Supervisor
Professor of the Practice
Engineering Systems Division

Accepted by ______________________________

Patrick Hale
Director
System Design and Management Program

[This Page Intentionally Left Blank]

Algorithmic Simulation in System Design and Innovation

by

Timothy Harsh

Submitted to the System Design and Management Program
in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Engineering and Management

Abstract

This thesis explores the use of genetic programming as a tool in the system design and innovation process. Digital circuits are used as a proxy for complex technological designs. Circuit construction is simulated through a computer algorithm which assembles circuit designs in an attempt to reach specified design goals. Complex designs can be obtained by repeatedly combining simpler components, often called building blocks, which were created earlier in the algorithm's progression. This process is arguably a reflection of the traditional development path of systems engineering and technological innovation.

The choice of algorithm used to guide this process is crucial. This thesis considers two general types of algorithms—a blind random search method, and a genetic programming search method—with variations applied to each. The research focused on comparing these algorithms in regard to: 1) the successful creation of multiple complex designs; 2) resources utilized in achieving a design of a given complexity; and 3) the inferred time dependence of technological improvement resulting from the process. Also of interest was whether these algorithms would exhibit exponential rates of improvement of the virtual technologies being created, as is seen in real-world innovation. The starting point was the hypothesis that the genetic programming approach might be superior to the random search method.

The results found however that the genetic programming algorithm did not outperform the blind random search algorithm, and in fact failed to produce the desired circuit design goals. This unexpected outcome is believed to result from the structure of the circuit design process, and from certain shortcomings in the genetic programming algorithm used.

This work also examines the relationship of issues and considerations (such as cost, complexity, performance, and efficiency) faced in these virtual design realms to managerial strategy and how insights from these experiments might be applied to real-world engineering and design challenges. Algorithmic simulation approaches, including genetic programming, are found to be powerful tools, having demonstrated impressive performance in bounded domains. However, their utility to systems engineering processes remains unproven. Therefore, use of these algorithmic tools and their integration into the human creative process is discussed as a challenge and an area needing further research.

Thesis Supervisor: Christopher L. Magee
Title: Professor of the Practice of Mechanical Engineering and Engineering Systems

[This Page Intentionally Left Blank]

Acknowledgements

I would first like to offer my sincere thanks to Wolfgang Polak and W. Brian Arthur for sharing the source code of their simulation model with me, as well as for offering their time and energy in providing instruction and answering my questions. This model formed a central element of the research in this thesis, and this product simply would not have been possible in its current form without their assistance. I hope that my characterizations of this model and their earlier papers—a pivotal part of the foundations for this thesis—do justice to the important insights gained from both. Any errors or misunderstandings are of course my own.

I also owe a great debt of gratitude to Dr. Chris Magee for his patience, understanding, valuable time, and generous help in guiding me through this process as my advisor. As work on this pursuit over the past many months progressed through a meandering maze of ups and downs, jubilations and disappointments, and many roadblocks, his wisdom and guidance provided the means to finally succeed. The thesis work has been challenging—at times fun, and at other times frustrating—and in the end, a highly rewarding endeavor.

Finally, I would also like to thank my family, friends, and colleagues, and especially, my parents, for their understanding, support, and generous patience over the past two years as I completed the SDM program. It has been a profound experience.

[This Page Intentionally Left Blank]

Table of Contents

Abstract **3**

Acknowledgements **5**

List of Figures and Tables **9**

Chapter 1: Introduction **11**
- 1.1: Background and Motivation 11
- 1.2: Summary of Technical Approach 12
- 1.3: Thesis Overview 12

Chapter 2: Literature Review **13**
- 2.1: Evolutionary Computation 13
- 2.2: Genetic Algorithms 14
- 2.3: Genetic Programming 17
- 2.4: Innovation and Design 23
- 2.5: Digital Circuit Construction 25
- 2.6: Engineering Process Models 26

Chapter 3: Methodology **29**
- 3.1: Research Approach 29
- 3.2: Algorithms 30
  - 3.2.1: Blind Random Search 31
  - 3.2.2: Genetic Programming 32

Chapter 4: Results **35**
- 4.1: Overview 35
- 4.2: Blind Random Search 35
- 4.3: Genetic Programming 51

Chapter 5: Conclusion 57
5.1: Future Work 57
5.2: Conclusion 58
5.3: Challenges 59

Bibliography 61

List of Figures and Tables

Figure 2.1: Graphical Representation of Algorithmic Methods and Relationships **14**

Figure 2.2: Illustration of a Binary Format Chromosome with Three Genes **15**

Figure 2.3: Illustration of a Genetic Programming Tree Chromosome with Five Terminals and Four Functions **20**

Table 3.1: List of Technology Goals for Digital Circuit Construction **30**

Figure 4.1: Results of a Typical Run of Arthur and Polak Random Model **36**

Figure 4.2: Trial 1 – Typical Run of Arthur and Polak Random Model with Event-Based Performance Intervals **38**

Figure 4.3: Trial 2 – Typical Run of Arthur and Polak Random Model with Event-Based Performance Intervals **39**

Figure 4.4: Trial 3 – Typical Run of Arthur and Polak Random Model with Uniform Performance Intervals **40**

Figure 4.5: Trial 4 – Typical Run of Arthur and Polak Random Model with Maximum Concurrent Working Goals Constraint Removed **42**

Figure 4.6: Trial 5 – Typical Run of Arthur and Polak Random Model with Frequent Removal of Unused Technologies **43**

Figure 4.7: Trial 6 – Typical Run of Arthur and Polak Random Model Using Only Prior Solved Goals as Available Technology Library **44**

Figure 4.8: Trial 7 – Typical Run of Arthur and Polak Random Model Using Doubled "Max Complexity" Parameter... **47**

Figure 4.9: Trial 8 – Typical Run of Arthur and Polak Random Model Using "Max Complexity" Parameter = 100 .... **48**

Figure 4.10: Trial 3 Data Scaled by Circuit Output Size **49**

Figure 4.11: Trial 3 Data Scaled by 2^N * Circuit Output Size... **50**

Figure 4.12: Four Runs of the Genetic Programming Model with Varying Fitness Evaluation Parameters **52**

[This Page Intentionally Left Blank]

Chapter 1: Introduction

This chapter introduces the problem to be examined in this thesis and the rationale for doing so. It also provides a very brief summary of the technical research approach to be pursued, followed by a structural overview of the content contained in this work.

1.1: Background and Motivation

Using modern computational power to run simulations as a means of creating or optimizing things of interest, particularly when the methods used are inspired by nature, is a subject of fascination for many researchers. One such method is known as genetic programming, an extension of the somewhat more widely-known genetic algorithm, which is a technique inspired by natural processes. This tool goes far beyond more traditional optimization methods, which merely make adjustments to an already-specified design framework, in that it adds the power of creativity to the process. In other words, this tool can potentially not only optimize a design, but also formulate initial and improved designs from very minimal initial specifications. Such a tool offers a powerful new potential to enhance traditional engineering processes if it can be successfully applied and harnessed to full effect on real-world problems of noteworthy concern.

Genetic programming has shown impressive results in some limited design contexts, such as in designing analog circuits and radio antennas. However, this progress to date is still limited to the technical aspects of the problem at hand and the algorithm itself. While this is a promising first step, the art of real-world engineering and design is far more complicated, particularly in that it necessitates human involvement. Thus, systems engineering, innovation, and design are not just facets of a technical discipline, but rather are socio-technical processes. From this, it follows that in order to fully harness the true power and potential of tools such as genetic programming, not only the technical but also the managerial and human components of interest must be integrated into a cohesive framework to support future progress.

This thesis seeks to take an initial, modest attempt at studying these two connected elements.

1.2: Summary of Technical Approach

Research work in this thesis first attempts to build a foundation upon earlier related work by Arthur and Polak as a starting point and then extends focus toward genetic programming issues and challenges. The algorithmic simulation model from that earlier work is reused to generate initial results. Then, a number of experiments and variations are performed using that model to gather insights into the process at large. Finally, a customized genetic programming model is used to produce solutions to the same design problem for comparison.

1.3: Thesis Overview

Chapter 2 provides a review of pertinent literature and develops a background understanding of the concepts and technologies covered in this thesis. The general class of algorithmic simulation approaches is first introduced, followed by an increasing specificity in discussing evolutionary computation methods, then genetic algorithms, and finally genetic programming, a tool used for part of the research in this thesis. Due to its importance in this work, a survey of certain techniques and issues relevant to genetic programming is also provided. Finally, innovation and design elements, digital circuit technologies, and engineering process models, including elements interrelated to social and managerial issues, are also briefly discussed.

Chapter 3 outlines the methodology and research approach followed in this thesis. This includes a description of the two algorithmic simulation tools used in this research: a blind random search model, and a genetic programming model.

Chapter 4 presents the results of the research conducted using the two algorithmic models. Original research data is presented and discussed. For the genetic programming model, a number of suspected causal factors pertaining to the results obtained are proposed.

Chapter 5 concludes the discussion and suggests areas where future work may prove fruitful, along with challenges that may be encountered.

Chapter 2: Literature Review

This chapter examines the historical roots of genetic programming and related methods. It also explores earlier work in technological innovation and design processes, with particular emphasis on digital circuits and their construction as a representative technology—the substance of which forms the basis for much of the work in this thesis. Finally, models of the engineering process are surveyed to understand the connection between real-world challenges in organizations and simulations of technological evolution.

2.1: Evolutionary Computation

Evolutionary Computation is a term applied to a large class of related methodologies for computational problem solving that are based loosely on or inspired by biological processes. The principal ideas involved have a surprisingly long history, with Alan Turing having proposed a method of "genetical or evolutionary search" as early as 1948, and active computer experiments appearing by the 1960s [Eiben and Smith 2007].

Evolutionary computation methods can be considered a subset of a larger collection of *stochastic optimization* algorithms, such as random searches, genetic algorithms, and particle swarm methods. This family of techniques shares the common trait of employing randomness in some aspect of their progression through the solution space. This feature is hypothesized to improve overall algorithm effectiveness compared to non-stochastic techniques such as by avoiding entrapment in local optima or overcoming the effects of noise in the optimization objective [Spall 2004]. However, it is difficult to generalize among these methods as their performance is often highly problem-specific and sensitive to a multitude of algorithmic tuning parameters [Hassan *et al*. 2005; Eiben and Smith 2007]. A graphical representation showing various algorithmic methods and relationships among them is presented in Figure 2.1. This thesis focuses on an implementation of genetic programming as well as a random search method used in earlier work by [Arthur and Polak 2006].

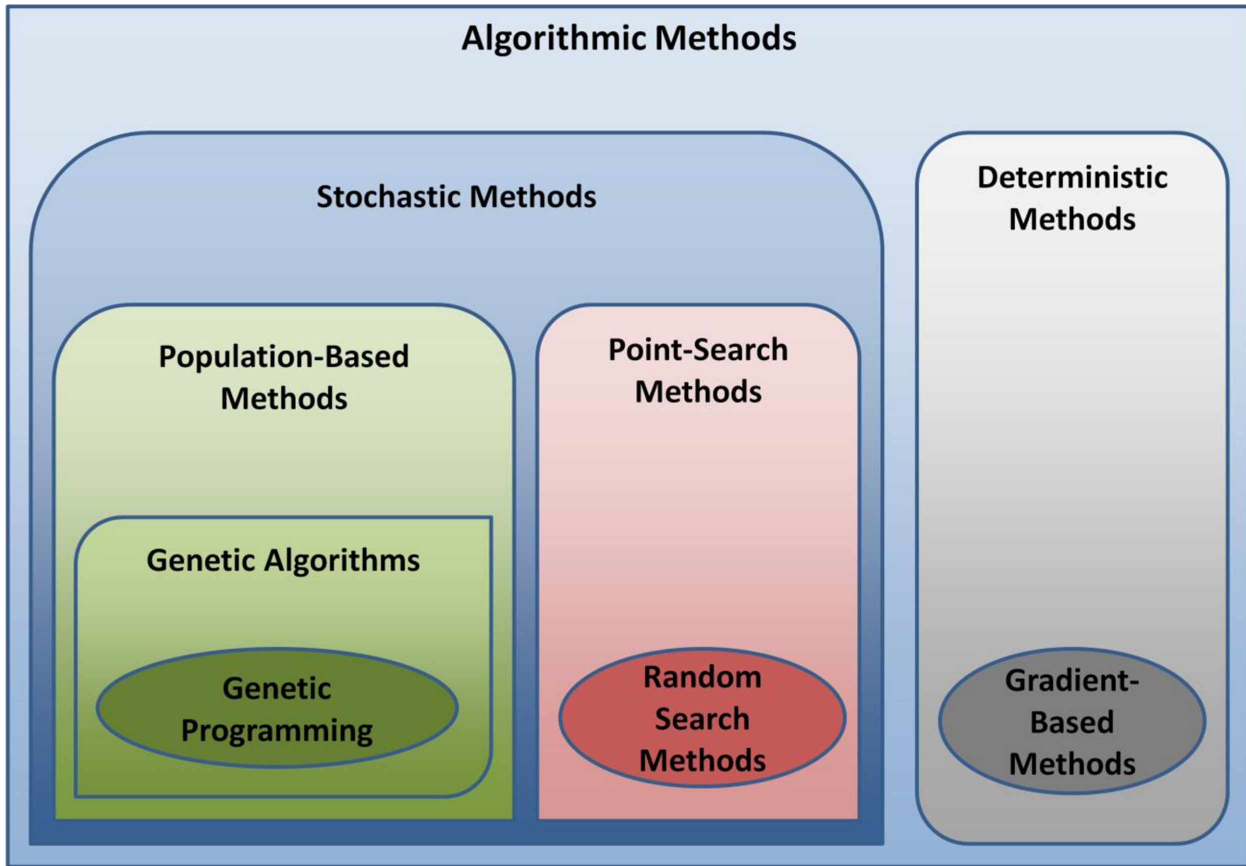

Figure 2.1: Graphical Representation of Algorithmic Methods and Relationships

2.2: Genetic Algorithms

As mentioned above, the genetic algorithm is an evolutionary computation method. The development of genetic algorithms (commonly called GAs) was arguably launched by John Holland in 1975 when he proposed a computational methodology mimicking processes seen in living organisms [Holland 1975]. Genetic algorithms, along with various other evolutionary computation approaches (such as particle swarm methods), employ a degree of parallelism in their search through the problem's solution space by employing a *population of individuals*, with each one representing a possible solution, as opposed to a single-point-at-a-time search procedure [Hassan *et al*. 2005]. Genetic algorithms are generally noted to be robust and efficient tools, but they are non-deterministic and thus offer no guarantee of optimality (i.e., that the global optimum has been found) [Goldberg 1989]. These algorithms also generally employ some form of a survival and/or reproduction mechanism which has the effect of preserving the

better individuals (solutions) contained in the population. Thus, genetic algorithms possess a type of memory effect or continuity feature.

Early implementations of genetic algorithms were generally constructed using fixed-length binary strings roughly analogous to DNA encoding found in biological organisms, as seen in [Goldberg 1989]. In this representation, each individual in the population is composed of a pre-determined length of “0” or “1” characters called a *chromosome* [Mitchell 1998]. These binary values are encoded or mapped in some pre-defined way by the algorithm operator to represent the control variables to be included in the resulting design or contained in the problem to be solved. Blocks of one or more adjacent bits then represent a single control variable in the problem, with the number of bits needed within each block determined by the number of possible permutations the experimenter wishes to represent within the algorithm. Again borrowing biological terms, the functional block units are often called *genes* and the entire genetic code representing a problem design solution contained within an individual may be called its *genome* [Mitchell 1998]. A visual representation of this concept is shown in Figure 2.2. The encoding format need not be binary, however, and various other encoding schemes have been devised for specialized problem applications.

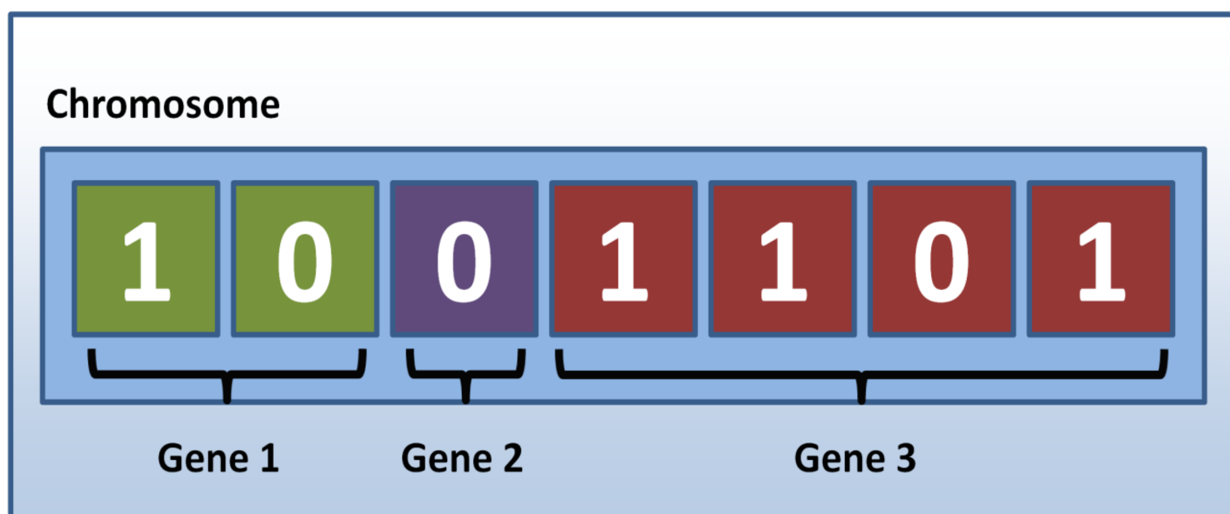

Figure 2.2: Illustration of a Binary Format Chromosome with Three Genes (adapted from [Eiben and Smith 2007])

Genetic algorithms proceed by repetitively modifying the gene expressions contained within the population of individuals on which it is operating. These modification processes, termed

operators, now exist in a multitude of variations, but many are based loosely on the functions observed in biological organisms. The primary operators used generally include: *selection*, *reproduction*, *crossover* or *recombination*, and *mutation*. Other secondary or specialized operators are also seen, and a nearly limitless range of variations exists for genetic operators in general. The selection operator is responsible for choosing one or more individuals out of the population for which some other action (often provided by some other genetic operator) is to be applied. Typically, individuals are sampled randomly from the population or a subset thereof, often with some bias factor applied, such as with respect to the individual's fitness score, as discussed below. The reproduction operator simply copies or promotes an individual into the next *generation*, or iteration of the algorithm. The crossover operator mixes the genes from two or more parent individuals by cutting and splicing segments of their chromosomes, creating an entirely new individual from its parent components. Finally, the mutation operator makes a random modification to some gene within an individual in the population, mimicking the occasional errors that occur in biological DNA [Eiben and Smith 2007; Goldberg 1989; Mitchell 1998; Koza 1992].

A core element of evolutionary algorithms is that each individual in the population must have some assessment of how well that particular individual solves the objective problem. This measure, called a *fitness score*, or simply *fitness*, is most commonly represented as a single real number derived from some fitness evaluation function designed by the experimenter to capture the desired traits of the design or solution being sought. This is a significant challenge and for true innovation, fitness functions must be at least partially approximate. As each individual within the population is created or modified, the design encoded by that individual's genome is passed through the fitness evaluation function to generate a fitness score for that individual. The fitness measure is then available for heuristic use by the various genetic operators (discussed above) as the algorithm proceeds. Likewise, the design space in which the experimenter is operating the algorithm is characterized by a *fitness landscape*, and the algorithm can be viewed simply as a mechanism for navigating a collection of points—the individuals—through that space in search of the optimal design fitness [Mitchell 1998].

The exact mechanism through which genetic algorithms are able to operate and produce success has been a matter of debate since their debut. In work dating back to Holland's original 1975 proposal, an implicit assumption existed that the algorithm succeeded by creating, mixing,

and amplifying good partial solutions to the problem being run [Eiben and Smith 2007; Mitchell 1998]. These elements are called *schemas* and can be thought of either as similarity templates among the design solutions [Goldberg 1989], or hyperplanes through the solution space [Mitchell 1998]. Additional study of this phenomenon led to a formalization known as the Schema Theorem. In this framework, genetic algorithms succeed by repetitively and exponentially increasing the incidence of smaller, simpler schemas whose average fitness is above the overall mean fitness of all schemas [Mitchell 1998]. Thus, small, basic elements of the design that are valuable are collected together into larger and more complex elements.

An extension of this idea leads to the Building Block Hypothesis which asserts that genetic algorithms solve problems by repeatedly assembling smaller, simpler components into larger, more complex components to eventually arrive at the final optimal solution [Eiben and Smith 2007]. Such a mechanism is proposed as the source of much of the power of genetic algorithms and related search techniques; by building ever larger components out of smaller elements already discovered, the overall algorithmic complexity of the problem is dramatically reduced, potentially making a large problem computationally feasible that would otherwise be intractable [Goldberg 1989]. This concept is a recurring theme in the literature and it forms a component of much of the prior work on which this thesis is based.

One topic of much interest related to building blocks is the concern of adequate supply of building blocks within the genetic population while the algorithm runs. In order for the algorithm to successfully "discover" higher-order, more complex building blocks, it must be supplied with or otherwise generate the necessary simpler building blocks to be used as components in construction (according to the Building Block Hypothesis). Since building blocks exist as subsets of an individual's genome, the supply of total available building blocks is implicitly linked to the population size. Consequently, the study of population sizing models has been an active area of research, as in [Sastry, O'Reilly, and Goldberg 2005] and [Sastry *et al.* 2003].

2.3: Genetic Programming

Genetic programming, often called GP, can be viewed as an extension or specialization of the class of genetic algorithms discussed above. The modern form of genetic programming was

effectively introduced by John Koza and popularized in [Koza 1992], although earlier elements of the concept existed in the broader evolutionary algorithm arena dating back to at least the 1960s [Banzhaf *et al*. 1998]. Genetic programming, as originally proposed in [Koza 1992], was designed to evolve computer programs (hence the name) in the Lisp programming language. Although genetic programming can be implemented in any programming language, Lisp was originally chosen largely because of its rather unique representation of both data and program instructions in the same format—as symbolic expressions. This meant that Lisp programs could be dynamically altered and then executed directly. In essence, this provided a means to create *self-modifying* computer programs [Koza 1992].

Although GP was originally proposed as a means of creating or discovering computer programs, the basic technique has since been enhanced and extended to more general constructs that may simply represent the structure, or the instructions to create the structure, of a solution to a given problem, rather than executable computer instructions in the form of a program. Other problems and fields where GP has now been applied include: product and system design (circuits, antennas, etc.), pattern recognition, data mining, control systems, neural network structures, machine learning, optimization, data modeling and symbolic regression, bioinformatics, visualization, and even music [Langdon and Poli 2002; Koza *et al*. 2003; Poli, Langdon, and McPhee 2008]. More recently, GP has been credited with generating several dozen inventions described as "human-competitive," including re-discovery of already-patented inventions and at least two newly patentable inventions [Koza *et al*. 2003].

The primary difference between genetic programming and the older genetic algorithm is that GP traditionally employs a *tree-shaped* genome or chromosome composed of objects rather than the binary string most commonly seen in genetic algorithms. This is a distinguishing trait of GP, and it implies that genomes can be of both variable and infinite size, which contrasts sharply with the rigid, fixed-length string representations seen in GAs. This flexibility allows for arbitrary representation of design solutions, and also permits designs of problem solutions to be made of infinite levels of detail and specificity [Banzhaf *et al*. 1998]. Other representations of GP do exist (such as graph or matrix structures), but they appear to be less prevalent than the tree-based structures which are the form used in this thesis.

The genome tree is formed as a connected set of objects called *nodes* drawn from some population provided by the experimenter. Each object may be either a *function* or a *terminal*,

depending on its behavior. Functions accept one or more inputs and produce one output while terminals accept no inputs and return one output. Therefore, terminal objects always form the "leaf" nodes of the tree while functions form the connecting nodes of the tree. The "root" node at the top of the tree returns the final result from the instructions being encoded within the tree [Banzhaf *et al*. 1998].

Figure 2.3 shows a stylistic example of a genetic programming tree genome encoding a collection of logic functions over three input variables, A, B, and C. In this illustration, the terminal objects are represented as red circles while functions are shown as named rectangles. This particular collection of functions and terminals might be used to represent the design of a digital circuit. The function and terminal sets from which the genome trees are constructed are chosen based on the problem domain for which the GP engine will be executed.

Implementations vary, but the instructions encoded within a genome tree are typically evaluated in a depth-first order such that the lowest function branches on the tree are evaluated first, and then evaluation proceeds recursively up the tree until the root node is reached. In Figure 2.3, the **XOR** function would be the first function evaluated and the **NAND** function of two sub-function results at the root of the tree would be the last evaluation.

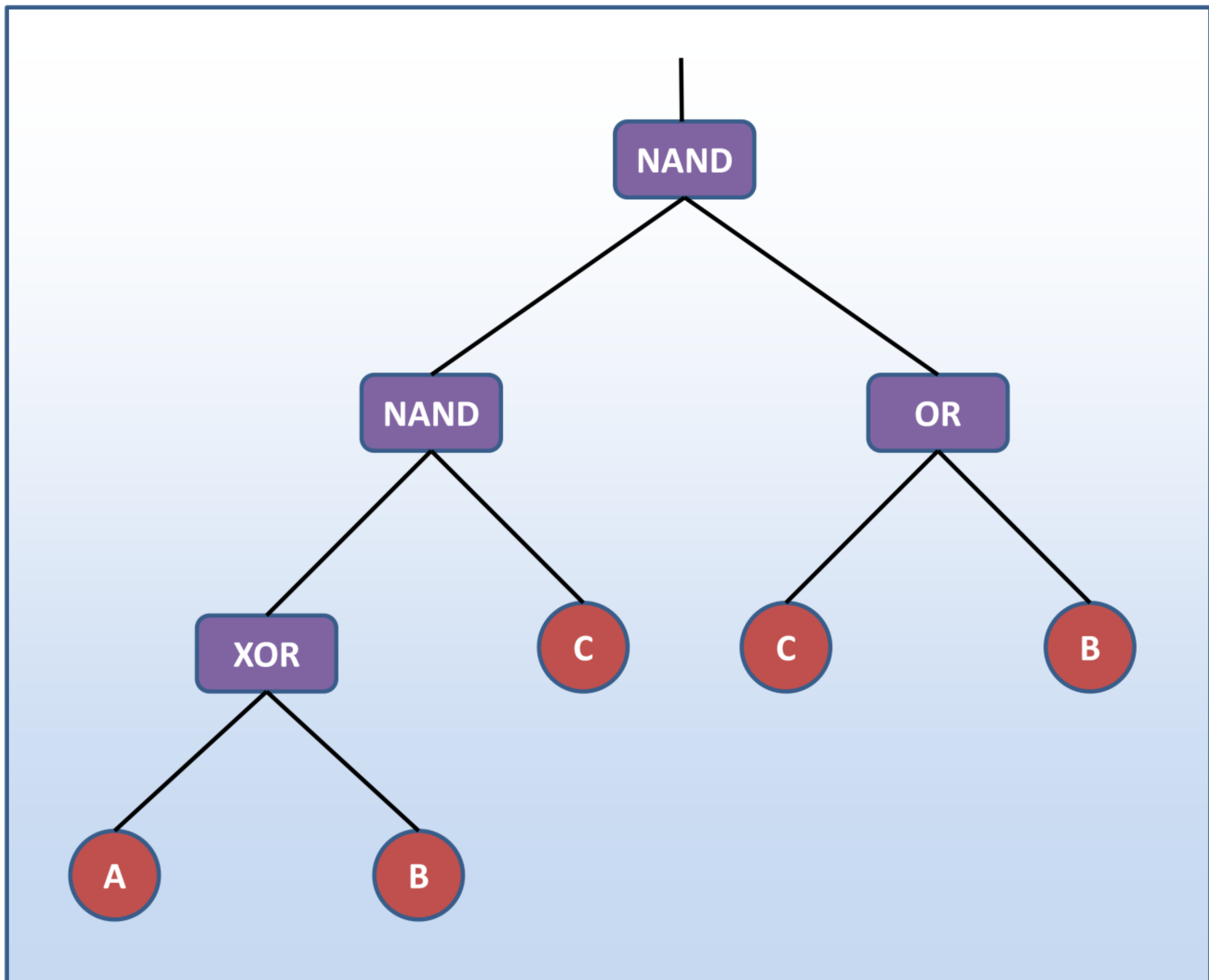

Figure 2.3: Illustration of a Genetic Programming Tree Chromosome with Five Terminals and Four Functions (adapted from [Luke 2000])

The fundamental genetic algorithm principles and operators, such as mutation, selection, crossover, and so on (described earlier) are extended to genetic programming as well, though with some adaptations necessary to function with the tree-shaped genome structures. For example, the mutation operator is more complex than that seen in traditional genetic algorithms where it merely flips a "0" and "1" value. In GP, it is adjusted to randomly select a node from the genome tree and then make a random modification to it, such as substituting one function or terminal for another from the available set, deleting an entire branch from the tree, or attaching a randomly generated new branch to the tree. Similarly, the crossover operator is modified to select a node (typically at random) within each of the genome trees to be crossed, and then switch the entire sections, called *subtrees*, beneath the crossover points to generate a new

offspring individual, although ensuring data type safety and function compatibility complicates implementation. Numerous variations on these and other GP-specific features exist throughout the literature in the field [Banzhaf *et al*. 1998].

As with genetic algorithms, the concept of building blocks applies to genetic programming too. In GP terminology, a building block is manifested as a subtree section of a genome tree, and even the entire tree itself can be a building block. There is, however, debate in the literature over the importance and implications of schemas and building blocks relative to GP implementations [Banzhaf *et al*. 1998; Luke 2000].

The tree-shaped genome structure which allows infinite and flexible design encoding may be responsible for much of the creative power seen in GP, but it also brings a severe and persistent side-effect that often challenges practical applications of GP as a design tool. Given the unlimited flexibility of the genome construction and size, the tree structures can (and frequently do) grow to enormous sizes, leading to severe degradation of algorithm performance and acute consumption of system resources. Often, large portions of such an inflated genome tree accomplish little or nothing toward the fitness of the individual. This excess material is referred to as *junk DNA*, *bloat*, or *introns*. Furthermore, bloated genomes tend to grow at exponential rates, quickly resulting in stagnation of the run after which little additional progress is made. Although the phenomenon is still not well understood, some research suggests that introns (specific segments of genome code that have no practical effect) may actually serve a necessary and beneficial purpose by protecting valuable building blocks of code within the population from the destructive effects of the crossover operators by minimizing the chance of splicing through a building block [Banzhaf *et al*. 1998; Luke 2000].

Given the severity of the problem of bloat in GP, a good deal of attention has been paid to controlling its impact within an algorithm run. Perhaps the simplest method is simply to impose a maximum size limit on each genome tree. Another common approach involves *parsimony pressure* in some form, which seeks to penalize the fitness scores of individuals in the population who carry excessively large genomes [Luke and Panait 2004].

In the original genetic programming form proposed by John Koza in [Koza 1992], all information about the construction of a genetic individual was stored within the structure of the tree-based genomes. Since then, numerous extensions and enhancements have been proposed to further improve the efficiency and performance of GP projects. One of the earliest and most

common extensions seen in the literature is Automatically Defined Functions (ADFs), proposed in [Koza 1994a]. ADFs provide a means to enable code reuse and reduce the size and complexity of a solution, just as human programmers use functions to achieve the same benefits when writing software. To implement ADFs, the genome structure is altered so that each individual contains one main tree which encodes its final solution, called the *result-producing branch*, and one or more additional trees, called *function-defining branches*, which encapsulate segments of genetic encoding that can be referenced by the main result-producing branch. In effect, the ADF structure provides a mechanism to store genetic code that would have to reside within the main genome tree—often redundantly—in the original, classic form of GP, and to enable that encapsulated code logic to be used repeatedly in constructing the evolved solution [Koza 1994a].

Koza further elaborates that ADFs are most useful in problems that can be solved as a decomposition of the main problem into subproblems, and where the problem contains regularities, symmetries, or patterns. A large body of his analysis showed that total computational effort of the GP algorithm can be significantly reduced on non-trivial problems when ADFs are in use [Koza 1994a]. One example problem in [Koza 1994b] showed a 283-fold improvement (decrease) in the computational burden expended to solve the problem when ADFs were used versus solving the problem without them.

The general concept embodied in the ADF construct was later extended to other elements and ideas from traditional software engineering, including Automatically Defined Loops (ADLs), Automatically Defined Macros (ADMs), Automatically Defined Storage (ADSs), Automatically Defined Recursion (ADRs), Automatically Defined Iterations (ADIs), and Automatically Defined Copies (ADCs) [Koza *et al*. 1999]. However, these features do not appear to have attained the same level of interest and attention in the literature as ADFs did.

Another mechanism to reuse parts of a genome tree has been proposed called Module Acquisition (MA). It is conceptually similar to ADFs though it bears significant structural and implementation differences. It operates on the GP population of individuals as a new genetic operator where it occasionally selects an individual at random, then selects some random subtree within that genome. The chosen subtree is then cut out and placed into a central "module library" as a packaged module available for reuse globally among other members of the population, and a reference to the new module is inserted back into the original individual in

place of the excised code subtree [Kinnear 1994]. Curiously however, this mechanism performed much worse than ADFs and even the original, basic GP when tested on a particular problem in [Kinnear 1994].

The Module Acquisition method is of particular interest here because it bears a close resemblance to the method of technology or invention encapsulation and reuse seen in [Arthur and Polak 2006] and used in this thesis. However, a significant difference is that Module Acquisition selects its code for module contents at random from the population, and thus adds much content to the module library that may be of little value, while Arthur and Polak add only fully successful designs to their technology library (as discussed below).

Overall, genetic programming attempts to re-invent the evolutionary process to obtain better design solutions to a problem, whereas early GA work simply pursued mimicking of biological evolution. It seems appropriate to look wider than biological processes for innovation and design as these processes clearly do not evolve simply by biological mechanisms. Differences include artificial or human selection as opposed to natural selection, the potential for inheriting and sharing newly invented traits, and many others. However, a complete model of the human (and especially the normal group) invention process does not exist.

2.4: Innovation and Design

The notion of building blocks is of significant interest for consideration of processes in the realm of technological innovation. Recent works have begun to consider the connections between "computational innovation" (using tools and methods such as genetic algorithms or genetic programming) and systems engineering, organizational design, technology transfer, creativity, and the inventive process. In this sense, genetic engines (GAs and GP) can be viewed as models of innovation. (See Epilogue in [Goldberg 2002].)

An offshoot of this is an idea by Goldberg for genetic engines described as *competent*: those that can solve hard problems quickly, reliably, and accurately [Goldberg 2002; Sastry and Goldberg 2003]. To that end, researchers are beginning to devise enhanced methods and techniques in second-generation genetic engines to more effectively mirror the processes of invention, design, creativity, and innovation seen in real-world human practices. These extended methods may provide significant improvements in the capabilities and performance of genetic

engines as tools for innovation and design within engineering practice. In addition, study of the effects often observed within genetic engine runs—taken as models of innovation—may provide unique insights to help us better understand the analogous properties and issues encountered in engineering and management problems [Goldberg 2002].

Another framework of technological innovation has been put forth in [Arthur 2007] and further expanded in [Arthur 2009]. In this model, technological design and innovation is an iterative, hierarchical, and recursive process whereby technologies are formed out of simpler component technologies. This is closely aligned to the notions of *systems of systems*, *subsystems*, *assemblies*, *modules*, *parts*, and similar terms that are commonplace within the field of systems engineering. Additionally, this idea of recursive construction of complex technologies from combinations of simpler ones strongly echoes the idea of building blocks discussed earlier, particularly in the context of genetic engines.

In the Arthur framework, a distinction is made between the development of radical new technologies and incremental improvement of existing technologies, with focus placed on the former. Invention is described as a process of recursive problem solving. Technologies arise through some structure or process whereby a phenomenon is exploited through a principle to harness an effect as a means to fulfill a purpose. Novel technologies may be created through one of two motivations: a needs-driven process where some opportunity or necessity exists that must be met, or through a phenomenon-driven process where some observation or discovery leads to or suggests the use of a new principle. In the course of developing a new technology, challenges may be encountered with its constituent sub-technologies, each of which must be solved through this same process, thereby giving rise to the iterative and recursive feature. Furthermore, once a new technology is created, it may suddenly enable the completion of other larger technologies using itself as a component, often leading to bursts or waves of inventions that Schumpeter observed in economics [Arthur 2007; Arthur 2009].

Interestingly, the invention and design process need not involve humans. Significant literature (e.g., [Koza *et al*. 1999; Koza *et al*. 2003]) exists using genetic programming to autonomously design analog circuits, including various filters, amplifiers, and controllers. More recent work has produced designs in various domains including analog circuitry, antennas, biological metabolic pathways that are deemed "human-competitive," along with several patented or patentable inventions [Koza *et al*. 2003]. In fact, Koza states: "Genetic

programming is an automated invention machine" [Koza *et al*. 2003, pg. 530]. Other approaches and algorithms have also been used to demonstrate automated invention of technologies, including a random search construction process described below.

2.5: Digital Circuit Construction

Understanding the exact nature by which innovations and new technologies arise, and older ones fail, has been of interest to both academics and practitioners alike for a long time. A series of experiments contained in [Arthur and Polak 2006] ties together several of the concepts discussed above, including building blocks and the recursive "stepping stone" manner of technological construction. This work was carried out by designing a virtual technology invention program driven by a blind random search algorithm. The technologies to be created are digital circuits, such as various adders and logic functions. Digital circuits were chosen because they are readily analyzable from a pure computation perspective, and because digital circuits can be readily modularized into parts and subsystems, just as is commonly seen in physical systems and technologies. Therefore, digital circuits serve as a proxy for real-world systems and technologies, and by studying the effects occurring within this model, the potential exists to gain important insights about the innovation process.

The Arthur and Polak simulation model uses a simple, blind random search algorithm (i.e., one that does not possess any particular domain knowledge or any self-enhancement ability) which was initialized with a primitive circuit component, such as a **`NAND`** gate. A list of technologies or "goals" for which invention is sought is input into the algorithm. The engine then attempts to "invent" technologies specified in the goal list by randomly hooking together components from its primitive set and its accumulated library of earlier inventions. As the algorithm runs, once it discovers a design solution that perfectly satisfies a goal technology, it is added to an accumulated list of inventions, thus forming a module which may then be reused in future designs [Arthur and Polak 2006].

This paper found that complex circuits, such as an 8-bit adder, could be constructed if simpler, intermediate technologies were first invented by the algorithm and then reused as modular components. Conversely, if modularity and reuse were not permitted, the algorithm failed to create complex technologies. This finding provides strong empirical evidence in

support of the building block paradigm discussed earlier. More broadly, it provides insight into the system design and innovation process, and how these effects relate to economic influences, such as creative destruction [Arthur and Polak 2006].

One of the goals of this thesis is to explore whether these algorithms are consistent with empirically observed exponential growth rates of technological progress. This thesis extends the work in [Arthur and Polak 2006] by using the same simulation platform used in that study, along with a genetic programming platform, as the two experimentation approaches performed in this work.

2.6: Engineering Process Models

An important consideration when using algorithmic models for simulation of innovation and design is whether the model is sufficiently detailed and robust to capture the core dynamics observed in human-led processes, such as the acceleration of the growth rate of technological progress. One of the key interests in this thesis is whether the results produced by the models tested in this research agree with the empirical evidence from real-world systems and processes.

Innovation and design processes within engineering practice today can be viewed as predominantly human social processes. Although various technological tools are available and frequently used throughout the design process to aid the engineer, the process itself remains chiefly driven by human engineers. For most modern product development initiatives of any practical significance, this design effort is likely to involve more than a single individual. Consequently, these engineering efforts involve teams of people, and with this comes the many nuances and considerations surrounding team dynamics and organizational management.

Given this central importance of human socio-technical interaction, some studies have attempted to capture the essence of the human-led design process. One such model is the Pugh Controlled Convergence method (PuCC). This relatively simple model describes the design concept phase (after specification development but prior to detailed design work) of the engineering lifecycle involving an engineering team. It provides a somewhat structured and disciplined process of narrowing down a set of design concepts under consideration by iteratively reviewing and comparing strengths and weaknesses of competing designs in a matrix format. As the process ensues, information is gathered, learned, and shared among team

members. This may lead to changes within the repertoire of design concepts under consideration, such as enhancing some aspects of a given design to create a new, stronger candidate solution, weeding out inferior ("dominated") designs, or pursuing additional information to further improve decision making. This cycle is then repeated and the result is a whittling down of the set of remaining designs worth further consideration. After a few iterations, the design choice should converge to a superior design candidate agreeable to all team members [Frey *et al*. 2008].

The rates of technological progress have been studied over relatively long periods of time (100 to 150 years) in domains such as information technology (in [Koh and Magee 2006]) and energy (in [Koh and Magee 2008]). These studies found persistent rates of exponential improvement across the various technologies and functional performance metrics tested. Progress within information technology grew at a significantly faster rate than in energy. This exponential rate of growth is hypothesized to be the result of humans building new technologies from cumulative prior knowledge—a theme consistent with the building block idea discussed earlier [Koh and Magee 2008].

One of the objectives in this thesis was to see whether the algorithmic simulation models were able to produce similar effects of knowledge accumulation and reuse so as to enable exponential rates of technological improvement during the simulation runs.

[This Page Intentionally Left Blank]

Chapter 3: Methodology

This chapter describes the research and experiments performed for this thesis. This research focused primarily on comparing two methods of algorithmic simulation of digital circuit construction. These circuits are intended as models of technologies and as a representative proxy for understanding effects and behaviors observed in the system design and innovation processes.

3.1: Research Approach

This thesis focused on testing the blind random search algorithm and the genetic programming algorithm based on beliefs stated in [Goldberg 1989] and [Eiben and Smith 2007] that the genetic programming algorithm is generally a more efficient and effective tool than a blind random search algorithm. Therefore, this thesis tested the hypothesis that a GP engine would successfully create new complex technologies—digital circuits—from simpler building block components discovered earlier during the algorithm run, and that this effect would occur faster and/or more efficiently than when using a blind random search algorithm.

The choice of digital circuits as the medium of study was made because digital circuits are relatively straightforward to model and evaluate programmatically, and because this allowed the work in this thesis to extend and build upon prior results found in [Arthur and Polak 2006].

The blind random search model used in this thesis (the "Arthur and Polak model") was responsible for generating the results presented in [Arthur and Polak 2006], [Arthur 2007] and in [Arthur 2009]. Those publications collectively establish the following hypotheses: technology is *autopietic* (meaning self-creating); invention is a process of recursive problem solving; technologies are formed through combinations of earlier designs (building blocks); and invention and innovation result as a process of linking human needs and goals with some phenomenon or effect where the role of technology is to harness that effect.

Another aspect of technology innovation which this thesis attempted to test is the commonly observed rate of exponential progression seen in real-world technologies (e.g., Moore's Law). Numerous runs were conducted on both the Arthur and Polak model and the genetic programming model, which are described below, with various parameter settings and

configurations to ascertain whether evolved technologies appeared to be developing at an increasing rate of progression. These results are discussed in Chapter 4.

3.2: Algorithms

This thesis tested and compared two types of algorithms from the literature, each powering a model of the system design and technology innovation processes. The algorithms are described in detail below.

Both algorithms were given the same task: to design and construct complex digital circuit technologies starting from only rudimentary components. The desired designs are specified in advance by the experimenter as "goals" for each algorithm to work towards. The list of goals used in this research is the same for both algorithms, and is shown in Table 3.1.

Goal	Technology Identifier	Inputs (n)	Outputs (m)	Description
GOAL 1	(not-)	1	1	Negation
GOAL 2	(imply-)	2	1	Implication
GOAL 3	(and-)	2	1	Conjunction of 2 inputs
GOAL 4	(or-)	2	1	Disjunction of 2 inputs
GOAL 5	(xor-)	2	1	Exclusive Or of 2 inputs
GOAL 6	(equiv-)	2	1	Equality of 2 inputs
GOAL 7	(and3-)	3	1	Conjunction of 3 inputs
GOAL 8	(1-bit-adder-)	2	2	Addition of 1-bit inputs
GOAL 9	(full-adder-)	3	2	Addition of 2 inputs and carry
GOAL 10	(2-bit-adder-)	4	3	Addition of 2-bit inputs
GOAL 11	(3-bit-adder-)	6	4	Addition of 3-bit inputs
GOAL 12	(4-bit-adder-)	8	5	Addition of 4-bit inputs
GOAL 13	(5-bit-adder-)	10	6	Addition of 5-bit inputs
GOAL 14	(6-bit-adder-)	12	7	Addition of 6-bit inputs
GOAL 15	(7-bit-adder-)	14	8	Addition of 7-bit inputs
GOAL 16	(8-bit-adder-)	16	9	Addition of 8-bit inputs

Table 3.1: List of Technology Goals for Digital Circuit Construction (adapted from [Arthur and Polak 2006])

3.2.1: Blind Random Search

Testing of the blind random search method used the same algorithm and code[1] used to generate the results in [Arthur and Polak 2006]. This model is programmed in the Common Lisp programming language. It represents both its goals and the technology designs (circuits) being evolved in the form of Binary Decision Diagrams (BDDs). BDDs provide a compact, efficient, unique, and canonical method of representing any Boolean expression [Andersen 1998]. This representation also makes possible a measurement of similarity between any two circuit designs [Arthur and Polak 2006]. This feature allows the algorithm to easily test whether a given candidate design solution correctly implements a desired goal, and to measure the degree of "correctness" against such a goal. This measure is translated into a fitness score for each candidate circuit design.

The blind random search algorithm is so named because it possesses no particular knowledge about the problem domain in which it operates, and it does not evolve specific intelligence or adaptation to guide or enhance its construction process. It is initialized with a set of one or more *primitives*, the simple building block elements from which all other technologies can be constructed. For the experiments performed in this thesis, the set of primitives included the **`NAND`** logic function and the Boolean constants for **`TRUE`** and **`FALSE`**. The algorithm then constructs a new circuit design by randomly choosing a number of components from this set of primitive building blocks as well as from a growing library of already-solved design goals, and then randomly connects them to form a new design as an invention to be considered against the remaining list of unsolved goals. As the algorithm execution proceeds, it may eventually discover a correct solution for a goal, at which point that design is encapsulated as a new technology and added to the library of primitive components available for use in future design construction, thus enabling reuse of prior work. This process of encapsulation and reuse enables the algorithm to succeed at designing complex circuits through recursive construction of simpler modules and subassemblies, even though it is a blind random search methodology.

[1] The code was graciously shared with this author by W. Brian Arthur and Wolfgang Polak.

3.2.2: Genetic Programming

Testing of the genetic programming method employed the Evolutionary Computation in Java ("ECJ") package, version 19.[2] ECJ is a free, flexible, open-source platform written in the Java programming language and developed by Sean Luke *et al*. for performing evolutionary computation research. It supports genetic programming (GP) amongst many other variations in the broader class of evolutionary algorithms, such as evolutionary strategies, genetic algorithms, and particle swarms. During testing, it was found to be full-featured and robust in this author's opinion.

The ECJ GP framework contains nearly unlimited flexibility to extend or alter various parts of the framework, along with numerous configuration parameters and settings. Many of these are pre-configured with default values popularized in [Koza 1992], and those settings were generally retained for the experiments conducted in this thesis. Anecdotal evidence from various sources in the literature suggests that the various configuration parameters and settings used in GP can have dramatic influence over the outcome of the simulations, so the results found in the research work for this thesis may be somewhat situation-specific. The sheer number of parameters and their combinatorial interactions made testing for all of these sensitivities infeasible, although some major ones were varied without significant impact on the results reported herein.

Initial efforts with GP for this thesis attempted to represent circuit design goals and candidate design solutions (individuals) in the population as vectors of BDDs, just as the Arthur and Polak model does. However, significant difficulties in implementation and irregularities of behavior necessitated the abandonment of this approach. Instead, both goals and design solutions encoded within individuals' genomes were represented in the form of Boolean logic *truth tables*. This representation has notable drawbacks and undoubtedly impacted the results produced by the GP model execution, as discussed in Chapter 4.

The primitive set given to the GP for this research was similar to the setup used by the Arthur and Polak model: a **`NAND`** function, the Boolean **`TRUE`** and **`FALSE`** constants, and two variables **`A`** and **`B`**. The model was executed with a population size of 1000 individuals over 250

[2] ECJ source code is freely available at: http://cs.gmu.edu/~eclab/projects/ecj/

generations to mirror (as closely as practicable) the parameters used for the Arthur and Polak model, which was 250,000 single-trial iterations.

The fitness function for individuals was designed to measure the similarity of the individual's truth table to the truth table of a given design goal. The fitness score represented the proportion of "correct" entries in the individual's truth table vector when matched against the truth table vector of one or more design goals. The number of design goals against which to evaluate an individual was used as a control parameter, and various settings were tested; see Figure 4.12 for results. A fitness score of zero would be produced with no matching truth table values, while a score of one would indicate a perfect match of the design goal. Thus, discrete gradations were possible for partial matches. This approach is reflective of the "Hamming distance" measure described in [Mitchell 1998].

The truth table necessary for fitness scoring of a given individual is derived dynamically by evaluating the individual's genome tree. Refer to Figure 2.3 for a visual representation. The management of the genome tree is handled automatically by the ECJ GP framework, including construction, crossover, mutation, type safety, evaluation, etc. The ECJ engine evaluates the genome tree by locating the deepest function node (the **XOR** function shown in Figure 2.3) and then processing the inputs to that node via the specified logic function to return a resultant truth table for that node, then recursively propagating results upward through the tree until the root node is reached. The root node contains the final truth table representation of that individual's genome, which is then evaluated against the design goals to produce a fitness score.

[This Page Intentionally Left Blank]

Chapter 4: Results

This chapter presents the findings of the experiments performed in evolving complex digital circuits using two different algorithms—the Arthur and Polak blind random search model, and the ECJ genetic programming model. Attributes of the models and an explanation of the experimental setup are discussed in Chapter 3.

4.1: Overview

The initial premise at the start of this research was that the genetic programming model would be able to design complex technologies in the form of digital circuits by using building blocks from earlier solutions to construct ever more sophisticated products, just as the Arthur and Polack blind random search model successfully does, but that GP would do so more efficiently, more quickly, and more powerfully (and also possibly show exponential progress in capability over time). This process of recursively constructing technologies is said to mimic the human innovation process.

The research performed for this thesis was able to validate the work done by the Arthur and Polak model. However, the GP engine did not perform as expected in these experiments. In the particular implementation tested in this thesis, GP did not succeed in reliably constructing even relatively simple digital circuits as technology goals. The sections below discuss in depth each of the models and the specific results found. For GP, several suspected contributing factors in the failure are offered, and Chapter 5 presents some possible features that may help overcome the limits encountered in this work.

4.2: Blind Random Search

The blind random search model was executed with minimal modifications from its original form, with minor changes being made mostly to facilitate data capture and study of results. The model operated successfully and was consistently able to create complex circuit designs, thus validating the results reported in [Arthur and Polak 2006]. Experiments were run with many different variations of the control parameters and settings contained within the model to enable

better understanding of its behavior and the effects observed in the generated output. A selection of progress profiles is presented here with a brief discussion of each. Note that due to the stochastic nature of the algorithm, the results will differ from one execution to another.

Figure 4.1 shows the results from a typical run of the algorithm. The colored lines represent the fitness progression of each of the circuit technologies as they arise from initial invention and evolve toward completion. The list of goals is arranged in a deliberate order from simple designs to complex ones (as described in [Arthur and Polak 2006]) so that more complex designs can benefit from solutions to simpler goals found earlier in the simulation run.

Figure 4.1: Results of a Typical Run of Arthur and Polak Random Model

We observe recurring patterns in the data produced. Simpler goals are successfully solved early in the simulation run, and these successes pave the way for more complex and far more difficult goals to begin forming. Fitness values for a particular goal technology begin when a random construction of components (primitive set values plus earlier solved goals) reaches a fitness score (a measure of correctness against the prescribed goal) of at least 50%. The fitness score of that technology then progresses on a generally upwards trajectory towards completion with a fitness score of 100% when a perfect design is located. On occasion, the fitness score of

the design may dip downwards as the random nature of the model may permit a worse-scoring design to be selected. This can serve as a means of escaping entrapment in localized optima within the search space, particularly if such a lower-scoring design is "cheaper," meaning that it contains fewer constituent components. The algorithm does consider "cost" as a secondary objective in the course of circuit design.

Another recurring pattern observed in the data is that progress on a given circuit design is often rapid at first when the design first appears, but then tends to slow and often undergoes long flat periods of no progression. Similarly, the overall progress of the algorithm as a whole when combining the progress curves of all the goals exhibits a clear slowdown in the rate of progress as the run proceeds. Thus, this data does not agree with the empirical fact that technologies evolve and improve at an exponential (or at least an increasing) rate as is observed in real-world technological progress. This suggests that the model misses some crucial element of human-led innovation, such as a learning effect. Note that this finding does not undermine Arthur's hypothesis or results, as the model was not developed to test this particular aspect of innovation, only that complex technologies can be formed out of simpler modules without human intervention.

Below, Figures 4.2–4.4 present various trials of the algorithm with its default settings. Multiple trials are presented to illustrate the differing outcomes resulting from the stochastic variation within the model. Figures 4.2 and 4.3 use event-based sampling intervals (i.e., a data snapshot was taken each time the algorithm made some improvement in any circuit design), while the remainder of the figures in this section use uniform interval performance snapshots.

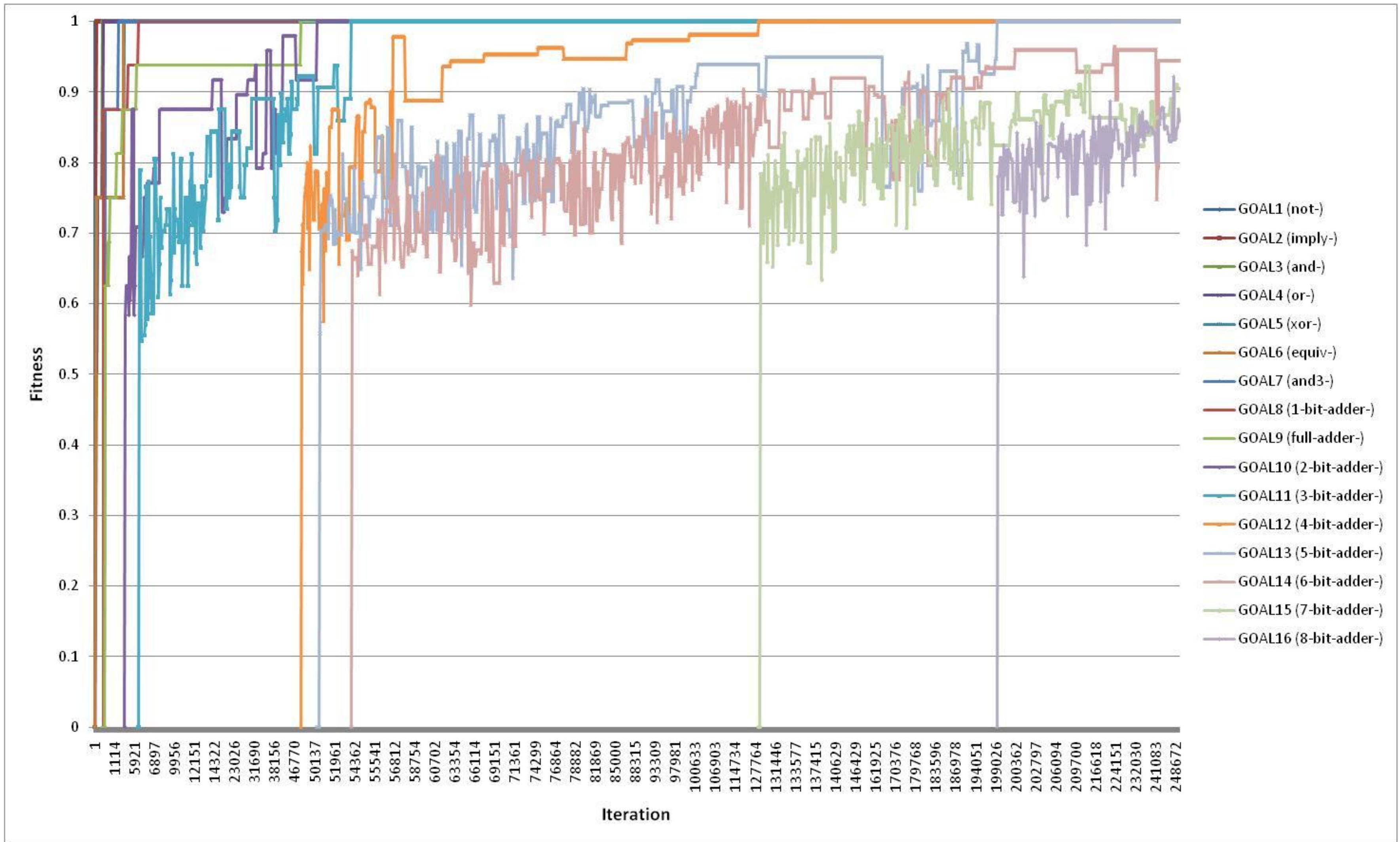

Figure 4.2: Trial 1 – Typical Run of Arthur and Polak Random Model with Event-Based Performance Intervals

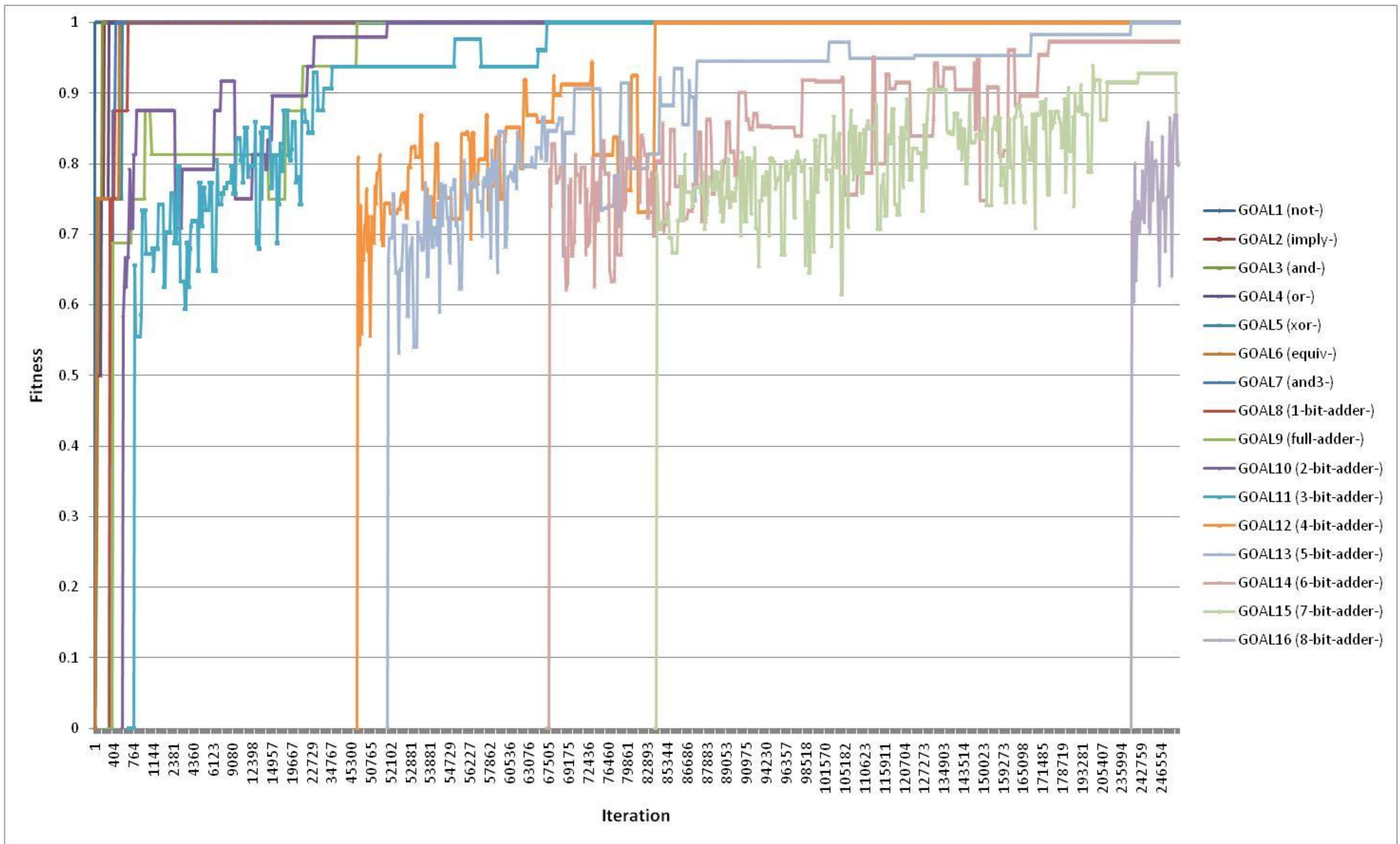

Figure 4.3: Trial 2 – Typical Run of Arthur and Polak Random Model with Event-Based Performance Intervals

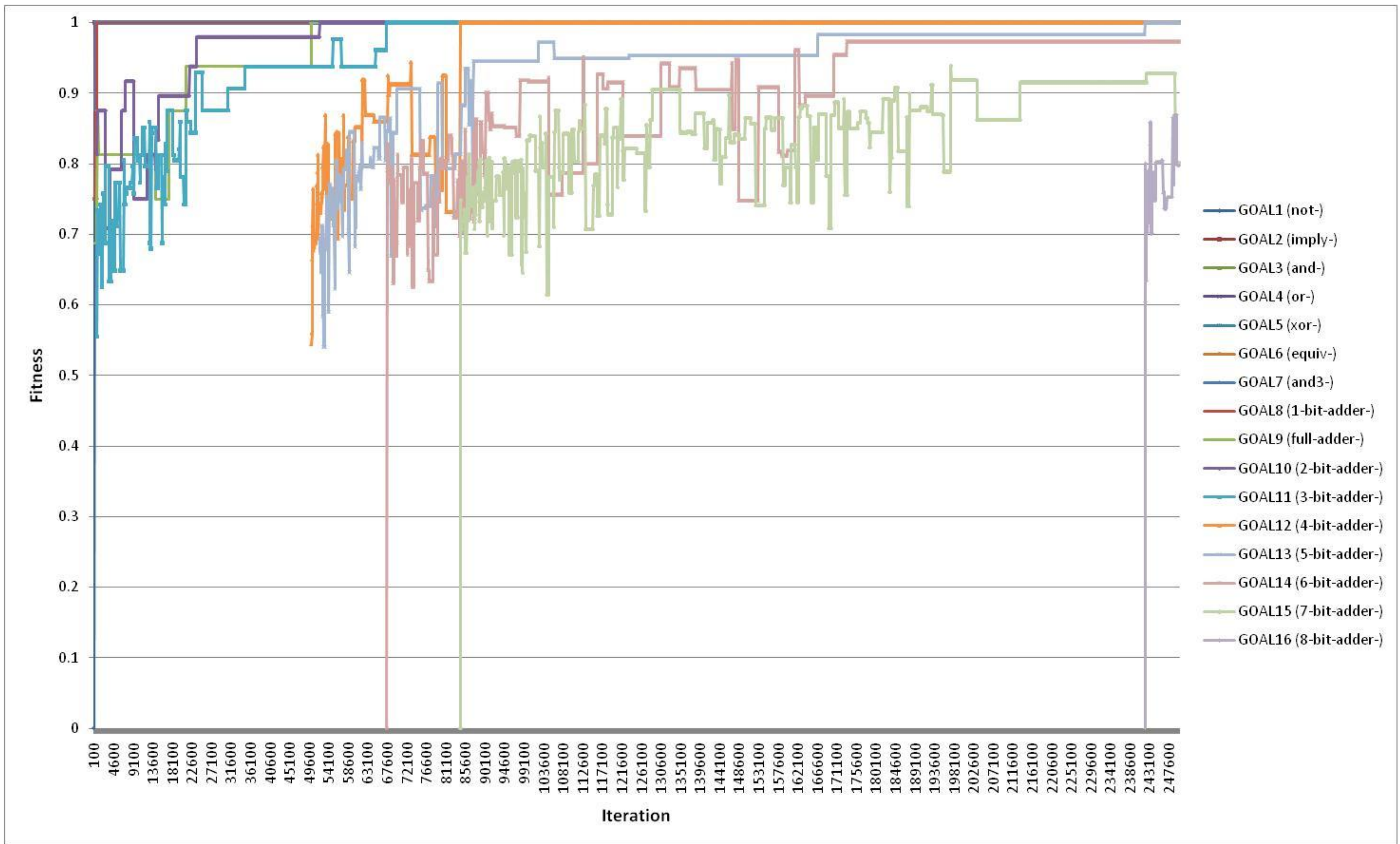

Figure 4.4: Trial 3 – Typical Run of Arthur and Polak Random Model with Uniform Performance Intervals

Other trials of the algorithm were conducted by varying certain control parameter settings to gauge the impact on behavior and performance. Figure 4.5 shows a run of the algorithm with the constraint on maximum concurrent working goals removed. By default, the algorithm normally only processes the next three unsolved goals (from Table 3.1) at a time, and the completion of one goal enables the commencement of work on another remaining unsolved goal. This parameter has the effect of channeling the algorithm's focus to a few narrow problems beginning with simpler technologies and working toward more complex ones, rather than permitting a concurrent broad-based search among all unsolved goals. As this figure shows, removing this channeling constraint had a notable deleterious impact on the algorithm's performance. Although goal technologies were clearly started much earlier in the simulation than in the prior trials, they tended to take much longer to complete, with many of them never finishing successfully within the allotted run time. It seems that removing the constraint had the effect of dispersing the algorithm's focus across multiple technologies simultaneously, rather than forcing it to use earlier solutions as building blocks toward more complex designs.

Figure 4.6 shows the performance of the algorithm when frequently sweeping all unused interim technologies from its library. By default, the algorithm is able to construct intermediate designs that have some degree of potential usefulness and add them to a growing library of designs for reuse as components or modules in other designs (this is in addition to its library of fully-solved goals, which similarly become available for reuse). Some of these designs may not have practical use, and others may become obsolete as the run progresses and better designs are discovered. Thus, by default, the model is programmed to periodically sweep out unused parts to maintain a reasonable working set size. In this experiment, the sweep parameter was altered to frequently remove unused designs. This had the effect of deleting many new designs before they had a chance to be incorporated into other technologies, and the resulting performance of the run was much worse than the default configuration. Figure 4.7 extends this notion by disabling this library feature of temporary designs, thus forcing the algorithm to construct solutions using only the set of primitives and its earlier solved goals. This change had a drastic negative impact on performance and the model was not able to fully solve even the relatively simple design goals. These observations reinforce Arthur's findings that the algorithm succeeds only by using building blocks of simpler technologies to generate more complex designs.

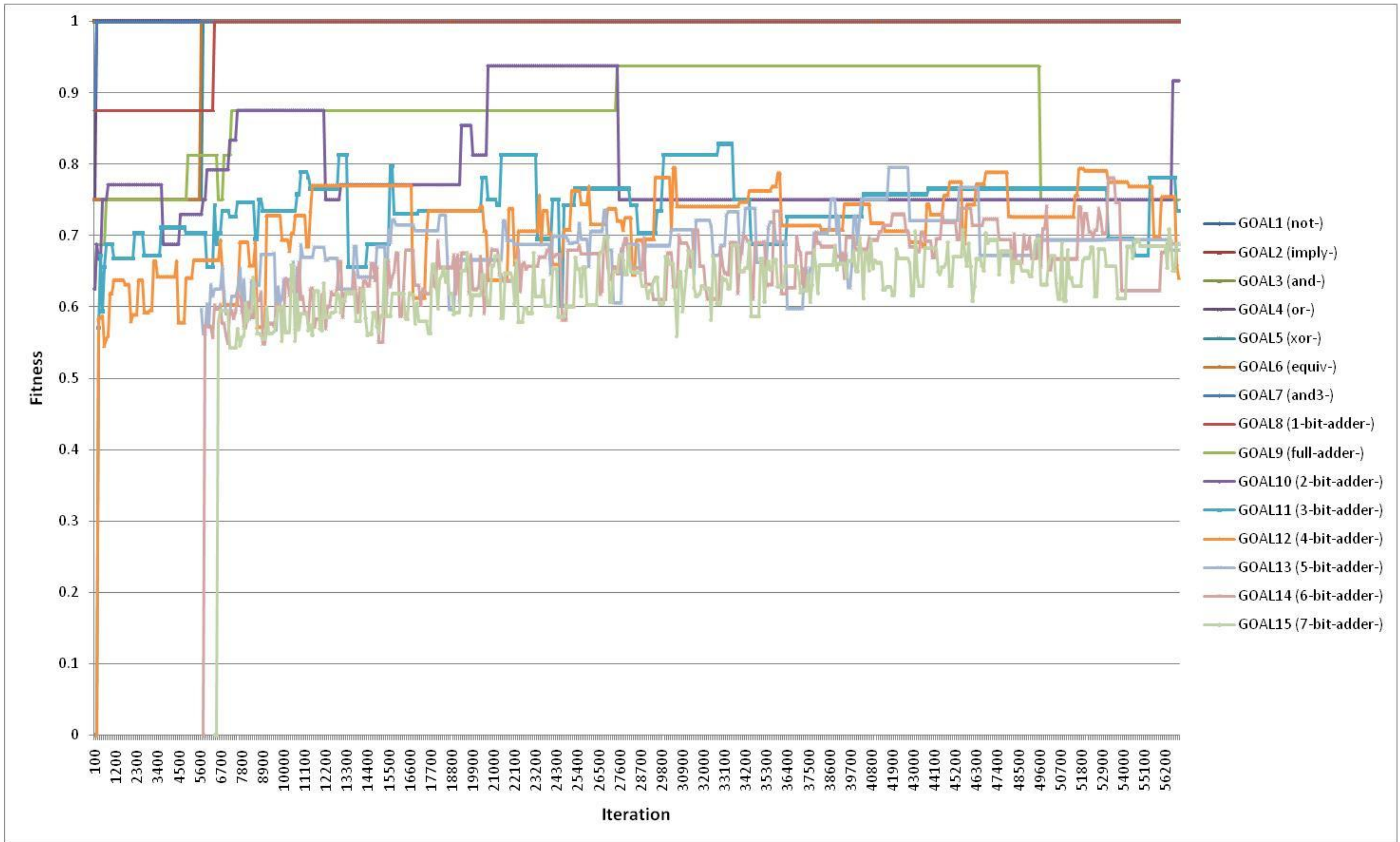

Figure 4.5: Trial 4 – Typical Run of Arthur and Polak Random Model with Maximum Concurrent Working Goals Constraint Removed

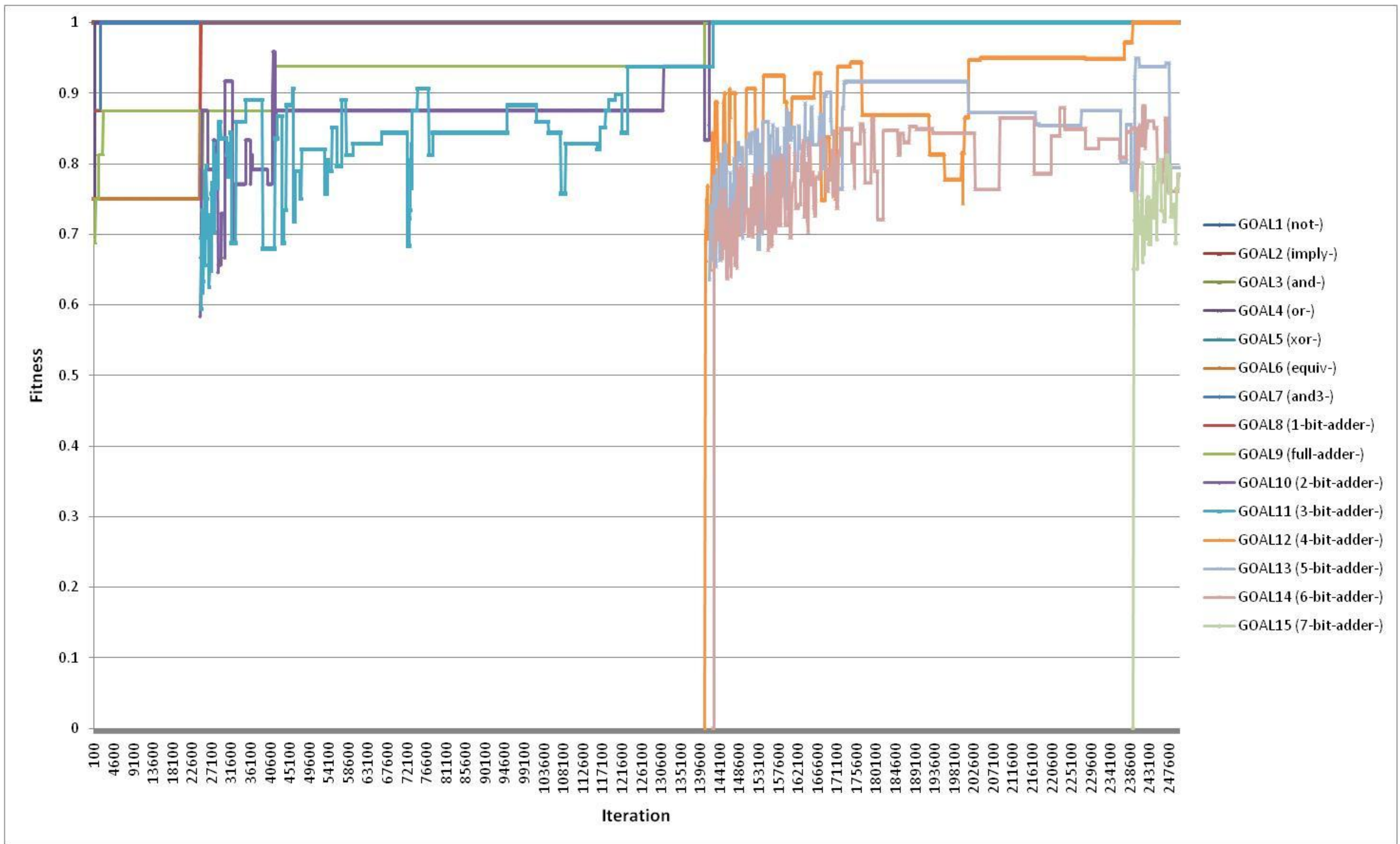

Figure 4.6: Trial 5 – Typical Run of Arthur and Polak Random Model with Frequent Removal of Unused Technologies

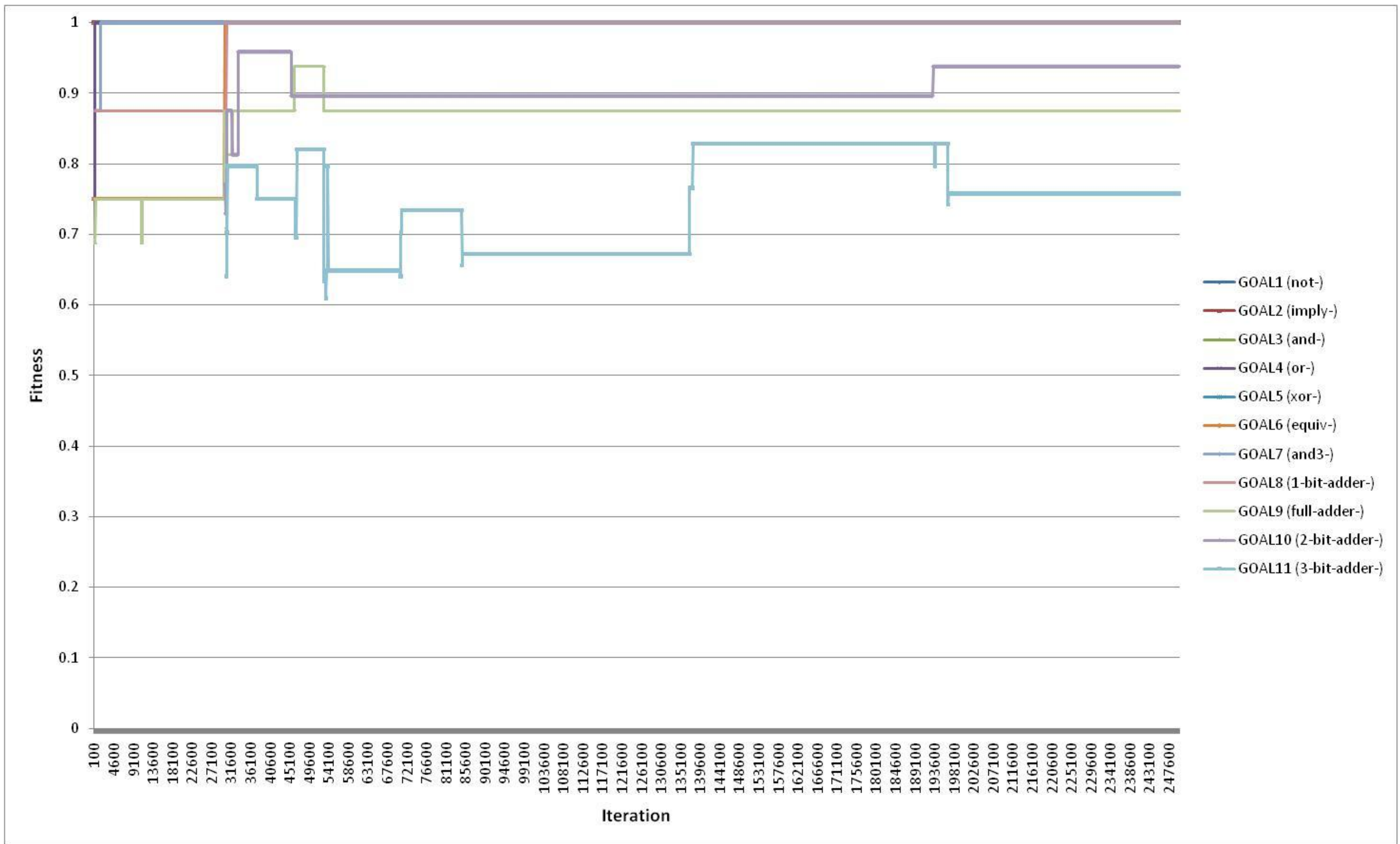

Figure 4.7: Trial 6 – Typical Run of Arthur and Polak Random Model Using Only Prior Solved Goals as Available Technology Library

Figure 4.8 shows the effect of changing the "max complexity" parameter within the algorithm. This limit controls how many components may be selected for assembly into a potential design candidate. The default limit is 12 components, and for this test it was doubled to 24. This had a positive impact on the algorithm run and this trial produced the best results of any test. Technology goals were solved earlier in general and some of the most complex goals were completely solved in this trial that were never completely solved in other experiments. Also of interest here is the observation that fitness score progress on the technologies tended to have much less random fluctuation than in earlier trials, or in other words, the scores tended to increase monotonically or have plateau periods with occasional drops instead of the jitter observed in other runs. However, as Figure 4.9 shows, this particular parameter is apparently subject to some optimality condition as drastically increasing the limit to 100 resulted in an unexpected degradation of performance.

Another matter of interest in this thesis was the rate of advancement of technological progress and whether, for example, the model would show an exponential (or at least an increasing) rate of improvement in the virtual technologies being developed. In the real world, this is typically measured using some functional performance metric, such as cost, size, weight, speed, or some other measurable parameter of interest. However, virtual digital circuits as a technology proxy have the disadvantage of bearing no obvious metric as a measure of value. Intuition suggests that an 8-bit adder (Goal 16) is more valuable than a 4-bit adder (Goal 12), for instance, but the degree of additional value provided by the more complex technology is not clear. Two potential concepts for value measurement were considered as possible metrics, and both are related to the size of the circuit as measured by the number of output bits it computes. The scaling values considered were: **`Fitness * N`**, and **`Fitness * 2`**^{N} (where **`N`** is the number of output bits of the circuit) as shown in Figures 4.10 and 4.11. Data from Trial 3 was used here as a representative sample to apply scaling. This did not, however, change the fundamental relationships among the technology progressions.

In these charts, progress of the innovation process is being achieved at a slowing rate, not at an increasing rate as might have been expected. Part of the likely problem with this metric is that fitness, as computed in this study, is fundamentally a measure of correctness of the design rather than the capability or value of the design. A more appropriate measure of the true effects of innovation occurring within this model is still needed. However, it also likely that the models

are not strong enough replicas of engineering invention over time to fully describe the empirical world. This motivates the search for a stronger model.

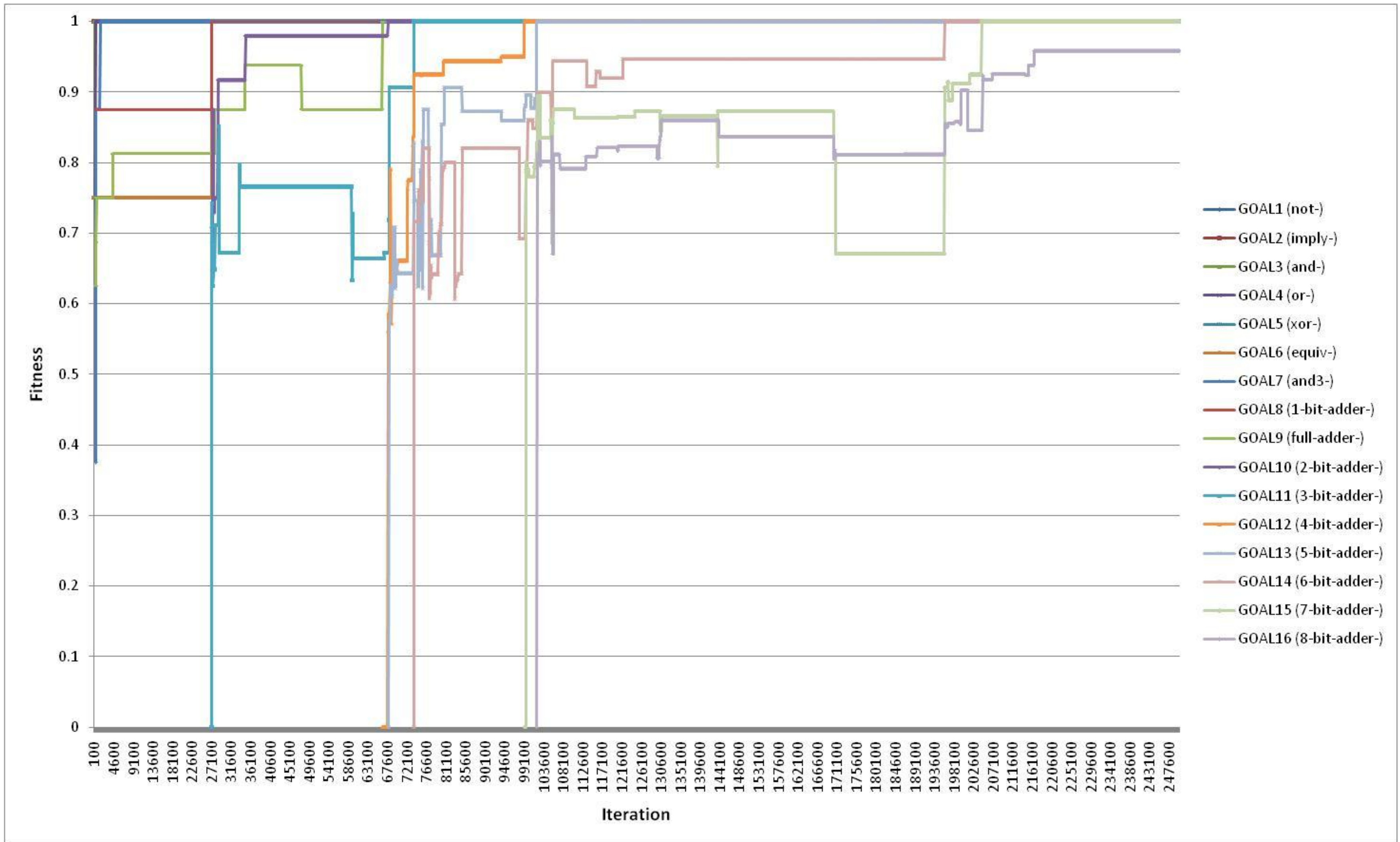

Figure 4.8: Trial 7 – Typical Run of Arthur and Polak Random Model Using Doubled "Max Complexity" Parameter

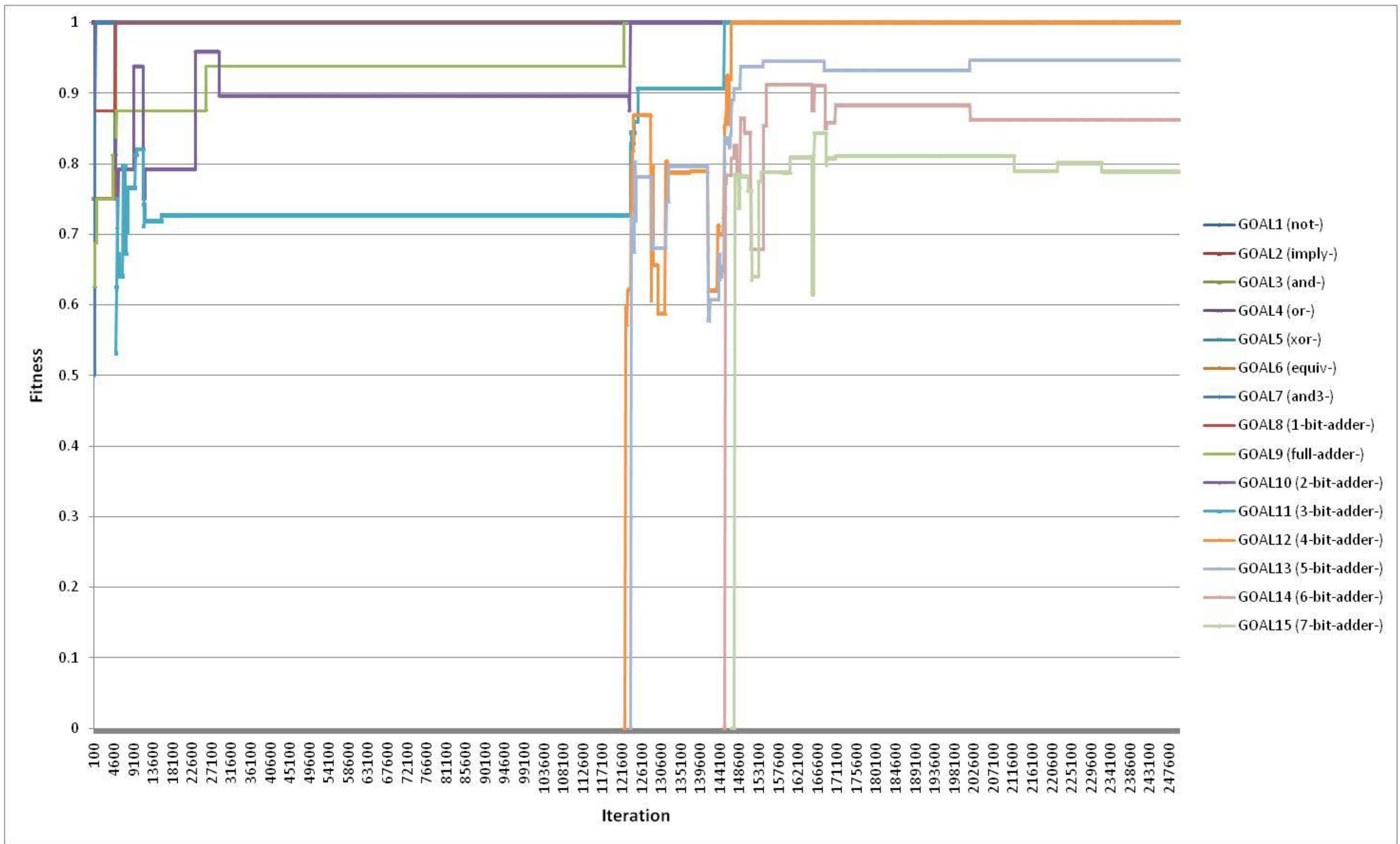

Figure 4.9: Trial 8 – Typical Run of Arthur and Polak Random Model Using "Max Complexity" Parameter = 100

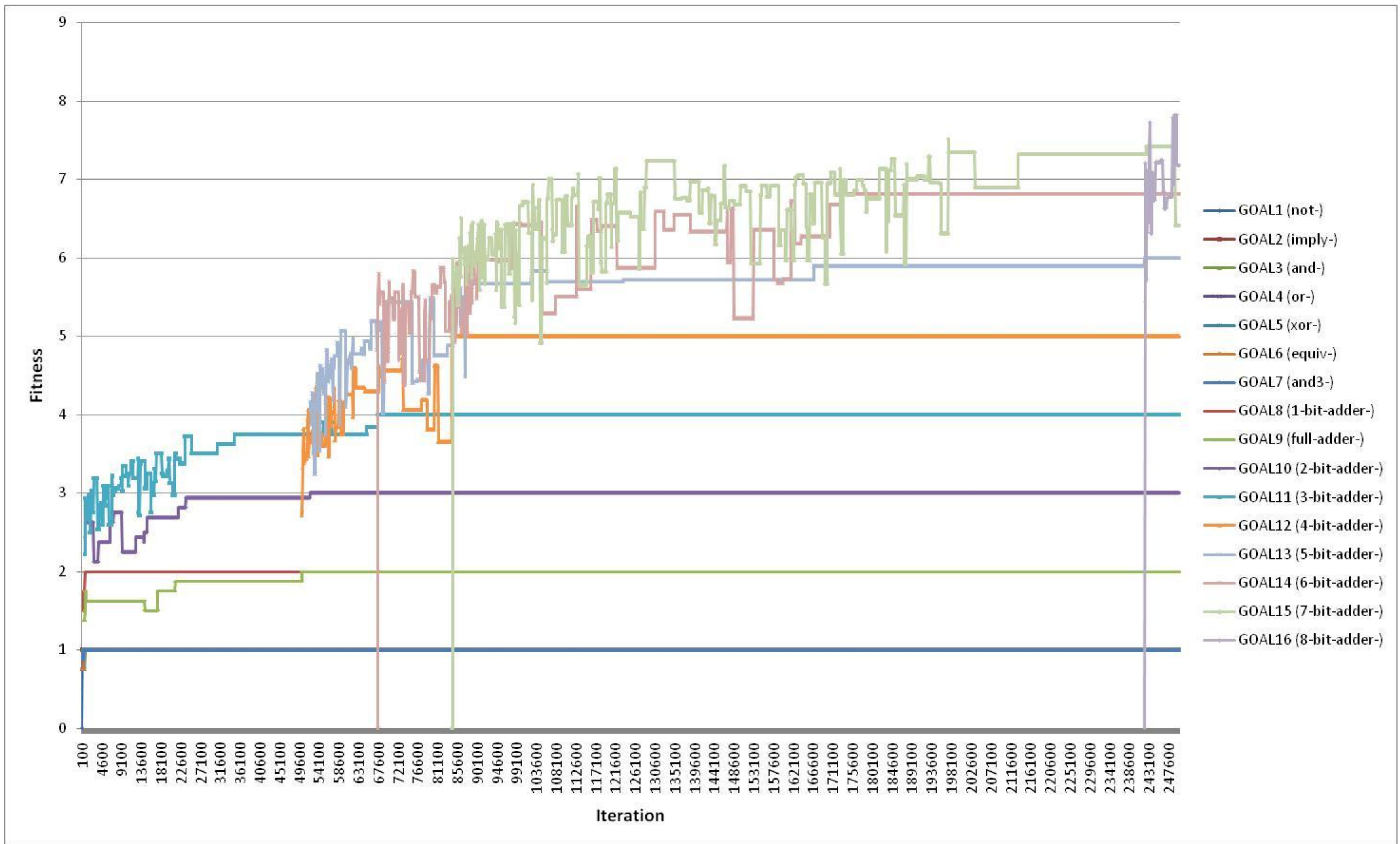

Figure 4.10: Trial 3 Data Scaled by Circuit Output Size

Figure 4.11: Trial 3 Data Scaled by 2^N * Circuit Output Size

4.3: Genetic Programming

The ECJ genetic programming framework is, in its default state, able to handle the standard genetic programming functions and operators. The only modification required to the code is to define the problem at hand (in this case, digital circuits) and to customize the GP engine to the problem-specific parameters as needed. See Section 3.2.2 for a discussion of the ECJ framework.

Results of several executions with different parameters are shown in Figure 4.12. The data is not directly comparable to the early results from the Arthur and Polak model due to core differences in the algorithms, since GP operates on a population of solutions simultaneously rather than a single point at a time. Thus, the fitness curves plotted here show the average of all fitness scores across the population at a given generation interval within the simulation run.

The four curves plotted reflect differing methodologies used to determine fitness scores and goal management. As discussed in Section 3.2.2, this research utilized truth table matching as the scoring mechanism for determining fitness values of individuals within the GP population. Initially, the scoring algorithm required that the length of the truth table for a given individual exactly match the length of the truth table of the goal technology before further evaluating the individual elements within the truth table for correctness. This constraint appeared to make it very difficult for the algorithm to find good-scoring designs. The length constraint was then removed in an attempt to help the algorithm find high-scoring designs (albeit designs that would be potentially incorrect due to their excess functionality). As the chart shows, this attempt slightly improved performance, but much less than expected, indicating that the length constraint was not the primary difficulty the algorithm was facing.

Another strategy that was tested was to control the number of goals that could be pursued simultaneously by the algorithm. As discussed in Section 4.2, the Arthur and Polak model restricts the number of goals that may be pursued at once, and that in testing, removing this constraint led to a decrease in performance. The GP engine was initially permitted to process all goals at once. Other experiments were then performed where only a single goal could be active at one time, such that a particular goal had to be perfectly solved before proceeding to the next goal. As Figure 4.12 shows, this generally worsened performance rather than improving it.

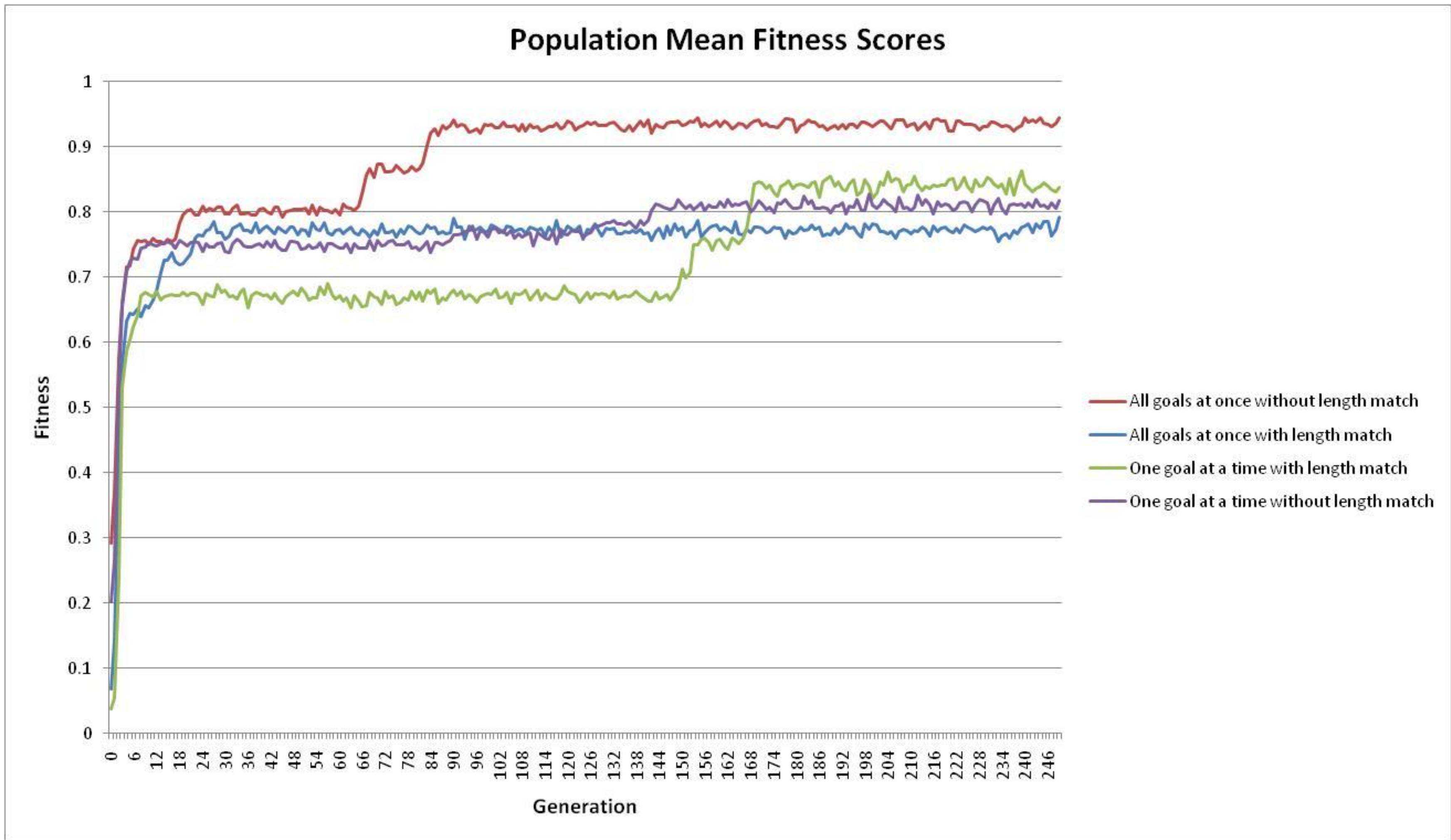

Figure 4.12: Four Runs of the Genetic Programming Model with Varying Fitness Evaluation Parameters

The use of genetic programming in this problem domain proved to be highly challenging, and ultimately, disappointing. The original hypothesis at the outset of this thesis was that GP would be able to construct complex technologies (in the form of digital circuits) just as the Arthur and Polak model does, but to do so much faster and more efficiently, perhaps even attaining an exponential rate of progress. That did not occur in the experiments for this thesis despite many attempts. In this work, the GP engine failed to generate complex circuit technologies, and did not even successfully design the mid-level "full-adder" goal (see Table 3.1). In fact, it rarely succeeded in creating more than a few of the most simplistic goal technologies, and even those were suspected to be the result of random chance rather than from deliberate operation of the algorithm. How could GP fail so spectacularly in this endeavor when several sources in the literature (see Chapter 2) indicated that GP should function more efficiently than the blind random search model, and what can be learned from this unexpected result? The remainder of this section offers several suspected causes for the failures observed based on insights gained from working with the GP framework.

- **Fitness function not smooth:** The failure of the GP engine to successfully invent digital circuits in this study is somewhat surprising given that significant work exists in the literature (see Chapter 2) with success reported in using GP to devise analog circuits. However, deeper consideration reveals an important difference between these two seemingly related problem domains. Analog circuits have the likely property of being "continuous" within their design space. In other words, a minute change to a component within the circuit is likely to result in a minute change in behavior, and therefore a small change to the design's fitness score. In contrast, digital circuits are, by definition, discrete and discontinuous, so that a small change in the design may result in either a drastic change or no change at all in the behavior of the design, and therefore a corresponding effect is induced in the fitness score. Thus, analog circuits are suspected to have "well-behaved" and relatively smooth fitness landscapes whereas digital circuits do not. Since GP works by repeatedly amplifying small superiorities within the population, irregularities in the digital circuit fitness landscape could cause difficulties for the algorithm. This is potentially a significant challenge in the use of GP as a design tool as many real-world problems likely have non-continuous aspects and/or poorly-behaved fitness features.

- **Problem/goals not stationary:** Some evidence in the literature (e.g., [Grefenstette 1992; Yang 2003]) indicates that having a non-stationary problem or fitness landscape poses a serious challenge to genetic engines. In this context, stationarity refers to whether or not the problem and its associated fitness landscape remain static during the course of the algorithm run. In the digital circuits problem used in this thesis, the problem is decidedly non-stationary, particularly when using the simultaneous active goals constraint. The implication is that as the population evolves toward the current goal, and once that goal is correctly solved, the focus shifts to another goal which may have very different characteristics, implying that the population at large is suddenly highly unfit for the new challenge.

- **No effective reuse of solved goals:** In the Arthur and Polak model, once a circuit design goal is perfectly solved, the design is added to the set of *primitives*—a small library of building block elements where it is available for explicit reuse as a module in constructing another design. But in the classic genetic programming framework used for this research, no such obvious mechanism exists. Instead, the design logic of a successfully-solved circuit goal exists within the tree-based genome of some individual in the population. It is thus available for reuse in building other designs as the genetic crossover operator mixes chromosome trees from the population to create new individuals and thus new designs. However, this exposes an important distinction between GP and the blind random search model. In the GP framework, the successful design most likely exists only inside a single individual within the large population. In order for the design to be reused, that individual would have to be selected as a parent for the new offspring design—a probabilistically low chance due simply to the size of the population. Furthermore, even if the successful individual is chosen as a parent during crossover, the only way to reuse the design encoded within its genome tree is for that individual's entire tree to be selected and grafted into the new individual. Since the splicing point where the crossover operator mixes genomes from the two parents can be chosen as any node within the tree (and it may be a large tree), this also results in a low probability of selecting the root node for crossover. Finally, both of these steps must occur to successfully reuse the solved goal design, so the product of two low-probability

events results in a very, very low chance of success. A discussion contained in [Langdon and Poli 2002] corroborates this explanation.

- **Restrictive circuit representation:** As discussed earlier, this work with the GP engine originally attempted to represent circuit designs with Binary Decision Diagrams (BDDs), just as the Arthur and Polak model does, but implementation issues forced the abandonment of that approach in favor of a more constrained representation using truth tables. Admittedly, the BDD representation is superior in many ways, not the least of which is that it allows for multiple designs to achieve the same goal. There are many possible designs of a given circuit which deliver equivalent behavior, thus there is generally more than one solution to these problems. The truth table representation used here however forces the expression of a very specific design. This had the effect of sharply limiting the freedom available to the GP algorithm, and this inevitably made the problem much harder to solve.

- **Bloat:** The issue of "bloat" was discussed in Section 2.3. It is often a serious problem within GP runs, and a significant amount of research has gone into controlling it or managing it. Bloat is the tendency of GP's tree genomes to grow in size exponentially as the run proceeds—a side effect of the unconstrained flexibility of the tree-based genome design. Once bloat begins to set in, it can quickly bring further progress of the algorithm to a halt. This was certainly evidenced in the experiments for this thesis. Often, after even the first few generations of a run, the resulting truth tables of most individuals in the population had become so large that there was no effective hope of correctly encoding even the simple unsolved goals remaining.

- **Insufficient population size:** A fair amount of research exists on the effect of population size for GP runs. In [Koza 1992], several examples of Boolean parity and multiplexer (a class of problems related to the digital circuit design problem) studies are presented which examine the minimum population size necessary to successfully find the problem solution with a certain probability. This data generally suggests that population sizes of several hundred thousand to several million individuals may be required to solve this class of problems. For the analog circuit design work that Koza has done, population

sizes of a half-million were often used, with more recent work in [Koza *et al*. 2003] using populations of 10 million or more spread across hundreds or thousands of computers. The research in this thesis used a population size of 1000—paltry by comparison. Larger sizes were attempted but became infeasible on the computing equipment available.

In addition to each of these problems described above, the issue is further complicated by complex interactions between the factors listed. For instance, the occurrence of bloat will drive up system memory consumption which in turn will constrain the feasible population size that can be run, and the population size impacts the probabilities of successful goal designs being selected as parents of new designs, and so on. Thus, future experimenters are cautioned that overcoming one or two of these hurdles in isolation may not bring the expected benefits, and could even have unforeseen negative consequences due to multiple interactions.

Chapter 5: Conclusion

This chapter presents a brief summary of the findings of this thesis. The genetic programming approach used here did not outperform the blind random search method as had been expected. Several suggested improvements are offered that may enable the genetic programming model to successfully invent complex technologies in the form of digital circuits in future work within this field. Finally, some concluding remarks are presented.

5.1: Future Work

Although the genetic programming engine used in this thesis ultimately failed to function as expected, this outcome still provides valuable insights. From an examination of the results of the experiments, several suspected causes and contributing factors to that failure were provided in Section 4.3. Building upon that knowledge, several improvements can be suggested that might enable a better outcome in future work with the GP tools.

First, additional efforts with this model should almost certainly switch to Binary Decision Diagram (BDD) representations of the circuit designs, rather than the truth table incarnation used in this thesis. This would provide the algorithm the ability to create any number of designs that correctly achieve the desired goal, rather than attempting to discover a prescribed, specific version (which is a "needle-in-the-haystack" problem). The BDD implementation has additional computational benefits as well, and likely presents a much more compact representation of the solution, thus sharply lowering system memory requirements and making equality computations faster.

Second, one or more bloat control mechanisms will likely be needed. Various proposals and means for accomplishing this are present in the literature, such as in [Luke and Panait 2004]. A few methods include: limiting the size or depth of the genome trees to a specified level; modifying the behavior of one or more of the genetic operators (such as crossover); penalizing the fitness scores of large genomes; and altering the generation of trees during the initialization process.

Third, a larger population size is needed. Some evidence in the literature suggests that the size used in this thesis is but a fraction of what is really necessary for this class of problems.

This potentially calls for larger and more powerful computing equipment. Even without different equipment, using the compact BDD representation and controlling genome bloat would have allowed larger population sizes to be run on the same equipment.

Fourth, and most strikingly, some mechanism *must* be found to achieve effective reuse of solved goals as building blocks. This feature seems to be key in the success of Arthur's work, and the lack of it is likely the main root cause of the failure in the GP work performed for this thesis. In fact, it is not at all surprising that no complex technologies emerged in this framework. This key insight comes from [Arthur and Polak 2006]:

> *"We should therefore not expect complicated circuits to appear without intermediate elements and without the simpler intermediate needs that generate these."*

In some sense, the behavior observed in the GP experiments implicitly reinforces Arthur's findings. Some mechanism that replicates the ability to reuse designs both explicitly and with high probability when building new designs would need to be constructed. Some form of Koza's Automatically Defined Functions (ADFs) or an improved variant of Module Acquisition might be suitable.

5.2: Conclusion

Genetic programming has shown itself to be a powerful tool. [Koza *et al*. 2003] and [Poli, Langdon, and McPhee 2008] report at least two patentable inventions created by GP. [Eiben and Smith 2007] recounts a recent problem where a genetic engine was given the task of designing a support boom on a spacecraft with the goal of maximum vibration dampening ability at minimum weight. The resulting design was tested to be an astounding 20,000% better than a conventional human-engineered configuration. Yet the output was a twisted, irregular, organic shape that no professional engineer would likely think of or even consider. This occurs because GP does not possess any preconceived notions or human cognitive biases about the form of the solution. This ability offers the field of engineering an exciting new realm of opportunity in the years ahead. As computing power grows ever larger, the potential to apply GP to far more difficult and sophisticated engineering challenges increases immensely. Similarly, using these algorithmic simulation models can improve understanding of the system design and innovation

processes at work in traditional efforts. With better understanding of both the technical and the human elements involved, engineers are empowered to better manage, organize, and facilitate the engineering practice. This thesis embodies the synergy of those two elements—engineering and management—in the continuing pursuit of addressing the complex challenges faced by society.

5.3: Challenges

Although genetic programming is a powerful tool, it is only a tool, not a replacement for human experience, wisdom, and judgment. Although GP has demonstrated impressive successes in bounded design domains (circuits, antennas, etc.), using it for a large systems engineering project (a central concern of this author's degree program) would be anything but straightforward. In truth, it probably exceeds current engineering and managerial capabilities. This tool should be viewed as a means of enhancing human capabilities rather than replacing them. At their core, the design process and the engineering discipline are both fundamentally socio-technical processes. To maximize effectiveness, there is a need to harness both the social aspects and technical aspects, and then integrate these into a cohesive, concrete framework through which innovation can be better understood and nurtured. As technical tools such as GP are gradually mastered, the human and social aspect becomes the next frontier—and the next challenge.

[This Page Intentionally Left Blank]

Bibliography

[Andersen 1998]
Andersen, H. R. (1998.) "An Introduction to Binary Decision Diagrams." In *Lecture notes for 49285, Advanced Algorithms, E97*, Department of Information Technology, Technical University of Denmark. Lyngby, Denmark.

[Arthur and Polak 2006]
Arthur, W. B. and Polak, W. (2006.) "The Evolution of Technology within a Simple Computer Model." In *Complexity*, vol. 11, No. 5, pp. 23–31. Wiley Periodicals, Inc.

[Arthur 2007]
Arthur, W.B. (2007.) "The Structure of Invention." In *Research Policy*, 36, pp. 274–287. Elsevier B.V.

[Arthur 2009]
Arthur, W.B. (2009.) *The Nature of Technology: What It Is and How It Evolves*. Free Press.

[Banzhaf *et al*. 1998]
Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998.) *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann.

[Eiben and Smith 2007]
Eiben, A.E. and Smith, J.E. (2007.) *Introduction to Evolutionary Computing*. Springer-Verlag.

[Frey *et al*. 2008]
Frey, D. D., Herder, P. M., Wijnia, Y., Subrahmanian, E., Katsikopoulos, K., and Clausing, D. P. (2008.) "The Pugh Controlled Convergence method: model-based evaluation and implications for design theory." In *Research in Engineering Design*, vol. 20, No. 1, pp. 41–58. Springer-Verlag.

[Goldberg 1989]
Goldberg, D. (1989.) *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

[Goldberg 2002]
Goldberg, D. (2002.) *The Design of Innovation: Lessons From and For Competent Genetic Algorithms*. Kluwer Academic.

[Grefenstette 1992]
Grefenstette, J. J. (1992.) "Genetic algorithms for changing environments." In Manner, R. and Manderick, B., eds., *Parallel Problem Solving from Nature*, 2, pp. 137–144. Amsterdam, North Holland.

[Hassan *et al*. 2005]
Hassan, R., Cohanim, B., de Weck, O., and Venter, G. (2005.) "A Comparison of Particle Swarm Optimization and the Genetic Algorithm." In AIAA 2005-1897, *46th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, Austin, TX, April 2005.

[Holland 1975]
Holland, J. (1975.) *Adaptation in Natural and Artificial Systems*. University of Michigan Press.

[Kinnear 1994]
Kinnear, Jr., K. E. (1994.) "Alternatives in Automatic Function Definition: A Comparison of Performance." In Kinnear, Jr., K. E., ed., *Advances in Genetic Programming*, pp. 119–141. MIT Press.

[Koh and Magee 2006]
Koh, H. and Magee, C. L. (2006.) "A functional approach for studying technological progress: Application to information technology." In *Technological Forecasting & Social Change*, vol. 73, pp. 1061–1083. Elsevier.

[Koh and Magee 2008]
Koh, H. and Magee, C. L. (2008.) "A functional approach for studying technological progress: Extension to energy technology." In *Technological Forecasting & Social Change*, vol. 75, pp. 735–758. Elsevier.

[Koza *et al*. 1999]
Koza, J. R., Bennett III, F. H., Andre, D., and Keane, M. A. (1999.) *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann.

[Koza *et al*. 2003]
Koza, J. R., Keane, M. A., Streeter, M. J., Mydlowec, W., Yu, J., and Lanza, G. (2003.) *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Springer.

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

[Koza 1994a]
Koza, J. R. (1994.) *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press.

[Koza 1994b]
Koza, J. R. (1994.) "Scalable Learning in Genetic Programming using Automatic Function Definition." In Kinnear, Jr., K. E., ed., *Advances in Genetic Programming*, pp. 99–117. MIT Press.

[Langdon and Poli 2002]
Langdon, W. B. and Poli, R. (2002.) *Foundations of Genetic Programming*. Springer-Verlag.

[Luke and Panait 2004]
Luke, S., and Panait, L. (2004.) "A Comparison of Bloat Control Methods for Genetic Programming." *Evolutionary Computation*.

[Luke 2000]
Luke, S. (2000.) *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat*. Ph.D. Dissertation, University of Maryland, College Park.

[Mitchell 1998]
Mitchell, M. (1998.) *An Introduction to Genetic Algorithms*. MIT Press.

[Poli, Langdon, and McPhee 2008]
Poli, R., Langdon, W. B., and McPhee, N. F. (2008.) *A Field Guide to Genetic Programming*. Creative Commons.

[Sastry and Goldberg 2003]
Sastry, K., and Goldberg, D. E. (2003.) "Probabilistic Model Building and Competent Genetic Programming." IlliGAL Report No. 2003013, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign.

[Sastry *et al*. 2003]
Sastry, K., O'Reilly, U.-M., Goldberg, D. E., and Hill, D. (2003.) "Building-Block Supply in Genetic Programming." IlliGAL Report No. 2003012, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign.

[Sastry, O'Reilly, and Goldberg 2005]
Sastry, K., O'Reilly, U., and Goldberg, D. E. (2005.) "Population Sizing for Genetic Programming Based on Decision-Making." In O'Reilly, U.-M., Yu, T., Riolo, R., and Worzel, B., eds., *Genetic Programming Theory and Practice II*, pp. 49–65. Springer.

[Spall 2004]
Spall, J. (2004.) "Stochastic Optimization." In Gentle, J., Härdle, W., and Mori, Y., eds., *Handbook of Computational Statistics*, pp. 170–197. Springer, Heidelberg.

[Yang 2003]
Yang, S. (2003.) "Non-Stationary Problem Optimization Using the Primal-Dual Genetic Algorithm." In *Proceedings of the 2003 Congress on Evolutionary Computation* (CEC 2003), vol. 3, pp. 2246–2253. IEEE.