Computational Complexity and the Genetic Algorithm

A Dissertation

Presented in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

With a

Major in Computer Science

In the

College of Graduate Studies

University of Idaho

By

Bart Rylander

June 2001

Major Professor: James A. Foster

Jun 7,2001

Abstract

A genetic algorithm is a biologically inspired method for function optimization that is loosely based on the theory of evolution. It is typically used when there is little knowledge of the solution space or when the search space is prohibitively large. Despite years of successful application to a variety of problems ranging from superstructure design to simulation of bacteria, little has been done to characterize the complexity of problems as they relate to this method.

Complexity is critical however. Though the metaphor is seductive, the genetic algorithm is only a simulation of evolution. The accuracy of this simulation changes daily as more knowledge is gained from the biological communities and as computer researchers find better methods for solving problems regardless of the strict adherence to the biological inspiration. In the end, the genetic algorithm is a search method that must be implemented as a program consisting of **loops** and **if** statements. Given this, it seems reasonable to evaluate whether programs developed using this method provide any advantage in terms of efficiency over other methods. To do this, there must first be a way for evaluating the complexity of problems specifically for the genetic algorithm. This technique must enable a direct comparison between the GA-complexity of problems and what is already known about the complexity of problems. In this way it is possible to evaluate whether there is a "gain" in using genetic algorithms beyond programmer ergonomics.

This dissertation describes a method for evaluating the complexity of a problem specifically for genetic algorithms. The method is used to define two specific genetic algorithm complexity classes. GA-hardness is defined as well as a method for GA reduction. In addition, the complexity of problems specifically for Genetic Programming (GP) is analyzed. Finally, the impact of quantum computers upon the complexity classes for evolutionary computation is examined. All of these areas are presented as peer-reviewed publications that have been developed as part of this research.

Contents

List of Figures

List of Tables

List of Theorems

List of Definitions

1 Introduction

The genetic algorithm (GA) is a biologically inspired method for function optimization that is loosely based on the theory of evolution. It is typically used when there is little knowledge of the solution space or when the search space is prohibitively large. Despite years of successful application to a variety of problems ranging from superstructure design to simulation of bacteria, little has been done to characterize the complexity of problems as they relate to this method.

Complexity is critical however. Knowing whether a method can efficiently solve a problem is usually the primary impetus for using the method. At the very inception of the GA the importance of efficiency was understood. Originally described in 1975, Holland wrote that though enumerative methods guaranteed a solution, they were a false lead. "The flaw, and it is a fatal one, asserts itself when we begin to ask, 'How long is eventually?' To get some feeling for the answer we need only look back ... that very restricted system there were 10^{100} structures ... In most cases of real interest, the number of possible structures vastly exceeds this number ... If 10^{12} structures could be tried every second (the fastest computers proposed to date could not even add at this rate), it would take a year to test about $3*10^{19}$ structures, or a time vastly exceeding the estimated age of the universe to test 10^{100} structures." [Hol75]

Despite this early recognition that efficiency is important, little has been done to characterize the efficiency of the GA. Most prior work has concentrated on determining the probabilistic convergence time of the GA [Ank91],[Gol89b] and on what features of representations of problem instances cause difficulty for GAs to converge [Dav91],[Whi91],[DG92]. While work to determine probabilistic convergence time has been marginally successful, the efforts to characterize what features of problems are difficult (or GA-hard) have been flawed. The reasons are multiple but stem from a fundamental misunderstanding of computational complexity and a lack of formalism in the search.

In order to determine whether a GA should be employed to solve a particular problem there must first be a method for evaluating that problem specifically for a GA. This

method must characterize the GA-specific complexity of the problem in a way that can be directly compared to that problem's traditional complexity. In a purely economical way, we must be able to answer the question: why use a GA? Currently, there is no known method for evaluating the complexity of a problem specifically for GAs. Consequently, we don't even have the means to attempt to answer the question of whether a GA should be used for a particular problem.

This dissertation presents a method for evaluating problem complexity specifically for the GA. I show how this method can be used to determine whether a problem is "hard" in the computational complexity sense. I provide definitions for unique GA complexity classes and define the term "GA-semi-hard" which refers to a problem that takes longer to solve using a GA than if some other method is used. Since a GA can be used to implement a Genetic Program (GP), the complexity of problems as they relate to GPs are analyzed leading to an intriguing observation about the advantages of Quantum Computers. Finally, the proposition of implementing a GA with a Quantum computer is explored.

The remainder of this dissertation is structured as follows: section 2 describes a simple genetic algorithm. Section 3 details the cogent essentials of computational complexity. Section 4 introduces the key publications that were produced as a result of this research, and describes the format in which they will be presented in this dissertation. Section 5 presents C*omputational Complexity and Genetic Algorithms* [RF01a]. Section 6 is *Genetic Algorithms and Hardness* [RF01b]. Section 7 is *Computational Complexity, Genetic Programming, and Implications* [RSF01], and section 8 is *Quantum Evolutionary Programming* [RSF+01]. Sections 9 and 10 discuss the implications of this research and present promising areas for future work.

The bulk of this dissertation is closely related published work. As such, many of the definitions, figures and proofs appear more than once. To avoid confusion, they will be numbered in a manner that identifies which section(s) they are in. Consequently, many will have more than one identifying number. For example, the figure of a graph that is included in sections 5 and 6 might be numbered figure 5.1 and figure 6.2. Conversely,

references for each individual publication have been combined at the end in the reference section for the entire dissertation. This was done to omit redundancy as much as possible. While this may lead to some confusion, the manner in which this dissertation is presented makes this seem the best avenue for clear exposition.

2 Introduction to the Genetic Algorithm

This section describes the common features of the simple GA. Enumerating and describing the myriad ways a GA can be implemented is a large undertaking. The purpose of this research is to show that the *problem* itself forces search space growth and that the Minimum Chromosome Length (MCL) Theory can be used as the primary way to evaluate the complexity of a problem *specifically for* GAs. Since the growth will occur with any particular collection of GA operators and their variants it is unnecessary to provide an exhaustive dissertation of them. The Maximum Clique (MC) problem is used as an example application in the depiction of the GA.

The GA consists of a number of elements. Once a problem has been identified, the elements can be broken down as follows:

- A representation of a potential solution;
- A population of chromosomes;
- A fitness function for evaluating the relative merit of a chromosome;
- A selection method;
- One or more operations for modifying the selected chromosomes (typically crossover and mutation);

These elements are then employed in an iterative process until a solution is found or a termination condition is met. An abstraction of a typical GA is given in Figure 2.1.

```
Generate initial population, G(0);
Evaluate G(0);              (apply fitness function)
t=1;
Repeat
        Generate G(t) using G(t-1);     (apply operators)
        Evaluate (decode(G(t)));
        t=t+1;
Until solution is found or termination
```

Figure 2.1: Typical Genetic Algorithm

Though it has been said that the biological metaphor is perhaps misleading, the terminology is so entrenched that omitting it in favor of more mathematical terms would serve only to confuse the issue. Consequently, to satisfy the need for rigor, mathematical definitions are provided as needed.

2.1 Notation

In order to be clear when analyzing any system it is necessary to describe the components in a way that minimizes confusion. Consequently it is important to define certain applicable notation. The usual set theory definitions apply. Non-typical definitions, definitions central to this research, and definitions unique to this field will be included throughout the dissertation.

Definition 2.1: Search Problem Instance

Given a finite discrete domain D and a function f: $D \rightarrow R$, where R is the set of real numbers, find the best, or near best, value(s) in D under f. f is referred to as the domain function [Raw90].

Definition 2.2: Encoding

A function e: $S^l \rightarrow D$ where S is the alphabet of symbols such that $\|S\| \geq 2$ and $l \geq \lceil \log_{\|S\|} \|D\| \rceil$. The encoding is sometimes referred to as the representation. The composition of functions f and e, is g:$S^l \rightarrow R$ where $g(x) = f(e(x))$ [Raw90].

Definition 2.3: Chromosome

A chromosome is a string of length l in S^l .

Definition 2.4: Allele

An allele is a logical component of a chromosome.

2.2 A Problem

At this point it becomes instructional to use a problem as an example of how to employ a GA. First, there must be a clear distinction between a problem, and a problem instance. Typically, the latter is simply referred to as an *instance*. A *problem* is a mapping from problem instances onto solutions. For example, consider the Maximum Clique (MC) problem (see Definition 2.5), which is to find the largest complete subgraph H of a given graph G. An instance of MC would be a particular graph G, whereas the *problem* MC is essentially the set of all pairs (G, H) where H is a maximal clique in G.

Definition 2.5: Maximum clique (MC)

Given: Graph G

Find: Largest complete sub-graph of G.

See Figure 2.2 for a pictorial example of an instance. In this case, the maximum clique consists of nodes 1,2, and 3.

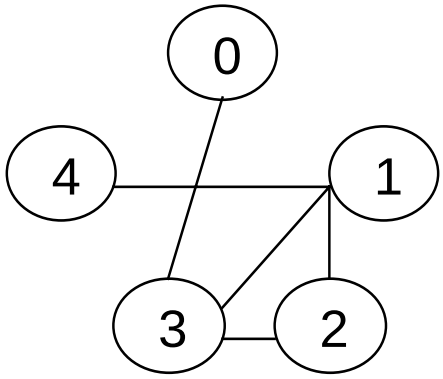


Figure 2.2: Maximum Clique Graph

2.3 Problem Representation

Informally, a representation is a way of describing a potential solution to the problem instance by using a predefined alphabet. In the case of GAs this usually means a series of

1s and 0s. It is possible to describe a GA solution using an alphabet other than binary, but since binary is the most common one, it is the one used here.

One possible representation for the MC example is to have each bit represent a node of the graph. Using this representation and the instance in Figure 2.2, each chromosome has five bits. The chromosome 11001 corresponds to a potential solution in which, starting from the right, the subset being considered as a possible maximum clique consists of nodes 0,3, and 4. Likewise, the solution to the sample problem instance is the clique of nodes 1,2, and 3, or the chromosome 01110.

Though this type of representation may seem obvious, the question of what constitutes a representation is deceptively important. Even though it is commonly defined as "a function e: $S^l \rightarrow D$ where S is the alphabet of symbols such that $||S|| \geq 2$ and $l \geq \lceil \log_{||S||} ||D|| \rceil$," this is insufficient in describing what a representation actually is. In this particular example there is a bit for each node of the graph. But this does not describe the graph in its entirety. As can readily be seen in Figure 2.2, node 0 is connected to node 3, and node 4 is connected to node 1. The question of exactly *what* features must necessarily be encoded in a representation still needs to be answered.

On casual inspection, it seems that a representation can be described as an encoding that uniquely and completely describes a potential solution to a given problem instance. This is done when relevant components of a problem instance are described in the representation. In the case of MC, each node must somehow be represented. Though the above representation will work [SF96] it seems that it somehow fails to describe important features of the graph such as the relations between nodes. Also, what representations can be developed for problems that are not so easily broken down such as in the Factoring problem (i.e. for a number n, what are the factors, if any, for n)? What are their *relevant* components? Despite the fact that these questions are posed in relation to the GA, they are in fact questions that must be answered for any method of programming. In any algorithm for solving a problem there must be an adequate way for representing that problem. Likewise, implicit in the definition of an algorithm is the notion that it must work. "More generally, an algorithm or effective procedure is a

mechanical rule, or automatic method, or programme (sic) for performing some mathematical operation" [Cut80]. Implicit in this description is the idea that the algorithm must work. Obviously if the algorithm doesn't work, it isn't an automatic method for performing any *intended* mathematical operation. In fact, sans the requirement for *working* any action whatsoever can be described as a failing algorithm for any particular mathematical operation.

This may seem like semantic wrangling. However, since this research focuses on the efficiency of the Genetic *Algorithm*, it is essential that what constitutes an algorithm be clearly spelled out. Otherwise, one could characterize a highly efficient algorithm as one that simply fails, but does so quickly. This would allow for a complete mischaracterization of a problem's complexity. (I.e. NP-complete problems can clearly be solved in polynomial time if we don't have to worry about accuracy.) In particular, since the method that will be introduced for analyzing a problem's unique GA-complexity is based on the idea of a minimum representation, it is incumbent that non-working minimum representations explicitly be excluded. Consequently, the answer for what constitutes an adequate representation is arrived at practically. Since GAs are probabilistic algorithms, a representation can be defined as in Definition 2 with the caveat that it must also probabilistically drive a population of chromosomes to a solution. In other words, it must *work.*

2.4 The Population

The population is the aggregation of chromosomes in a given generation. There has yet to be a proven or even generally accepted optimum population size for all problems [DeJ75], [Gre86], [SCE+89]. Since earlier work indicated that no single optimum existed, there has been little recent work attempting to determine such an optimum. A programmer can have a large population exceeding even the size of the solution space. Or conversely, he or she may use a small population; on the order of tens of individuals. The most notable tradeoffs are as follows: the larger the population the more quickly a solution space can be explored, the smaller the population the more quickly it can

converge to the optimum solution once it is found. Unfortunately there is no agreement as to how large *large* is or conversely how small *small* is. One theory is that the large number of chromosomes increases the chance that the optimum solution will be more quickly discovered. Problems with this theory include the fact that most problems of interest have such vast solution spaces that it is computationally impossible to represent even a small percentage (say 1 percent) of the solution space.

Despite these shortcomings, researchers have had good success with their implementations. Typically, a researcher experiments with a variety of population sizes and chooses the one that works best. In some cases the researcher will implement a dynamic population size in which the population is larger at the beginning in the discovery phase and then smaller at the end when quick convergence is desired [DeJ75]. There has been no recent work that contradicts the advantages of such a strategy.

2.5 The Fitness Function

For the Maximum Clique problem, and for any problem, the fitness of a particular chromosome can be arrived at in an infinite number of ways since there are an infinite number of equivalent representations. To see this, note that one can transform any l-bit representation into an l+1-bit one without changing the fitness function, by simply ignoring the new bit.

The important requirement for a fitness function is that the closer a chromosome is to the solution, the higher the relative fitness should be. This too, is ill defined. What constitutes closeness? Again, we must appeal to practicality. The fitness function must reward chromosomes in a manner that drives the population to the desired solution. As examples, we can describe two different fitness functions for the instance of MC in Figure 2.2.

The first fitness function which can be called "node-based" rewards chromosomes based on the number of nodes that actually represent a clique in the given graph. For example, one hundred points can be assigned to each node in a three-node clique. This means that the chromosome 01110 receives 300 of possible 300 points. Conversely, the

chromosome 10110 receives no points since it includes a node (node 0) which is not in a clique. The chromosome 00110 receives 200 points since it represents a two-node clique that is in the graph.

The next fitness function is "edge-based". In this case, the relevant features are edges between nodes instead of nodes themselves. So instead of there being a possible maximum clique of 5 nodes, there will be a possible maximum clique of 10 edges. (Obviously the same clique but measured differently.) This change highlights the relationship of representations and fitness functions. In order to use this fitness function, the representation must be modified so that all of the 10 possible edges can be included; consequently a ten-bit chromosome is required.

2.6 Selection

In selection, chromosomes are picked based on their fitness to be parents of the future population of chromosomes. In some cases, the programmer may wish to automatically pass some chromosomes into the next generation. This is called elitism, when the special ones are chosen because of their fitness. In other cases, only the selected chromosomes contribute their genetic material. In one typical selection scheme (called fitness proportional selection), the total fitness of the population is summed and a chromosome is selected probabilistically based on its relative proportion of that fitness. There are many other methods of selection, however, each with its proponents and detractors. This fact alone suggests that there is not universal agreement as to the best method [BH91], [GD91], [Han94], [DS93].

2.7 Crossover

Crossover is the method by which selected chromosomes contribute their genetic material to the next generation. A typical example of crossover is called single-point. In this case, two chromosomes are selected from the population, cut at a random point, and fragments from each parent are swapped. For example, suppose the chromosomes have 5 bits and the random cut point is between bits 2 and 3, see Figure 2.3.

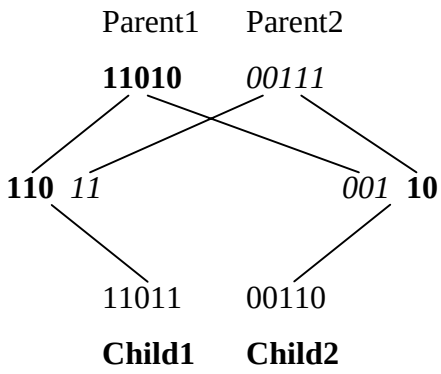


Figure 2.3: Crossover Producing Two Chromosomes

The last three bits of Parent1 are combined with the first two bits of Parent2 to create Child1. Likewise, the first two bits of Parent1 are combined with the last three bits of Parent2 to create Child2.

In addition to single-point crossover, a programmer may choose multiple-point crossover, or even no crossover whatsoever. In the case where no crossover is chosen, mutation is used as the method for chromosome modification. As with the previous options available to GA practitioners, crossover is currently a point of research and there is no universal agreement as to the optimum method [Jon95], [SD91].

2.8 Mutation

Mutation occurs when a chromosome's genetic code changes without undergoing crossover. Typically, in each generation, each member of the population will have a very small probability of one of its bits (alleles) being switched. Most commonly, mutation is implemented with a 1-% probability of each allele changing per generation. This is not universal though. Researchers have experimented with a wide range of mutation rates and generally settle on the one that works best with the problem at hand [Spe92], [Muh92]. This itself varies with the problem. In fact, it's easy to show that different mutation rates drive the population to converge more quickly simply by varying the instance size of a particular problem. This is true even when the mutation rate is per

allele. Likewise, as different crossover techniques are tried, different mutation rates produce different convergence rates.

Mutation was initially inspired by what is believed about biological evolution rather than by practical requirement. Nonetheless it seems to be effective in helping a population avoid being stranded at a local optima. In some cases, there are circumstances when the entire population has converged uniformly to a suboptimal solution. Without the presence of mutation the population has no means to change to a more optimal solution since all chromosomes would be identical.

2.9 A Sample Generation

Now that the parts have been described, it is possible to show how the algorithm works. First, randomly develop an initial population. Then apply the fitness function to rank the chromosomes. Using your chosen selection method, apply crossover and mutation to the parent chromosomes to produce the next generation. These steps are repeated until all or most chromosomes in the population represent the optimum solution or some other end criterion is met. Almost surprisingly, this method works.

Below, in Figure 2.4, is an example of the initial population of chromosomes going through one generation in an algorithm designed to solve our Maximum Clique problem instance. Selection is simply picking the top half of the chromosomes based on fitness. These individuals are automatically passed to the next generation (a form of elitism). Single-point crossover is applied to produce the remaining individuals. A 1-percent mutation rate is then applied. Notice that generation 1 is more fit, on average, than generation 0.

Step 1: Generate initial population

01010	10101	01010	00011
11011	00110	00111	01101
01000	01000	10010	00011
00100	11110	01010	11110
00001	00011	01101	10000

Step 2: Apply fitness Function

Average Fitness = 95

Step 3: Apply Selection (top half are chosen)

01010	00110
00011	01010
00011	00011
00011	01010
10000	01000

Step 4: Apply Operators (crossover and mutation)

01010	00110	00011	01010
00011	00011	01010	01000
00100	00001	10100	01000
01010	00111	00011	00000
01000	00010	00011	00011

Step 5: Apply Fitness Function

Average Fitness = 140

Step 6: Test for end condition, else Go to **Step 3**

Figure 2.4: Sample Generation

2.10 Simple GA Discussion

One of the main reasons that an adequate characterization of GA efficiency has not been achieved before is the flexibility and variability of the algorithm itself. Most research has been experimentally driven. As an example, a researcher may do an exhaustive analysis

of 2-point versus 1-point crossover. In the first study 1-point crossover may appear best. In the second, 2-point crossover appears best. Upon analysis, several difficulties typically arise. In the first case, for one method to be better than the other, all other factors must be held constant. That is, the populations must be equal, the mutation must be equal, the representation must be comparable, the PRNG must be the same, and the fitness function must not insert a bias. Since none of these features are in fact universally agreed upon, the results, even if the researcher did hold these factors constant, must be deemed inconclusive. One could simply argue that the factors that were held constant do not adequately reflect the best or most typical features of the GA. Another possibly more serious problem stems from the fact that there are an infinite number of problems. If a researcher somehow manages to prove that his or her version of crossover induces a faster convergence for the Parity problem, for example, there is no guarantee that it will induce a faster convergence for the Three Processor Scheduling problem. Like species in the biological world, each version of an operator may have a particular niche. But even this argument is suspect because of the difficulty of producing an adequate test. That being said, there have been several attempts at test suites that have achieved success in analyzing operators on a per-instance, per-representation basis [DeJ75], [FM93b], [MFH92].

Not only are there an infinite number of problems; there are typically an infinite number of problem instances for each problem. Just because we can solve MC for a five-node graph does not imply we can solve it in a timely manner (or at all) for a 700-node graph [MC98]. In addition, different population sizes may induce different convergence rates as problem instance size varies. The same may be true for variations in crossover and mutation.

Finally, and maybe most importantly, there are an infinite number of representations for each problem instance. Herein lies the real problem. When a researcher comes up with a result regarding crossover, mutation, or selection, the result must be questioned simply because a better or worse representation may have fared differently. There are currently no studies that address this fact. There has yet to be a study that claims to have

the best crossover or selection method for all possible representations, or even for the average representation. As such, all research in this area must be considered incomplete at best.

3 Background

The field of Computational Complexity is the study of classes of problems based on the rate of growth of space, time, or other fundamental unit of measure as a function of the size of the input. It seeks to answer, among other things, what problems are computationally tractable in the sense that their resource requirements do not increase prohibitively with larger instance sizes. A complexity analysis of a method of programming such as a GA seeks to answer another question as well. Namely, is there an advantage or disadvantage, in the point of view of resource consumption rates to using a particular method over others? This is in contrast to a study that seeks to identify what features of a problem instance cause the method, such as a GA, to converge more slowly. In the former, the interest is over the class of all computable problems whereas the latter is concerned only with particularly vexing features of a problem instance. To illustrate this difference, consider the problem of Sort (i.e. given a list of n elements, arrange the list in some predefined manner). If there is only one element that is out of order then a particular algorithm may converge more quickly than if all the elements are out of order, as with the case of the well known Insertion-Sort algorithm. An analysis based on problem instances would be concerned with what features of the particular instance cause the convergence to occur more slowly such as the number of elements being out of order. The computational complexity analysis is concerned with the complexity of the *problem* over all possible instances, sizes, and representations.

3.1 Traditional Complexity

Traditionally, in order to identify the complexity of a problem, the model of computation must first be identified. Turing machines, random access machines, circuits, probabilistic Turing machines, etc. are well-studied models of computation [BDG88]. Depending upon the model of computation chosen, the complexity of specific problems may vary. For example, primality testing is not known to be possible in polynomial time on a deterministic Turing machine, but it is possible in polynomial time on a probabilistic Turing machine.

Fortunately, these differences have not led to a proliferation of theories. The strong Church-Turing thesis [Lee90], which states that one can translate from any reasonable model to any other with at most a polynomial slowdown, has been proven to hold for all interesting computation models-- with the exception of quantum Turing machines. Therefore, despite the many possible models of computation, we may provisionally proceed with an analysis of complexity assuming a deterministic Turing machine model.

Further discussion of computational complexity requires more definitions. Since many problems that GAs are used to solve are optimization problems, PO and NPO are the relevant classical complexity classes.

Definition 3.1: Complexity Class PO

> The class PO is the class of optimization problems that can be solved in polynomial time with a Deterministic Turing Machine (DTM). [BC94].

An example of such a problem is described in Definition 3.2 (Minimum Partition).

Definition 3.2: Minimum Partition

> *Given*: A = $\{a_1 \ldots a_n\}$ of non-negative integers such that for all i>j, $a_i \geq a_j$.
>
> *Find*: Partition of A into disjoint subsets A1 and A2 which minimizes the maximum of $\left(\sum_{a \in A1} a, \sum_{a \in A2} a\right)$

Definition 3.3: Complexity Class NPO

> The class NPO is the class of optimization problems that can be solved in polynomial time with a nondeterministic Turing Machine (NTM).

An example of a problem in this class is MC (see Definition 2.5). The class NPO contains the class PO. This means that problems that can be solved in polynomial time with a DTM can also be solved in polynomial time with an NTM. PO and NPO are the

optimization analogues of P and NP, which are classes of decision problems. (It is well known that P=NP if and only if PO=NPO [BC94].)

These two classes are by no means the only interesting or important ones. Nonetheless, they are sufficient for our purposes, since NPO most likely includes all tractable optimization problems. That is, these problems are computable in a reasonable amount of time.

Given a complexity class, the classical way to define a *hard* problem is to show that every problem in the class reduces to the *hard* problem. A reduction R is a mapping from problem A onto problem B in such a way that one can optimize (or decide) any problem instance *x* of A by optimizing (or deciding) B(R(*x*)). For example, there is a polynomial time computable transformation from an instance of Min-partition, which is a list of non-decreasing non-negative integers A, into an instance of MC, which is a graph G. The interesting property of such a reduction is that if one finds the maximum clique in G one can recover the minimum partition of A. However, there is no known reduction from MC to Min-partition. *Hardness* then, is an upper bound relative to the reduction R, since one need never do more work to solve any problem in the class than to transform the instance and solve the instance of the hard problem. If MC were reducible to Min-partition, for example, this would place an easy (PO) upper bound on a hard (NPO) problem—implying that MC wasn't so hard after all. The formal definition of hardness is included in Definition 3.4.

Definition 3.4: Complexity, Hard

> A problem H is Hard for class C if, for any problem A$\in$C, and any instance *x* of A, there exists a polynomial time transformation R such that optimizing R(H(x)) optimizes A(x).

Hardness in this sense is a rigorously defined concept that can only correctly be used in conjunction with a specified class of problems and a specified reduction. This contrasts with the usually subjective notion of the term in current GA theory. One contribution of

ours is to develop an appropriate reduction and notion of hardness for GAs (see chapter 6).

3.2 Kolmogorov Complexity

In addition to the more traditional complexity theory, Kolmogorov Complexity analysis seems uniquely suited for application to genetic algorithm implementation. It was created, among other reasons, to provide a way to evaluate objects statically. This section details some of the definitions and theorems of Kolmogorov Complexity that are pertinent to this study.

Informally, the amount of information in a finite object (like a string) is the size of the smallest program that, starting with a blank memory, outputs the string and then terminates. As an example, consider pi. It is a provably infinite series. And yet, despite this infinity, it can be represented in much less than an infinite string. In fact, the complexity of pi is $\in O(1)$. Another example is an infinite sequence of 1's. Clearly, the smallest string that can output a repeating sequence of 1's is not infinite. Below is the formal definition for an object's Kolmogorov complexity.

Definition 3.5: Kolmogorov Complexity, K

> $K_S(x) = \min \{|p|: S(p) = n(x)\}$. The Kolmogorov complexity of an object x, using method S, is the length of the smallest program p that can output x (where n(x) represents the numbers of some standard enumeration of objects x) and then terminate. [LV90]

Therefore, K(x) is the length of the shortest program that outputs the string x. A question that should be asked is whether K exists. Though K is not computable, it does indeed exist for every string. Below is the Invariance Theorem (due to Solomonoff, Kolmogorov, and Chaitin) which states the existence of K. [LV90].

Theorem 3.1: Invariance

There exists a partial recursive function f_0, such that, for any other partial recursive function f, there is a constant c_f such that for all strings x,

$K_{f0}(x) \leq K_f(x) + c_f$.

In addition to the existence of K, Kolmogorov Complexity theory also provides two important tools for our investigation. Firstly, some strings are incompressible (see Theorem 3.2). Secondly, random strings are themselves in this category.

Theorem 3.2: Shortest Program Incompressibility

If p(x) is defined as the shortest program that outputs x, it is incompressible in the sense that there is a constant $c > 0$ such that for all strings x, $K(p(x)) \geq |p(x)| - c$.

Given these definitions a number of points may jump out. Two that should be stressed are that the length of the shortest program that produces an incompressible string is greater than or equal to, plus a constant, the length of the string itself; and that a shortest program is itself incompressible.

3.3 Previous Work

The last thing that must be done before describing new work is to provide a background from which to objectively observe the study. In fact, little work has been published that describes any aspect of the computational complexity of problems specifically for GAs or even of any available method or technique for evaluating such complexity. Tangential work that on casual glance seems to address complexity, on closer inspection typically turns out to evaluate convergence time for a particular problem instance for a particular GA implementation. Even the No-Free-Lunch (NFL) theorems don't address the complexity of problems as they relate to GAs [WM95], [WM96a], [WM96b]. Rather, they address all possible algorithms over all possible problem instances of a fixed size then show their results are independent of that size. The caveat for their work is that each

search algorithm must be fixed. With respect to GAs, this means that all possible search problems must be addressed with the same fitness function. This isn't even possible to do. The main significance of this result is that there is no "silver-bullet" that can solve all problems efficiently. This does not address the efficiency of the GA to any particular problem or provide a way to determine what problems exist that GAs are "good" at solving or conversely "bad" at solving, in terms of efficiency.

There is currently no published work that specifically addresses GA convergence for a *problem* allowing for *any* representation (i.e. complexity work). This appears to be a gaping lapse in the field. To fully appreciate this lapse, it may help to imagine a discussion between the GA community and the rest of the computer science community. Consider the essential arguments.

> **GA community:** We have developed a new programming method for difficult problems.
> **Response:** Great! Is there any advantage over current methods?

Despite 35 years of research and application development, there appears to have been no attempt to answer this obvious question. One reason for this is that it has been very difficult to even develop a theory to describe the behavior of the algorithm. Another reason, which must be regarded as speculation, is the thought that since biological evolution has been so successful, the evolutionary metaphor must be useful. Also, many researchers have not been attempting to claim that GAs are useful for function optimization at all. Rather, their focus has been to *understand* biological evolution by using GAs as a model. Even for *those* researchers however, it seems that understanding the computational complexity of problems specific to GAs would be valuable, if only to provide a greater understanding of evolution.

The apparent success of evolution in our everyday world is widespread. In terms of nature, economics, politics, and everyday capitalism, survival-of-the-fittest seems to be a tremendously successful mechanism. As earlier computer researchers contemplated

areas of interest that would be tremendously useful if they were only computationally tractable, it seemed logical to try to mimic nature.

Unfortunately, the simulation is just that. Each day biologists understand more about their field. Each day the original simulation becomes outdated. The fact that it must not have been accurate to begin with becomes lost. In addition, in recognition that the simulation was just a simulation, there have been no efforts to assert that it is anything more. Consequently, researchers routinely modify the GA to suit their computational efforts. What has been lost is the notion that the simulation should be analyzed as a program. There seems to be an underlying suspicion that since the GA is a model of evolution it must necessarily be a good idea for programming.

In recognition of this notion, research has concentrated on understanding the simulation, regardless of its accuracy to biology. There are studies to determine the best crossover method. There are studies to determine the best methods for selection, mutation, etc. As with biology the studies are done experimentally. But, rarely do these studies try to determine if their techniques match nature. Rather, they focus on whether their technique makes the algorithm converge more quickly to a solution. So in effect there is an understanding that the algorithm is just an algorithm, and yet the metaphor of evolution, and perhaps inertia, seems to insulate the community from actually analyzing the algorithm mathematically.

There have been some studies that are tangential to complexity and seem important to describe as background [Ank91], [Gol89a], [Gol89b], [Vos99], [WM95], [WM96a], [WM96b]. Though the list is by no means exhaustive, it briefly describes what are the most prevalent works of theory in this field. As mentioned before, their focus has been primarily on why the GA works. The Schema theorem is an attempt to understand how and why chromosomes converge to a solution [Gol89b]. The research into GA-hardness is concerned with what features of a problem instance cause difficulty for the GA to converge. The work with Random Heuristic Search (RHS) description jettisons the biological metaphor and describes the convergence as a series of probabilistic events [Vos99].

While this dissertation briefly describes these theories it does not purport to be an analysis of any of them. They are included only as a description of remotely tangential work. In fact, none of the theories claim to provide a method for analyzing the computational complexity of problems specifically for GAs or to provide any analysis whatsoever of computational complexity. However, they do represent the closest efforts to a complexity analysis for this field, such as they are. Below are more detailed descriptions of these works.

3.3.1 Schema Theorem

The Schema Theorem [Gol89b] and the closely related Building Block hypothesis [Hol75] were developed as ways for understanding how and why a population of chromosomes converge to a solution. Essentially they both assert that subsets of the chromosomes are responsible for the convergence. An example seems instructional at this point. Referring back to the MC problem where the node-based fitness function is used, the Schema Theorem asserts that a subset of the chromosome 01110 (which is the optimum solution) is the string *111* where *s indicate a “don’t care”. The string *111* is called a hyperplane and consists of all the strings that have 111 in the bit positions 1,2, and 3. The Schema Theorem asserts that this hyperplane is rewarded by the fitness function more heavily than a hyperplane such as *010* and thus steers the convergence toward the optimum solution. It is a compelling idea and one that seems to provide adequate predictions for how and why a population converges. The notion of the hyperplane seems to suggest that during convergence the GA is exploring a subset of 3^l hyperplanes while only using a subset of 2^l steps. This is the implied advantage of the evolutionary metaphor. Recently this notion has come under attack as simply being unproved wishful thinking [Vos99], [WM95], [WM96a], [WM9b]. Despite these attacks, it is possible to see the Schema Theorem as a good way to "think" about what is happening during execution without relying on the idea that the GA is somehow plumbing an exponential amount of information in a polynomial amount of time.

3.3.1.1 Expected Convergence

Using the Schema Theorem as a foundation, several proofs have been developed that describe the expected convergence time of a GA. Figure 2.1 is the same GA that was used to prove worst-case and average-case convergence time [Ank91]. The many assumptions used in these proofs mostly parallel those used to derive the Schema Theorem itself. Ankenbrandt shows that, with proportional selection, GAs have average and worst case time complexity in $O(Evaluate \times m \lg m / \lg r)$, where m is the population size, r is the fitness ratio, and *Evaluate* represents the combined domain dependent evaluation and decoding functions [Ank91]. Though these are in fact two separate functions, they were referred to in the proofs jointly as the evaluation function. Adding Mutation to the formula was deemed cumbersome with no increase in fundamental understanding. Notice in particular that the length of the chromosomes and the evaluation function have dramatic effects on the overall GA complexity. Unfortunately, this work ignores interesting possibilities, such as that the evaluation function might be infeasible, or that the decoding function might be uncomputable.

It should be stressed that these formulae describe the complexity of the GA, rather than that of a particular problem. To illustrate, the complexity of BubbleSort is in $O(n^2)$ whereas the complexity of the *problem* Sort is in O(nlgn). More importantly, however, they do not describe what the *GA-complexity* of a specific problem is. They are excellent however, in providing a baseline in the sense that they characterize how a GA will react to *any* problem (probabilistically) under the given assumptions. GAs that do not meet these assumptions because they are in fact slightly different algorithms may well have slightly different complexities depending on the specific differences. Nonetheless, they would still be algorithms not problems. Therefore, to provide a provable bound on a problem there must be a bound on the evaluation function as well as a manner by which the underlying problem can be tied to the representation.

3.3.1.2 Empirical Results

In addition to the proofs developed by Ankenbrandt, there are numerous results from many researchers that indicate that GAs converge in quadratic time. At this point it is important to be *very* careful about what is discussed. In the first case, the collection of results that indicate quadratic time are not part of one orchestrated research project [LGP00]. Consequently it is unclear exactly what the accumulation of results actually indicate. A much more serious problem with this work is that there has been no attempt to insure that the terminology is uniform. "Quadratic convergence" talk coupled with nonstandard usage of the word "class" have caused a great deal of misunderstanding, and perhaps fed the implication that something magical was happening with this “evolutionary method”. The term quadratic, in this case, only refers to quadratic convergence on a given instance. The only bearing this has on computational complexity is that it provides a constant in front of the big O function that truly describes the value of the algorithm. The term *class* that is frequently being used in this community does not refer to a formal computational complexity class of problems, but instead refers to an informal group of problems that may or may not have similar complexities.

Despite these difficulties, the work is meaningful. Determining the probabilistic convergence time for the GA, in fact, would further enhance the theory of complexity for GAs. Though the growth of the search space will be seen as the most important factor in the computational complexity of problems as they relate to GAs, knowing exactly the expected convergence time per instance size is valuable as well.

3.3.2 GA-hardness

Early on it became clear that GAs might not be an effective tool for every problem. It seems logical that if GAs are in fact *good* at solving some problems they may in fact be *bad* at others. The NFL theorems prove this to be true [WM95], [WM96b]. Essentially the term "GA-hard" arose from a search for these problems.

The search for these problems began by first identifying what would be the hallmark of a GA-hard problem. This hallmark would of course be a problem that either

consistently failed to converge or consistently converged more slowly than expected. This search has academic as well as practical benefits. Knowing when and where to best use a GA seems a requirement to effective application as well as tantamount to a clear understanding of the underlying theory.

Efforts in this search have produced fruitful results in the analysis of representations. Researchers have shown that representations which are deceptive or epistatic may in some instances cause a failure to converge or a slowing of convergence [Whi91], [Dav91]. The flaw to the argument that these efforts address a universal "GA-hardness" is that they fail to address the fact that there are an infinite number of representations for each problem instance. Just because one representation exhibits deception does not mean that all representations do. In fact, the opposite is true. By merely reversing the assignments of the representation, a maximally deceptive problem instance can be made non-deceptive [Whi91]. In addition, even *with* a deceptive representation it is unclear just how harmful to convergence this is [Gre92]. So too, the research of epistasis focuses only on features of a specific representation [Dav91].

A very similar analogy can be used as an argument to illustrate the flaw with assuming that what is known about a specific representation (or algorithm) can be applied to "all" representations. Suppose a list of several hundred elements is to be searched. A researcher produces a program that searches the list serially beginning at the first element. Unfortunately for the researcher, the desired element is the last in the list. The researcher may wrongfully conclude that *any* program attempting to find the element will take several hundred steps. He may in fact compile statistics, such as the relative differences of the elements, about the problem given the program he is using. Only later may he find that in fact the list has been sorted.

While this analogy may seem somewhat simple-minded, it nonetheless exhibits the same problem as compiling statistics for one representation does. Because of this, it must not be construed that features of a particular representation are necessarily features of a problem. Knowing what features cause a representation to be GA-hard may not imply that the underlying problem is GA-hard. In fact, most researchers recognize this, and

don't attempt to expand beyond identifying particularly vexing features of specific representations. Nonetheless, it is important to explain why this prior work does not apply to efficiency or GA-hardness in general.

Further, because most previous research in this vein has been representation specific it has been difficult to achieve a definition of what a GA-hard *problem* is, beyond the notional idea of it being something that is difficult for a GA to do. Beyond representational difficulties, previous approaches to GA-hardness actually only explain why a GA is unlikely to converge quickly to a solution on a particular problem instance. Landscape analysis, for example, examines the fitness landscape of a particular problem instance. If, however, we are to understand the *a priori* applicability of GAs to problems, we need a more general approach.

3.3.3 Random Heuristic Search

Recently, a more mathematical approach has been taken to understand the convergence of GAs. Rather than relying on the evolutionary metaphor, the theories associated with Random Heuristic Search assert that GA convergence is a probabilistic event that reflects what happens as a result of a series of smaller probabilistic events. A complete description of this work is involved and daunting [Vos99] and highly recommended reading. However, this work too concentrates on providing a description of the GA and why and how it converges. It does not provide an analysis of complexity or a means for evaluating problem complexity specifically for a GA. Unfortunately, unlike the probabilistic convergence times based on the schema theorem, the mathematics used here have not yet produced a meaningful expected convergence time.

3.3.4 No-Free-Lunch

The No-Free-Lunch (NFL) theorems [WM95], [WM96a], [WM96b] assert that when averaged over all possible cost functions no search algorithm outperforms any other. In particular, if a search algorithm A outperforms another algorithm B on some cost functions, then there is an equal number of other cost functions in which B outperforms

A. The implications of these theorems seem to make the question of GA complexity moot. On closer inspection however there are a number of reasons to think otherwise.

It can be argued that the class of all cost functions is too great to be meaningful. Consider that the class of all problems (which can be presented as cost functions) is infinitely larger than the class of computable problems. This means that if you could randomly select a problem, the probability that the resulting problem is uncomputable is one. This stems from the fact that the class of computable problems is denumerable and the class of all problems is not. Since no algorithm can solve an uncomputable problem, it seems obvious that no algorithm outperforms any other over the class of *all* problems.

In addition, to fairly evaluate all search methods, the methods must remain fixed. In the case of GAs, this means that the fitness function must be generic enough that it apply to all possible problems. This isn't even possible. No researcher would attempt to apply a GA to a problem without a specific fitness function. So the requirement for a level playing field explicitly excludes an implementable representation of a GA.

It may be argued that NFL is most meaningful as a means of evaluating the biological metaphor itself as a search method (i.e. an attempt to tarnish the "silver bullet" of evolution). This however, is also somewhat less than very meaningful. All methods when confronted with the unknown apply what *a priori* knowledge they retain, but essentially the response to the truly unknown is a guess. As such, the average of all guesses will be the expected value based upon the number of available choices. Clearly, some sequence of guesses will provide better results than others. But also clearly, over all possibilities, no sequence of guesses or method for producing such sequences will outperform any other. This does not mean that biological evolution is not an efficient method for solving the problems that are most typically encountered though. It may be that the search problems most meaningful for us to solve are in fact the class of problems that evolutionary methods are good at solving.

The results of these theorems are not entirely meaningless to the complexity of problems as they relate to GAs however. Since no method outperforms any other including linear search, this means that the worst case complexity for any problem

instance for any search method, over the class of all problems, is no worse than linear search. This means that over the class of all problems, GAs and all other search methods, have worst case convergence time in O(n/2) where n represents the size of the search space. This means that a method which can characterize the growth of the size of the search space will provide the answer to the question of: what is the complexity of a problem for a GA?

4 Publications

Several publications have been produced as a result of this research. They are included as references to this work. To date they are [RF00], [RSF+00], [Ryl00], [RF01a], [RF01b], [RSF01], and [RSF+01]. The publications most significant to this dissertation are included in full in the following chapters. In addition to noting these publications, this section describes the manner in which they are presented. They are reproduced identically to how they were originally published with the exceptions that the author's names have been omitted, the format has been changed to conform to the format used in the dissertation, and all references are combined in the reference section for the dissertation. In addition to these modifications, a typographical error (pg. 49) has been corrected in *Genetic Algorithms and Hardness*. Definition 6.6 GA-semi-hard now reads:

> An optimization problem P is called semi-hard if there is a DTM M that solves it in polynomial time and $MCL(P,n) \in \Omega(n)$.

Previously the definition ended with $MCL(P,n) \in O(n)$. Finally, the original author's names and the conference proceedings in which the publications appear are included in the reference section.

The papers are presented in the order in which they were originally published. Fortunately, this is the most logical way to present the material. Each paper is the written culmination of attempting to answer fundamental questions about Genetic Algorithm theory. Each answer became the basis for more questions that, in turn, became the foundation for the succeeding paper. Below is a brief description of the ideas and questions that led to the development of each paper.

The research began as an attempt to answer the question of what GA-hard was. After concluding that no definition existed, I determined to derive my own definition. While others had focused on a problem-instance and representation-specific notion, I wanted a more general definition. In classical complexity theory, a problem that is hard, is hard relative to a class of problems and a specific reduction. Therefore, if I were to define

GA-hardness in a similar manner, I must first be able to identify classes of problems specific to GAs and also describe a method for GA-reduction. Since no method had been developed for evaluating the complexity of a problem specifically for a GA, there was no way to identify a GA-specific problem class. So two questions immediately presented themselves. These questions are: what is a method for evaluating the complexity of a problem specifically for a GA, and, what complexity classes can be developed using this method? Answering these questions became the basis for *Computational Complexity and Genetic Algorithms*.

Once a method for evaluating the GA-complexity of a problem was developed, and specific GA complexity classes had been identified, it was possible to tackle the original question of what does it mean for a problem to be GA-hard? This question was the basis for *Genetic Algorithms and Hardness*. In this paper, we describe the definition for GA-hard, GA-semi-hard, and what a GA reduction is. In addition, we identify a problem that meets the criteria for being GA-semi-hard.

Since my goal had been to find a general method for evaluating the complexity of problems specifically for GAs that is independent of representation, I had to address all problems that GAs could be applied to. This presented a difficulty since a Genetic Program (GP) could be implemented with a GA. Since GPs evolved programs rather than solutions, my previous work didn't directly apply. Consequently, answering the question of what is the complexity of a problem specifically for a GP became the basis for *Computational Complexity, Genetic Programming, and Implications*.

The work to characterize problem complexity for GPs produced a proof that showed that with respect to Kolmogorov complexity, the output solution of a GP could not contain as much information as the GP itself. An interesting feature of this proof was that it did not hold if the GP had been implemented on a quantum computer. Because of this result, and because it was theoretically possible to implement a GA on a quantum computer, it seemed prudent to explore how this could be accomplished. This question was the basis for *Quantum Evolutionary Programming*.

The results of these papers represent only a beginning of an understanding of computational complexity as it relates to the genetic algorithm. They form a foundation from which more questions can be posed and implications can be explored. Many of the implications are discussed in section 9. Section 10 provides a brief description of possible ideas for future research.

5 Computational Complexity and Genetic Algorithms

Rylander, B., Foster, J., Computational Complexity and Genetic Algorithms, *Proceedings of the World Science and Engineering Society's Conference on Soft Computing, Advances in Fuzzy Systems and Evolutionary Computation*, pp. 248-253, World Science and Engineering Press, 2001.

5.1 Abstract

Recent theory work has suggested that the genetic algorithm (GA) complexity of a problem can be measured by the growth rate of the minimum problem representation [RF00]. This paper describes a method for evaluating the computational complexity of a problem specifically for a GA. In particular, we show that the GA-complexity of a problem is determined by the growth rate of the minimum representation as the size of the problem instance increases. This measurement is then applied to evaluate the GA-complexity of two dissimilar problems. Once the GA-complexities of the problems are known, they are then compared to what is already known about the problems' complexities. These results lead to the definition of a new complexity class called "NPG". Future directions for research are then proposed.

5.2 Introduction

Informally, computational complexity is the study of classes of problems based on the rate of growth of space, time, or other fundamental unit of measure as a function of the size of the input [BC94]. It seeks to determine what problems are computationally tractable, and to classify problems based on their best possible convergence time over all possible problem instances and algorithms. A complexity analysis of a method of automatic programming such as a GA seeks to answer another question as well. Namely, does the method in question, with all of its method-specific computational baggage, provide an advantage or disadvantage to solving a particular problem over all other methods. This is in contrast to a study that seeks to identify what features of a problem

instance cause the method to converge more slowly, or a study that seeks to analyze the expected convergence time of the method itself.

To illustrate this difference, consider the well-known problem of Sort (i.e. given a list of n elements, arrange the list in some predefined manner). If there is only one element that is out of order then a particular algorithm may converge more quickly than if all the elements are out of order, as is the case with the algorithm Insertion Sort. An analysis based on problem instances seeks to learn what features of the particular instance cause the convergence to occur more slowly (such as the number of elements being out of order). An analysis of the algorithm itself may come to the conclusion that Insertion Sort converges $\in O(n^2)$. The *computational complexity* analysis is concerned with the complexity of the *problem* over all possible instances, sizes, and algorithms. In this case, the complexity of Sort is the complexity of the fastest *possible* algorithm for solving Sort, not just the fastest *known* algorithm.

By confining this analysis to the complexity of problems specifically when an evolutionary algorithm is applied, we can compare our findings to what is already known about the complexity of problems. In this way, we may be able to ascertain when best to apply evolutionary algorithms. This paper is a continuation of an ongoing effort to understand the computational complexity of problems specific to evolutionary algorithms. In particular, it is a report that describes how to determine a problem's GA-complexity. Two example applications are described, as well as the definition of a new complexity class. Finally, future directions for research are suggested.

GAs are a biologically inspired method for evolving solutions to particular problems [Hol75]. They are typically used when a more direct form of programming cannot be found. Years of successful application to a wide variety of problems have given ample reason to be optimistic that GAs are an efficient method for solving such problems. This evidence is primarily anecdotal however. Little formal theoretic work has been done to compare the efficiency of GAs to all other methods. Most research has concentrated on what features of a problem instance cause the GA to converge more slowly, or work that seeks to describe what the complexity of the GA as an *algorithm* is. As described above,

computational complexity is a property of a problem and not an algorithm. Consequently, it seems valuable to ascertain what the computational complexity of a problem is specifically when a GA is being used.

5.3 Minimum Chromosome Length

GAs have a probabilistic convergence time [Ank91]. By repeating an experiment a number of times it is possible to determine the average convergence time (typically measured as the number of generations to convergence) for a specific GA for a specific problem instance. This average convergence may be mistaken for the complexity of the problem. However, by increasing the size of the instance we will in many cases arrive at a different convergence time. It is this *growth rate* as the instance size increases that we are looking for. For example, a problem whose convergence is in $O(x^2)$ will converge at a rate equivalent to the square of the instance size. In this example, an instance size of 8 will result in an expected convergence at 64 generations. Consequently our goal is to determine the function that best describes the expected convergence time for a problem across all instance sizes.

As an example, consider the problem of Maximum 1's (i.e. for a given string length n, find the maximum number of 1's that can be held in the string). This is possibly the easiest problem for a GA to implement. Simply let each bit position represent a possible 1. Then, for each chromosome, count the number of 1's to determine that chromosome's fitness. For a string of length 5, the maximum fitness for a chromosome is 5. As the problem instance size increases, the length of the chromosome increases. This means that the size of the search space doubles for every increase in instance size because the number of possible solutions is equivalent to the number 2 raised to the length of the chromosome, or 2^l.

Observe that regardless of the expected convergence time per instance size, the growth rate of the search space determines the complexity of the problem for the GA. As an example, imagine the GA will converge after examining 1/10 of the search space. If the search space doubles for every increase in instance size, the GA-complexity of the

problem will be (1/10)* 2^{l}. This is clearly an exponential convergence time. The search space growth rate will dominate over every expected convergence rate regardless of how quickly a particular implementation can converge.

However, if the minimum chromosome length doesn't have to change *every* time the instance size increases, this indicates that the problem convergence and consequently the GA-complexity for the problem are *less* than exponential time. Note that with the problem Maximum 1's there is a smaller representation.

A representation can be created in which the maximum number of 1's in a string can be encoded in binary. This means that instead of 5 bits being required to represent a string of length 5, 3 bits will suffice (e.g. 101). The growth rate of the search space changes as well. Notice, for our new representation, we don't always have to increase the length of the chromosome. A 5 is encoded with 3 bits. Also, a 6 is encoded with 3 bits. Consequently, there is no change in convergence time when going from a string of length 5 to a string of length 6. In fact, our representation doesn't cause the chromosome length to increase until the instance size is increased from 7 to 8 (because 8 requires 4 bits, e.g. 1000). This fact alone suggests that the GA-complexity of Maximum 1's is some constant c times lg(x).

In order to validate our theoretical analysis we constructed an experiment for verification. To conduct our study we employed a typical GA implementation strategy. We chose an arbitrarily fixed population of 10 chromosomes, single point crossover, and a fixed mutation rate of 1% per bit. There was 50% elitism thus providing a 50% survival rate for each generation. Selection was implemented randomly based on the chromosomes that survived.

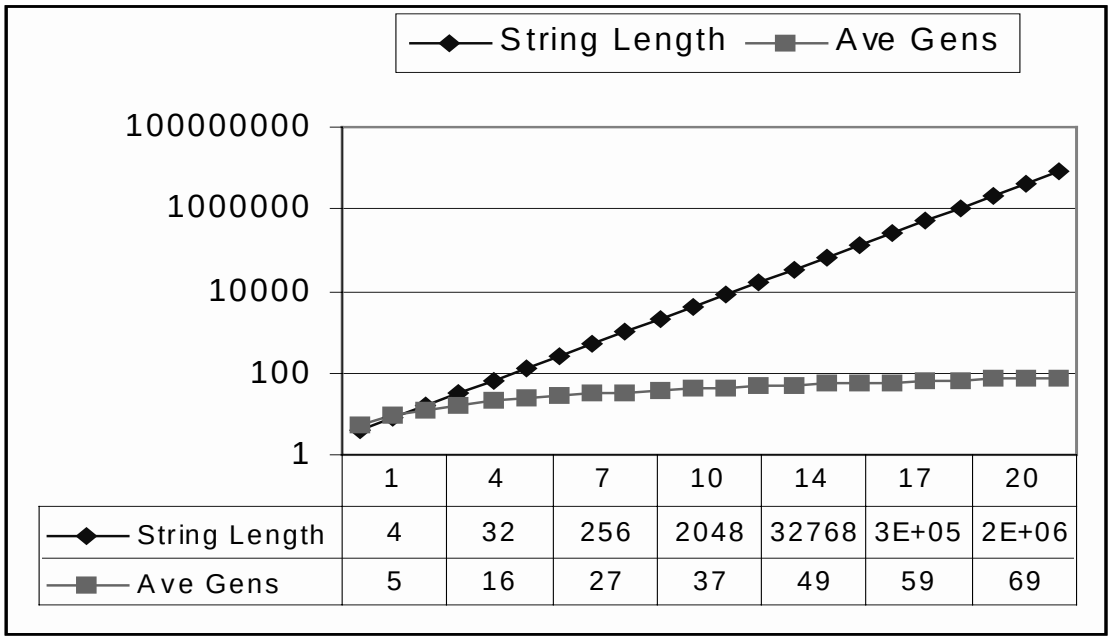
This particular GA implementation worked well with the problems in this study and in fact has worked very well for a variety of problems that it has been applied to. Note that the principle of measuring the growth rate of the minimum length representation can be used with *any* GA implementation, since what is actually being measured is the growth rate of the search space. Therefore it is not necessary to demonstrate the usefulness of this measure with *every* GA implementation.

We then let the GA run until all chromosomes had converged to the optimum solution. This was repeated 5000 times to produce an average convergence time for a given chromosome length. Then the problem instance was increased and the 5000 runs were repeated. This experiment tested the range of string lengths from 4 to 8,421,376. The length of the chromosomes ranged from 3 bits to 24 bits. See Table 5.1.

String Length	4	8	16	32	64	512	32k	8m
Chrom. Length	3	4	5	6	7	10	16	24
Average Conver.	5	9	12	16	20	30	49	77

Table 5.1: Average Generations to Converge Per Instance Size of String Length.

As can be seen, the convergence time is less than exponential. In fact, the convergence varied linearly with the chromosome length. This particular experiment had a convergence of roughly 3*|b(x)| (3 times the length in binary of x). Since the chromosome itself grew at lg(x), this means that the convergence is in O(lg(x)). It should be noted that this rate of convergence held for a very wide range. Consequently, unless it is possible to derive a smaller representation, we can assert from both experimental as well as theoretic evidence that the GA-complexity of Maximum 1's is $\in$ O(lg(x)). A graphical depiction of the convergence times over the entire range can be seen in Figure 5.1.



	1	4	7	10	14	17	20
String Length	4	32	256	2048	32768	3E+05	2E+06
Ave Gens	5	16	27	37	49	59	69

Figure 5.1: Average Convergence for Maximum 1's Problem, Logarithmic

A *desideratum* for a representation is to minimize the number of bits in a chromosome that still uniquely identifies a solution to the problem. In so doing, one minimizes the search space that must be examined to find an answer. This notion can be formalized. See Definition 5.1 for the definition of the Minimum Chromosome Length (MCL).

Definition 5.1: Minimum Chromosome Length (MCL)

> For a problem P, and D_n the set of instances of P of size n, let MCL(P,n) be the least *l* for which there is an encoding $e:S^l \rightarrow D_n$ with a domain dependent evaluation function g, where g and e are in FP (the class of functions computable in polynomial time).

That is, MCL(P,n) measures the size of the smallest chromosome for which there is a polynomial time computable representation and evaluation function. Notice that for a fixed problem P, MCL(P,n) is a function of the input instance size n, and so is comparable to classical complexity measures. Also, since a smaller representation means

a smaller number of solutions to search, there must be a limit as to how small the representation can become. So MCL(P,n) is well-defined for every P and n.

The MCL *growth* rate can be used to bound the worst case complexity of the problem for a GA. If a problem is in NPO, then the MCL growth rate will be no more than linear, since an NP algorithm can search a space which grows no more quickly than exponentially, and linear MCL growth implies exponential search space growth. Conversely, if MCL grows slowly enough, then the search space will be small enough to place the problem in PO. (PO and NPO are the optimization equivalents of P and NP, which are classes of decision problems. [BDG88]) In particular

Theorem 5.1: Sublinear MCL Growth

For any problem P, if $2^{MCL(P,n)} \in O(n^k)$ for some *k*, then P$\in$ PO.

It is currently unclear if the converse is true. However, if $2^{MCL(P,n)} \notin O(n^k)$ then the problem is most likely in NPG (the class of problems that take more than polynomial time for a GA to solve).

Definition 5.2: GA Complexity Class NPG

A problem P is in the class NPG if MCL(P,n)$\in$ O(n).

At this point it seems warranted to demonstrate the method on a problem that may fit the requirements for our new complexity class, NPG. Maximum Clique (i.e. for a graph G, find the largest complete subgraph H) is a problem that has been shown to be NP-complete [Has96]. A representation can be developed in which 1 bit can represent 1 node in the desired graph. Consider the graph G with 5 nodes in Figure 5.2. We will label the nodes 0 through 4. The maximum clique in G is the set of nodes {1,2,3}. For an n-node graph, we will use the n-bit representation that designates subgraph H, where the ith bit of the chromosome is 1 (0) if node i is (is not) in H. 01110 represents the maximum

clique in our example. This representation has been used several times for this problem [MF98].

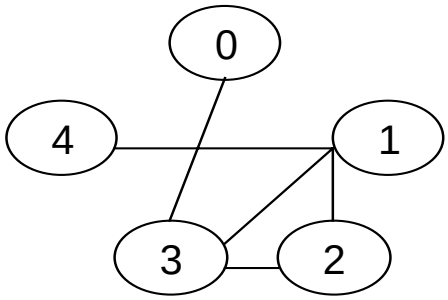


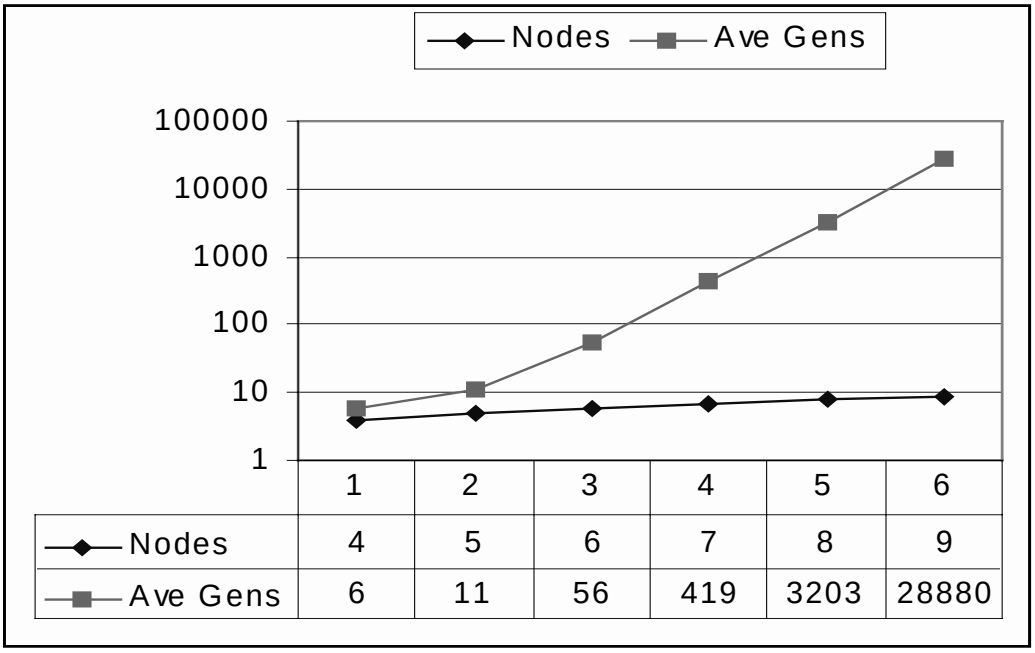
Figure 5.2: Instance of Maximum Clique (MC), with solution {1,2,3}

This representation is an MCL for a GA implementation of Maximum Clique, since any smaller representation would be unable to represent all 2^n subsets of nodes for an n-node graph, and each subset is a maximal clique for some input instance. In fact, MCL(MC,n)=n is easy to verify. Given this, we see that Maximum Clique's GA-complexity is exponential. Every time the instance size is increased, we must add another bit to our representation. To verify our prediction we used the same GA implementation that was used for the Maximum 1's problem. The experiment was conducted in the same manner. That is, for each instance size the average number of generations to convergence was calculated from a total of 5000 runs. Unfortunately, due to the exponential nature of the problem, it was impossible to view the problem over the same range of instances. Table 5.2 contains the results.

Nodes	4	5	6	7	8	9
Ave Gens	6	11	56	419	3203	28880

Table 5.2: Average Generations to Converge Per Instance Size of Graph

Clearly, this is exponential growth. In addition to the table it may be helpful to see the graph in Figure 5.3.



	1	2	3	4	5	6
Nodes	4	5	6	7	8	9
Ave Gens	6	11	56	419	3203	28880

Figure 5.3: Average Convergence for the MC problem, Logarithmic

Note that this is a logarithmic graph, which is why the convergence time seems relatively linear. In fact, using a traditional graph, the average convergence time for Maximum Clique would exhibit the classical asymptotic bend that is characteristic of an exponential problem. Note also that for Figure 5.1 (Maximum 1's) the lower line represents the average generations to converge whereas in Figure 5.3 (Maximum Clique), the upper line depicts the average generations to converge.

In both of our study problems, the GA-complexity is similar to the general complexity of the problem. Maximum 1's is trivially a problem in P. Also, since there is no known polynomial time algorithm for Maximum Clique, the exponential time required for the GA to solve it is not unexpected. Despite these facts it is not impossible or even unlikely that some problems in PO could take more than polynomial time for a GA to solve. A problem could be in PO, despite having a rapidly growing MCL and therefore a

rapidly growing search space. This would happen if, for any input instance, there was a small (polynomial in the size of the input) portion of the space that is assured to contain the solution, and this region could be identified in polynomial time. However, the only way this could happen would be if any alternative representation, which maps problem instances, requires more than polynomial time. In a sense, the *interesting* part of the search space must be efficiently hidden (by the representation) in the actual search space in such a way that no efficient algorithm can discover it. Such a function is a classical one-way function, and it is well known that one-way functions exist iff P≠NP (actually, iff P≠UP). This is very close to a proof of:

Conjecture 5.1: If P∈PO, and $2^{MCL(P,n)} \notin O(n^k)$, then P≠NP

This is also evidence that if a particular MCL grows linearly, then the underlying problem is most likely in NPO\PO (unless P=NP). After all, we explicitly disallow unnecessarily inefficient representations in our definition of MCL. Put another way, even though the encoding $e:S^l \rightarrow D_n$ minimizes *l*, the search space may be larger than the solution space. This happens if some redundant encoding is forced by the requirement that encoding and evaluation be polynomial time. In this case, it would clearly be more efficient to avoid algorithms such as GAs that require a representation. If the solution space for a problem P grows polynomially, while MCL(P,n) grows super-polynomially, then the P is clearly in PO, but a GA would be a poor choice of methods to solve P.

5.4 Conclusions

We have introduced a method (MCL) for evaluating the GA-complexity of problems. It was shown that in using this method it is possible to predict the complexity of problems specific to GAs. This was verified in two specific cases experimentally. In addition, we have laid the foundation for applying this method to other problems and for theoretical analysis. Since this method is based on the growth rate of the search space as a function of the size of the input instance, it is similar to classical measures of problem complexity.

One of the most important contributions is that we now have the beginning of a theory that will let us evaluate whether GAs are indeed efficient for *all* or even *any* problems.

There are many possible directions for future research. Our study used a fixed population of 10 chromosomes. If the population were fixed as a *percentage* of the search space as the search space is allowed to grow, it seems possible that the convergence time may remain fixed. Also, if the population were some other fixed number of chromosomes, it seems likely that there would be no difference in problem complexity. In addition, there are other problems that may now be tackled. What is a GA-hard problem in this context? In traditional complexity, a *Hard* problem is one in which all other problems in its class can be reduced to (in other words, solving an instance of the *Hard* problem would also solve the same size instance of all the others in the class, providing a polynomial time transformation existed). And what about problems that are not *Hard* for which a GA must take exponential time where other methods need only polynomial time? Would these problems be *Semi-hard*?

Finally, it is unlikely that MCL is computable, consequently each case must be proven inductively. However, it may be possible to arrive at an arbitrarily good *approximation* of a problem's MCL. But this too must wait till another day.

6 Genetic Algorithms and Hardness

Rylander, B., Foster, J., Genetic Algorithms, and Hardness, *Proceedings of the World Science and Engineering Society's Conference on Soft Computing, Advances in Fuzzy Systems and Evolutionary Computation*, pp. 323-329, World Science and Engineering Press, 2001.

6.1 Abstract

Genetic algorithm (GA) researchers have long desired to describe the key characteristics of GA-hardness. Unfortunately this goal has remained largely unmet. In this paper, we describe a new way for evaluating GA-hardness that is based on the foundations of formal complexity theory. In particular, a GA-reduction is introduced. Using this reduction, we show that a reasonable definition of "GA-hardness" is essentially the same as the definition of "hardness" in complexity theory. We then provide a definition called "GA-semi-hard" that describes a problem that takes longer to solve with a GA than it does using any other method. The problem "Sort" is shown to meet this criterion. Finally, implications and possible future research directions are discussed.

6.2 Introduction

The GA is a biologically inspired search method that seeks to converge to a solution using an evolutionary process. It is typically used when a more direct form of programming cannot be found. GAs have successfully been applied to problems from such diverse fields as economics, game theory, genetics, and artificial intelligence, to name a few [Raw91]. This broad success has prompted researchers to question whether there may be problems that do not lend themselves so easily to GAs. The goal of finding such problems has been informally described as the search to identify what is "GA-hard" [Whi91][LV90]. Unfortunately, though this search has been in progress for over twenty years, efforts have largely failed to achieve acceptance for identifying what is truly GA-hard. This, in part, is due to the fact that GA-hard has never been formally defined. Instead, most work has concentrated on trying to find problem characteristics that cause a

GA to fail to converge, or at least to converge more slowly. This, despite the fact that "converge more slowly" has never been defined as the *essence* of GA-hard. Perhaps because of this, most efforts have only produced results in the analysis of representations and not in the identification of a universal *GA-hard*. While this is significant, it should be noted that features of a particular representation are not necessarily features of a *problem*. Knowing what features cause a representation to be GA-hard may not imply that the underlying problem is GA-hard. To the contrary, it is well known that there are an infinite number of representations for every problem instance. Given this, it is easy to see that not all representations for a given problem instance necessarily exhibit the same degree of epistasis or deception [Whi91]. Another difficulty lies with the fact that there has been little attempt to distinguish between a *problem*, and a problem instance (usually referred to simply as an "instance"). A *problem* is a mapping from problem instances onto solutions [BDG88]. For example, consider the Maximum Clique (MC) problem (see Definition 6.2), which is to find the largest complete subgraph H of a given graph G. An *instance* of MC would be a particular graph G, whereas the *problem* MC is essentially the set of all pairs (G, H) where H is a maximal clique in G.

Our main contention then, is that most current approaches to GA-hardness actually explain why a GA is unlikely to converge quickly to a solution on a particular problem instance given a particular representation. Since this is clearly inadequate, it seems pressing that a new approach be described. This paper represents a shift to a more formal approach to identifying GA-hardness. Using the techniques of formal computational complexity theory, we define what a GA-hard problem is. Section 6.3 introduces basic concepts of computational complexity theory. Section 6.4 applies these concepts to the analysis of the GA, introducing several definitions specific for GAs. Finally, section 6.5 is a discussion of the implications of this work.

6.3 Complexity and Hardness

Informally, computational complexity is the study of classes of problems based on the rate of growth of space, time, or other fundamental unit of measure as a function of the

size of the input. It is a study that seeks to classify problems based on their relative computational intractability. Since the search for GA-hardness is in fact a search for degrees of intractability specifically for a GA, we begin by first describing what *intractable* means in general.

6.3.1 Complexity Classes

Since many problems that GAs are used to solve are optimization problems, PO and NPO seem the most relevant complexity classes to describe. Though P and NP may be more familiar, PO and NPO are in fact the optimization equivalents of P and NP, which are classes of decision problems [BC94].

Definition 6.1: Complexity Class PO

> The class PO is the class of optimization problems that can be solved in polynomial time with a Deterministic Turing Machine (DTM) [5].

An example of such a problem is:

Minimum Partition:

> ***Given*:** A = $\{a_1 \ldots a_n\}$ of non-negative integers such that for all i>j, $a_i \geq a_j$.
>
> ***Find*:** Partition of A into disjoint subsets A1 and A2 which minimizes the maximum of $\left(\sum_{a \in A1} a, \sum_{a \in A2} a\right)$

Definition 6.2: Complexity Class NPO

> The class NPO is the class of optimization problems that can be solved in polynomial time with a nondeterministic Turing Machine (NTM) [BC94].

For example,

Maximum clique (MC):

> ***Given*:** Graph G
>
> ***Find*:** Largest complete sub-graph of G.

The class NPO contains the class PO. This means that problems that can be done in polynomial time with a DTM can also be done in polynomial time with an NTM. These two classes are by no means the only interesting or important ones. Nonetheless, they are sufficient for our purposes, since NPO most likely includes all tractable optimization problems. That is, these problems are computable in a reasonable amount of time.

6.3.2 Hardness

Given a complexity class, the formal way to define a *hard* problem is to show that every problem in the class reduces to the *hard* problem. In other words, a reduction R is a mapping from problem A onto problem B in such a way that one can optimize any problem instance *x* of A by optimizing B(R(*x*)). For example, there is a polynomial time computable transformation from an instance of Min-partition, which is a list of non-decreasing non-negative integers A, into an instance of MC, which is a graph G with the interesting property that if one finds the maximum clique in G one can recover the minimum partition of A. *Hardness* then, is an upper bound relative to the reduction R, since one need never do more work to solve any problem in the class than to transform the instance and solve the instance of the hard problem. As an example, if MC were reducible to Min-partition, this would place an easy (PO) upper bound on a hard (NPO) problem—implying that MC wasn't so hard after all. The formal definition of hardness is:

Definition 6.3: Complexity, Hard

> A problem H is Hard for class C if, for any problem $A \in C$, and any instance x of A, optimizing R(H(x)) optimizes A(x) [BDG88].

Hardness in this sense, is a rigorously defined concept that can only correctly be used in conjunction with a specified class of problems and a specified reduction. This contrasts with the usually subjective notion of the term in current GA theory.

6.4 GA-Hardness

Having a formal understanding of classical hardness, we can begin to provide a definition of GA-hardness. It's important at this point to note what an appropriate definition should be. A useful notion will single out problems that are upper bounds on the complexity of some interesting class. For example, equating "GA-hard" with "nonrecursive" problems, or with "needle in a haystack" problems, would be singularly uninformative. Before this can be done however, there are a few items that need to be addressed. In the first place, we must describe a method for evaluating a problem specifically for a GA. Fortunately, this has already been addressed. A brief description of such a method is provided here, a more complete one can be found in [RF01a].

6.4.1 Minimum Chromosome Length (MCL)

The goal for a representation is to minimize the number of bits in a chromosome that still uniquely identifies a solution to the problem [RF01a]. In so doing, one minimizes the search space that must be examined to find an answer.

Definition 6.4: Minimum Chromosome Length (MCL)

> For a problem P, and D_n the set of instances of P of size n, let MCL(P,n) be the least l for which there is an encoding $e:S^l \rightarrow D_n$ with a domain dependant evaluation function g, where g and e are in FP (the class of functions computable in polynomial time).

That is, MCL(P,n) measures the size of the smallest chromosome for which there is a polynomial time computable representation and evaluation function. Notice that for a fixed problem P, MCL(P,n) is a function of the input instance size n, and so is comparable to classical complexity measures. Also, since a smaller representation means a smaller number of solutions to search, there must be a limit as to how small the representation can become. So MCL(P,n) is well-defined for every P and n.

It is important to stress that the domain dependent evaluation function must be feasible. Otherwise, it would be possible to skew the analysis by having an exponential time or even uncomputable function in the main loop of the GA. By bounding this function with a polynomial we ensure that the evaluation does not materially increase the worst-case analysis in terms of problem classes.

Given this, we can now provably describe the complexity of a problem instance as implemented with a GA. However, since complexity is typically based on the rate of growth of a fundamental unit of measure as a function of the input size, there is another step in our analysis. We must show how the growth of the problem instance size, n in our definition, correlates with the MCL.

As an example, consider the graph G with 5 nodes in Figure 6.1. We label the nodes 0 through 4. The maximal clique in G is the set of nodes {1,2,3}. For an n-node graph, we have an n-bit representation that designates subgraph H, where the ith bit of the chromosome is 1 (0) if node i is (is not) in H. 01110 represents the maximum clique in our example. This representation has been used several times for this problem [MF98][SFD96].

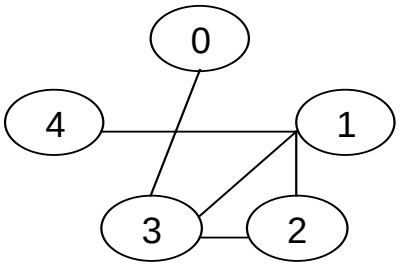


Figure 6.1: Instance of Maximum Clique, with solution {1,2,3}

This representation is an MCL for the problem Maximum-clique, since any smaller representation would be unable to represent all 2^n subsets of nodes for an n-node graph, and each subset is a maximal clique for some input instance. In fact, MCL(MC,n)=n is easy to verify.

Now, consider what happens when you add a sixth node to this graph. You must expand your chromosome to six bits. This means that the MCL grows by one as the size of the input instance grows by one. This also means that the search space has increased to 2^6. Clearly, as the size of the input increases linearly, the MCL grows linearly and the search space grows exponentially. This is the one hallmark of a hard problem. In the case of MC or any other NP hard problem, there is no known deterministic algorithm to search this exponentially growing space in polynomial time.

Recent research has shown that the MCL growth rate can be used to bound the worst case complexity of a problem for a GA [RF01a]. If a problem is in NPO, then the MCL growth rate will be no more than linear, since an NP algorithm can search a space which grows no more quickly than exponentially, and linear MCL growth implies exponential search space growth. Conversely, if MCL grows slowly enough, then the search space will be small enough to place the problem in PO.

Now that there is a method for evaluating the complexity of a problem specifically for a GA, it is possible to address what a reduction for a GA must be.

Definition 6.5: GA Reduction

> Let A and B be subsets of $\sum^*$ and T* and let *f* be some function from $\sum^*$ to T*. If |MCL(A,n)| = |MCL(B,n)| and |MCL(A,n+1)| = |MCL(B,n+1)|, we say that *f* reduces A to B, in case *f*-1(MCL(B,n))=MCL(A,n).

This means a GA-reduction is a length -preserving translation from one problem's representation to another. Given this, it is easy to see that the same definition for hardness is appropriate for GAs as it is classically. (see Definition 6.3, brought forward).

Definition 6.3: Complexity, Hard

> A problem H is Hard for class C if, for any problem A$\in$C, and any instance *x* of A, optimizing R(H(x)) optimizes A(x).

6.4.2 GA-semi-hard

One of the requirements for being GA-hard is that all other problems in a particular class must reduce to it. But this doesn't describe those problems that do not fit the classical definition of hardness and yet are inherently less efficient than if solved using a method other than a GA. More specifically, suppose a problem such as Min-partition, for which there is a known polynomial time algorithm, takes exponential time if solved with a GA (remembering of course that the problem is not simply solved in the encoding or decoding functions). Problems such as these are called GA-semi-hard (Definition 6.6).

Definition 6.6: GA-semi-hard

> An optimization problem P is called semi-hard if there is a DTM M that solves it in polynomial time and $MCL(P,n) \in \Omega(n)$.

In fact, these problems may be the most interesting, since they are clearly ones for which a GA would be a poor selection for application.

Such problems are not just conjecture. Consider the problem of Sort (i.e. given a list of n elements, arrange the list in some predefined manner). In order for a GA to perform this task, as a minimum, there must be a representation that includes an ordination of each element. For example, let our list comprise of 8,4,3,and 7. One possible representation is to produce a chromosome where 4 bits are required for each element. In this case, 1000 0100 0011 0111 represents all elements. However, it is possible to provide for an ordinal representation of the elements allowing for a translation in the decoding function. Since there are only 4 elements, you would only need 3 bits per element. In this case, the chromosome would be 100 010 001 011 (for 8,4,3,7 being translated to 4,2,1,3, their respective rank in terms of size). This results in a search space of 2^{12}. By adding a 5th element, the chromosome must grow by 3 bits making the search space 2^{15}. Clearly this is not polynomial growth. In addition, as the number of elements becomes 8 or greater, the number of bits per element has to increase to 4. So the growth

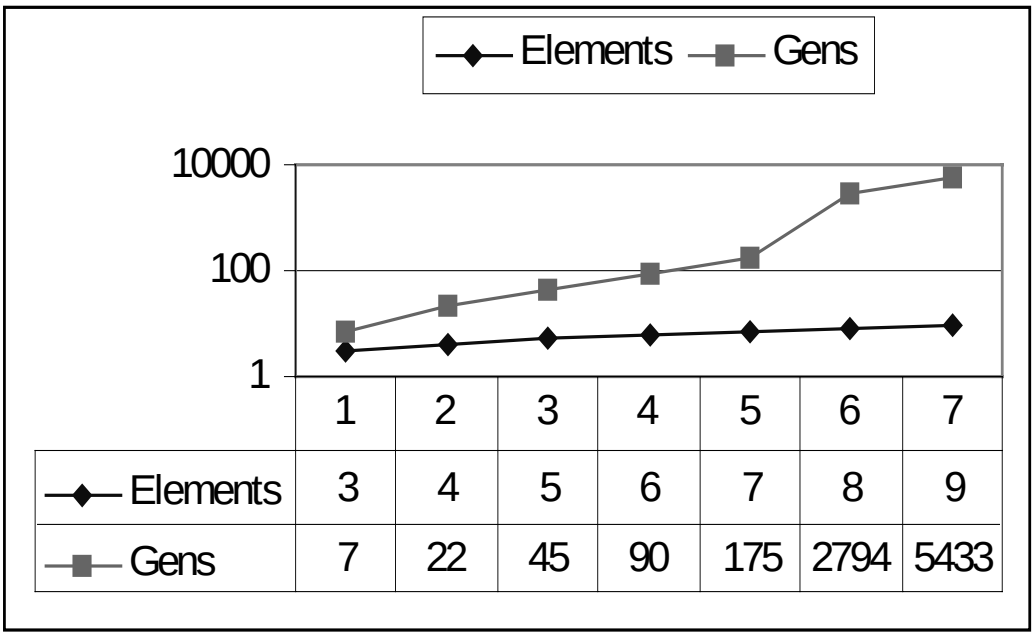
rate actually increases as the instance size does. Conducting an experiment, the results are verified empirically (see table 6.1).

Elements	3	4	5	6	7	8	9
Average Generations	7	22	45	90	175	2794	5433

Table 6.1: Average Generations to Converge Per Number of Elements to Sort.

Note that for each extra element the number of generations to converge roughly doubles. This is with the exception of the jumps from 3 to 4 elements and from 7 to 8 elements. This is because in the latter, in addition to adding an element, each individual element increased in size by 1 bit, thus causing a larger than expected jump in search space size as well as convergence time.

The experiment was conducted using a fixed population of 10 chromosomes. There was random single-point crossover, 1-% mutation, 50% elitism (thus 50% survival rate) and selection based randomly from the surviving chromosomes. Each instance size was tested 5000 times to produce the average number of generations per convergence per instance size. As can clearly be seen, Sort as implemented with a GA takes exponential time. Below is a logarithmic graph of the average convergence times.



	1	2	3	4	5	6	7
Elements	3	4	5	6	7	8	9
Gens	7	22	45	90	175	2794	5433

Fig. 6.2: Average Convergence Per Elements to Sort, Logarithmic

It may be possible to produce a smaller representation for Sort. Suppose for example, that ignoring crossover problems, you could represent each ordinal element by its own smallest representation. That is, for a 4-element list, using 100 11 10 1 as the representation. But even with this representation, each additional element requires additional bits in the chromosome. And, as the number of elements increases, the largest element will require increasingly more bits for its representation. So even with this representation, the GA growth rate for Sort is exponential. Since we know that Sort is in PO, it seems clear that Sort is GA-semi-hard.

6.5 Conclusions

We have introduced the concepts of formal computational complexity to the search for GA-hardness. In particular, we described how a GA-reduction is accomplished. With this, we gave reasons why GA-hardness is essentially the same as traditional complexity

hardness. We introduced the idea of GA-semi-hard to describe problems that take longer for a GA to solve than other methods, but yet don't meet the criterion to be GA-hard. In addition, it was demonstrated that the problem Sort is GA-semi-hard, and thus probably not a problem one would choose to solve with a GA.

By introducing these ideas, we hope to have provided a framework from which practitioners can evaluate problems for GA implementation, hopefully providing insight into which problems are suitable and which problems are not. As a result, some questions remain unanswered. For example, it might be interesting to evaluate problems on the high end of the complexity hierarchy such as NP-complete problems, or even Pspace problems to see if GAs are perhaps more efficient for problems that are inherently difficult. In the converse, it seems that the evolutionary baggage that GAs must lug may prevent them from being prudent choices for problems in PO.

Interestingly, we did not have to introduce a new model of computation for GAs in order to do this analysis. This makes it more likely that traditional proof strategies may be applicable, and it also makes it very likely that this theory is applicable to search algorithms other than GAs.

This paper raises interesting questions that need to be answered. Rather than enumerating them here, we encourage the interested researcher to find a question that interests them, then let us know the answer.

7 Computational Complexity, Genetic Programming, and Implications

Rylander, B., Soule, T., Foster, J., Computational Complexity, Genetic Programming and Implications, *Proceedings of the 4th European Conference, EuroGP 2001*, pp. 348-360, Springer-Verlag, 2001.

7.1 Abstract

Recent theory work has shown that a Genetic Program (GP) used to produce programs may have output that is bounded above by the GP itself [Ryl00]. This paper presents proofs that show that 1) a program that is the output of a GP or any inductive process has complexity that can be bounded by the complexity of the originating program; 2) this result does not hold if the random number generator used in the evolution is a true random source; and 3) an optimization problem being solved with a GP will have a complexity that can be bounded below by the growth rate of the minimum length problem representation used for the implementation. These results are then used to provide guidance for GP implementation.

7.2 Introduction

Informally, computational complexity is the study of classes of problems based on the rate of growth of space, time, or other fundamental unit of measure as a function of the size of the input. It seeks to determine what problems are computationally tractable, and to classify problems based on their best possible convergence time over all possible problem instances and algorithms. A complexity analysis of a method of automatic programming such as a Genetic Algorithm (GA) or a GP seeks to answer another question as well. Namely, does the method in question, with all of its method-specific computational baggage, provide an advantage or disadvantage to solving a particular

problem over all other methods. This is in contrast to a study that seeks to identify what features of a problem instance cause the method to converge more slowly, or a study that seeks to analyze the expected convergence time of the method itself.

To illustrate this difference, consider the well-known problem of Sort (i.e. given a list of n elements, arrange the list in some predefined manner). If there is only one element that is out of order then a particular algorithm may converge more quickly than if all the elements are out of order, as is the case with the algorithm Insertion Sort. An analysis based on problem instances is concerned with what features of the particular instance cause the convergence to occur more slowly (such as the number of elements being out of order). An analysis of the algorithm itself may come to the conclusion that Insertion Sort is $\in O(n^2)$. The *computational complexity* analysis is concerned with the complexity of the *problem* over all possible instances, sizes, and algorithms. In this case, the complexity of Sort is the complexity of the fastest *possible* algorithm for solving Sort, not just the fastest *known* algorithm.

By confining this analysis to the complexity of problems specifically when an evolutionary algorithm is applied, we can compare our findings with what is already known about the complexity of problems. In this way, we may be able to ascertain when best to apply evolutionary algorithms. This paper is a continuation of an ongoing effort to understand the computational complexity of problems specific to evolutionary algorithms. In particular, it is a report that provides proofs and analysis that describe the complexity of problems as they relate to GPs and the implications of these results.

GP is a biologically inspired method for evolving programs to solve particular problems. In addition to evolving programs, GPs have been successfully used to optimize functions [SFD98]. It is difficult to characterize the complexity of a problem specific to a method of programming. Holding all things constant, you measure what must change as the size of the input instance increases. It is even more difficult to describe the complexity of a problem that can be solved by a program that is *itself* the output of a program, as is the case with the typical GP. In general, this type of question cannot be answered. What *can* be done however, is to compare the information content

of a program with the information content of its output and in this way provide a bound on that output. This, in addition to an analysis for function optimization, is what will be described below.

7.3 Kolmogorov Complexity

Kolmogorov complexity analysis is uniquely suited for addressing this problem. It was created, among other reasons, to provide a way to evaluate objects statically. This section details some of the definitions and theorems of Kolmogorov complexity that are pertinent to this study.

Informally, the amount of information in a finite object (like a string) is the size in bits of the smallest program that, starting with a blank memory, outputs the string and then terminates. In the case of an infinite string, it is the smallest program that can continue to output the infinite series. As an example, consider pi. It is a provably infinite series. And yet, despite this infinity, it can be represented in much less than an infinite string. Another example is an infinite sequence of 1's. Though the sequence of 1's is infinite, a program to produce it can be written in far fewer bits. See Figure 7.1.

```
while(true){cout << "1";}
```

Figure 7.1: Sample program to output an infinite series of 1's

By this type of measure, a number such as 9,857,384,002 is more complex than the number 10,000,000,000 even though the second number is larger than the first. This is because the amount of *information* contained in the first number is greater than that contained in the second. This *information* can be measured by the size of the program required to output it. This is the basis for scientific notation and, in a nutshell, is the basis for Kolmogorov complexity. Below is the formal definition for an object's Kolmogorov complexity.

Definition 7.1: Kolmogorov complexity, K

$K_S(x) = \min \{|p|: S(p) = n(x)\}$. [LV90]

This says that the complexity of an object x, is the length of the minimum program p, using programming language (method) S, that can output x and then terminate. Other proofs, which can be found in [LV90], show that the complexity of an object is a property specific to that object, thus justifying dropping the subscripts and becoming:

$$K(x)=\min\{|p|:p=n(x)\}$$

A question that might be asked is whether K exists for every object. Though K is not computable, it does indeed exist for every string. Below is the Invariance Theorem (due to Solomonoff, Kolmogorov, and Chaitin) which states the existence of K. [LV90].

Theorem 7.1: Invariance

There exists a partial recursive function f_0, such that, for any other partial recursive function f, there is a constant c_f such that for all strings x,y, $K_{f0}(x|y) \leq K_f(x|y) + c_f$.

Again, though a more detailed description can be found elsewhere, for the purposes of our study we will provide a brief explanation before moving on. y is included because it was easier to prove the complexity of an object x *given* an object y. f is called the *interpreter* or *decoding* function that provides f(p,y)=x. In this case, f refers to the universal partial recursive function computed by a universal Turing machine *U*. Since any f_0 that satisfies the Invariance Theorem is optimal, we can fix a particular reference machine *U* with its associated partial recursive function f_0, and are justified in dropping the subscripts. We can then define the unconditional complexity $K(x|\varepsilon)$ where ε denotes the empty string ($|\varepsilon| = 0$). Consequently the complexity of x can be expressed as K(x).

In addition to the existence of K, Kolmogorov complexity theory also provides two important tools for our investigation. Firstly, some strings are incompressible (see Theorem 7.2). Secondly, random strings are themselves in this category.

Theorem 7.2: Shortest Program Incompressibility

> If p(x) is defined as the shortest program for x, it is incompressible in the sense that there is a constant $c > 0$ such that for all strings x, $K(p(x)) \geq |p(x)| - c$.

This concludes a *very* brief presentation of some of the fundamental theorems of Kolmogorov complexity. The interested researcher is encouraged to read [LV90]. Those more interested in *this* particular study now have enough of a background to evaluate the meaningfulness of the research. In particular, for our study, these three points should be stressed:

1) the length of the shortest program that produces an incompressible string is greater than or equal to (plus a constant) the string itself;
2) a random string is incompressible; and,
3) a shortest program is incompressible.

7.4 GP Complexity

The class of problems solvable by a GP can be grouped into two distinct subsets. The first group is the class of problems in which the GP is used to produce an output program that can be used to solve a desired problem. This is the class that GPs are typically associated with, evolutionary computation to produce programs. The second group is the class of optimization problems in which a GP is used to optimize the solution. In this case, the output of the GP is not a program but a solution to a particular problem. An example of this would be if a GP were used to solve an instance of the Maximum Clique problem. Though this is not how GPs are typically implemented, it has been shown that they are perhaps equally adept at optimizing functions [SFD98].

7.4.1 GPs for Programs

For GPs that produce programs, the Kolmogorov complexity analysis is straightforward. Though it is impossible to classify the complexity of a problem that can be solved by the output program in advance, it *is* possible to relate the amount of information contained in the output program to the GP itself and thus provide a bound on that complexity. By applying the theorems from Kolmogorov complexity, it can be shown that the complexity of the output program of a GP using a pseudo random number generator (PRNG) can be bound above by the GP itself.

Theorem 7.3: GP, Fundamental

For all strings x,y, if x is the shortest program that outputs y, that is $K(y)=|x|$, then $K(x) \geq K(y) + c$.

Proof:

Let x be the shortest program (by definition, incompressible) that outputs y. That is, $K(y) = |x|$. Suppose $K(y)>K(x)$. By substitution, $|x| > K(x)$, which is impossible since x was defined as incompressible.

Though this proof applies to GPs it is not specific to them. It applies to any method for inductively producing programs and therefore should not be seen as a limitation only to GPs.

A brief explanation may now be warranted. By assuming that the GP *x* is a shortest program, we are only stating that in the best case, where *x* is a shortest program, the upper bound for the complexity of the output *y* is the length of *x*. It may well be that any *particular* implementation of a GP contains much extraneous code and is nowhere near the shortest program. This does not mean that the complexity of the output *y* is any greater. K(x) still refers to the shortest program that can output *x*. And, K(x) must still be greater than K(y). So if the GP *x* is not a shortest program, then there must be a shorter program that can output *x*. This means that the complexity of the output *y* is

even smaller. A simple diagram may be helpful.

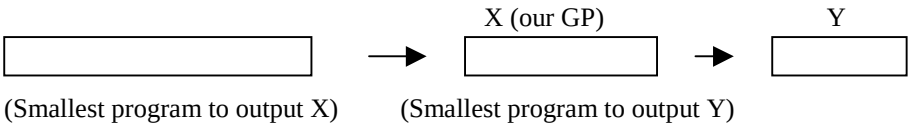


Figure 7.2: Smallest Program to Output *x* (our GP)

Though this proof applies to *any* method for inductively generating programs and is thus not reliant on any particular definition or implementation of GP, it may be instructional to describe exactly what constitutes the GP in reference to the complexity bound and what constitutes the output.

There are many types of GPs, and each implementation is relatively unique. Consequently a specific definition of a GP would not only be confining but also needlessly imply that the application of the proof is limited to that definition. We will instead provide an informal description of what is included in the GP. Essentially, it is all of the software required at execution time. In addition to the software written by the practitioner, all of the libraries used for implementation, the environment (if one is used), as well as the PRNG itself are included as part of the GP. The bound on the output is the Kolmogorov complexity of all of the software required for the GP implementation at execution time. If a more complex PRNG is used, the upper bound on the output will be greater. However, this does not mean that the output will necessarily be more complex. This only means that the upper bound is greater.

The output that is being bounded is the sum total of all of the individuals that are output, not just the final chosen program. To help explain this apparent paradox, think of a string of five 1's. Now think of a string of ten 1's. The shortest program that can output ten 1's is certainly not twice the length of the shortest program that can output five 1's. In fact, there is at most a constant difference in complexity. See Figure 7.3.

```
for(i=1;i<=5;i++)          for(i=1;i<=10;i++)
    cout<<'1';                 cout<<'1';
```

Figure 7.3: Two *for-loops*

But this proof only applies to GPs using a PRNG. This is because a *truly* random string is by definition incompressible. This leads to the following theorem.

Theorem 7.4: GP Complexity with True Randomness

For all strings x, y, if x is a shortest length program that outputs y, and x uses a true random source for its generation of y, then:

1) K(x) is undefined during execution;
2) $K(y) \leq |y| + c$; (a well known Kolmogorov result)
3) K(y) can be > K(x) - random input.

Proof:

Let K(x) = n, where x is a GP. Below is x

```
returnstring = "";
        for i= 1 to n+1{
    get a random bit, b;
    returnstring += b;
        }
        return returnstring;
```

Since a random string is incompressible, K(y) = n+1. Therefore, K(y) > K(x). Since it may be unknown how many times a GP will access a random bit, K(GP) is undefined during execution.

These proofs give reason to pause when considering how best to implement a GP. Before we discuss this however, we must address the complexity of the optimization problems solvable by GPs.

7.4.2 GPs for Function Optimization

In addition to producing programs, GPs have often been used to optimize functions. To describe how to evaluate the complexity of a problem that can be solved by this sort of GP, it is necessary to examine the complexity of problems for GAs.

Since a GP is a GA designed for program discovery [O'Re95] it is possible to apply the methods developed for analyzing problems for GAs. Though GPs are typically written in a higher level language, it is easy to see that they can be implemented with strings composed of 1's and 0's, as is typical for GAs, and are in fact, converted to strings of 1's and 0's at execution. As such, it is clear that the problems solvable by GPs are a subset of the problems solvable by GAs. (The converse may equally be true, since both methods essentially manipulate strings of 1's and 0's.) Consequently, we may introduce the theorems described for the complexity analysis of GA function optimization without a loss of specificity.

Essentially, the complexity of an optimization problem for a GA is bound above by the growth rate of the smallest representation [Minimum Chromosome Length - (MCL)] that can be used to solve the problem [RF00],[RF01a]. This is because the probabilistic convergence time will remain fixed as a function of the search space. All things held constant, the convergence time will grow as the search space grows. A brief, but more formal description is provided in the following definitions.

Definition 7.2: Minimum Chromosome Length (MCL)

> For a problem P, and D_n the set of instances of P of size n, let MCL(P,n) be the least *l* for which there is an encoding $e:S^l \rightarrow D_n$ with a domain dependent evaluation function g, where g and e are in FP (the class of functions computable in polynomial time).

That is, MCL(P,n) measures the size of the smallest chromosome for which there is a polynomial time computable representation and evaluation function. The MCL *growth* rate can be used to bound the complexity of the problem for a GA. If a problem is in

NPO, then the MCL growth rate will be no more than linear, since an NP algorithm can search a space which grows no more quickly than exponentially, and linear MCL growth implies exponential search space growth. Conversely, if MCL grows slowly enough, then the search space will be small enough to place the problem in PO. (PO and NPO are the optimization equivalents of P and NP, which are classes of decision problems.) In particular

Theorem 7.5: Sublinear MCL Growth

For any problem P, if $2^{MCL(P,n)} \in O(n^k)$ for some *k*, then P∈PO.

It is currently unclear if the converse is true. However, if $2^{MCL(P,n)} \notin O(n^k)$ then the problem is most likely in NPG (the class of problems that take more than polynomial time for a GA to solve).

Definition 7.3: GA Complexity Class NPG

A problem P is in the class NPG if MCL(P,n)∈O(n).

As an example, consider the problem of *Maximum 1's* (i.e. for a given string length n, find the maximum number of 1's that can be held in the string). This is possibly the easiest problem for a GA to implement. Simply let each bit position represent a possible 1. Then, for each chromosome, count the number of 1's to determine that chromosome's fitness. For a string of length 5, the maximum fitness for a chromosome is 5. As the problem instance size increases, the length of the chromosome increases. This means that the size of the search space doubles for every increase in instance size because the number of possible solutions is equivalent to the number 2 raised to the length of the chromosome, or 2^l.

Observe that regardless of the expected convergence time per instance size, the growth rate of the search space determines the complexity of the problem for the GA. As an example, imagine the GA will converge after examining 1/10 of the search space. If

the search space doubles for every increase in instance size, the GA-complexity of the problem will be $(1/10)* 2^{l}$. This is clearly an exponential convergence time. The search space growth rate will dominate over every expected convergence rate regardless of how quickly a particular implementation can converge.

However, if the minimum chromosome length doesn't have to change *every* time the instance size increases, this indicates that the problem convergence and consequently the GA-complexity for the problem are *less* than exponential time. Note that with the problem *Maximum 1's* there is a smaller representation.

A representation can be created in which the maximum number of 1's in a string can be encoded in binary. This means that instead of 5 bits being required to represent a string of length 5, 3 bits will suffice (e.g. 101). The growth rate of the search space changes as well. Notice, for our new representation, we don't always have to increase the length of the chromosome. A 5 is encoded with 3 bits. Also, a 6 is encoded with 3 bits. Consequently, there is no change in convergence time when going from a string of length 5 to a string of length 6. In fact, our representation doesn't cause the chromosome length to increase until the instance size is increased from 7 to 8 (because 8 requires 4 bits, e.g. 1000). This fact alone suggests that the GA-complexity of *Maximum 1's* is some constant c times lg(x).

This prediction can be validated empirically. To conduct our study we employed a typical GA implementation strategy. We chose an arbitrarily fixed population of 10 chromosomes, single point crossover, and a fixed mutation rate of 1% per bit. There was 50% elitism thus providing a 50% survival rate for each generation. Selection was implemented randomly based on the chromosomes that survived.

It should be noted that the principle of measuring the growth rate of the minimum length representation can be used with *any* implementation, since what is actually being measured is the growth rate of the search space. Therefore it is not necessary to demonstrate the usefulness of this measure with *every* GA implementation.

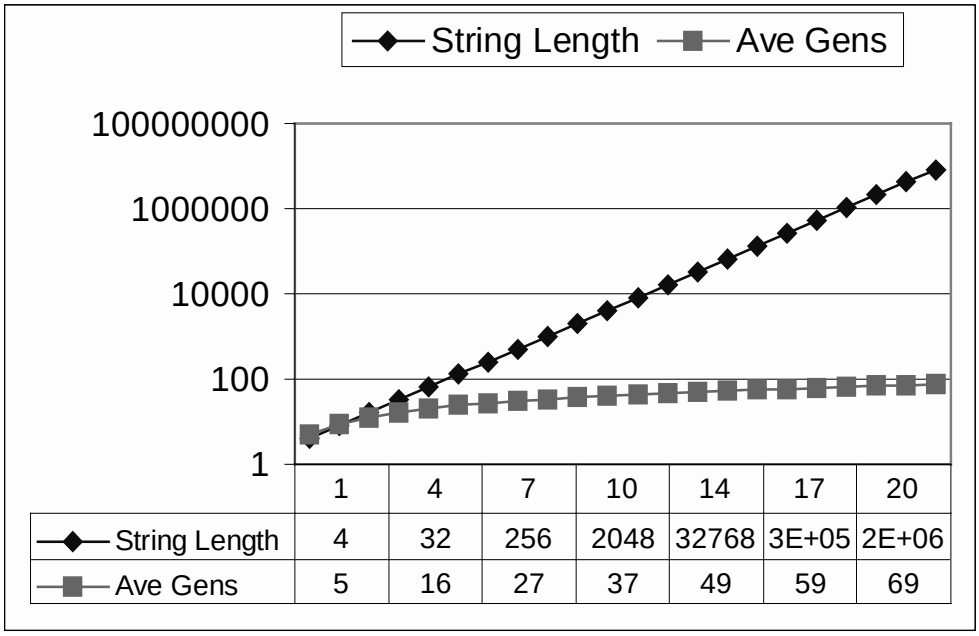
We then let the GA run until all chromosomes had converged to the optimum solution. This was repeated 5000 times to produce an average convergence time for a

given chromosome length. Then the problem instance was increased and the 5000 runs were repeated. This experiment tested the range of string lengths from 4 to 8,388,608. The length of the chromosomes ranged from 3 bits to 23 bits. See Table 7.1.

String Length	4	8	16	32	64	512	32k	8m
Chrom. Length	3	4	5	6	7	10	16	24
Ave. Conv.	5	9	12	16	20	30	49	77

Table 7.1: Average Generations to Converge Per Instance Size of String Length

As can be seen, the convergence time is less than exponential. In fact, the convergence varied linearly with the chromosome length. This particular experiment had a convergence of roughly 3*|b(x)| (3 times the length in binary of x). Since the chromosome itself grew at lg(x), this means that the convergence is in O(lg(x)). It should be noted that this rate of convergence held for a very wide range. Consequently, unless it is possible to derive a smaller representation, we can assert from both experimental as well as theoretic evidence that the GA-complexity of Maximum 1's is $\in$ O(lg(x)). A graphical depiction of the convergence times over the entire range can be seen in Fig.7.4.



	1	4	7	10	14	17	20
String Length	4	32	256	2048	32768	3E+05	2E+06
Ave Gens	5	16	27	37	49	59	69

Figure 7.4: Average Convergence for Maximum 1's Problem, Logarithmic

There is one further twist to applying this theory to GPs that optimize functions. Namely, GAs typically have fixed length chromosomes whereas GPs have no such bound. Consequently, the MCL problem complexity bound will actually be a lower bound for the problem when a GP is applied, since it assumes the GP will converge to an optimum solution with the minimum chromosome length, per instance size.

This completes an introduction to the complexity analysis for problems that are solvable with GPs. However, though we can now define what the complexity of problems are when GPs are applied, there still must be guidance for GP implementation. Since GPs are in fact a very promising avenue of artificial intelligence theory, in light of these results, it seems appropriate to address how best to implement a GP. This will be discussed in the next section.

7.5 Implications to GP Design

The Kolmogorov complexity results for traditional GPs (i.e. those used to inductively generate programs) should be reviewed with some concern. The implications seem far-

reaching. Before delving into them however, it may be helpful to note that typical compressibility can serve as an approximation of Kolmogorov complexity [LV90]. Thus, in considering these results it is reasonable to consider the GP as a compressed file and it's output as a compressed file. Our proofs indicate that the compressed output file must be smaller than the compressed GP.

Note that the output of the GP is not just the best program produced. The GP's output consists of all of the programs produced in all of the generations. In fact, convergence and the resulting redundancy in GP populations make them fairly compressible. Recall also that for this result the GP includes the GP itself, the PRNG, the training data, and any other code or data that is used in running the GP. As noted, some problems require a simulated environment for training the GP; this environment is also considered part of the GP for complexity measurements. However, any data or code used to *test* the final evolved programs are not considered part of the GP. This is because they do not influence the final program. They are only used for measurement. Thus, these results compare the compressed size of the GP, including training data etc. to the compressed size of all programs generated in all generations.

7.5.1 Large vs. Small Populations

There are several interesting and significant implications derivable from this result. First, there *must* be a limit to the advantages of larger populations and/or longer trials. The GP is finitely complex (in most cases, we will talk about infinitely complex GPs in a moment), therefore its output must also be finitely complex. Because the output complexity includes all individuals from all populations, producing more individuals through larger populations or longer runs must eventually stop producing new solutions because these solutions would necessarily increase the output complexity beyond the finite limit imposed by the GP.

This is not as limiting as it may appear however. First, we expect all GPs to converge, at which point they generally do not produce significantly new solutions or additional complexity. Second, the most complex solution possible is one that perfectly

fits the training data. This introduces a natural limit to the output complexity. Therefore, a perfect solution is generally not desired as they often over-fit the training data and generalize poorly. Instead a simpler program that generalizes well is usually preferable.

7.5.2 Increasing Complexity

The above observations lead to a fairly intuitive relationship: *evolving more complex programs requires a more complex set of training data*. More complex training data often comes from having a more complex problem. Thus, the complexity of the solutions a GP can produce is *directly related to the complexity of the problem it is trying to solve.* Of course, this is only the potential complexity of the solution. The GP could produce much simpler solutions. This may be desired, for example, if they generalize well. Conversely, it could indicate that the GP is failing to solve the problem.

As an extreme case, very complex (i.e. incompressible) training data is random data. In this case the only perfect solution is to reproduce the training set entirely; i.e. evolve the same random data. Thus, a complex training set leads to complex solutions. Although GPs are not trained on random data, many systems include random noise in the training data. This noise actually increases the complexity of the total GP and thus increases the potential complexity of the generated solutions. Again, this additional allowed complexity may or may not be beneficial. Clearly a solution that is complex because it is fitting random noise is not desired.

A more practical example is evolving control programs for robots or other autonomous agents. Often the evolutionary process occurs in a relatively simple, simulated environment and the best programs are transferred to actual robots in the physical world. However, because the training environment (the simulated environment) is simple it can only produce equally simple programs, which may not be capable of dealing with the complexities (such as noise) of the physical world. Thus, our results with Kolmogorov complexity prove that an oversimplified training environment can make it *impossible* to evolve a program complex enough to deal with more complex testing environments such as those found in the real world. In contrast, programs that are

trained in a real world environment are exposed to effectively infinite complexity (though some of this complexity may be filtered out by whatever sensory devices that are being used) and thus may produce programs of unlimited complexity.

7.5.3 Quantum GPs

Recently there has been interest in combining evolutionary techniques with quantum computers [RF01a], [RF01b], [LSP+99]. One unexplored advantage of this merger is that quantum computers have inherent *true* random number generators. Because true random numbers are incompressible, the complexity of a GP executing on a quantum computer using true randomness is bounded below by the number of random numbers used. This is in contrast to the complexity of pseudo random numbers, which is fairly low and is fixed.

First, we note that a typical GP uses many random numbers, introducing a large amount of complexity (when the numbers are truly random). Second, and more importantly, larger populations and longer runs both require additional random numbers. If the numbers are from a truly random source, then they introduce more complexity into the GP, making more complex solutions possible. Thus, indefinitely increasing the population size or number of generations *may* be useful when a true random number source (such as a quantum computer) is used.

7.6 Conclusions

This paper has provided a proof that the complexity of the output of a GP using a PRNG can be bound above by the GP itself. It has also been shown that this result does not hold if the GP instead has access to a truly random source. An analysis of the complexity of problems solvable by a GP as optimizer has been presented. The implications of this have shown that an optimization problem's complexity can be determined by measuring the growth rate of the minimum representation. We noted that this is essentially the same method used for analyzing GA problem complexity. This observation is important because GAs and GPs are clearly related. The main difference being that GP problem

complexity is bounded *below* by the growth rate since GP chromosome length is dynamic during execution. We have proposed that the complexity of the training data has a direct correlation to the complexity of the desired output. Finally, we have briefly explored the significance of employing a GP on a quantum computer and showed that there may be significant advantages to such an endeavor.

There are many unanswered questions relating to this research. It is clear that there are several avenues from which to expand this work. Perhaps the most fruitful direction may be to conduct experiments to verify how closely these theoretical bounds are adhered to in practice.

7.7 Acknowledgments

This research was funded in part by NSF EPS0080935, NIH F33GM20122-01, and NSA MDA 904-98-C-A894.

8 Quantum Evolutionary Programming

Rylander, B., Soule, T., Foster, J., Alves-Foss, J., Quantum Genetic Programming, *Proceedings of the Genetic and Evolutionary Computation Conference*, Morgan Kaufman, 2001.

8.1 Abstract

Recent developments in quantum technology have shown that quantum computers can provide dramatic advantages over classical computers for some problems [Sho94] [Gro96]. These quantum algorithms rely upon the inherent parallel qualities of quantum computers to achieve their improvement. In this paper we provide a brief background of quantum computers. We present a simple quantum approach to genetic algorithms and analyze its benefits and drawbacks. We describe the quantum advantage of true randomness. We show that in some cases, such as program induction, there is a measurable difference [Ryl00]. These algorithms are significant because to date there are only a handful of quantum algorithms that take advantage of quantum parallelism [WC97] and none that show an advantage due to true randomness. Finally, we provide ideas for directions of future research.

8.2 Introduction

The first major breakthrough in quantum computing came in 1985 with the development of the Quantum Turing Machine (QTM) [Deu85]. Then 1996, two researchers independently proved that a Universal Quantum Simulator was possible [Llo96], [Zal98]. As such, anything computable by a classical computer would be computable by a quantum computer in the same time, if and when such a computer was in fact built. This cannot be said of the converse however. Quantum computers have a feature called quantum parallelism that can not be replicated by classical computers without an exponential slowdown. This unique feature turns out to be the key to most successful quantum algorithms.

Quantum parallelism refers to the process of evaluating a function once on a "superposition" of all possible inputs to produce a superposition of all possible outputs. This means that all possible outputs are computed in the time required to calculate just one output with a classical computer. Unfortunately, all of these outputs cannot be as easily obtained. Once a measurement is taken, the superposition collapses. Consequently, the promise of massive parallelism is offset by the inability to take advantage of it.

This situation changed in 1994 with the development of a fast, hybrid algorithm (part QTM and part TM) for factoring that took advantage of quantum parallelism by using a Fourier transform [Sho94]. With this algorithm and a suitably sized quantum computer it is possible to provide a solution for factoring in polynomial time. This is an important development because the security of most public-key encryption methods relies upon the difficulty of factoring large numbers. Though not proven intractable, factoring had previously seemed secure. Consequently, quantum technology has already made a very tangible impact on some communities.

Previous work that explored the application of quantum parallelism to evolutionary programming has been promising. The first attempt was a "quantum inspired" GA in which many of the operators were modeled after quantum behavior [GWC98]. While this algorithm wasn't actually quantum-based it did imply that a quantum genetic algorithm would outperform a classical one if it could be implemented. The next two attempts showed that a quantum GA maybe possible, but that the restrictions imposed by the quantum paradigm seemed to negate the parallelism introduced by the evolutionary paradigm [NM98][RSF+00]. We build on these efforts to produce an algorithm that seems to take advantage of both kinds of parallelism. The next section provides a brief introduction to quantum concepts. Section 3 outlines our quantum genetic algorithm (QGA). Section 4 provides an analysis of the advantages of quantum programming. Section 5 details the difficulties that still remain to developing such an algorithm (beyond the obvious fact that a practical quantum computer has yet to be built). Finally, Section 6 gives conclusions and directions for potential research.

8.3 Quantum vs. Classical

There are two significant differences between a classical computer and a quantum computer. The first is in storing information, classical bits versus quantum *q-bits*. The second is the quantum mechanical feature known as *entanglement*, which allows a measurement on some q-bits to effect the value of other q-bits.

A classical bit is in one of two states, 0 or 1. A quantum q-bit can be in a *superposition* of the 0 and 1 states. This is often written as $\alpha |0> + \beta |1>$ where α and β are the *probability amplitudes* associated with the 0 state and the 1 state. Therefore, the values α^2 and β^2 represent the probability of seeing a 0 (1) respectively when the value of the q-bit is measured. As such, the equation $\alpha^2 + \beta^2 = 1$ is a physical requirement. The interesting part is that until the q-bit is measured it is effectively in *both* states. For example, any calculation using this q-bit produces as an answer a superposition combining the results of the calculation having been applied to a 0 and to a 1. Thus, the calculation for both the 0 and the 1 is performed simultaneously. Unfortunately, when the result is examined (i.e. measured) only one value can be seen. This is the "collapse" of the superposition. The probability of measuring the answer corresponding to an original 0 bit is α^2 and the probability of measuring the answer corresponding to an original 1 bit is β^2.

Superposition enables a quantum register to store exponentially more data than a classical register of the same size. Whereas a classical register with N bits can store one value out of 2^N , a quantum register can be in a superposition of all 2^N values. An operation applied to the classical register produces one result. An operation applied to the quantum register produces a superposition of all possible results. This is what is meant by the term “quantum parallelism.”

Again, the difficulty is that a measurement of the quantum result collapses the superposition so that only one result is measured. At this point, it may seem that we have gained little. However, depending upon the function being applied, the superposition of answers may have common features. If these features can be ascertained by taking a measurement and then repeating the algorithm, it may be possible to divine the answer

you're searching for probabilistically. Essentially, this is how the famous quantum algorithm for factoring works. First, you produce a superposition and apply the desired functions. Then, take a Fourier transform of the superposition to deduce the commonalties. Finally, repeat these steps to pump up your confidence in the information that was deduced from the transform.

The next key feature to understand is entanglement. Entanglement is a quantum connection between superimposed states. In the previous example we began with a q-bit in a superposition of the 0 and 1 states. We applied a calculation producing an answer that was a superposition of the two possible answers. Measuring the superimposed answer collapses that answer into a single classical result. Entanglement produces a quantum connection between the original superimposed q-bit and the final superimposed answer, so that when the answer is measured, collapsing the superposition into one answer or the other, the original q-bit also collapses into the value (0 or 1) that produces the measured answer. In fact, it collapses to *all* possible values that produce the measured answer. Given this very brief introduction to superposition and entanglement, we can begin to address our GA. (Interested researchers may refer to [4] for a more detailed description of quantum computing.)

8.4 A Quantum Genetic Algorithm

We now present a quantum genetic algorithm (QGA) that exploits the quantum effects of superposition and entanglement. This QGA differs from previous QGA's in several regards. Primarily, our QGA is more similar to classical GA's. This allows the use of any fitness function that can be calculated on a QTM without collapsing a superposition, which is generally a simple requirement to meet. Our QGA differs from a classical GA in that each individual is a quantum individual. For example, consider the fitness landscape in Figure 8.1.

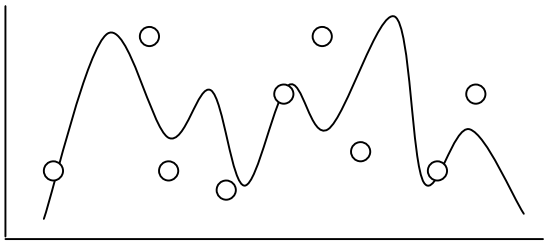


Figure 8.1: Fitness landscape with individuals

Each point represents a unique position on the fitness landscape. In the case of a fitness function such as multiplication, the fitness of 24 can be produced in a variety of ways. Individuals that have subcomponents such as 6*4, 4*6, 12*2, and 24*1 all have the same fitness despite the fact that they are fundamentally different. Each of these unique individuals will reside somewhere on the landscape. Consequently, when selecting an individual to perform crossover, or mutation, exactly 1 individual is selected. This is true regardless of whether there are other individuals with the same fitness. (See Figure 8.2)

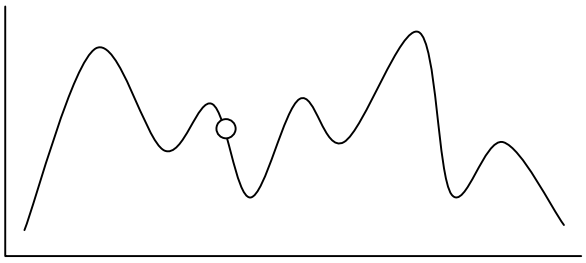


Figure 8.2: A classical individual is selected

This is not the case with a quantum algorithm. By selecting an individual, *all* individuals with the same fitness are selected. (See figure 8.3) In effect, this means that a single quantum individual in reality represents multiple classical individuals.

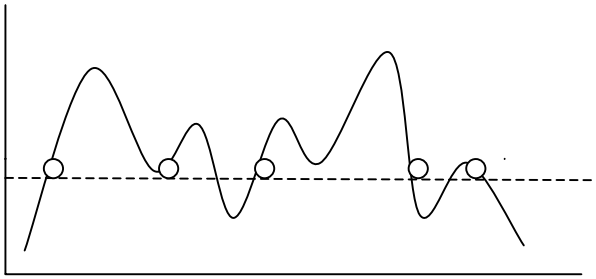


Figure 8.3: A Quantum Individual (all those with the same fitness)

In our QGA each *quantum* individual is a superposition of one or more classical individuals. To do this we use several sets of quantum registers. Each quantum individual uses two quantum registers. We refer to these as the *individual register* and the *fitness register*. (See Figure 8.4) The first register stores the superimposed classical individuals. The second register stores the quantum individual's fitness. At different times during the QGA the fitness register will hold a single fitness value or a quantum superposition of fitness values. A population will be N of these quantum individuals.

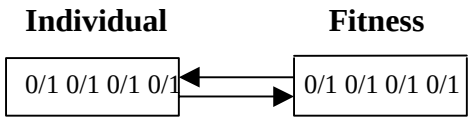


Figure 8.4: Two 4-qubit entangled quantum registers (representing a single individual)

The key step in our QGA is the fitness measurement of a quantum individual. We begin by calculating the fitness of the quantum individual and storing the result in the individual's fitness register. Because each quantum individual is a superposition of classical individuals, each with a potentially different fitness, the result of this calculation is a superposition of the fitnesses of the classical individuals. This calculation is made in

such a way as to produce an entanglement between the register holding the individual and the register holding the fitness(es).

For example, consider a quantum individual consisting of a superposition of six classical individuals with fitnesses: 4, 4, 7, 9, 9 and 11. The fitness calculation will produce a superposition of the values 4, 7, 9, and 11. The values 4 and 9 will have higher probabilities associated with them because classical individuals with those fitnesses occurred more times in the quantum individual.

Next, we measure the fitness. That is, we 'look' in the fitness register. This measurement collapses the register and we measure (or observe) only a single fitness (either 4, 7, 9, or 11 in the above example, with 4 and 9 being twice as likely as the other values). When the fitness superposition collapses to a single value the individual register partially collapses, so that it only holds those individuals with the measured fitness. (In the above example, if we observed a fitness of 4 the individual register would automatically change from holding a superposition of six classical individuals to holding a superposition of two individuals; the two whose fitness is 4.)

This process reduces each quantum individual to a superposition of classical individuals with a common fitness. This allows the selection process to proceed as in a classical GA, based on a classical fitness function.

Crossover and mutation can then be applied (as previously developed [8]). For the initial population each individual is in a fully mixed state (i.e. contains a superposition of all possible solutions). The fitness calculation described above reduces these individuals to superpositions consisting of all classical individuals with a common fitness. This assures a wide diversity in the initial population.

The complete algorithm is described below:

Generate a population of fully mixed quantum individuals. (Each individual is superposition of all possible classical individuals.)

Calculate the fitness of the individuals.

observe the fitness of each individual. (This collapses the individuals into superpositions of only those classical individuals with the observed fitness.)

Repeat

Selection based on the observed fitnesses.

Crossover and **Mutation**. (Superimposed classical individuals in a single quantum individual will no longer have a common fitness.)

Calculate the fitness of the individuals.

observe the fitness of each individual. (This collapses the individuals into superpositions of only those classical individuals with the observed fitness.)

Until done.

8.5 Analysis of the QGA

Now we can begin to describe what has been gained by this algorithm. Unfortunately comparing convergence times between a GA and a QGA cannot currently be directly performed. This is because there is no current theory to predict convergence time for a QGA. Consequently the convergence times must be compared deductively. Another measure of complexity, the amount of information that is contained in a string, *can* be measured [LV90]. This type of complexity is more meaningful when evaluating algorithms for program induction, such as Genetic Programming. For this type of complexity, the QGA is superior to the classical GA. Both of these measures will be described in detail below.

8.5.1 Convergence Times

Currently the answer of whether a QGA converges more quickly than a GA cannot be directly quantified. Only recently has there been a way to characterize the complexity of problems related to *classical* GAs [RF01a]. Since it is currently unknown exactly how this form of probabilistic selection will drive convergence, it would be disingenuous to firmly assert any type of quantitative advantage. Still, it may be possible to gain insights through a discussion of the pros and cons of the QGA in an informal manner.

8.5.1.1 Increased Diversity

The major advantage for a QGA is the increased diversity of a quantum population. A quantum population can be exponentially larger than a classical population of the same *size* because each quantum individual is a superposition of multiple classical individuals. Thus, a quantum population is effectively much larger than a similar classical population.

This effective size decreases during the fitness operation, when the superposition is reduced to only individuals with the same fitness. However, it is increased during the crossover operation. Consider two quantum individuals consisting of N and M superpositions each. One point crossover between these individuals results in offspring that are the superposition of N*M classical individuals. Thus, in the QGA crossover increases the effective size of the population in addition to increasing it diversity.

There is a further benefit to quantum individuals. Consider the case of two individuals of relatively high fitness. If these are classical individuals, it is possible that these individuals are relatively incompatible; that is that any crossover between them is unlikely to produce a very fit offspring. Thus, after crossover it is likely that the offspring of these individuals will not be selected and their good ‘genes’ will be lost to the GA.

If these are two quantum individuals then they are actually multiple individuals, all of the same high fitness, in a superposition. As such, it is very unlikely that all of these individuals are incompatible and it is almost certain that some highly fit offspring will be produced during crossover. Unfortunately, the likelihood of measuring these individuals

may not be very good. Therefore, it is possible that *on average* the quantum algorithm will not have a great advantage. However, at a minimum the good offspring are somewhere in the superposition, which is an improvement over the classical case. This is a clear advantage of the QGA.

It seems likely that the more significant advantage of QGA's will be an increase in the production of good building blocks. In classical GA theory good building blocks are encouraged because statistically they are more likely to produce fit offspring, which will survive and further propagate that building block. However, when a new building block appears in the population it only has one chance to 'prove itself'. Originally a good building block only exists in a single individual. It is crossed with another individual and to survive it must produce a fit offspring in that one crossover. If the individual containing the good building block happens to be paired with a relatively incompatible mate it is likely that the building block will vanish.

The situation is very different in the QGA. Consider the appearance of a new building block. During crossover the building block is not crossed with *only* one other individual. Instead, it is crossed with a superposition of many individuals. If that building block creates fit offspring with most of the individuals, then by definition, it is a good building block. Furthermore, it is clear that in measuring the superimposed fitnesses, one of the "good" fitnesses is likely to be measured (because there are many of them), thereby preserving that building block. In effect, by using superimposed individuals, the QGA removes much of the randomness of the GA. Thus, the statistical advantage of good building blocks should be much greater in the QGA. This should cause the number of good building blocks to grow much more rapidly. This is clearly a significant benefit.

One can also view the evolutionary process as a dynamic map in which populations tend to converge on fixed points in the population space. From this viewpoint the advantage of QGA is that the large effective size allows the population to sample from more basins of attraction. Thus, it is much more likely that the population will include members in the basins of attraction for the higher fitness solutions.

Interestingly, all of the advantages of our QGA depend on the mapping from individual to fitness. If this mapping is one-to-one then measuring the fitness function collapses each quantum individual to a single solution and the benefits are lost. The greater the degree of "many to oneness" of the mapping from individual to fitness the greater the potential diversity advantage of a QGA.

8.5.2 Quantum Genetic Programming

Another advantage for quantum computers is the ability to generate true random numbers. This becomes important for Genetic Programming (GP) and other methods for automatic program induction. In particular, recent theory work has shown that the output of a GP can be bounded above by the information content of the GP itself [RSF01]. Given this, the advantages of true randomness become readily apparent. By application of Kolmogorov complexity analysis, it has been shown that classical implementations of GPs which use a pseudo random number generator (PRNG) are bounded above by the GP itself whereas with the benefit of a true random number generator, there is no such bound [13].

Briefly, Kolmogorov complexity measures the size of the smallest program that can output a string and then terminate. It is often used when trying to evaluate the information content of static objects (such as strings). (See Definition 8.1)

Definition 8.1: Kolmogorov Complexity, K

$K_S(x) = \min \{|p|: S(p) = n(x)\}$.

(In this instance, p can be thought of as a program and S as the programming language.) [LV90]

Essentially, using the tools from Kolmogorov complexity the following Theorem was developed.

Theorem 8.1: GP, fundamental

For all strings x, if x is the shortest program that outputs y, that is K(y)=|x|, then K(x)≥K(y) + c.[RSF01]

A graphical depiction maybe helpful. (See Figure 8.5)

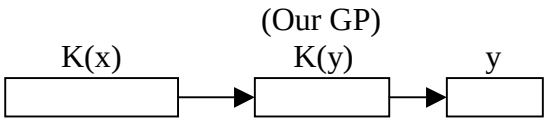


Figure 8.5: Smallest Program to Output X (our GP)

This says that the length of the shortest program to produce our GP must be greater than the shortest program to produce the output of our GP ("x" in this case). However, this theorem does not hold for GPs that have access to a true random source, such as a quantum computer. In particular, for GPs implemented on a quantum computer, Theorem 2 holds.

Theorem 8.2: GP Complexity with True Randomness

For all strings x, y, if x is a shortest length program that outputs y, and x uses a true random source for its generation of y, then:

1) K(x) is undefined during execution;
2) K(y) ≤ |y| + c ; (a well known Kolmogorov result)
3) K(y) can be > K(x) - random input.

Proof:

Let K(x) = n, where x is a GP. Below is x

```
returnstring = " "
    for i= 1 to n+1{
    get a random bit, b
    returnstring += b
    }
    return returnstring
```

Since a truly random string is incompressible, K(y) = n+1. Therefore, K(y) > K(x). Since it may be unknown how many times a GP will access a random bit, K(x) is undefined during execution.

It is hard to quantify exactly how great this advantage is. It is clear that the Kolmogorov complexity of the output of a classical GP is bounded by the GP itself. Likewise, it is clear that a GP implemented on a quantum computer has no such bounds. The difference is of course infinite. However, it is hard to fathom how a sequence of truly random numbers could produce such dramatic improvement over an equal length string produced by a PRNG. Trying to verify such an advantage is also difficult since no such practical quantum computer currently exists. Nonetheless despite these difficulties it is a provable advantage.

8.6 Difficulties with the QGA

There are some potential difficulties with the QGA presented here, even as a theoretical model. Some fitness functions may require "observing" the superimposed individuals in a quantum mechanical sense. This would destroy the superposition of the individuals and ruin the quantum nature of the algorithm. Clearly it is not possible to consider all fitness functions in this context. However, since mathematical operations can be applied without destroying a superposition, many common fitness functions will be usable.

As noted previously a one-to-one fitness function will also negate the advantages of the QGA. Another, more serious difficulty, is that it is not physically possible to exactly

copy a superposition. This creates difficulties in both the crossover and reproduction stages of the algorithm. A possible solution for crossover is to use individuals consisting of a linked list rather than an array. Then crossover only requires moving the pointers between two list elements rather than copying array elements. However, without a physical model for our quantum computer it is unclear whether the notion of linked lists is compatible with maintaining a quantum superposition.

The difficulty for reproduction is more fundamental. However, while it is not possible to make an exact copy of a superposition, it is possible to make an inexact copy. If the copying errors are small enough they can be considered as a "natural" form of mutation. Thus, those researchers who favor using only mutation may have an advantage in the actual *implementation* of a QGA.

8.7 Conclusions

We have presented a quantum GA that uses the quantum features, superposition and entanglement. Our simple analysis of the algorithm suggests that it should have three advantages over a normal GA. First, because individuals in the QGA are actually the superposition of multiple individuals it is less likely that good individuals will be lost. Secondly, and more significantly, the effective statistical size of the population appears to be increased. This means that the advantage of good building blocks has been magnified. Presumably this will greatly increase the production and preservation of good building blocks thereby dramatically improving the search process. Finally, in the case of inductive generation of programs, there is a provable improvement over the classical method. Though it is currently hard to evaluate how significant this improvement is, the magnitude of this improvement causes one to wonder what the significance may be in the future.

Unfortunately, the first two advantages can not be presently proven. Therefore, a good direction for future research would include providing a mathematical analysis of the convergence time of the QGA. Once this has been determined, it should be easy to evaluate the relative complexity of QGAs and their classical counterpart. Another

potential fruitful direction would be to compare the ease of implementation of crossover methods versus mutation methods. Whereas with classical GAs applying only mutation is in effect a “numerative method” [Hol75] which must contend with the time complexity involved in searching a vast search space, this problem may not exist for a QGA.

8.8 Acknowledgments

This paper was supported in part by NSF EPS0080935, NIH F33GM20122-01, and NSA MDA 904-98-C-A894.

9 Discussion

The genetic algorithm was created for a variety of reasons. One of the most important reasons was to plumb a vast search space to provide an optimum or near optimum solution in a reasonable amount of time. Many problems of interest were and continue to be computationally intractable. By developing a method that could adapt, it seemed possible that some of these problems could be addressed. Another reason that the genetic algorithm was developed was to be a simulation of biological evolution. As such, it is a natural mechanism for exploring the computational power of one of the most dominant forces in our world.

Despite these noble goals, little prior work has concentrated on quantifying the value of this method from the perspective of computational complexity. Some of the reasons for this apparent lack have been expressed in Section 3. Briefly, the seduction of the biological metaphor as well as the relative difficulty in understanding computational complexity (itself a young field) have led to this shortfall. Because of this however, no method for even *evaluating* the complexity of a problem independent of a representation, specifically for a genetic algorithm, had even been developed.

The research presented in this dissertation concentrated on developing a method for evaluating a problem specifically when a genetic algorithm is applied. Upon developing this method, it became possible to compare the complexity of problems for a genetic algorithm with the complexity of those same problems when other methods were applied. In so doing, it became obvious that some problems are better off being solved with methods other than genetic algorithms. These problems were then defined as being GA-semi-hard. After applying the method to a variety of problems it became clear that there was a complexity hierarchy unique to the genetic algorithm which differs from the classical complexity hierarchy. That is, some problems that take exponential time when a genetic algorithm is applied take polynomial time when another algorithm is used. As a result, two genetic algorithm complexity classes have been defined. There are most certainly other natural ones.

Though there are differences between the two complexity hierarchies, there has yet to be an instance when a genetic algorithm takes provably less time than expected in the classical hierarchy. This makes sense, because if such a problem did exist, the classical hierarchy would just be amended since it allows for the use of *any* algorithm (and clearly a genetic algorithm meets this requirement). So a primary result of these findings is that there are some problems which provably should not be solved with genetic algorithms.

It may seem that this result means that there is in fact no reason to be using a genetic algorithm. This is not necessarily the case however. Genetic algorithms were developed to deal with uncertainty. There are many instances when a practitioner knows enough to develop a successful genetic algorithm and yet does not know enough about the search space to employ a more direct method. In cases where there is uncertainty, algorithms that require complete knowledge of a search space will be useless regardless of their computational efficiency.

Genetic Programming presented an interesting challenge for analysis. Since a GP can be implemented with a GA, no analysis could be complete without considering the implications of evolving programs. Because the goal of a GP is to evolve another program, there was no direct way for evaluating the complexity of a problem that could be solved by a GP. Instead a comparison was made between the GP and its output. By applying the techniques developed in Kolmogorov Complexity theory it was possible to prove that the output of a GP could not contain as much information as the GP itself. While the implications of this fact are still being debated, it seems clear that the GP is not a panacea in the sense that we have found a "silver bullet" that can somehow solve our problems for us.

Despite this proof, a couple of interesting alternatives have developed for which the proof does not apply. One alternative is to develop a Universal Genetic Program (UGP). Since the UGP gets its "specification" from the input, its initial complexity is undefined. The other alternative is to use a quantum computer. Though no practical quantum computer yet exists, this work proved that the limiting complexity proof did not apply to a GP that was implemented on a quantum computer and that such implementations are

possible. The limiting complexity proof does not apply because quantum computers have access to true random numbers rather than numbers generated from a psuedo-random number generator.

Since in some cases quantum computers are provably superior to classical ones, it became important to explore the possibility of employing a genetic algorithm *on* a quantum computer. Interesting quantum features such as superposition and entanglement provide assets that seem to provide additional computational power to the unique parallelism that genetic algorithms already possess. Unfortunately, there has yet to be developed a coherent theory of convergence for a Quantum Genetic Algorithm (QGA). As such, the apparent advantages must be regarded as educated speculation. Nonetheless, there is still the proof that was developed for GP analysis that clearly shows the superiority of a QGA used for program induction with respect to Kolmogorov complexity. Until further theoretical work has been done, this is unfortunately the most that can be said.

10 Future Work

The research conducted as part of this dissertation has provided numerous avenues for future work. Theoretical studies can expand this work. New parametric studies should be conducted. Empirical convergence analysis can be conducted comparing the convergence of MCL representations and greater than minimum representations. A GA complexity hierarchy can be explored. A new model of GP can be developed and evaluated. QGA modeling needs to be done. These are just some of the areas that can be explored as a result of this work. Below, each of the above suggestions is briefly explored.

10.1 MCL Theory Work

The MCL method has been initially explained. There are many things that are currently unknown about the MCL. It seems likely that MCL is uncomputable. This needs to be proven. Also, what are the implications of MCL on classical complexity? Since the growth rate of the MCL has been shown to be useful in bounding the complexity of a problem for a GA, there must be a relation between MCL and the P vs. NP question. Can we provide a representation whose MCL has a sublinear growth rate for an NP-complete problem? If this is impossible, there should be a proof to explain why.

In addition, only two GA complexity classes have been defined. The traditional hierarchy is infinite. There may be differences between the two hierarchies. The GA-complexity hierarchy needs to be mapped out. In so doing, the researcher would be identifying areas that either lend themselves to GA application or ones that should be solved with other methods. This seems clearly to be a valuable area to explore.

Also, MCL was defined with a polynomial bound on the encoding and decoding functions within the GA. This allows for the possibility of ignoring the evolutionary mechanism altogether and solving a problem directly within the encoding or decoding functions. It may be interesting to explore using a more restrictive linear bound on the encoding and decoding functions

10.2 Parametric Studies

Parametric studies can be redone. Previous studies only evaluated the best crossover, mutation operators, and population sizes based on specific representations. Now that there is a method for evaluating problem complexity specifically for a genetic algorithm, we can redo these studies based on a non-specific representation. There is now an objective viewpoint from which to evaluate the findings. Though it cannot necessarily be said that the MCL for a problem is the best representation in terms of convergence (it may prove to be), it is nonetheless a fixed point from which all statistics can be measured. While this type of study could be done before, there was no rationale for the representation that was used as the baseline for the experiment. Also, it should be noted that no study has evaluated an operator as the instance size has varied. It may be interesting to see whether higher or lower mutation rates are better for a 100-node maximum clique instance relative to a 10-node maximum clique instance, for example.

10.3 Comparative Representations

Does an MCL representation converge more quickly than a greater than minimal representation? Many researchers have tried to improve upon previous work by improving upon the representation. Does it really make a difference? It would be interesting to see if a greater than minimal representation would converge more or less quickly than an MCL representation with all other parameters held constant. Would the possible increase in convergence speed be offset by the greater growth in the search space?

10.4 Genetic Programming

Proofs developed from this research may lead to a reevaluation of genetic programming. The limiting proof on GP complexity implies that making the training data denser, from an information theoretic perspective, may produce more complex results. This result may be verifiable empirically. If so, it may place a greater emphasis on the selection of quality PRNGs. If the result is not so easily verifiable, we need to know why. In

addition, it may be worthwhile to reevaluate how a GP is typically implemented. One alternative would be the development of a universal genetic program (UGP). Like a Universal Turing Machine, a UGP would get its specification from its input. In doing so, its complexity would be undefined initially and upon execution derive most of its information content from the universe.

10.5 Quantum Genetic Algorithms

The work with quantum genetic algorithms is far from being complete. A theory of convergence for a QGA needs to be developed. Because of this, it is not even possible to be certain as to whether there are any quantifiable advantages to using a QGA. Despite this, it seems likely that since we live in a quantum world, the knowledge to be gained from understanding a QGA might be beneficial. It may be possible that problems which are GA-hard or GA-semi-hard using a classical computer do not meet these criteria using a quantum computer.

A Main code

This is the main body of the code used for the analysis of the genetic algorithm. This was the same code that was used to produce the empirical results for all of the publications presented in this dissertation.

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

const int MAXPOP=400;
const int MAXSTR=1000;


void init_pop(int, int);
void print_pop(int, int);
void fitness(int,int);
void crossover(int,int);
void survives(int,int);
void sortchrom(int,int);
int* gene(int pop, int str);
long double  fit[MAXPOP];

int* TheLifePool;
int  MaxPopulation;
int  MaxStr, operand;
long double  fitnesstot;


//     max clique stuff

const int vert =     4;
const int maxclique = 3;

int g[vert][vert] ={

      {1,1,0,0},
      {1,1,1,1},
      {0,1,1,1},
      {0,1,1,1}};

int g[vert][vert]={

      {1,1,0,0,0},
      {1,1,1,1,0},
      {0,1,1,1,0},
      {0,1,1,1,1},
      {0,0,0,1,1}};
```

```
int g[vert][vert] ={

        {1,1,0,0,0,0},
        {1,1,1,1,0,0},
        {0,1,1,1,0,0},
        {0,1,1,1,1,0},
        {0,0,0,1,1,1},
        {0,0,0,0,1,1}};



int g[vert][vert] ={

        {1,1,0,0,0,0,0},
        {1,1,1,1,0,0,0},
        {0,1,1,1,0,0,0},
        {0,1,1,1,1,0,0},
        {0,0,0,1,1,1,0},
        {0,0,0,0,1,1,1},
        {0,0,0,0,0,1,1}};



int g[vert][vert] ={
        {1,1,1,1,0,0,0,0},
        {1,1,1,1,0,0,0,0},
        {1,1,1,1,1,1,1,0},
        {1,1,1,1,1,1,1,0},
        {0,0,1,1,1,1,1,0},
        {0,0,1,1,1,1,1,0},
        {0,0,1,1,1,1,1,0},
        {0,0,0,0,0,0,0,1}};



int g[vert][vert] ={
        {1,1,1,1,0,0,0,0,0},
        {1,1,1,1,0,0,0,0,0},
        {1,1,1,1,1,1,1,0,0},
        {1,1,1,1,1,1,1,0,0},
        {0,0,1,1,1,1,1,0,0},
        {0,0,1,1,1,1,1,0,0},
        {0,0,1,1,1,1,1,0,0},
        {0,0,0,0,0,0,0,1,0},
        {0,0,0,0,0,0,0,0,1}};
*/
```

```
//
//
//
//    Sort stuff
/*

int    sarray[14][4] = {                    // Solution array for Sort
                {1,1,1,0},
                {1,1,0,1},
                {1,1,0,0},
                {1,0,1,1},
                {1,0,1,0},
                {1,0,0,1},
                {1,0,0,0},
                {0,1,1,1},
                {0,1,1,0},
                {0,1,0,1},
                {0,1,0,0},
                {0,0,1,1},
                {0,0,1,0},
                {0,0,0,1},
                };

*/
//
//          Main function preliminaries
//
int   main()
{
      int i,z,stringnum,stringsize,evocount,busted=0,gens=0,evocnt=0;
      int gentot=0;

      double bestfit;

      srand((unsigned)time(NULL));//     Set the randomizer

            stringnum = 10;                        //     The number of chromomsomes in a population
      stringsize = vert;                  //     For clique stuff
//    stringsize = 50;

//    bestfit=pow(2,stringsize) - 1;
      bestfit= maxclique*100;

//          cout << "How many chromosomes do you want?  (up to 100) ";
//          cin >> stringnum;

//          cout << "What is the string size? \n";
//          cin >> stringsize;

      TheLifePool = (int*) malloc(sizeof(int) * stringnum * stringsize);
            MaxPopulation = stringnum;
      MaxStr = stringsize;
```

```
    for(z=0;z<=4999;z++){                    //    The big loop

        init_pop(stringnum,stringsize);      //    generate initial pop.

        print_pop(stringnum,stringsize);          //    print initial pop.

        evocount = 0;

        fitness(stringnum,stringsize);


        while(fitnesstot < bestfit*.98 && evocount<500){ evocount++;

            survives(stringnum,stringsize);

            crossover(stringnum,stringsize);

            fitness(stringnum,stringsize);

        }

        if(evocount>499)busted++;            //    statistics check

        cout<<"the final fitness is:  "<<fitnesstot<<"\n\n";
        cout<<"evocount is :  "<<evocount<<"\n\n";
        cout<<"The best individual at end is: \n";

        for(i=0;i<=stringsize-1;i++)
            cout<<*gene(0,i);
        cout<<"\n\n";

        cout<<"The second to worst individual is: \n";
        for(i=0;i<=stringsize-1;i++)
            cout<<*gene(stringnum-2,i);
        cout<<"\n\n";

        gentot=gentot + evocount;
    }

    print_pop(stringnum,stringsize);

    gentot=gentot/5000;

    cout<<" The average number of generations to a solution is: \n\n "<<gentot<<"\n\n";
    cout<<" The number of busted runs,  "<< busted <<"\n\n";
    print_pop(stringnum,stringsize);
    return 0;

}


//
```

```
//                      Various Support Functions
//


int* gene(int pop, int str)
{
      return TheLifePool + (pop * MaxStr) + str;
}



void init_pop(int num, int size)
{

    int i,j;

       for(i=0;i<=num-1;i++)
             for(j=0;j<=size-1;j++)
             {
                   *gene(i,j) = rand() % 2;
             }
}



void print_pop(int num,int size)
{

       int i,j;
       cout<<"\n\n";
       for(i=0;i<=num-1;i++){
             for(j=0;j<=size-1;j++)
                   cout << *gene(i,j);
        cout << "\n";
       }
       cout << "\n";
}



void survives(int num,int size)

{
       int i,j;

       for(i=num/2;i<=num-1;i++)
             for(j=0;j<=size-1;j++)
                   *gene(i,j) = 0;

}
```

```
void sortchrom(int num, int size)
{
      int i,j,x,h;
      long double t;

      for(i=0;i<=num-2;i++){
            for(j=i+1;j<=num-1;j++){
                  if(fit[i]<fit[j]){

                                    t=fit[i];
                                    fit[i]=fit[j];
                                    fit[j]=t;
                                    for(x=0;x<=size-1;x++){

                                          h= *gene(i,x);
                                          *gene(i,x) = *gene(j,x);
                                          *gene(j,x) =h;

                                    }
                  }
            }
      }
}


void crossover(int num, int size)

{
      int i,j,crosspt,par1,par2,mut;

      for(i=num/2;i<=num-1;i++){
            crosspt= rand() %(size);
            par1= rand() %(num/2);
            par2= rand() %(num/2);

          for(j=0;j<=crosspt-1;j++)
            *gene(i,j) = *gene(par1,j);

          for(j=crosspt;j<=size-1;j++)
                  *gene(i,j) = *gene(par2,j);

            for(j=0;j<=size-1;j++){                 //    1% chance of a mutation
                  mut= rand() %(100);
                  if(mut<=1){
                        if(*gene(i,j)==1)
                              *gene(i,j)=0;
                        else
                              *gene(i,j)=1;
                  }
            }

      }
}
```

```
//                      Various Fitness Functions
//
//
//    Fitness for Max Clique
//

void fitness(int num, int size)

{
      int clique,flag;
      int i,j,k;

      fitnesstot=0;
      for(i=0;i<=num-1;i++)            //    initialize fitness of strings to zero
            fit[i]=0;

      for(i=0;i<=num-1;i++){
            flag=1;
            for(j=0;j<=size-1;j++){
                  if(*gene(i,j)==1){
                        for(k=0;k<=size-1;k++){
                              if(*gene(i,k)==1){
                                    if(g[j][k]!=1)
                                          flag=0;
                              }
                        }
                  }
            }
            clique=0;
            if(flag==1){
                  for(j=0;j<=size-1;j++){
                        if(*gene(i,j)==1)
                              clique++;
                  }
            }

            fit[i]=clique*100;

            fitnesstot=fitnesstot+fit[i];

      }

      fitnesstot=fitnesstot/num;
      sortchrom(num,size);
}

//
//    End of Clique fitness
//
```

```
//
//    Fitness for 1's binary rep
//

void fitness(int num, int size)
{

      int i,j;

      fitnesstot=0;

      for(i=0;i<=num-1;i++)      //     initialize fitness of strings to zero
            fit[i]=0;


      for(i=0;i<=num-1;i++){
            for(j=0;j<=(size-1);j++)
                  if (*gene(i,j) == 1) fit[i]=fit[i]+pow(2,size-j-1);

            fitnesstot=fitnesstot+fit[i];
      }

      sortchrom(num,size);

}

//
//    Fitness for 1's long rep
//

void fitness(int num, int size)
{

      int i,j,sum;

      fitnesstot=0;

      for(i=0;i<=num-1;i++)      //     initialize fitness of strings to zero
            fit[i]=0;

      for(i=0;i<=num-1;i++){
            sum=0;
            for(j=0;j<=size-1;j++){
                  if(*gene(i,j)==1) sum++;
            }
            fit[i]=sum;
            fitnesstot=fitnesstot+fit[i];
      }


      sortchrom(num,size);

}
```

```
//
//    fitness for sort
//

void fitness(int num, int size)
{

      int i,j,k,l,sum;

      fitnesstot=0;

      for(i=0;i<=num-1;i++)      //     initialize fitness of strings to zero
            fit[i]=0;

      for(i=0;i<=num-1;i++){
            k=0;
            for(j=0;j<=13;j++){
                  sum=0;

                  for(l=0;l<=3;l++){
                        if(*gene(i,k)==sarray[j][l]){
                              sum++;
                        }
                        k++;
                  }

                  if(sum==4) fit[i]=fit[i]+10;

            }


            fitnesstot=fitnesstot+fit[i];

      }
      fitnesstot=fitnesstot/num;

      sortchrom(num,size);

}
```

```
//    fitness for simple rectangle
//
//

void fitness(int num,int size)
{

      int i,j;
      long double tot1,tot2;

      fitnesstot=0;

      for(i=0;i<=num-1;i++)      //    initialize fitness of strings to zero
            fit[i]=0;


      for(i=0;i<=num-1;i++){     //    This will check each string for fitness

                  tot1=0;
                  tot2=0;

                  for(j=0;j<=(size-1)/2;j++)
                    if (*gene(i,j) == 1) tot1=tot1+pow(2,size/2-j-1);


                    for(j=size/2;j<=size-1;j++)
                          if (*gene(i,j) == 1) tot2=tot2+pow(2,size-j-1);



                    fit[i] = tot1*tot2;
                    if(fit[i]>ans) fit[i]=0;
                    fitnesstot =fitnesstot + tot1*tot2;


      }
      fitnesstot=fitnesstot/num;

      sortchrom(num,size);
}
```

```
//
//              Fitness for factor
//

void fitness(int num,int size)
{

    int i,j,temp[MAXSTR];
    long double tot1,tot2;

    fitnesstot=0;

    for(i=0;i<=num-1;i++)     //    initialize fitness of strings to zero
        fit[i]=0;

    for(i=0;i<=size-1;i++)//    initialize a temporary chromosome
        temp[i]=0;


    for(i=0;i<=num-1;i++){    //    This will check each string for fitness

            tot1=0;
            tot2=0;

            for(j=0;j<=(size-1)/2;j++)
              if (*gene(i,j) == 1) tot1=tot1+pow(2,size/2-j-1);              // totals left side


              for(j=size/2;j<=size-1;j++)
                  if (*gene(i,j) == 1) tot2=tot2+pow(2,size-j-1);     // totals right side


              if(tot1*tot2 <= operand)
                  fit[i] = tot1*tot2;
              if(tot1*tot2 < operand)
                  fit[i] = tot1*tot2/operand;
              if(tot1*tot2 > operand)
                  fit[i] = 0;

              cout<<"fitness for "<<i<<"is : "<<fit[i]<<"\n";
              fitnesstot =fitnesstot + fit[i];


    }

    cout<<"The average fitness is:  "<<fitnesstot/num<<"\n";
    sortchrom(num,size);
}
```

References

[Ank91] Ankenbrandt, C., An Extension To the Theory of Convergence and A Proof of the Time Complexity of Genetic Algorithms, *Foundations of Genetic Algorithms*, pp. 53-68, Morgan Kaufman, 1991.

[BC94] Bovet, D., Crescenzi, P. *Computational Complexity*, Prentice Hall,1994.

[BDG88] Balcàzar, J., Díaz, J., and Gabarró, J. *Structural Complexity Theory I*, Springer-Verlag, 1988.

[BH91] Back, T., Hoffmeister, G., Extended Selection Mechanisms in Genetic Algorithms, *Proceedings of the Fourth International Conference on Genetic Algrotihms*, Morgan Kaufman, 1991.

[Dav91] Davidor, Y., Epistasis Variance: A Viewpoint on GA-Hardness, *Foundations of Genetic Algorithms*, pp. 75-92, Morgan Kaufman, 1991.

[DeJ75] DeJung, K. An Analysis of the Behavior of Genetic Adaptive Systems, Ph.D. Thesis, University of Michigan, Ann Arbor

[Deu85] Deutsch, D., Quantum Theory, the Church-Turing Principle, and the Universal Quantum Computer, *Proceedings Royal Society London, Vl. A400*, pp. 97-117, 1985.

[DG92] Deb, K., Goldberg, D., Analyzing Deception in Trap Functions, *Foundations of Genetic Algrithms 2*, pp. 93-108, Morgan Kaufman, 1992.

[DS93] DeJung, K. Sarma, J., Generation Gaps Revisited. *Foundations of Genetic Algorithms 3*, pp. 19-28, Morgan Kaufman, 1993.

[FM93B] Forrest, S., Mitchell, M., Relative Building-Block Fitness and the Building-Block Hypothesis, *Foundations of Genetic Algorithms 3*, pp. 109-126, Morgan Kaufman, 1993.

[GD91] Goldberg, D., Deb, K., A Comparative Analysis of Selection Schemes used in Genetic Algorithms. *Foundations of Genetic Algorithms*, pp. 69-93, Morgan Kaufman, 1991.

[Gre92] Grefenstette, J., Deception Considered Harmful, *Foundations of Genetic Algorithms 2*, pp. 75-92, Morgan Kaufman, 1992.

[Gol89b] Goldberg, D. E., *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[Gro96] Grover, L., A Fast Quantum Mechanical Algorithm for Database Search, Proceedings *of the 28th Annual ACM Symposium on the Theory of Computing*, pp. 212-219, 1996.

[GWC98] Ge, Y., Watson, L., Collins, E.: Genetic Algorithms for Optimization on a Quantum Computer, Unconventional Models of Computation, Springer-Verlag, London, 1998.

[Han94] Hancock, P., An Empirical Comparison of Selection Methods in Evolutionary Algroithms. *Evolutionary Computing: AISB Workshop* Leeds, U.K. April 1994, Selected Papers, Springer-Verlag, 1994.

[Has96] Håstad, J. (1996). Clique is Indeed Hard ,*Proc.on STOC*, 1996.

[Hol75] Holland, J., *Adaptation in Natural and Artificial Systems*, Ann Arbor, MI, University of Michigan Press, 1975.

[LGP00] Lobo, F., Goldberg, D., Pelikan, M., Time Complexity of Genetic Algorithms on Exponentially Scaled Problems, *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 151-158, Morgan Kaufman, 2000.

[Llo96] Lloyd, S., Universal Quantum Simulators, Science, Vol. 273, pp. 1073-1078, 1996.

[LV90] Li, M., Vitanyi, P., Kolmogorov Complexity and its Applications, *Handbook of Theoretical Computer Science Volume A. Algorithms and Complexity*, pp. 189-254. The MIT Press, Cambridge, Massachusetts,1990.

[LVo90] Liepins, G., Vose, M., Representation Issues in Genetic Optimization, J. Experimental and Theoretical Art. Intell. 2, pp.101-115, Morgan Kaufman, 1990.

[LV91] Liepins, G., Vose, M., Deceptiveness and Genetic Algorithm Dynamics, *Foundations of Genetic Algorithms*, pp. 36-52, Morgan Kaufman, 1991.

[LSP+99] Langdon, W., Soule, T., Poli, R., Foster, J.: The Evolution of Size and Shape, *Advances in Genetic Programming Volume III*, pp. 163-190. The MIT Press, Cambridge, Massachusetts, 1999.

[MF98] Marconi, J., Foster, J., Finding Cliques in Keller Graphs with Genetic Algorithms, *Proc. International Conference on Evolutionary Computing*, 1998.

[MFH92] Mitchell, M., Forrest, S., Holland, J., The Royal Road for Genetic Algorithms: Fitness Landscapes and GA Performance. Toward a Practice of Autonomous Systems, *Proceedings of the First European Conference on Artificial Life*. MIT Press, 1992.

[Muh92] Muhlenbein, H., How Genetic Algorithms Really Work: 1. Mutation and Hillclimbing, *Parallel Problem Solving from Nature 2*. North-Holland, 1992.

[NM98] Narayan, A., Moore, M.: Quantum Inspired Genetic Algorithms, Technical Report 344, Department of Computer Science, University of Exeter, England, 1998.

[O'Re95] O'Reilly, U.: An Analysis of Genetic Programming, Doctoral Thesis, School of Computer Science, Carleton University, Ottawa, Ontario, Canada, pp. 14, 1995.

[Raw91] Rawlins, G.J.E., Introduction, *Foundations of Genetic Algorithms*, pp. 1-10, Morgan Kaufman. 1991.

[RF00] Rylander, B., Foster, J.: GA-Hard Problems, *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 367, Morgan Kaufman, 2000.

[RF01a] Rylander, B., Foster, J., Computational Complexity and Genetic Algorithms, *Proceedings of the World Science and Engineering Society's Conference on Soft Computing, Advances in Fuzzy Systems and*

Evolutionary Computation, pp. 248-253, World Science and Engineering Press, 2001.

[RF01b] Rylander, B., Foster, J., Genetic Algorithms, and Hardness, *Proceedings of the World Science and Engineering Society's Conference on Soft Computing, Advances in Fuzzy Systems and Evolutionary Computation*, pp. 323-329, World Science and Engineering Press, 2001.

[RSF00] Rylander, B., Soule, T., Foster, J.: Quantum Genetic Algorithms, *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 373, Morgan Kaufman, 2000.

[RSF01] Rylander, B., Soule, T., Foster, J. (2001) Computational Complexity, Genetic Programming, and Implications, *Proceedings of the European Genetic Programming Conference*

[RSF+01] Rylander, B., Soule, T., Foster, J., Alves-Foss, J., Quantum Genetic Programming, *Proceedings of the Genetic and Evolutionary Computation Conference*, Morgan Kaufman, 2001.

[Ryl00] Rylander, B., On GP Complexity, *Proceedings of the Genetic and Evolutionary Computation Conference Workshop Program*, pp. 309-311, Morgan Kaufman, 2000.

[SCE+89] Schaffer, J., Caruana, R., Eshelman, L, Das, K., A Study of Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization, *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufman,1989.

[SD91] Spears, W., DeJung, K., An Analysis of Multi-Point Crossover, *Foundations of Genetic Algorithms*, pp. 301-315, Morgan Kaufman, 1991.

[SFD96] Soule, T., Foster, J., Dickinson, J., Code Growth in Genetic Programming, *Proceedings Genetic Programming Conference*, pp. 215-223, Morgan Kaufman, 1996.

[SFD98] Soule, T., Foster, J., and Dickinson, J., Using Genetic Programming to Approximate Maximum Cliques, *Proceedings Genetic Programming*. pp. 400-405, Morgan Kaufman, 1998.

[Sho94] Shor, P., Algorithms for Quantum Computation: Discrete Logarithms and Factoring, *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pp. 124-134, 1994.

[Spe92] Spears, W., Crossover or Mutation?, *Foundations of Genetic Algorithms 2*, pp. 221-238, Morgan Kaufman, 1992.

[Vos99] Vose, M., *The Simple Genetic Algorithm*, Morgan Kaufman, 1999.

[WC97] Williams, C., Clearwater, S., *Explorations in Quantum Computing.* Springer-Verlag, New York, Inc., 1997.

[Whi91] Whitley, L.D., Fundamental Principles of Deception in Genetic Search, *Foundations of Genetic Algorithms*, pp. 221-242, Morgan Kaufman, 1991.

[WM95] Wolpert, D., Macready, W., No Free Lunch Theorems for Search. The Santa Fe Institute. Santa Fe, NM: 1995.

[WM96a] Wolpert, D., Macready, W., No Free Lunch Theroems for Optimization. The Santa Fe Institute. Santa Fe, NM:1996.

[WM96b] Wolpert, D., Macready, W., What Makes an Optimization Problem Hard? The Santa Fe Institute. Santa Fe, NM:1996.

[Zal98] Zalka, C., Efficient Simulation of Quantum Systems by Quantum Computers, Los Alamos National Laboratory, preprint archive, quant-ph/9603026, 1998.