University of Alberta

A Theoretical and Empirical Analysis of α-ary Landscapes for Genetic Algorithms

by

Jonathan Lichtner

Technical Report TR 97-04
May 1997

DEPARTMENT OF COMPUTING SCIENCE
The University of Alberta
Edmonton, Alberta, Canada

University of Alberta

A Theoretical and Empirical Analysis of α-ary Landscapes for Genetic Algorithms

by

Jonathan Lichtner

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 1997

Abstract

Genetic algorithms (GAs) are probabilistic search algorithms that are loosely based on biological evolution. Analyzing genetic algorithms has proven difficult, for a variety of reasons, but a landscape paradigm that rigorously models the search of GAs has become increasingly popular in their analysis. So far much of this analysis concerns binary representations, where each member of the population is a binary string. In this thesis we consider using α-ary representations—using strings where each character can be one of α characters—and conduct a theoretical and empirical study of the resulting α-ary landscapes.

In terms of theory, we discuss the types of landscape graphs produced by various α-ary crossover and α-ary mutation operators. We relate these landscapes to a common class of graphs, the α-ary hypercubes. We then generalize a binary mutation-crossover isomorphism to higher alphabets and use this isomorphism to show that α-ary crossover can simulate α-ary mutation. Since crossover can simulate mutation, crossover must be at least as powerful as mutation, in the sense of computational power. Because this α-ary mutation-crossover isomorphism is closely related to an α-ary Gray code, extending our simulation to "hyper orders" is a generalization of iterating the landscape's representation (and thus the landscape) using this α-ary Gray code. If we repeatedly apply this Gray code, the landscapes will eventually cycle. We prove a theorem on the maximum number of landscapes produced when this Gray code is iterated.

We explore the long path problem for α-ary mutation and α-ary crossover. We construct exponentially long distance-preserving paths for α-ary mutation, improving on a previous method. We also construct exponentially long distance-preserving paths for crossover between two complementary binary strings, and discuss the problem of creating long distance-preserving paths for populations greater than two.

In Chapter 7, we create the schizophrenic function, a function designed to discriminate between mutation and crossover. This function has two optima classes, and is constructed in such a way that mutation should find one optimum, crossover the other. Empirical tests show that the schizophrenic function is a useful but imperfect tool. More importantly, these tests offer insight into how crossover works.

Finally, we show how to generate an α-ary Gray code efficiently in sequence (constant amortized time per codeword). We generalize this Gray code to "multary" strings, and generalize our algorithm to generate this code in constant amortized time per codeword.

Acknowledgements

There are several people who made a direct contribution to this thesis, all deserving thanks. First, and foremost, is my supervisor, Joseph Culberson, for time spent deciphering and editing earlier versions of this thesis, and for helping to focus my research in the right direction and for giving me some of his research problems. I thank the members of my examining committee, Maziar Shirvani and Lorna Stewart, for time spent reading my thesis and their helpful suggestions. I'd also like to thank Jim Hoover for chairing my examining committee.

Several chapters of this thesis were based on papers, either published or submitted, and the comments of the anonymous referees helped me to polish these portions of my thesis. I also thank Kevin Charter, Kurt Lichtner, and Basil Vandegriend for reading my thesis and giving helpful advice.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We divide our introduction into three sections. In the first, we provide some motivation for studying α-ary landscapes and landscapes in general. This is done by showing that many search algorithms, including genetic algorithms (GAs) and hill climbers, can be modelled using the landscape paradigm. In the second, we give some necessary background material on landscapes. Finally, we provide an overview of this thesis.

1.1 Motivation

Genetic algorithms [22, 27, 39] (GAs) are a diverse class of probabilistic algorithms that are loosely based on biological evolution. In this thesis we consider, for the most part, using GAs to optimize functions. That is, we have a function $f : \mathcal{S}^\ell \to \mathcal{R}$, where $\mathcal{S} = \{0, 1, \cdots, \alpha - 1\}$ and $\mathcal{R}$ is the set of real numbers, and we want to find an optimal or near-optimal string $x \in \mathcal{S}^\ell$.

Since there is no precise definition of what does or does not constitute a GA, we instead list four distinguishing features that encompass most GAs: the domain of the function to be optimized is represented by strings, which are akin to chromosomes in biological genetics; the GA maintains a population of these strings, a multiset of the representation space; each string has a fitness, a measure of the worth of the solution represented by the string; the populations are changed, or evolved, using genetic operators such as mutation or crossover, biased by the fitness of the strings.

Mutation, in its most general form, takes a single string, picks certain characters of that string, and randomly replaces those characters with new characters. For example, the string 00000 can be mutated to 00230. Crossover takes two strings $x = x_1 x_2 \cdots x_\ell$ and $y = y_1 y_2 \cdots y_\ell$ and produces the strings x' and y' where ($x'_i = x_i$ and $y'_i = y_i$) or ($x'_i = y_i$ and $y'_i = x_i$). Crossover is sometimes called recombination.

These common features can be combined to form a generic "template" GA:

1. Create an (initial) population of strings. This initial subset is often chosen randomly, but this need not be the case.

2. Assign each string a fitness value using a *fitness function*, $f()$.

```
Create an initial random, n-string population.
While (not done) {
        Repeat { /* create new population */
              - Stochastically select two parent strings from the old
                population (with replacement). The probability a string
                is selected is an increasing function of its fitness.
              - With some crossover probability, p_c, cross this
                pair to make a pair of children. With some mutation
                probability, p_m, mutate each character of each child.
              - Add the children to the new population.
        } until (≥ n children strings have been produced);
        Replace old population with children.
        Drop one string from population if n is odd.
}
```

Figure 1.1: Outline of a simple genetic algorithm, based on Mitchell [39, pp. 10-11].

3. Create a new population by applying various genetic operators, such as mutation and crossover, to the old population, biased by the fitness of the strings.

By iterating steps 2 and 3, the population will evolve over time.

An infinite number of algorithms will fit into the generic template just given, but most GAs can be represented by a *simple genetic algorithm* (see Figure 1.1). While many GAs are based on this simple GA, some are not. Some non-traditional genetic algorithms include CHC [13] and GENITOR [50]. The CHC algorithm merges the old population with the new, keeping the n most fit strings in the merged population. GENITOR replaces the least fit string of the population with a newly generated string iff the new string has a higher fitness than the old. GIGA and NQ-GIGA, two of the GAs used in this thesis, are also not based on simple GAs. See Appendix B for descriptions of GIGA, NQ-GIGA, and some other related search algorithms.

Analyzing genetic algorithms has proven difficult. This is amply demonstrated by the controversy over whether crossover is more powerful than mutation, or vice versa [46]. There are two main camps in the GA community: those who believe that crossover is the power behind GAs while mutation is just a secondary operator, used only to introduce variation into the population; and those who believe that crossover is not needed or is intrinsically less powerful than mutation. (In addition to these two traditional opinions about crossover and mutation, there is a view that holds that both mutation and crossover are useful search operators [41].) This crossover-mutation debate is important because, as some have noted [34, 14], GAs are usually distinguished from other evolutionary algorithms by using populations with crossover. If crossover is not useful, then GAs themselves may not be useful as function optimizers.

Part of the difficulty with this mutation-crossover debate is that it is hard to define precisely what it means for one operator to be more powerful than another. One possible approach is to try both mutation and crossover on a suite of test functions. If

one does better than another, then that can be taken as evidence that that operator is more powerful on those functions. In this way, classes of problems may be discovered that seem easier for one operator than for another. For instance, Fogel and Atmar [17] find that mutation is more useful than crossover on a particular class of functions, and further make the claim that crossover "cannot be the hallmark of a broadly useful algorithm."[1] There is a danger in extrapolating from one class of functions to the general case: tests on a suite of functions say little about functions or problems not modelled in that suite. However, experiments can help determine where crossover seems to be useful. Eshelman and Schaffer [14] suggest that crossover is useful, but only in a small niche of problems. Jones [34] suggests that even when the addition of crossover seems to help the search, it may just be doing a macro-mutation, and so crossover's niche may be even smaller than it appears.

While useful, experiments alone cannot answer the mutation-crossover debate, because there are many possible crossover operators, mutation operators, and implementations that the researcher has to choose from, and each choice may affect the results dramatically. It may be that mutation is better than one type of crossover operator yet worse than another. Or crossover may be potentially better than mutation, but an algorithm may not use that potential.

For these reasons, this debate has also been addressed theoretically: Spears [46] analyzes crossover and mutation in terms of their ability to construct and disrupt useful information, and Culberson [8] uses an isomorphism between mutation and crossover to compare their relative search capabilities. Part of this thesis extends the work of Culberson to further address the mutation-crossover debate.

Another example of the difficulty in analyzing GAs can be seen in the analysis of GAs on Royal Road functions [18, 19, 40]. The Royal Road functions are, intuitively, better suited to crossover than mutation. However, when tested, a mutation hill climber outperformed a traditional GA [19], even though an "idealized" GA has been theoretically shown to be faster than the mutation hill climber (by a linear factor). Thus, crossover has the potential to do better than mutation, but an algorithm may not use this potential even though it uses crossover. Interestingly, GIGA performs very well on the Royal Road functions [7].

There are many other examples of analysis gone wrong in the GA literature. For example, Davis [11] notes that a simple mutation hill climber often outperformed GAs on a suite of test functions designed to demonstrate the power of GAs. Grefenstette [25] observes that certain classes of problems that are supposed to be easy for GAs are actually hard, and some classes that are supposed to be hard are actually easy.

One of the reasons analyzing GAs can be difficult is that there are many different genetic algorithms and an analysis of one may not apply to another. It is tempting to think that because different GAs may share much in common (e.g., use of populations and similar operators) an analysis will carry over from one to the other. However, an analysis of a particular GA may become irrelevant if the algorithm is modified,

[1] They use mutation and crossover on real-valued strings, and so their results say little about discrete forms of mutation and crossover, just as our analysis says little about real-valued operators.

even slightly. For example, we have shown [10] that small changes in search strategy (elitism vs. non-elitism, full-neighbourhood sampling vs. partial-neighbourhood sampling) can make a problem go from being easy to exponentially difficult. An analysis may often be robust, but extrapolation must be done with caution.

Even if we restrict our focus to one genetic algorithm, further difficulties result because GAs are applied to many problems, and it is often expected that any analysis should apply to all possible problems. This "black box" mind-set—i.e., treat the fitness function as a black box module separate from the GA—is an analytical dead-end since an analysis on one problem may not carry over to another. It is impossible using the "black box" or "blind" model of search to do better on average than a random enumerative search over all problems. This is noted as early as 1991 by Rawlins [42], and is proven formally by Wolpert and Macready [52]. Culberson [9] gives a more intuitive perspective on the limitations of blind search. This means we cannot separate the problem from the algorithm; the two are intertwined, and any analysis must take both into account.

GAs are also very complex, which adds further difficulty. For instance, we may be forced to make several abstractions in order to simplify the analysis. It is possible that the assumptions may be incorrect, and while our analysis on the simplified problem may be correct, it may not apply to the more general instance.

Landscapes [8, 33] are one possible approach to analyzing genetic algorithms. In the landscape paradigm, all probabilistic searches can be seen as wandering through some space of possible solutions, or *search space*. A search can be represented by a graph where each node in the graph represents some point in the search space. For GAs, for instance, each possible population is a point in the search space. The search algorithm moves through this landscape by applying various operators, and these operators define edges in the graph. That is, they define how the landscape can be traversed. Each node is labelled with its fitness value. Thus we can speak of various landscape features such as paths, peaks, and slopes, etc.[2]

The landscape paradigm does not solve the problem of GA analysis, but it does give a precise model to work with which is important since it is easy to prove theorems that are not tied to any algorithm. It is also relatively easy to simplify the model. Some other advantages of using landscapes are:

- Landscapes are very general and can be used for almost any search problem. Techniques developed for GAs and hill climbers may also apply to other algorithms. For instance, Jones notes that the fields of AI and Operations Research have views of search that are related to landscapes [33].

- Landscapes allow us to divide the analysis of most search algorithms into two main components [33]: the landscape, and the navigation strategy (how the algorithm traverses the landscape graph).

- Landscapes fall into classes, and analysis can be applied to those classes.

[2] We leave the definitions for the next section.

The second point is useful, since there are often "natural" operations that define a landscape. The features of this landscape may affect which navigation strategy should be used. For example, different navigation strategies—working on the same landscape—can result in either linear or exponential performance [10].

The last point is especially important. Being able to lump landscapes into classes instead of having to consider all landscapes at once or only a single landscape at a time allows us to get an analysis of GAs that is general but that is not so general as to be useless.

Most landscape analysis so far has only examined landscapes based on binary string representations. If we change our representation to α-ary strings, then different landscapes result, and these landscapes may be more suitable for searching than their binary counterparts. At the very least, studying α-ary landscapes and landscapes in general should lead to a better understanding of GAs.

Finally, much of our analysis is closely connected to graph theory since landscapes are graphs with a fitness measure on the vertices (and, possibly, probabilities on the edges). Some of our results may be of interest to non-GA researchers. For example, we explore the problem of generalizing distance-preserving paths, known in the GA community as the long path problem, to α-ary hypercubes. In Chapter 5 we prove a theorem on the number of unique codes produced when an α-ary Gray code is iterated, and in Appendix A, we prove that this α-ary code really is a Gray code and develop an algorithm to generate it efficiently in sequence (constant amortized time per code word). We further generalize this code and code generation algorithm to include a class of "multary" Gray codes.

1.2 Landscape Background

A landscape is an abstract way of viewing many search algorithms. Because landscapes are commonly misused it is prudent to define what we mean by a landscape. We will also define several measures and features of landscapes. The reader may also wish to see Jones's thesis [33] or Culberson [8] for similar definitions. We base some of our definitions on the two former works and also use some of the definitions from the paper, "On Searching α-ary Hypercubes and Related Graphs" [10].

For simplicity we restrict our discussion of landscapes to algorithms that work on length ℓ, α-ary strings. For example, if $x = x_1 x_2 \cdots x_\ell$ is an α-ary string, then each character of x (the x_is) can take one of α characters. We use the character set $\{0, 1, \cdots, \alpha\text{-}1\}$. The *search space* is the set of all possible populations, where each member of a population is an α-ary, length ℓ string. Each population in this search space is represented by a vertex v in the landscape.

Each vertex is labelled with the fitness of the population represented by this vertex, $g(v)$. The fitness function on strings, $f()$, and the fitness function on the population, $g()$, are equivalent only when the population is of size one, although $g()$ is usually based on $f()$. For example, the fitness of a population could be the maximum fitness of all strings within the population. In effect, $g()$ assigns a "height" to each vertex; the goal of a search algorithm (when maximizing) is to find the highest such point.

Two vertices v and w are connected by an edge *iff* the population represented by w is a result of applying a genetic operator to v. Thus the operators induce edges in the graph. It is possible to model algorithms that apply more than one operator at a step by treating these operators as a single compositive "population" operator.

We illustrate what we mean by population operators inducing edges in the landscape graph using a simple example on binary strings. Consider an algorithm that maintains a population of three strings, and generates a new population by uniformly applying one-point mutation, one-point crossover[3], or both to the old population. Then the populations $v_1 = \{01101, 00010, 11011\}$, $v_2 = \{01101, 00110, 11011\}$, $v_3 = \{01011, 00010, 11101\}$ and $v_4 = \{01011, 00110, 11101\}$ are four points in the search space. Further, v_1 and v_2 are connected by the one-point mutation operator (mutate the second string), v_1 and v_3 are connected by the one-point crossover operator (cross the first and third strings), and v_4 is connected to v_1, v_2, and v_3 (mutation *and* crossover on v_1, crossover on v_2, mutation on v_3). However, if our algorithm *only* applied one-point crossover or one-point mutation to the population (not both at the same time) then there would be no edge between v_1 and v_4.

Each edge (v, w) is labelled with the probability that an operator produces w from v; this is different than the probability that a specific search algorithm will move from v to w. For example, if $g(v) > g(w)$, then an elitist algorithm will never move from v to w, but a non-elitist algorithm might.

Thus a landscape can be represented as a graph $G = (V, E)$ with population fitness function $g()$ on the vertices in V and probabilities on the edges. There are directed and undirected landscapes, but in this thesis we only consider operators that define undirected landscapes, since we use symmetric operators and populations of constant size. (Crossover between two strings to produce one child will not produce symmetric landscapes; Jones's landscapes [33] is more general than Culberson's [8] in that such crossover types and the resulting directed landscapes are modelled.) Further, if the operators used generate all the neighbours of a vertex v with equal probability, then we do not need to label edges with probabilities. This will frequently be the case. We call the graph underlying a landscape the *landscape graph*.

Given two vertices in the landscape, say v_0 and v_i, there is a *path* between them *iff* there is a sequence of vertices $v_1, v_2, \cdots, v_{i-1}$ such that $\forall j, 0 \leq j < i, (v_j, v_{j+1}) \in E$. The *length* of a path is the number of edges contained within the path. Let $N(v)$ be the set of neighbours of v. The *degree* of a landscape graph is $\max_{v \in V} |N(v)|$. The *distance*, $dist(v, w)$, between two points v and w is the length of the shortest path between them. The *diameter* is $\max_{v,w \in V} dist(v, w)$. Two points are *connected* iff there is a path between them. A *region* in the landscape graph is a set of points that induce a connected subgraph. Let X be a region. Then $N(X) = \{v : w \in X, (v, w) \in E, v \notin X\}$; that is, we overload $N()$ to work on points and regions.

Recall that $g()$ gives the fitness of the vertices in the landscape. A region X is a *peak iff* $\forall v \in X, \forall w \in N(X), g(v) = c$ and $g(w) < c$ where c is a constant. A peak is

[3]These are defined formally in Chapter 2. One-point mutation takes a string and mutates a single character. One-point crossover takes two strings, picks a crossover point, and exchanges the tail-ends of the strings to produce two children; the two children then replace their parents.

Generate initial population v
While (not done) {
- generate some subset S of $N(v) \cup v$
- let $v = \max_{v_i \in S} g(v_i)$
}

Table 1.1: Outline of a generic hill climbing algorithm.

optimal iff it has at least as high a fitness value as any point not in the peak. A peak is a *false optimum iff* it is not optimal. A path $p = v_i, v_{i+1}, \cdots, v_j$ (with start v_i) is *strictly increasing iff* $g(v_k) < g(v_{k+1}) \forall k, i \leq k < j$.

In summary, a landscape is a graph, where each vertex represents a population. Edges in the graph are induced by genetic or "population" operators, and $g()$ takes the vertices of a landscape graph and assigns a "height" to them, which gives us the landscape paradigm. As well, an edge (v, w) may be labelled with the probability that the operator produces w from v.

Almost any probabilistic search algorithm can be analyzed using landscapes, where the algorithm walks along the landscape graph, an edge at a time. This means that even GAs can be seen as doing a local search. Further, this landscape view has no "magic" operators: the landscape of a GA is *not* the binary hypercube and crossover does not "warp" or "jump" through this space. Instead, both mutation and crossover are modelled together.

The landscape for a GA can be very complex, and so we usually reduce its complexity by making abstractions or simplifications. This will usually be done by studying the graphs induced by a single genetic operator on minimal populations, and conducting experiments with search algorithms on these simplified landscapes (e.g., using simple hill climbers). See Table 1.1 for an outline of a generic hill climber. If v is included in S, then the hill climber is elitist, otherwise it is non-elitist. An elitist hill climber that only moves uphill is *strictly* elitist. This definition allows even some population-based search algorithms to be viewed as hill climbers.

1.3 Overview of this Thesis

In this thesis we conduct a theoretical and empirical analysis of α-ary landscapes for genetic algorithms. We do this from an algorithmic point of view, in which the search of GAs is modelled as searching graphs. We use the term landscape for this combination of graph and fitness of the graph vertices.

Having decided to use α-ary strings to represent members of our population, in Chapter 2 we examine the landscape graphs induced by several variants of α-ary crossover and α-ary mutation. We also show that α-ary mutation searches an α-ary hypercube, and because of this, we discuss some properties of the hypercube. We then show in Chapter 3 that α-ary mutation is isomorphic to a form of α-ary crossover. This is a generalization of a binary crossover-mutation isomorphism [8].

This generalized isomorphism can be used (Chapter 4) to show that a GA or hill

climber that uses mutation and crossover can be simulated by one that uses only crossover. Additional operators can be included in this simulation. This shows that crossover is at least as powerful as mutation in the sense of computational power, and suggests that the old question, "Which is better, crossover or mutation?" should be replaced with two new questions: "Are the various operators being used to their potential?" and "Which operator is better suited to a particular problem?" The technique underlying this simulation can be extended to hyper-orders (hyper-order simulation), and this analysis illustrates the complexity of crossover.

A special case of this hyper-order simulation leads to the notion of iterating an α-ary Gray code. That is, we start with the code representing the natural numbers of base α, length ℓ, and repeatedly apply a Gray code mapping until the code cycles. We iterate this Gray code in Chapter 5 and prove a theorem on the number of unique codes produced when this Gray code is iterated. Alternatively, we can view iterating strings under this Gray code mapping as iterating the representation or, equivalently, the landscape of a GA or hill climber. This theorem gives the maximum number of landscapes produced when iterated under the Gray code.

In Chapter 6 we discuss the long path problem for α-ary mutation and show how to construct these paths. Our construction gives a new lower bound on the maximum length of distance-preserving paths in α-ary hypercubes. We also develop long paths for α-ary crossover. We create an exponentially long distance-preserving path for crossover between a complementary pair of binary strings. We then try extending these long paths to bigger populations and do some tests on the resulting landscapes. The results are interesting in that crossover appears to follow exponentially long paths. However, these results are preliminary.

Having developed some theory about α-ary mutation and α-ary crossover—in particular, having shown that crossover is *potentially* as powerful as mutation—we develop the schizophrenic function in Chapter 7. This function has two optima classes, and is constructed in such a way that mutation should find one optimum, crossover the other; our goal is to construct a function able to discriminate between searches that use crossover well, and those that do not use crossover at all or do not use it well. Empirical tests for $\alpha = 2$ and $\alpha = 4$ show that the schizophrenic function is a useful but imperfect tool.

In Chapter 8 we give a quick survey of some previous landscape analysis in GAs. We conclude our thesis in Chapter 9. We prove in Appendix A that the code of Sharma and Khanna [43] really is a Gray code, and give an algorithm that generates each code word of it in sequence in constant amortized time. We generalize this α-ary Gray code to a "multary" Gray code, and generalize our constant amortized time code generation algorithm for this multary code.

In Appendix B, we describe some of the algorithms used throughout this thesis to test our various conjectures. This includes descriptions of GIGA, a crossover-only GA; NQ-GIGA, a modification of GIGA; GAC [47], a traditional genetic algorithm; and a mutation hill climber.

Chapter 2

Landscapes Induced by Mutation and Crossover

2.1 Search Space Structures and Some Notation

In this chapter we describe the landscapes induced by a single genetic operator on minimal populations. A population is considered minimal if, when an operator is applied, all the strings in the population must be used (i.e., there is no selection of strings from within the population). For example, a minimal population for one-point mutation consists of a single string. For one-point crossover, a minimal population consists of two strings. We discuss the landscapes of α-ary mutation and several variants of α-ary crossover. We call these landscape graphs on minimal populations *search space structures (SSSs)*, as done by Culberson [8].

There are several reasons for focusing on minimal population sizes in this initial analysis:

1. The landscapes generated by operators on minimal populations are much simpler than those with larger populations. Thus, any analysis of "simplified" landscapes can be seen as a first step towards an analysis of more complex landscapes.

2. These landscapes, in certain cases, may relate to more general landscapes.

3. There may be graph-theoretic properties of these landscapes that are worthy of individual study.

4. The landscapes examined in this section correspond to the landscapes of some simple hill climbers, and so our analysis will at least apply to hill climbers, even if it does not always extend to GAs.

5. Comparing test results from minimal populations and non-minimal populations can offer insight into how population and crossover interact in GAs.

These search space structures require some additional notation. Throughout this thesis, most of the mathematical derivations and equations are carried out under a

finite abelian[1] group with α elements and identity 0. If x is an element in the group, then x^{-1} is the inverse of x. The group operator is $*$.

At certain points in this thesis we will restrict our discussion to the group "addition mod α," or $\mathbf{Z}_\alpha$. Rather than writing $(x+y)$ mod α or $(x-y)$ mod α, we will write $x \oplus y$ and $x \ominus y$, respectively. The value of α will be implicit.

We will also sometimes write $x * y$ as $< x >_y$ or $< y >_x$. For example, $x * y * z^{-1}$ could be written as $< x >_{y*z^{-1}}$. We will also apply this notation to strings, when convenient. Consider

$$< x_i x_{i+1} x_{i+2} \cdots x_{j-1} x_j >_n \text{ , where } i \leq j$$

as a substitute for

$$< x_i >_n < x_{i+1} >_n < x_{i+2} >_n \cdots < x_{j-1} >_n < x_j >_n$$

or

$$(x_i * n)(x_{i+1} * n)(x_{i+2} * n) \cdots (x_{j-1} * n)(x_j * n)$$

This shorthand notation is used later to define several types of α-ary crossover.

We first discuss mutation-based search space structures, followed by crossover-based search space structures. In the last section we discuss α-ary hypercube graphs and some of their properties, and relate hypercubes to both mutation and crossover search space structures.

2.2 Landscape Graphs for α-ary Mutation

One-point binary mutation can be specified by denoting which bit gets mutated, i.e., the position k of that bit. On a string $x = x_1 x_2 \cdots x_\ell$, a one-point binary mutation at k is equivalent to complementing x_k, and leaving all other bits in x unchanged. More rigorously, one-point binary mutation at k on x creates a new string x' such that $x'_k \neq x_k$, and $\forall i$, such that $i \neq k$, $x'_i = x_i$. As an example of one-point binary mutation, the string 000010 can be mutated at $k = 3$ to get the string 001010.

One-point binary mutation can be generalized to α-ary strings by specifying both the position of mutation and the new value at that position. When $\alpha = 2$, one-point mutation reduces to one-point binary mutation. Rather than speaking of what the mutated character changes to, we introduce the symbol Δ, which refers to what the mutated character changes by (e.g., $\Delta = x'_k * x_k^{-1}$). For example, if $\alpha = 5$ and $x = 00300$ and we mutate x to x' at $k = 3$ for $\Delta = 4$, then $x' = 00(3 \oplus 4)00 = 00200$.

We focus on one-point mutation in this thesis, even though there are an infinite number of different mutation types. Two common mutation types are *k-point mutation* where up to k points are mutated and *uniform mutation* where each character is mutated with probability $p_m, 0 < p_m < 1$. We choose one-point mutation because it is simple and can be used to implement most common mutation types. Its landscapes

[1] The group operator is commutative.

can also be similar to those of other mutation types: for example, uniform mutation with $p_m = 1/\ell$ will mutate one character on average. Thus there may be some similarity between searches that use uniform mutation with low probabilities and those that use one-point mutation.

The *one-point mutation SSS* is defined on a population of size one. Two strings are adjacent in the search space structure iff they differ in exactly one character. In this thesis we make the assumption that the character to be mutated is chosen uniformly; this means we do not need probabilities on edges. This is the usual implementation of one-point mutation. We will refer to the SSS for mutation on α-ary, ℓ-character strings by the notation $\mathcal{H}_M(\ell, \alpha, *)$.

2.3 Landscape Graphs for α-ary Crossover

In this section we will define two types of crossover:

1. one-point crossover (normal crossover between two strings), and
2. one-point (α,$*$) crossover (crossover between α strings, driven by $*$),

and discuss their search spaces structures.

One-point Crossover SSS

One-point crossover on two strings $x = x_1 x_2 \cdots x_\ell$ and $y = y_1 y_2 \cdots y_\ell$ at crossover point $k, 1 \leq k < \ell$, produces the strings $x' = x_1 \cdots x_k y_{k+1} \cdots y_\ell$ and $y' = y_1 \cdots y_k x_{k+1} \cdots x_\ell$. It is possible to define other crossover operators in terms of one-point crossover; for example, two-point crossover at k_1, k_2 on x and y is equivalent to doing a one-point crossover on x and y at k_1, followed by a crossover at k_2.

For an example, consider crossing the pair of strings:

01101
10001

at crossover point $k = 1$, which yields the strings:

00001
11101

Not all crossovers in this SSS produce a change. For instance, crossing the strings above at positions $k = 3$ or $k = 4$ produces no change.

The *one-point crossover SSS* is defined by two strings x and y. The vertices consist of all the pairs (x', y') such that $x'_i = x_i$ and $y'_i = y_i$, or $x'_i = y_i$ and $y'_i = x_i, \forall i, 1 \leq i \leq \ell$. In this SSS, any point (x, y) in the search space, is connected by an edge to another point (x', y') *iff* (x, y) can be crossed (once) to get (x', y'). This search space

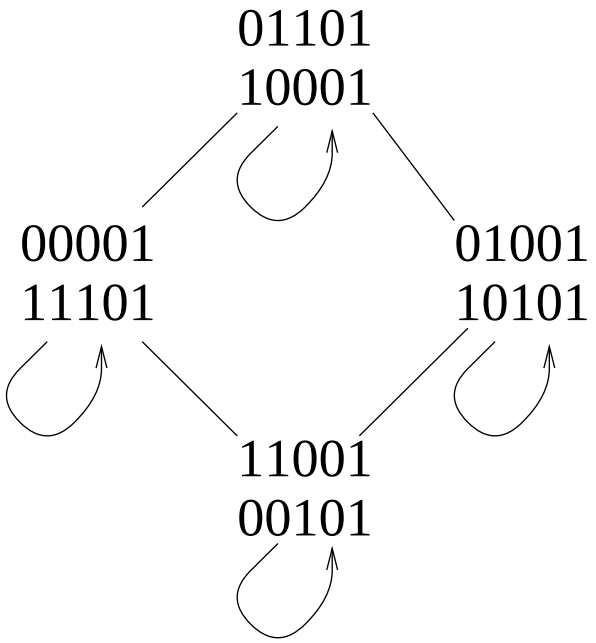


Figure 2.1: Example one-point crossover SSS. The loops have probability 1/2, all other edges have probability 1/4 (assuming crossover point is chosen uniformly).

is connected, and is equivalent to the search space resulting from taking an initial pair of strings (x, y) and repeatedly applying one-point crossover to them.

Jones [33] gives an alternative simplified crossover landscape: the vertices consist of all possible pairs of strings, with edges defined by one-point crossover. This landscape is not connected but consists of many connected components, where each component is a one-point crossover SSS. The number of one-point SSSs in Jones's landscape is given by $\sum_{h=2}^{\ell} \binom{\ell}{h} \binom{\alpha}{2}^h$ which is approximately $(1+\binom{\alpha}{2})^\ell$ since $\sum_{h=0}^{\ell} \binom{\ell}{h} \binom{\alpha}{2}^h = (1+\binom{\alpha}{2})^\ell$. For the binary case, there are roughly 2^ℓ one-point crossover SSSs in Jones's landscape.

See Figure 2.1 for an example one-point crossover SSS. The probabilities on the edges of a one-point crossover SSS are not equal unless the strings are identical or have equally spaced differing characters separated by equal-sized blocks of identical characters. A one-point crossover SSS will have self-loops only if the first or last character in each string are the same.

One-point (α,$*$) Crossover SSS

In this section we use the shorthand notation described previously to define one-point (α,$*$) crossover. These crossover types are used in the next two chapters to show that one-point crossover can simulate discrete mutation. That is, we will show that one-point (α,$*$) crossover can simulate α-ary mutation and that one-point crossover can simulate (α,$*$) crossover, implying (by transitivity) that one-point crossover simulates α-ary mutation.

Recall that $*$ is a finite abelian group operator. Then one-point (α,$*$) crossover acts on a set of α strings:

$$\begin{array}{c} < x_1 x_2 \cdots x_k x_{k+1} \cdots x_\ell >_0 \\ < x_1 x_2 \cdots x_k x_{k+1} \cdots x_\ell >_1 \\ \vdots \\ < x_1 x_2 \cdots x_k x_{k+1} \cdots x_\ell >_{\alpha-1} \end{array}$$

Given this set of strings, one-point (α,$*$) crossover with Δ and crossover point k gives the set of strings:

$$\begin{array}{c} < x_1 x_2 \cdots x_k >_0 < x_{k+1} \cdots x_\ell >_{\Delta} \\ < x_1 x_2 \cdots x_k >_1 < x_{k+1} \cdots x_\ell >_{\Delta * 1} \\ \vdots \\ < x_1 x_2 \cdots x_k >_{\alpha-1} < x_{k+1} \cdots x_\ell >_{\Delta * (\alpha-1)} \end{array}$$

In other words, one-point (α,$*$) crossover picks a crossover point k, and permutes the tail ends of the strings by applying Δ to each character in the tail.

One-point (α,$\oplus$) crossover is one-point rotational crossover (so named because the tail ends of the strings rotate upwards by Δ rows). For the binary case, rotational crossover is just crossover between complementary strings. In the following examples, the vertical bars mark the crossover point. The complementary binary pair

$$\begin{array}{c} 001101|1001 \\ 110010|0110 \end{array}$$

becomes the following pair after one-point rotational crossover at $k = 6$:

$$\begin{array}{c} 001101|0110 \\ 110010|1001 \end{array}$$

Consider the following set of strings, for $\alpha = 4$:

$$\begin{array}{c} 0132102|1223 \\ 1203213|2330 \\ 2310320|3001 \\ 3021031|0112 \end{array}$$

Rotational crossover at $k = 7$ with $\Delta = 1$ gives the following strings:

$$\begin{array}{c} 0132102|2330 \\ 1203213|3001 \\ 2310320|0112 \\ 3021031|1223 \end{array}$$

$*$	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

Table 2.1: Product table for the Klein 4-group.

Using a group other than $\mathbf{Z}_\alpha$ will generate other types of multi-string crossovers. For example, consider the Klein 4-group (its group product is given in Table 2.1). Then, letting $*$ be the Klein 4-group operator, one-point (4,$*$) crossover on the strings

$$\begin{array}{c} 01302|130 \\ 10213|021 \\ 23120|312 \\ 32031|203 \end{array}$$

at $k = 5$ with $\Delta = 2$ produces the strings

$$\begin{array}{c} 01302|312 \\ 10213|203 \\ 23120|130 \\ 32031|021 \end{array}$$

The *one-point (α,$*$) crossover SSS* is defined on populations of α strings, where each string has length $\ell+1$. The vertices consist of the sets $\{x, < x >_0, \cdots, < x >_{\alpha-1}\}$ for all strings x such that $x = 0x_2x_3\cdots x_{\ell+1}$. Two sets of α strings are adjacent in the search space structure iff one can be derived from the other by a single one-point (α,$*$) crossover operation. We use the notation $\mathcal{H}_X(\ell, \alpha, *)$ to refer to the graph of the one-point crossover SSS for $(\ell + 1)$-character strings.

2.4 Hypercubes and Some Properties of Hypercubes

The *Hamming distance* between two α-ary strings $x = x_1x_2\cdots x_\ell$ and $y = y_1y_2\cdots y_\ell$ is

$$h\_d(x, y) = \sum_{i=1}^{\ell} \begin{cases} 1, \text{ if } x_i \neq y_i \\ 0, \text{ otherwise} \end{cases}$$

For example, the strings 01234 and 11234 have a Hamming distance of one, while the strings 1001 and 0010 have a Hamming distance of three.

The Hamming (binary) hypercube of dimension ℓ is a graph that consists of all the binary numbers of length ℓ as vertices. Given any two vertices x and y, there is

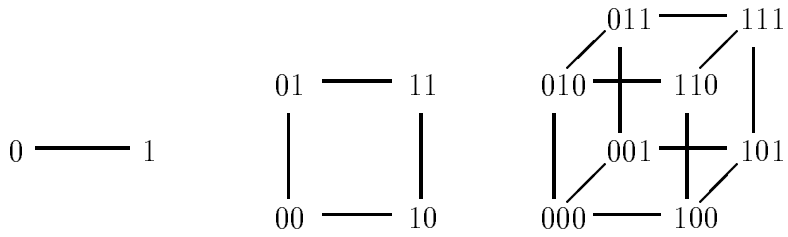


Figure 2.2: The one (l), two (c), and three (r) dimensional Hamming hypercubes.

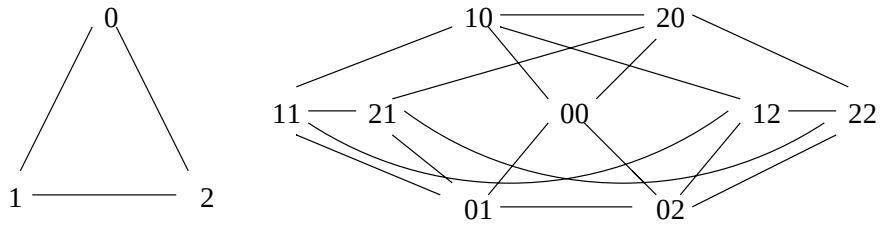


Figure 2.3: Some α-ary hypercubes: the (3,1)-cube (l), and the (3,2)-cube (r).

an edge (x,y) iff $h\_d(x,y) = 1$. For example, the string 000 is connected to the strings 001, 010, and 100 in the three-dimensional Hamming hypercube. The hypercube is an undirected graph. See Figure 2.2 for some example Hamming hypercubes. See Harary, *et al.* [26] for an alternative set definition of the binary hypercube and a survey on hypercube theory.

The hypercube can be generalized for strings that have α-ary characters. The hypercube of dimension ℓ, base α is a graph that has α^ℓ vertices (all the α-ary strings of length ℓ). Two vertices x and y are connected by an edge (x,y) iff $h\_d(x,y) = 1$. For example, the vertices 3456 and 3156 would be connected in a hypercube of length 4. As with the Hamming hypercube, base α hypercubes are undirected. When referring to hypercubes, we will sometimes use the notation *(α,ℓ)-cube* for the hypercube of dimension ℓ, base α as a shorthand. See Figure 2.3 for two example α-ary hypercubes. For more on (α,ℓ)-hypercubes, the reader may wish to read the paper by Barasch, *et al.* [1].

The search space structures discussed in the previous two sections are isomorphic to hypercubes or closely related to them. One-point mutation can be defined in terms of the Hamming distance between a string x and the mutated string x', since one-point mutation on x to get x' always gives $h\_d(x,x') = 1$. Therefore, the one-point mutation SSS, $\mathcal{H}_M(\ell,\alpha,*)$, is isomorphic to the (α,ℓ)-cube.

The crossover search space structures are also related to hypercubes. For the one-point crossover SSS, when two length ℓ strings have a Hamming distance of h, their SSS is equivalent to searching on the $\ell - h - 1$ Hamming hypercube [8] if the first and

last character in each string are different, and the $\ell - h - 1$ Hamming hypercube with self-loops, otherwise. The one-point $(\alpha,*)$ crossover SSS for strings of length $\ell + 1$ is isomorphic to the (α,ℓ)-cube. This isomorphism is proven in the next chapter.

Given that (α,ℓ)-cubes occur often in GA and hill climber landscape analysis, it should prove beneficial to note some measures of these graphs that may affect the performance (or limit the performance) of our various search algorithms. Some example measures include:

- maximum number of false optima
- longest distance-preserving paths
- degree vs. diameter trade-off in landscape graphs

Several of these measures have been discussed in the literature. For example, the maximum number of false optima on a hypercube landscape is $\alpha^{\ell-1}$ [10]. The longest possible distance-preserving path is exponential (in ℓ) for the Hamming hypercube [45, 26], and exponentially long paths for one-point mutation have been implemented [29, 30, 31, 10]. Exponentially long paths can still occur in hypercubes, for $\alpha > 2$, but the maximum length of these paths is reduced [10]. We discuss this "longest distance-preserving path" problem in Chapter 6.

The (α,ℓ)-cube's *diameter*, the maximum distance between any two vertices, is ℓ. Its *degree*, the maximum number of neighbours of any vertex, is $\ell(\alpha - 1)$. Notice that $n = \alpha^\ell$ and so $\ell = \log_\alpha n$. This can also be written as $\ell = \frac{\ln n}{\ln \alpha}$. In general, we want a diameter and degree that are not too small and not too large [8, 10], and in certain situations it may be useful to minimize *degree*diameter* [10] for a given number of vertices n. For example, the degree*diameter of a complete graph is $n - 1$ and that of a cycle graph is n, much worse than the degree*diameter product for the hypercube: $(\alpha - 1)\log_\alpha^2 n$. A hypercube of $\ell = 1$ is a complete graph and so not all hypercubes have desirable degree-diameter trade-offs. The degree*diameter product is minimized for $\alpha = 5$ [10].

Chapter 3

Generalized Crossover-Mutation Isomorphism

In this chapter we prove that the one-point $(\alpha, *)$ crossover SSS, $\mathcal{H}_X(\ell, \alpha, *)$, is isomorphic to the one-point α-ary mutation SSS, $\mathcal{H}_M(\ell, \alpha, *)$. That is, the search space structures of both operators are (α,ℓ)-cubes.

This isomorphism has several uses:

- It can illustrate key points about crossover's landscapes.
- It can be used to show that crossover can simulate mutation (next chapter).
- It can be used to create long paths for crossover (Chapter 6), and in the construction of discriminating functions (Chapter 7).

3.1 The Binary Crossover-Mutation Isomorphism

Define the mapping $\mathcal{I}$ to be:

$$\begin{aligned} \mathcal{I}(x) &= (a, \bar{a}) \\ a_i &= \begin{cases} 0 & \text{if } i = 1 \\ x_{i-1} \oplus a_{i-1} & 1 < i \leq \ell + 1 \end{cases} \end{aligned}$$

It can be shown [8] that $\mathcal{I}$ is an isomorphism between the search spaces induced by *one-point binary mutation* on a string of length ℓ and *one-point binary crossover* on two complementary strings of length $\ell + 1$.

Culberson [8] uses this isomorphism to transform functions that are hard (or easy) for one-point binary mutation to functions that are hard (or easy) for one-point crossover on a complementary pair of strings. He also uses it to illustrate key points about the one-point binary crossover SSS by allowing a complex operator, crossover, to be expressed in terms of a simpler operator, mutation.

3.2 An α-ary Crossover-Mutation Isomorphism

We can generalize the binary crossover-mutation isomorphism to an α-ary crossover-mutation isomorphism; that is, we show that one-point (α,$*$) crossover is isomorphic to one-point α-ary mutation.

The changes in $\mathcal{I}$ are that the $\oplus$ symbol becomes the $*$ symbol, and that the isomorphism maps a string x of length ℓ to α strings of length $\ell + 1$. The new generalized $\mathcal{I}$ and $\mathcal{I}^{-1}$ mappings are

$$\begin{aligned} \mathcal{J}(x) &= (a, < a >_1, \ldots, < a >_{\alpha-1}) \\ a_i &= \begin{cases} 0 & \text{if } i = 1 \\ x_{i-1} * a_{i-1} & 1 < i \leq \ell + 1 \end{cases} \end{aligned}$$

and

$$\begin{aligned} \mathcal{J}^{-1}((a, < a >_1, \ldots, < a >_{\alpha-1})) &= x \\ x_i &= (a_{i+1} * a_i^{-1}),\ 1 \leq i \leq \ell \end{aligned}$$

When $\alpha = 2$, these mappings reduce to $\mathcal{I}^{-1}$ and $\mathcal{I}$. Note that x_i also equals $< a_{i+1} >_m * (< a_i >_m)^{-1}$, $1 \leq i \leq \ell, 0 \leq m \leq \alpha - 1$, since the "$m$"s cancel. This mapping is also one-one and onto.

As an example of $\mathcal{J}$ for higher alphabets, let $x = 24103$ for α= 5, and let $* = \oplus$. Then $\mathcal{J}(x)$ is equal to

0212|20
1323|31
2434|42
3040|03
4101|14

If x is mutated to x', where $x' = 24113$ ($k = 4, \Delta = 1$), then $\mathcal{J}(x')$ is

0212|31
1323|42
2434|03
3040|14
4101|20

This mutation is equivalent to a rotational crossover, and indeed, α-ary mutation is isomorphic to an (α,$*$)-crossover. The following theorem and proof were presented in the paper, "On Searching α-ary Hypercubes and Related Graphs" [10].

THEOREM **3.2.1** *The mapping $\mathcal{J}$ is an isomorphism from $\mathcal{H}_M(\ell, \alpha, *)$ to $\mathcal{H}_X(\ell, \alpha, *)$. That is, a mutation of Δ at k on an α-ary string is equivalent to one-point (α,$*$) crossover at k with tail permutation Δ on the α strings generated by $\mathcal{J}$.*

Proof:
(Theorem 3.2.1):

Assume (x, y) is an edge in $\mathcal{H}_M(\ell, \alpha, *)$. Then $x_k \neq y_k$ for some $k, 1 \leq k \leq \ell$ and $x_i = y_i$ for all $i \neq k$. Let

$$\begin{aligned} \mathcal{J}(x) &= (a, < a >_1, \ldots, < a >_{\alpha-1}) \\ \mathcal{J}(y) &= (b, < b >_1, \ldots, < b >_{\alpha-1}) \end{aligned}$$

Simple induction shows that

$$b_i = a_i, \text{ for } 1 \leq i \leq k$$

We will now show through induction that

$$b_{k+i} =< a_{k+i} >_{y_k * x_k^{-1}}, \text{ for } k+1 \leq k+i \leq \ell+1$$

Basis:

$$\begin{aligned} b_{k+1} &= a_k * y_k \\ &= y_k * x_k^{-1} * (a_k * x_k) \\ &= y_k * x_k^{-1} * a_{k+1} \\ &= < a_{k+1} >_{y_k * x_k^{-1}} \end{aligned}$$

Induction Step (IS):
Induction Hypothesis (IH): assume $b_{k+i} =< a_{k+i} >_{y_k * x_k^{-1}}$ for $k+1 \leq k+i < \ell+1$. Then

$$\begin{aligned} b_{k+i+1} &= b_{k+i} * x_{k+i} \\ &= y_k * x_k^{-1} * a_{k+i} * x_{k+i} \text{ by IH} \\ &= y_k * x_k^{-1} * a_{k+i} * a_{k+i}^{-1} * a_{k+i+1} \\ &= y_k * x_k^{-1} * a_{k+i+1} \\ &= < a_{k+i+1} >_{y_k * x_k^{-1}} \end{aligned}$$

Therefore (by Basis and IS)

$$\begin{aligned} b &= a_1 \cdots a_k < a_{k+1} \cdots a_{l+1} >_{y_k * x_k^{-1}} \\ < b >_1 &= < a_1 \cdots a_k >_1 < a_{k+1} \cdots a_{\ell+1} >_{y_k * x_k^{-1} * 1} \\ \vdots &\quad \vdots \\ < b >_{\alpha-1} &= < a_1 \cdots a_k >_{\alpha-1} < a_{k+1} \cdots a_{l+1} >_{y_k * x_k^{-1} * (\alpha-1)} \end{aligned}$$

Since $\Delta = y_k * x_k^{-1}$, one-point mutation of Δ at position k is isomorphic to one-point (α,$*$) crossover at k on strings of length $\ell + 1$ with permutation of Δ. (Recall that $\mathcal{J}$ is one-one and onto.)

■

This isomorphism generalizes for k-point mutation (up to k-mutations), which is isomorphic to k-point $(\alpha, *)$ crossover. As well, α-ary uniform mutation is isomorphic to uniform $(\alpha,*)$ crossover. It is noted [8] that two adjacent one-point binary crossover operations on complementary strings mimics mutation in that both strings seem to undergo a mutation between the adjacent crossover points; this also applies to $\mathcal{J}$ except that the adjacent crossovers must have a Δ permutation followed by a Δ^{-1} permutation.

For another example of one-point $(\alpha,*)$ crossover, let $*$ be the Klein-4 group operator (see Table 2.1). In this case, the isomorphism maps a single string to four pseudo-complementary strings. For example:

$$2013 \rightarrow \begin{array}{l} 02|230 \\ 13|321 \\ \\ 20|012 \\ 31|103 \end{array}$$

As with the previous isomorphism, a mutation on a string x appears to be a crossover under $\mathcal{J}(x)$. If we mutate the 0 to a 3 in the above example we get:

$$2313 \rightarrow \begin{array}{l} 02|103 \\ 13|012 \\ \\ 20|321 \\ 31|230 \end{array}$$

If we think of 0 and 1 as being complementary and 2 and 3 as being complementary, then we can think of $\mathcal{J}(x)$ as a mapping from a single string x to two pairs of strings, the pair that starts with 0 and 1 and the pair that starts with 2 and 3. A pair is always complementary. It is noted [10] that this crossover type is similar to the DNA code (in a very limited way). It may be possible that this may be the first step towards an algebra for DNA[1].

[1] We warn the reader that this last paragraph is speculation.

Chapter 4

On the Power of Crossover and Mutation

In this chapter we discuss the question of which is more powerful, discrete crossover or discrete mutation. To help answer this question we use the generalized crossover-mutation isomorphism of the previous chapter to show that crossover can simulate discrete mutation, and further show that crossover can simulate crossover and mutation in combination. This means that crossover is at least as powerful as mutation, in the sense of computability. This simulation can be achieved in linear space and time.

We also discuss generalizations of this simulation, and this leads to the notion of hyper-order crossovers and hyper-order mutations. These hyper-order operators can help demonstrate the complexity of regular crossover.

Finally, our work suggests that the question of which operator is more powerful should be rephrased into at least two questions: "Are the various operators being used to their potential?" and, if so, "Which operator, crossover or mutation, is better suited to a particular problem?"

4.1 Introduction

There has been much debate in the GA community on whether the crossover operator is inherently more powerful than the mutation operator or vice versa [46]. In this chapter, we consider the relative "power" of one-point crossover and one-point mutation. We attack this problem by showing that one-point crossover can simulate one-point mutation, and thus is at least as computationally powerful as one-point mutation. By saying crossover can simulate mutation, we mean that a GA that uses mutation and crossover can be replaced with a computationally equivalent one that uses only crossover and no mutation; that is, crossover can be used to implement an operator isomorphic to mutation.

While our simulation will focus on one-point crossover and one-point mutation, it will be much more general in that we will be able to show that discrete mutation can be simulated by one-point crossover. By discrete mutation, we mean a mutation operator that can be expressed as some number of one-point mutations on strings

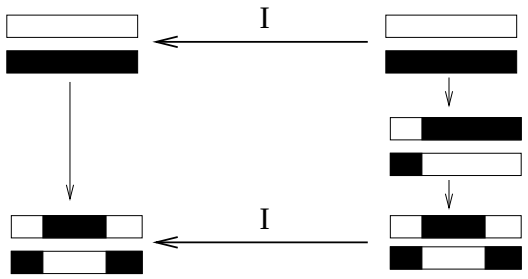


Figure 4.1: One-point crossover can simulate two-point crossover. I is the identity mapping.

made up of discrete (α-ary) characters. Discrete mutation includes k-point mutation and uniform mutation. Our results, however, will not apply to all mutation types, e.g., mutation on real numbers.

We can demonstrate what we mean by one operator simulating another by showing schematically how one-point crossover can simulate two-point crossover in Figure 4.1. There are two ways one-point crossover can simulate two-point crossover:

1. one-point crossover can be used in the implementation of a two-point crossover operator (simulation), and

2. two strings may be crossed, evaluated, then crossed again (mimicry).

For this chapter our focus will be on simulation, but we will also discuss mimicry, which can occur in GAs and hill climbers.

To show one operator simulates another, we must show that the states of the objects being acted on (in our case, strings) remain in correspondence. That is, there must be a 1-1 and onto mapping between the objects' initial and end states; in our example we used the identity mapping I, but in a more complex simulation, the objects can be different or even of different types, as long as there is a mapping that makes the initial and end states correspond. This will be the case in our simulation: strings will correspond to sets of strings.

We can also convert any algorithm that uses both mutation and crossover to an equivalent one, a hyper GA (HGA), that uses only crossover. If we use more general crossover types on the set representations, where the general crossovers include the highly restricted crossovers used in our simulation, then HGAs will still include GAs but the converse will not be true: HGAs will have actions that cannot be simulated by a GA. That is, crossover can do whatever mutation can do and more, even when both mutation and crossover are used. This will help demonstrate the complexity (and power) of crossover with respect to mutation. The cost to using these "relaxed" HGAs is an increase in the size of the neighbourhood in the landscape.

To show that one-point crossover can simulate discrete mutation, we will show that (α,$*$) crossover can simulate one-point mutation. Since one-point crossover can simulate one-point (α,$*$) crossover and one-point mutation can simulate discrete mu-

tation, this will imply (by transitivity) that one-point crossover simulates discrete mutation.

4.2 Simulating Mutation Using Crossover

To show that one-point crossover can simulate discrete mutation, we first show that one-point (α,$*$) crossover can simulate one-point mutation. In this section we only consider search algorithms that use mutation on α-ary strings (e.g., mutation hill climbers); in the next section, we show how algorithms that use mutation and other operators (e.g., GAs) can be simulated.

The first step in creating a modified algorithm is to change the basic search objects. We convert each individual string x into the set of strings given by the generalized crossover-mutation isomorphism of Chapter 3:

$$\begin{aligned} \mathcal{J}(x) &= (a, < a >_1, \ldots, < a >_{\alpha-1}) \\ a_i &= \begin{cases} 0 & \text{if } i = 1 \\ x_{i-1} * a_{i-1} & 1 < i \leq \ell + 1 \end{cases} \end{aligned}$$

Each set of strings directly corresponds to the string that generated it; this means each set of strings, for simulation purposes, is atomic. Thus our population of strings has been replaced with a population of sets of strings.

The second step in the simulation is then to replace all calls to the mutation operator with calls to the (α,$*$) crossover operator. This must be the same group operator as the one used to generate the sets of strings. For example, if $\alpha = 4$, it would be incorrect to generate the sets of strings using the Klein 4-group operator, and then use rotational crossover ($*$=$\oplus$).

The third and final step is to alter the fitness function f to a new fitness function f', where $f'() = f(\mathcal{J}^{-1}())$. If S is the set of strings representing the string x, or $S = \mathcal{J}(x)$, then $f'(S) = f(x)$.

Doing these three steps will convert a search algorithm that uses mutation on strings to a computationally equivalent one that uses no mutation whatsoever. One-point crossover can simulate one-point (α,$*$) crossover in at most $\alpha - 1$ crossover operations, which implies crossover can simulate one-point (α,$*$) crossover. (The sequence of one-point crossover operations that simulates one-point (α,$*$) crossover does not depend on the particular strings in the population.) By transitivity, crossover can simulate mutation. This simulation is computationally efficient since the required (extra) space is $\Theta(\alpha n \ell)$, where n is the population size.

It is also easy to see that discrete mutation can be simulated by one-point crossover, since one-point mutation can simulate discrete mutation. Some common examples of discrete mutation are uniform mutation and k-point mutation.

Instead of using a strict sequence of one-point crossover operations to simulate a one-point (α,$*$) crossover, we can use a more general crossover type that includes (i.e., can mimic) rotational crossover. For example, using one-point crossover, for $\alpha > 2$, on a rotational crossover population can mimic one-point (α,$*$) crossover and thus can mimic mutation, but also includes operations that cannot be directly mimicked

by mutation. The structure of this "relaxed" crossover SSS is related to the mutation SSS. For instance, if there is a monotonic path in the mutation SSS, then there is also a monotonic path in the "relaxed" crossover SSS.

Although it is trivially true that mutation can simulate crossover (if crossover on x and y gives x' and y', just mutate x to x' and y to y'), this simulation is weaker in that the character values of x and y must be used in the simulation and that it is a non-obvious form of mutation. Crossover appears to be more powerful than mutation, in some sense, as there appears to be no "natural" method for using mutation to simulate one-point crossover.

4.3 Including Other Operators in Our Simulation

In the previous section, we showed how to construct a crossover-only algorithm that is computationally equivalent to an algorithm that uses mutation. If the original algorithm uses other operators, it is easy enough to simulate these operators with a few minor alterations. For example, imagine the original algorithm is a GA that uses both mutation and crossover. One-point crossover is easily simulated. To do this, convert the sets of strings to their single-string representations, do the crossover, and then convert them back into their corresponding sets of strings. This can be done for any operator, and can be seen as an *indirect* approach.

Another, more direct, approach would be to find a new operator that acted on the set representation that was isomorphic to the old operator on the single string representation. We will do this for one-point crossover; in fact, it will be shown that one-point crossover on two sets of strings can simulate crossover between single strings. To do this we define another crossover type: first-order crossover. *First-order crossover* is a crossover operation applied on the $(a, < a >_1, \ldots, < a >_{\alpha-1})$ and $(b, < b >_1, \ldots, < b >_{\alpha-1})$ strings at k, $2 \leq k \leq l$, to give two new ordered sets of strings $(a', < a' >_1, \ldots, < a' >_{\alpha-1})$ and $(b', < b' >_1, \ldots, < b' >_{\alpha-1})$, where

$$a' = a_1 \cdots a_{k+1} < b_{k+2} \cdots b_{\ell+1} >_{a_{k+1} * b_{k+1}^{-1}}$$

and

$$b' = b_1 \cdots b_{k+1} < a_{k+2} \cdots a_{\ell+1} >_{b_{k+1} * a_{k+1}^{-1}}$$

and $< a' >_i$ and $< b' >_i$, $0 \leq i \leq \alpha - 1$ can be determined from a' and b', respectively.

Lemma 4.3.1 *One-point crossover at k between x and y is isomorphic to first-order crossover at $k+1$ between $\mathcal{J}(x)$ and $\mathcal{J}(y)$.*

The following proof takes two arbritrary α-ary strings of length ℓ, crosses them at some k to get two new strings x' and y', and then notes how $\mathcal{J}(x)$ is different from $\mathcal{J}(x')$. Any such observations will apply to y and y' under $\mathcal{J}$ by symmetry.

Proof:
(Lemma 4.3.1)

Let $\mathcal{J}(x) = (a, < a >_1, \ldots, < a >_{\alpha-1})$ and $\mathcal{J}(y) = (b, < b >_1, \ldots, < b >_{\alpha-1})$. One-point crossover between x and y at k where $1 \leq k < \ell$ yields

$$\begin{aligned} x' &= x_1 \cdots x_k y_{k+1} \cdots y_\ell \\ y' &= y_1 \cdots y_k x_{k+1} \cdots x_\ell \end{aligned}$$

Then let $a' = \mathcal{J}(x')$ and $b' = \mathcal{J}(y')$.

Inductive use (omitted) of $\mathcal{J}$ gives $b'_1 \cdots b'_{k+1} = b_1 \cdots b_{k+1}$.

We now show that $b'_{k+i} =< a_{k+i} >_{b_{k+1} * a_{k+1}^{-1}}$ for $k+2 \leq k+i \leq \ell+1$.

Basis:

$$\begin{aligned} b'_{k+2} &= b_{k+1} * x_{k+1} \\ &= b_{k+1} * a_{k+1}^{-1} * a_{k+2} \\ &= < a_{k+2} >_{b_{k+1} * a_{k+1}^{-1}} \end{aligned}$$

Induction Step (IS):
Induction Hypothesis (IH): assume $b'_{k+i} =< a_{k+i} >_{b_{k+1} * a_{k+1}^{-1}}$, for $k+2 \leq k+i < \ell+1$.
Then

$$\begin{aligned} b'_{k+i+1} &= b'_{k+i} * x_{k+i} \\ &= b_{k+1} * a_{k+1}^{-1} * a_{k+i} * x_{k+i} \text{ by IH} \\ &= b_{k+1} * a_{k+1}^{-1} * a_{k+i} * a_{k+i}^{-1} * a_{k+i+1} \\ &= b_{k+1} * a_{k+1}^{-1} * a_{k+i+1} \\ &= < a_{k+i+1} >_{b_{k+1} * a_{k+1}^{-1}} \end{aligned}$$

And therefore $b'_{k+2} \cdots b'_{l+1} =< a_{k+2} \cdots a_{\ell+1} >_{b_{k+1} * a_{k+1}^{-1}}$ by Basis and IS. Thus,

$$b' = b_1 \cdots b_{k+1} < a_{k+2} \cdots a_{\ell+1} >_{b_{k+1} * a_{k+1}^{-1}}$$

which is first-order one-point crossover between $(a, < a >_1, \ldots, < a >_{\alpha-1})$ and $(b, < b >_1, \ldots, < b >_{\alpha-1})$.

■

This means first-order crossover between $\mathcal{J}(x)$ and $\mathcal{J}(y)$ is isomorphic to crossover between x and y, and that first-order crossover can simulate crossover. As in the previous section, we must use the same group operator, $*$, in each stage of the simulation. One-point crossover can in turn simulate first-order crossover in at most $2\alpha - 1$ crossover operations. Thus any GA that uses both mutation and crossover can be simulated by one that only uses crossover.

For an example of first-order crossover between complementary binary strings, consider crossing

$$\begin{aligned} &0110|010 \\ &1001|101 \end{aligned}$$

with

$$1001|110$$
$$0110|001$$

at $k = 4$. We can immediately determine that the two new sets are

$$0110|001$$
$$1001|110$$

and

$$1001|101$$
$$0110|010$$

4.4 Hyper-Order Simulation

One-point (α,$*$) crossover and first-order crossover are useful in showing that crossover can simulate discrete mutation, and are also useful in that they can help illustrate key ideas about crossover's landscapes. For example, the isomorphism between mutation and crossover on binary strings can be used [8] to show that crossover between complementary binary strings searches a hypercube [26] of dimension $\ell - 1$. If the two strings are not complementary, but have a Hamming distance of h, then crossover searches a hypercube (possibly with self-loops) of dimension $\ell - h - 1$.

New insights into crossover's landscapes can be gained by iterating $\mathcal{J}$. By iterating $\mathcal{J}$, we mean mapping a string x under $\mathcal{J}$ to get a set of α strings, and then mapping each of these α strings under $\mathcal{J}$ to get α sets of α strings and so on. We can express this by the following notation: $\mathcal{J}^0(x) = \{\ x\ \}$, and $\mathcal{J}^i(x) = \{\mathcal{J}(a)$ for each $a \in \mathcal{J}^{i-1}(x)$, for $i \geq 1\ \}$. When $* = \oplus$, iterating $\mathcal{J}$ turns out to be highly similar to iterating the α-ary Gray code of Sharma and Khanna [43], an unsurprising fact considering that a slight change in $\mathcal{J}^{-1}$ will generate this Gray code: if $\mathcal{J}(x) = (a, < a >_1, \ldots, < a >_{\alpha-1})$, where $x = x_1 x_2 \cdots x_\ell$, then $a_2 a_3 \cdots a_{\ell+1}$ (dropping a_1) is the inverse Gray code element for x.

We can now define i-order mutation and i-order crossover. i-order mutation is the change in $\mathcal{J}^i(x)$ when x is mutated, and i-order crossover is the change in $\mathcal{J}^i(x)$ and $\mathcal{J}^i(y)$ when x and y are crossed. Thus one-point (α,$*$) crossover is first-order mutation, and a one-point mutation is 0-order mutation. Zero-order crossover is crossover between two arbitrary strings x and y. For the binary case, 0-order crossover is a generalization of a 1-order mutation, since 1-order mutation is a crossover between complementary strings, and i-order crossover is a generalization of $(i + 1)$-order mutation.

For an example of i-order mutation, consider the string $x = 1001$ mapped twice

under $\mathcal{J}$

$$1001 \rightarrow \begin{array}{c} 011|10 \\ 100|01 \end{array} \begin{array}{cc} \nearrow & \begin{array}{c} 0010|11 \\ 1101|00 \end{array} \\ \searrow & \begin{array}{c} 1000|01 \\ 0111|10 \end{array} \end{array}$$

If a mutation is done on x, say at $k = 3$, then this is equivalent to crossing $(01110, 10001)$ at $k = 3$, but we know from the previous subsection that crossing two strings (0-order crossover) is equivalent to first-order crossover at $k+1$. With the mutation the levels become

$$10\bar{0}1 = 1011 \rightarrow \begin{array}{c} 011|01 \\ 100|10 \end{array} \begin{array}{cc} \nearrow & \begin{array}{c} 0010|01 \\ 1101|10 \end{array} \\ \searrow & \begin{array}{c} 1000|11 \\ 0111|00 \end{array} \end{array}$$

It turns out that i-order mutation, $i \geq 1$, can be simulated by crossover, and that i-order crossover, $i \geq 0$, can also be simulated by crossover.

Lemma 4.4.1 *i-order mutation for $i \geq 1$ can be simulated by crossover.*

The proof uses induction on i. The induction hypothesis says that for all strings $a \in \mathcal{J}^{i-1}(x)$ there is a string b also in $\mathcal{J}^{i-1}(x)$ such that mutating x at k to get x' will cause a to be crossed with b, generating the string $a' = a_1 a_2 \cdots a_{k+i-2} b_{k+i-1} \cdots b_{\ell+i-1}$ in $\mathcal{J}^{i-1}(x')$. From this we will show that $\mathcal{J}(a')$ can be simulated by one-point crossover between $\mathcal{J}(a)$ and $\mathcal{J}(b)$, which will prove the lemma. The case $i = 1$ is covered by the isomorphism $\mathcal{J}()$.

Proof:
(Lemma 4.4.1)
Basis ($i = 1$): by the isomorphism $\mathcal{J}()$. That is, a mutation on any string x at k of Δ to get x' is isomorphic to a one-point (α,$*$) crossover on $\mathcal{J}(x)$ at k, which in turn can be simulated by one-point crossover. Thus for any string $a \in \mathcal{J}(x)$ there is also a string $b \in \mathcal{J}(x)$ such that there is a string $a' = a_1 a_2 \cdots a_k b_{k+1} \cdots b_{\ell+1}$ in $\mathcal{J}(x')$.

Induction Step ($i > 1$):
Induction Hypothesis (IH): assume $(i-1)$-order mutation can be simulated by crossover.

Let $a = a_1 a_2 \cdots a_{\ell+i-1}$ be a string in $\mathcal{J}^{i-1}(x)$, and by the IH we know that a mutation on x at k of Δ causes a to be crossed with the tail-end of another string, say $b = b_1 b_2 \cdots b_{\ell+i-1}$ to generate a new "crossed" string $a' = a_1 \cdots a_{k+i-2} b_{k+i-1} \cdots b_{k+i-1}$.

We must now show that any string in $\mathcal{J}(a)$ can be crossed with some string from $\mathcal{J}(b)$ to get a string in $\mathcal{J}(a')$. Showing this will prove the lemma.

Let $\mathcal{J}(a)$ generate the following set of strings:

$$\begin{array}{c} < g_1 g_2 \cdots g_{\ell+i} >_0 \\ < g_1 g_2 \cdots g_{\ell+i} >_1 \\ \vdots \\ < g_1 g_2 \cdots g_{\ell+i} >_{\alpha-1} \end{array}$$

Simple induction shows that $g_j = a_1 * \cdots * a_j, j \geq 2$, and $g_1 = 0$. Induction also shows that $\mathcal{J}(a')$ generates the set of strings

$$\begin{array}{c} < g_1 g_2 \cdots g_{k+i-1} h_{k+i} \cdots h_{\ell+i} >_0 \\ < g_1 g_2 \cdots g_{k+i-1} h_{k+i} \cdots h_{\ell+i} >_1 \\ \vdots \\ < g_1 g_2 \cdots g_{k+i-1} h_{k+i} \cdots h_{\ell+i} >_{\alpha-1} \end{array}$$

where $h_j = g_{k+i-1} * b_{k+i} * \cdots * b_j, k+i \leq j \leq \ell+i$. Since g_{k+i-1} is a factor in each h_j, the tail ends of $\mathcal{J}(a')$ (the h_j's) will be a permutation of the tail ends of $\mathcal{J}(b)$. This can be seen by noting that if $\mathcal{J}(b)$ generates the strings

$$\begin{array}{c} < d_1 d_2 \cdots d_{k+i-1} d_{k+i} \cdots d_{\ell+i} >_0 \\ < d_1 d_2 \cdots d_{k+i-1} d_{k+i} \cdots d_{\ell+i} >_1 \\ \vdots \\ < d_1 d_2 \cdots d_{k+i-1} d_{k+i} \cdots d_{\ell+i} >_{\alpha-1} \end{array}$$

where $d_j = d_{k+i-1} * b_{k+i} * \cdots * b_j, k+i \leq j \leq \ell+i$. That is, $h_j = (d_{k+i-1}^{-1} * g_{k+i-1}) * d_j$.

Thus any string generated by $\mathcal{J}(a')$ can be seen as a crossover between a string in $\mathcal{J}(a)$ and $\mathcal{J}(b)$.

■

This proof shows that crossover can simulate i-order mutation. Moreover, since i-order mutation on strings of length $\ell + i$ is isomorphic to mutation on ℓ length strings, this shows that large populations of strings with crossover can interact to search a hypercube.

It can also be easily shown that crossover can simulate i-order crossover. The proof is virtually the same, except that zero-order crossover is the basis. As with i-order mutation, i-order crossover can be used to show that large populations under crossover can search a hypercube.

Our results only show that crossover, in large and small populations, can search a hypercube; we do not think it likely that these restricted quasi-crossovers occur often (if ever) in GAs. However, they show crossover is much harder to analyze than mutation, and also help demonstrate the complexity of regular crossover (i.e., to simplify crossover to the level of mutation, we need highly constrained crossovers). It is also possible that hyper-order mimicry can be used in the construction of long path landscapes for crossover and to create other interesting functions.

4.5 Conclusion

In this chapter we showed that one-point crossover is at least as computationally powerful as discrete mutation, since one-point crossover can simulate discrete mutation. One-point crossover can also simulate both mutation and crossover. Further, this simulation generalizes to hyper-order crossovers and hyper-order mutations. By showing that one-point crossover can simulate i-order crossover ($i \geq 0$) and i-order mutation ($i > 0$), we showed that crossover in large populations can potentially search a hypercube.

Since these crossover types are so restrictive, they also help demonstrate the complexity and power of crossover. That is, if we use a more general type of crossover (e.g., one-point crossover) that includes one-point (α,$*$) crossover, then this "relaxed" HGA will be able to mimic a GA but the GA will be unable to simulate the "relaxed" HGA. That is, crossover can do everything crossover and mutation can do and more. (This may seem paradoxical, but we are using larger populations in the HGA). We can iterate this process: i.e., make HGAs of HGAs and so on, which demonstrates the additional power and complexity of using larger populations. These relaxed HGAs are not without cost: they have larger neighbourhoods to search.

This chapter addressed the crossover-mutation debates by showing that crossover is at least as powerful as mutation because crossover can simulate mutation. This does not mean that all implementations of crossover will use the potential of crossover, and though the relaxed-crossover neighbourhoods are less restrictive and thus broader in scope, a more focussed search may be better. Although crossover and mutation both are powerful search operators, which operator is better may also depend on the specific problem. The old question, "Which is better, crossover or mutation?" should be replaced with two new questions: "Are the various operators being used to their potential?" and "Which operator is better suited to a particular problem?"

Chapter 5

Iterating an α-ary Gray Code

In this chapter we prove a theorem on the number of unique codes produced when the α-ary Gray code mapping of Sharma and Khanna [43] is iteratively applied to an α-ary, dimension ℓ code; that is, starting with an α-ary, dimension ℓ code, repeatedly apply the permutation given by Sharma and Khanna's mapping. From this theorem, it is easy to show there are $\Theta(\ell^q)$ unique codes generated from this mapping, where q is the number of unique primes in α. To prove this theorem we show that any base α, dimension ℓ code word will cycle in $O(\ell^q)$ iterations of this Gray code mapping, and that this upper bound is attained. This theorem is a generalization of a theorem proven by Culberson [8] for the binary case.

The work of this chapter is a special case of hyper-order mimicry dealt with in the previous chapter. It can also be recast into GA theory, if we think of iterating a search space using this Gray code (i.e., iterating the representation, but keeping the same genetic operators). Our work will also show that iterating a search space (and thus iterating a landscape) with the Gray code of Sharma and Khanna will generate $O(\ell^q)$ unique landscapes.

5.1 Gray Codes

An α-ary Gray [23] code of dimension ℓ is a sequence of α^ℓ unique α-ary, length ℓ strings such that any two adjacent code words have a Hamming distance of one. Gray codes can be used to reduce errors when an analog signal is converted to a digital signal, they can be used in distributed memory architectures that are based on the α-ary hypercubes [1], and can be used in various combinatorial applications [2]. Gray codes also prove the existence of Hamiltonian paths on the hypercube graphs, and demonstrate the existence of Hamiltonian circuits if the Gray code is cyclic [20]. (For any Gray code on an α-ary hypercube of length ℓ, there are $(\alpha!)^\ell \ell!$ equivalent Gray codes that can be obtained by permuting the characters $0, 1, \cdots, \alpha$ in each column, and permuting each column.)

The Gray code discussed in this section will be represented by $\mathcal{G}(\alpha, \ell)$. We will sometimes use subscripts to refer to specific words in $\mathcal{G}(\alpha, \ell)$. For example, $\mathcal{G}_0(2,3) = 000$, $\mathcal{G}_1(2,3) = 001$, and $\mathcal{G}_2(2,3) = 011$.

A cyclic Gray code has the additional property that the Hamming distance of the first and last numbers in the Gray code is also one. $\mathcal{G}(\alpha,\ell)$ is a cyclic Gray code, and has the special property that if $g = \mathcal{G}_i(\alpha,\ell)$ and $h = \mathcal{G}_{(i+1) \bmod \alpha^\ell}(\alpha,\ell)$ differ in their kth character, then $h_k = g_k \oplus 1$.

We can define $\mathcal{G}(\alpha,\ell)$ in terms of a function that maps the base α, dimension ℓ integers, $\mathcal{N}(\alpha,\ell)$, to $\mathcal{G}(\alpha,\ell)$. For example, if x is the base α, dimension ℓ representation of the nonnegative integer i, then this function will map x to $\mathcal{G}_i(\alpha,\ell)$. This mapping was given in [43], and we give the same mapping in a slightly altered form.

The mapping is denoted by $\mathcal{K}$ and its inverse by $\mathcal{K}^{-1}$. Let an element in $\mathcal{G}(\alpha,\ell)$ be represented by the string $g = g_1 g_2 \ldots g_\ell$ and its corresponding base α integer be represented by the string $x = x_1 x_2 \ldots x_\ell$. The mappings are then defined as

$$\begin{aligned} \mathcal{K}(x) &= (g) \\ g_i &= \begin{cases} x_1 & \text{if } i = 1 \\ x_i \ominus x_{i-1} & 1 < i \leq \ell \end{cases} \end{aligned}$$

and

$$\begin{aligned} \mathcal{K}^{-1}(g) &= (x) \\ x_i &= \begin{cases} g_1 & \text{if } i = 1 \\ g_i \oplus x_{i-1} & 1 < i \leq \ell \end{cases} \end{aligned}$$

$\mathcal{G}(2,\ell)$ is the binary reflected Gray code. Both $\mathcal{K}$ and $\mathcal{K}^{-1}$ can be computed in parallel. That is, g_i can be written in terms of x, and x_i can be written in terms of g. For instance, $x_i = g_1 \oplus g_2 \oplus \cdots \oplus g_i$. See Figure 5.1 for two examples of this Gray code. Sharma and Khanna [43] discussed the structure of this Gray code, and described several other methods for generating it, including a direct method. They did further work [44] on this Gray code, as well. For another α-ary Gray code, the reader may wish to see Barasch, *et al.* [1]. A good discussion of the binary reflected Gray code and some of its uses is given by Bitner, *et al.* [2].

5.2 Iterating the Gray Code $\mathcal{G}(\alpha,\ell)$

In this section we prove a theorem on the number of unique codes produced when $\mathcal{N}(\alpha,\ell)$ is iteratively mapped using $\mathcal{K}^{-1}$. That is, we start with $\mathcal{N}(\alpha,\ell)$ and repeatedly apply the permutation given by $\mathcal{K}^{-1}$. Let $\mathcal{N}_j^i(\alpha,\ell) = \mathcal{K}^{-1}(\mathcal{N}_j^{i-1}(\alpha,\ell))$ and $\mathcal{N}_j^0(\alpha,\ell) = \mathcal{N}_j(\alpha,\ell)$. Iteratively applying $\mathcal{K}^{-1}$ to each code word can be seen as iterating $\mathcal{G}(\alpha,\ell)$, since $\mathcal{N}^1(\alpha,\ell) = \mathcal{G}(\alpha,\ell)$. (This can also be seen as iterating a landscape; that is, the representation of a GA or hill climber is a code, and by iterating this code/representation, we iterate the landscape.)

We want to know the number of unique codes that can be generated by iterating $\mathcal{N}(\alpha,\ell)$, or more formally , for what $i > 0$ does $\mathcal{N}^i(\alpha,\ell) = \mathcal{N}(\alpha,\ell)$ such that $\forall j, 0 < j < i, \mathcal{N}^j(\alpha,\ell) \neq \mathcal{N}(\alpha,\ell)$.

The following theorem follows easily from Theorem 5.3.1 (proven in Section 5.3):

$\mathcal{N}(2,3)$	$\mathcal{K}$	$\mathcal{G}(2,3)$
000	$\longmapsto$	000
001	$\longmapsto$	001
010	$\longmapsto$	011
011	$\longmapsto$	010
100	$\longmapsto$	110
101	$\longmapsto$	111
110	$\longmapsto$	101
111	$\longmapsto$	100

$\mathcal{N}(3,3)$	$\mathcal{K}$	$\mathcal{G}(3,3)$	$\mathcal{N}(3,3)$	$\mathcal{K}$	$\mathcal{G}(3,3)$
000	$\longmapsto$	000	112	$\longmapsto$	101
001	$\longmapsto$	001	120	$\longmapsto$	111
002	$\longmapsto$	002	121	$\longmapsto$	112
010	$\longmapsto$	012	122	$\longmapsto$	110
011	$\longmapsto$	010	200	$\longmapsto$	210
012	$\longmapsto$	011	201	$\longmapsto$	211
020	$\longmapsto$	021	202	$\longmapsto$	212
021	$\longmapsto$	022	210	$\longmapsto$	222
022	$\longmapsto$	020	211	$\longmapsto$	220
100	$\longmapsto$	120	212	$\longmapsto$	221
101	$\longmapsto$	121	220	$\longmapsto$	201
102	$\longmapsto$	122	221	$\longmapsto$	202
110	$\longmapsto$	102	222	$\longmapsto$	200
111	$\longmapsto$	100			

Table 5.1: Two example Gray codes: $\alpha = 2$ and $\ell = 3$ (l), and $\alpha = 3$ and $\ell = 3$ (r).

THEOREM **5.2.1** *Let $\ell > 1$ and $\alpha = p_1^{n_1} p_2^{n_2} \cdots p_q^{n_q}$ where p_i is prime, $p_i \neq p_j$ for $i \neq j$, (prime decomposition) and for each p_i, set h_i such that $p_i^{h_i - 1} < \ell \leq p_i^{h_i}$. Then $\mathcal{K}^{-1}$ will generate $m = p_1^{h_1 + n_1 - 1} p_2^{h_2 + n_2 - 1} \cdots p_q^{h_q + n_q - 1}$ unique codes.*

Proof:
We know, from Theorem 5.3.1, that for any string x, $x^m = x$. We also know that this upper bound is attained for any string such that $x_1 \neq 0$, $\mathrm{GCD}(x_1, \alpha) = 1$, and $\ell > 1$. Since this is true for some strings (e.g., any string whose first character is 1), we know that iterating $\mathcal{N}(\alpha, \ell)$ gives m unique codes.

■

If $\ell = 1$ then $m = 1$; for $\ell > 1$, $\ell^q \leq m < \alpha \ell^q$, where q is the number of unique prime factors in α. This implies that the maximum number of codes generated is $\Theta(\ell^q), \forall \ell$, and also implies that using $\mathcal{K}^{-1}$ to iterate a search space will produce $O(\ell^q)$ unique search spaces.

5.3 Iterating Strings Using $\mathcal{K}^{-1}$

In this section we will prove a theorem on the cycles induced when an α-ary, dimension ℓ string is iterated. We will use the notation $x^i = \mathcal{K}^{-i}(x) = \mathcal{K}^{-1}(\mathcal{K}^{-(i-1)}(x))$ and $x^0 = \mathcal{K}^0(x) = x$. We use subscripts to refer to a particular character in x. For example, if $x = x^0 = 12356$, then $x_3^0 = 3$.

A cycle on string x consists of the sequence $x^0, x^1, \cdots, x^{i-1}$, where $x^i = x^0$ and $\forall j$ such that $0 < j < i, x^j \neq x^0$. Since there are a finite number of strings with base α, length ℓ, we know that the length of any cycle must be finite. The following cycle

theorem gives an upper bound on the cycle length of any string x, and gives the actual cycle length when GCD(x_1, α)= 1 and $x_1 \neq 0$.

THEOREM **5.3.1** *Let $\ell > 1$ and $\alpha = p_1^{n_1} p_2^{n_2} \cdots p_q^{n_q}$ where p_i is prime, $p_i \neq p_j$ for $i \neq j$, (prime decomposition) and for each p_i, set h_i such that $p_i^{h_i - 1} < \ell \leq p_i^{h_i}$. If $m = p_1^{h_1+n_1-1} p_2^{h_2+n_2-1} \cdots p_q^{h_q+n_q-1}$, then $x^m = x$, and if $x_1 \neq 0$ and GCD(x_1, α)= 1 then for any $0 < m' < m, x^{m'} \neq x$.*

Given any α-ary string and a number m as described in Theorem 5.3.1, mapping the string with the inverse Gray code mapping m times will cause the iterated string to return to its original value. If $x_1 \neq 0$ and GCD(x_1, α)= 1 then m is the smallest integer for which the code word will cycle. For example, if $x = 1000$ and α= 2, then $x^1 = 1111$, $x^2 = 1010$, $x^3 = 1100$, $x^4 = 1000$, which implies that $m = 4$. The special case of this theorem for $\alpha = 2$ is proven by Culberson [8].

Before proving this theorem, we will prove a number of lemmas. Also note that all summations in this chapter are taken mod α.

Lemma 5.3.1

$$x_j^i = x_{j-1}^i \oplus x_j^{i-1}, i > 0, 1 < j \leq \ell$$

Proof:
From the definition of $\mathcal{K}^{-1}$.
■

Lemma 5.3.2

$$x_j^1 = x_1 \oplus x_2 \oplus \cdots \oplus x_j, 1 < j \leq \ell$$

and

$$x_1^i = x_1, i \geq 1$$

Proof:
Both statements follow from Lemma 5.3.1.
■

Lemma 5.3.3

$$x_j^i = \sum_{k=1}^{j} \binom{i+j-k-1}{j-k} x_k, 1 \leq j \leq \ell, i \geq 1$$

Proof:

Basis:

Lemma 5.3.2 is the Basis, and it can be seen that both of the statements in Lemma 5.3.2 are special cases of Lemma 5.3.3.

Induction Step:
I.H: assume Lemma 5.3.3 is true for x^i_{j-1} and x^{i-1}_j.

We know from Lemma 5.3.1 that

$$x^i_j = x^i_{j-1} \oplus x^{i-1}_j$$

and applying the induction hypothesis yields

$$x^i_j = \sum_{k=1}^{j-1} \binom{i+j-k-2}{j-k-1} x_k \oplus \sum_{k=1}^{j} \binom{i+j-k-2}{j-k} x_k$$

We now add the kth terms, $1 \leq k \leq j-1$ which gives us

$$\binom{i+j-k-2}{j-k-1} x_k \oplus \binom{i+j-k-2}{j-k} x_k$$

which is equal to

$$\binom{i+j-k-1}{j-k} x_k$$

which satisfies Lemma 5.3.3. The x_j term occurs only once, and also satisfies Lemma 5.3.3.

■

We now introduce the notation $c^i_{j,k}$, where $1 \leq k \leq j$. $c^i_{j,k}$ refers to the coefficient of the kth term of the equation given in Lemma 5.3.3 for x^i_j. That is

$$c^i_{j,k} = \binom{i+j-k-1}{j-k}$$

Lemma 5.3.4 *Let $x = x_1 x_2 \cdots x_\ell$ be any α-ary string. If $\forall j, 2 \leq j \leq \ell, c^i_{j,1} = 0 (\mathrm{mod} \alpha)$, then $x^i = x$.*

Proof:

To show $x^i = x$ we must show that $\forall j, 1 \leq j \leq \ell, x^i_j = x_j$. Note that $x^i_1 = x_1$; thus we need only consider an arbitrary j, $2 \leq j \leq \ell$.

Assume $\forall j', 2 \leq j' \leq \ell, c^i_{j',1} = 0 (\mathrm{mod} \alpha)$. Then

$$\begin{aligned} x^i_j &= \sum_{k=1}^{j} c^i_{j,k} x_k \\ &= (\sum_{k=1}^{j-1} c^i_{j,k} x_k) \oplus c^i_{j,j} x_j \\ &= (\sum_{k=1}^{j-1} c^i_{j,k} x_k) \oplus x_j \end{aligned}$$

since $c^i_{j,j} = 1$. But since

$$c^i_{j,k} = c^i_{j-k+1,1} = 0$$

the summation is equal to $0(\mathrm{mod}\alpha)$, and $x^i_j = x_j$.

■

Lemma 5.3.5 *Let $x = x_1x_2\cdots x_\ell$ be any α-ary string such that $x_1 \neq 0$ and GCD(x_1, α)= 1. If $\exists j, 2 \leq j \leq \ell$ such that $c^i_{j,1} \neq 0(\mathrm{mod}\alpha)$, then $x^i \neq x$.*

Proof:

Assume $x_1 \neq 0$, GCD(x_1, α)= 1, and $c^i_{j,1} \neq 0(\mathrm{mod}\alpha)$, for some $j, 2 \leq j \leq \ell$. If $c^i_{2,1} \neq 0(\mathrm{mod}\alpha)$, then $x^i_2 = ix_1 \oplus x_2 \neq x_2$ and we are done. Otherwise, $c^i_{j,1} \neq 0(\mathrm{mod}\alpha)$, for some j, $2 < j \leq \ell$ and that $c^i_{j',1} = 0(\mathrm{mod}\alpha)$, for $j', 2 \leq j' < j$. Then

$$x^i_j = (\sum_{k=1}^{j-1} c^i_{j,k}x_k) \oplus x_j$$

but note that

$$\begin{aligned} c^i_{j,k} &= c^i_{j-k+1,1} \\ &= 0, \forall k, 2 \leq k < j \end{aligned}$$

which means that

$$x^i_j = c^i_{j,1}x_1 \oplus x_j$$

since all the other terms are congruent to zero. Since $c^i_{j,1} \neq 0(\mathrm{mod}\alpha)$, $x^i_j \neq x_j$.

■

Lemma 5.3.4 shows that finding an m that sets $c^m_{j,1} = 0(\mathrm{mod}\alpha)$ for $2 \leq j \leq \ell$ implies that $x^m = x$. Lemma 5.3.5 shows that if we choose the smallest such m that sets $c^m_{j,1} = 0(\mathrm{mod}\alpha)$ for $2 \leq j \leq \ell$ and $x_1 \neq 0$ and GCD(x_1, α)= 1, then $x^{m'} \neq x$, for $0 < m' < m$. Our goal then will be to set the $c^m_{j,1}$ terms to zero $(\mathrm{mod}\alpha)$, picking the smallest such m that does so.

We now prove Theorem 5.3.1. To do this we use induction on j, and within the inductive proof, we will use the fact that $c^m_{j,1} = \frac{m+j-2}{j-1}c^m_{j-1,1}$. Since m increases with ℓ we will use the notation m_j, which refers to the cycle upper bound length on strings of length j. If $j > i$, then m_j will be a multiple of m_i. Let p_i be a prime divisor of α. Since h_i also varies with ℓ, we use the notation $h_{i,j}$ within the proof.

Proof:
(Theorem 5.3.1)

Basis (j=2):

$$c_{2,1}^{m_2} = m_2, \forall m_2 > 1$$

We pick the minimum m_2 that will set $m_2 = 0(\mathrm{mod}\alpha)$, or $m_2 = \alpha = p_1^{n_1} p_2^{n_2} \cdots p_q^{n_q}$.

It is easy to see that Theorem 5.3.1 is satisfied for $j = 2$, and that $c_{2,1}^{m_2}$ has n_i factors of $p_i, 1 \leq i \leq q$. In this case $h_{i,2} = 1$.

Induction Step ($j > 2$):

For the induction step we need only consider an arbritrary prime $p_i, 1 \leq i \leq q$:
I.H.1: Assume setting m_{j-1} as in Theorem 5.3.1 will set $c_{j',1}^{m_{j-1}} = 0(\mathrm{mod}\alpha), \forall j', 2 \leq j' < j$.
I.H.2: Assume $j - 1 = C_1 p_i^d + 1$ (where p_i does not divide $C_1, d < h_{i,j-1}$) implies $c_{j-1,1}^{m_{j-1}}$ has $n_i + h_{i,j-1} - 1 - d$ factors of p_i.

I.H.2 is needed because we must know how many factors of p_i are in $c_{j-1,1}^{m_{j-1}}$; if we know this, then using the fact that $c_{j,1}^{m_{j-1}} = \frac{m_{j-1}+j-2}{j-1} c_{j-1,1}^{m_{j-1}}$ we can determine the number of p_i factors that are in $c_{j,1}^{m_{j-1}}$. If there are n_i or more such factors, for each i, then $c_{j,1}^{m_{j-1}} = 0(\mathrm{mod}\alpha)$ and it is sufficient to set $m_j = m_{j-1}$; otherwise, m_j must be increased (while still being a multiple of m_{j-1}). This leads to two cases:

1. $j = C_2 p_i^{d'} + 1, 0 \leq d' < h_{i,j-1}$
2. $j = p_i^{d'} + 1, d' = h_{i,j-1}$

where C_2 contains no factors of p_i For each case we must now show that I.H.1 and I.H.2 hold for j.

Case 1 ($j = C_2 p_i^{d'} + 1, 0 \leq d' < h_{i,j-1}$):

Recall that $c_{j,1}^{m_{j-1}} = \frac{m_{j-1}+j-2}{j-1} c_{j-1,1}^{m_{j-1}}$. In this case $m_{j-1} + j - 2$ will have d factors of p_i and $j - 1$ will have d' factors of p_i, and the total number of factors of p_i will be $n_i + h_{i,j-1} - 1 - d + d - d' = n_i + h_{i,j-1} - 1 - d'$. Setting $m_j = m_{j-1}$ (and $h_{i,j} = h_{i,j-1}$) corresponds to Theorem 5.3.1; I.H.1 holds for j since we need at least n_i factors of p_i in $c_{j1}^{m_{j-1}}$, and this is the case. I.H.2 also holds.

Case 2 ($j = p_i^{d'} + 1, d' = h_{i,j-1}$):

In this case it can be easily seen that $c_{j,1}^{m_{j-1}}$ has $n_i - 1$ factors of p_i, but n_i factors are needed. We must show that setting $m_j = p_i m_{j-1}$ (as in Theorem 5.3.1, i.e., $h_{i,j} = h_{i,j-1} + 1$) will make $c_{j,1}^{m_j}$ have exactly one more factor of p_i than $c_{j,1}^{m_{j-1}}$, while leaving the number of all other prime factors unchanged. Then,

$$c_{j,1}^{m_j} = \frac{(m_j + j - 2)(m_j + j - 3) \cdots (m_j)}{(j-1)!}$$

and

$$c_{j,1}^{m_{j-1}} = \frac{(m_{j-1} + j - 2)(m_{j-1} + j - 3) \cdots (m_{j-1})}{(j-1)!}$$

The denominator of $c_{j,1}^{m_j}$ is equal to that of $c_{j,1}^{m_{j-1}}$, and so the denominators have the same number of factors of p_i. Thus we need only consider the numerator. Consider an arbitrary factor of the numerator of $c_{j,1}^{m_j}$, $m_j + k$, where $0 \leq k \leq j-2$. In the $k = 0$ case, $c_{j,1}^{m_j}$ has an extra factor of p_i. When k is non-zero, there are no extra factors of $p_s, 1 \leq s \leq q$. There are two cases to consider, $p_s \neq p_i$ and $p_s = p_i$. For the first case, $j - 2 < p_s^{h_{s,j}}$ which means that $m_j + k = (p_s E + F)p_s^t$ and $m_{j-1} + k = (p_s E' + F')p_s^t$ where F and F' have no factors of p_s, which means that the number of p_s factors is unchanged when $k \neq 0$. For the latter case a similar argument suffices, but uses the fact that $j - 2 < p_i^{h_{i,j-1}}$ (since $j = p_i^{h_{i,j-1}} + 1$).

■

When $x_1 \neq 0$ or GCD$(x_1, \alpha) \neq 1$, the m of Theorem 5.3.1 may be larger than the cycle length of x (though m will be a multiple of x's cycle length). For an example x where Theorem 5.3.1 describes the cycle length consider $x = 123456$ for $\alpha = 10$. In this case $m = 200$, as the theorem states. The strings 421 and 4211 for $\alpha = 8$ are two examples of strings whose minimum cycle lengths are 2 which is less than the m of Theorem 5.3.1.

5.4 Conclusion

In this chapter we discussed the problem of iteratively applying the inverse Gray code mapping to strings, and showed that a cycle on any string x will have length in $O(\ell^q)$ where ℓ is the length of x and q is the number of unique primes in α. If GCD$(x_1, \alpha) = 1$ and $x_1 \neq 0$, then x has a cycle length in $\Theta(\ell^q)$. This implies that the number of unique codes generated by iterating $\mathcal{N}(\alpha, \ell)$ (or any α-ary, dimension ℓ code) using $\mathcal{K}^{-1}$ is $\Theta(\ell^q)$. This implies that using $\mathcal{K}^{-1}$ to iterate a search space will yield $O(\ell^q)$ unique search spaces.

Chapter 6

Long Paths for α-ary Mutation and Crossover

In this chapter we discuss distance-preserving paths for α-ary crossover and α-ary mutation. A path $P = v_0, v_1, \cdots, v_n$ is *simple iff* $\forall v_i, v_j, 0 \leq i < j \leq n, v_i \neq v_j$. A simple path $P = v_0, v_1, \cdots, v_n$ in a graph G is *k-distance-preserving*, $DP(k)$, *iff* for any two vertices $i, j \in P$ that are t steps apart on the path, $dist(i, j) \geq \min(t, k + 1)$ For the binary hypercube, both simple paths and $DP(k)$ paths can be exponential in length [45, 26].

Since landscapes are graphs, it may be worthwhile trying to find these "long paths" or bounds on them for a particular landscape graph. This is done for the binary one-point mutation landscape (the Hamming hypercube) by Horn, *et al.* [29, 30, 31]. They construct a unimodal (no false optima and a single optimum) landscape that is hard for mutation by sloping the fitness of all points not in the path towards the start of the path and then making the $DP(1)$ path strictly increasing. This makes any elitist algorithm on *this* landscape follow the slope towards the start of the path, eventually falling onto the path (not necessarily at the start), followed by constant progress to the optimum (the end of the path).

In the worst case, such a hill climber does an exponential amount of work: start the hill climber on the first vertex on the path; it is then forced to follow the path to the end, and since this $DP(1)$ path is exponential in ℓ, the hill climber will take an exponential amount of time. Horn, *et al.*, also show empirically that an elitist hill climber that starts on a random vertex (chosen uniformly) takes an exponential amount of time on average to find the optimum. Long paths have also been used to discriminate between different types of elitist hill climbers [10].

The results of Horn, *et al.*, are interesting because they show how bad a supposedly easy (e.g., unimodal) landscape can be. In this chapter we extend the $DP(1)$ path construction to α-ary hypercubes, and explore the problem of creating distance-preserving paths for crossover. We present the counter-intuitive result that crossover can have $DP(k)$ paths.

6.1 Long Paths for α-ary Mutation

Long paths have previously been considered for the α-ary hypercubes [10]. While these paths are exponential in length, a better construction is possible. Let P be a path, which can be represented by the list of strings in P. Let the first string in P be $\mathbf{FIRST}(P)$ and the last be $\mathbf{LAST}(P)$. Let the reversal of P be denoted by $\overline{P}$; for example, if $P = \{00, 01, 11\}$, then $\overline{P} = \{11, 01, 00\}$. The concatenation of two or more lists $p_1, p_2, \cdots, p_n$ will be given by $\{p_1, p_2, \cdots, p_n\}$. Finally, given a path P, the notation cP, where c is a character, means that c is prepended to every string in P.

Using the above notation, we can now define our α-ary $DP(1)$ paths for the α-ary hypercube. The construction given is a generalization of the construction of Horn, *et al.* [29], and when $\alpha = 2$, our construction reduces to theirs. Let P_ℓ be a path on the (α, ℓ)-cube (where ℓ is odd). Then

$$\begin{aligned} P_{\ell+2} \;=\; & \{00P_\ell, \\ & 01\mathbf{LAST}(P_\ell), \\ & 11\overline{P}_\ell, \\ & 12\mathbf{FIRST}(P_\ell), \\ & 22P_\ell, \\ & \vdots \\ & (\alpha-2)(\alpha-1)\mathbf{FIRST}(P_\ell), \\ & (\alpha-1)(\alpha-1)P_\ell\} \end{aligned}$$

if α is odd; otherwise,

$$\begin{aligned} P_{\ell+2} \;=\; & \{00P_\ell, \\ & 01\mathbf{LAST}(P_\ell), \\ & 11\overline{P}_\ell, \\ & 12\mathbf{FIRST}(P_\ell), \\ & 22P_\ell, \\ & \vdots \\ & (\alpha-2)(\alpha-1)\mathbf{LAST}(P_\ell), \\ & (\alpha-1)(\alpha-1)\overline{P}_\ell\} \end{aligned}$$

The basis for this path is $P_1 = \{0, 1\}$. Thus a $DP(1)$ path on strings of length $\ell + 2$ consists of α sub-paths connected by $\alpha - 1$ "bridge" points. A $DP(k)$ construction is also easy to derive. Given a length ℓ, k-distance-preserving path P, an $\ell + k + 1$ path can be constructed by making α copies of P, say P_c, where $c = 0, 1, \cdots, \alpha - 1$, and appending $ccc\cdots c$ ($k + 1$ $c's$) to P_c. Then connect each sub-path P_c to the sub-path P_{c+1} using a bridge point. The base path is { 0,1 }.

Let $|P_\ell|$ be the length of a path on ℓ-character strings. Then $|P_{\ell+2}| = \alpha|P_\ell| + \alpha - 1$, with basis $|P_1| = 2$ and $|P_3| = 3\alpha - 1$. Solving this recurrence yields $|P_\ell| = 3\alpha^{\frac{\ell-1}{2}} - 1$ if

ℓ is odd.[1] Thus these $DP(1)$ paths are exponential in ℓ, and are longer than a previous α-ary $DP(1)$ path construction [10]. This construction also lends credence to the view that the fraction of the search space (for this construction, $O(\alpha^{-\ell/2})$) contained in a $DP(1)$ path will decrease with α [10]. This seems a reasonable conjecture, since increasing α increases the degree of each vertex; for each vertex used in the path, more must be excluded.

We can use this $DP(1)$ path to construct a landscape that is unimodal but exponentially hard for an elitist hill climbing algorithm that uses only one-point α-ary mutation. The landscape construction is similar to that done by Horn, *et al.* [29]. Let the summation function, $f_\Sigma(x) = \sum_{i=1}^{\ell} x_i$. Since the first point on the path is always the all-zeroes string, giving any point x not on the path the value $(\alpha-1)\ell - f_\Sigma(x)$ will lead a hill climber towards the start of the path. If a point is on the path P, then that point's fitness value is $(\alpha-1)\ell + x$'s position in P. We give the pseudo-code for the fitness function that generates the landscape of Figure 6.1. The last string in the path will be 1 for the base path; otherwise, it will be $(\alpha-1)(\alpha-1)00\cdots0$ for even α and $(\alpha-1)(\alpha-1)\cdots(\alpha-1)1$ for odd α.

In the worst case, an elitist hill climber will be forced to do an exponential amount of work to reach the optimum (e.g., start on the first vertex of the path). We show empirically that an elitist hill climber on this landscape will be expected to take exponential time on average to find the optimum, where the initial vertex is chosen uniformly. See Figure 6.2 for the results of running our mutation hill climbing algorithm, MHC (see Appendix B), on this landscape for $\alpha = 2, 3, 4, 5$. Clearly, MHC takes exponential time on average (the estimated standard deviations were small). It seems reasonable that any elitist hill climber on this landscape will take exponential time on average, since the overall fraction of the search space contained in the path decreases exponentially in ℓ; this means that a hill climber should be less likely to skip the first vertex in the path and take "large" short-cuts.

6.2 Long Paths for Crossover

In this section we discuss constructing distance-preserving paths for crossover landscapes. We first discuss long paths for binary one-point crossover. We create a $DP(1)$ path for one-point crossover on a complementary pair of strings, and suggest how $DP(1)$ paths can be constructed for the one-point crossover SSS. We then try extending these long paths to bigger populations by creating a "naive" long path landscape for crossover and do some tests on this landscape. These results are preliminary but interesting in that, even with non-minimal populations, crossover appears to follow exponentially long paths. We then discuss long paths for α-ary crossover and possible future research directions.

[1] We can construct a $DP(1)$ path for even $\ell > 0$ by ignoring the first character in a string, and constructing an $\ell - 1$ path on the remaining characters.

```
GLOBAL boolean onpath;

/* If x in path, return x's position, else set onpath to false. */
int path-position(x, ℓ, α) {

        char c = x_1;

        /* basis */
        if (x == "0") return 0;
        if (x == "1") return 1;

        /* possibly in path */
        if (x == cc x_3 x_4 ··· x_ℓ) {
                if (even(c)) return (c(|P_{ℓ-2}| + 1)+path-position(x_3 ··· x_ℓ, ℓ − 2, α));
                else return (c(|P_{ℓ-2}|+1)+|P_{ℓ-2}|−1−path-position(x_3 ··· x_ℓ, ℓ−2, α));
        }

        /* must be a bridge point, or not in path*/
        if (even(c)) {
                if (x == "c(c + 1)1" or
                   (even(α) and x == "c(c + 1)(α − 1)(α − 1)00 ··· 0") or
                   (odd(α) and x == "c(c + 1)(α − 1)(α − 1) ··· (α − 1)1"))
                        return ((c + 1)(|P_{ℓ-2}| + 1) − 1);
        } else {
                if (x == "c(c + 1)0" or x == "c(c + 1)00 ··· 0")
                        return ((c + 1)(|P_{ℓ-2}| + 1) − 1);
        }

        onpath=false;
        return -1;
}

/* Compute fitness of string x (α-ary mutation long path). */
int string-fitness(x, ℓ, α) {

        int temp;

        onpath=true;
        temp=path-position(x, α, ℓ);
        if (onpath) return ((α − 1)ℓ + temp);
        else return ((α − 1)ℓ − f_Σ(x));
}
```

Figure 6.1: Fitness function for α-ary mutation long path.

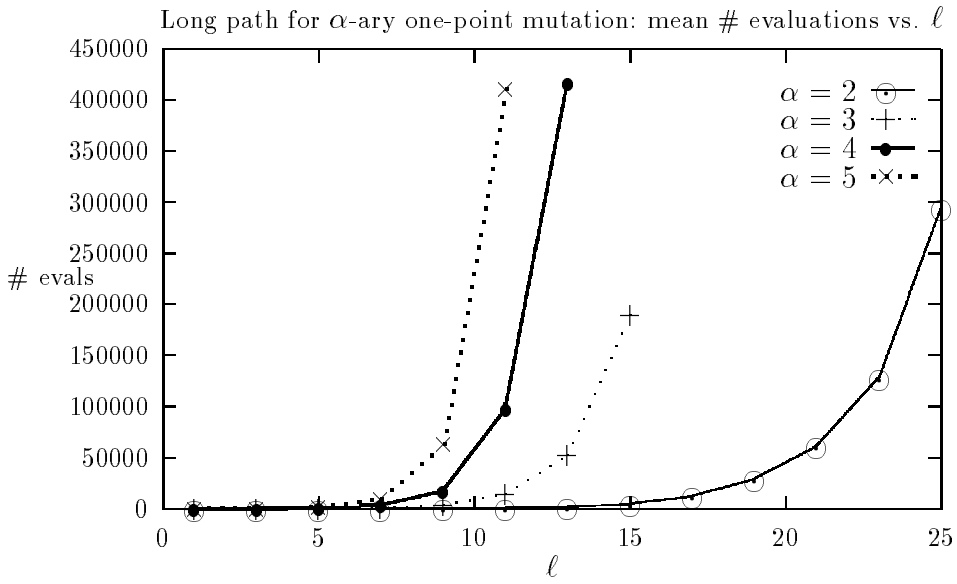


Figure 6.2: Testing the α-ary mutation long path landscape using MHC($n = 1, 50\ trials/point, \mathcal{F} = 5, elitism$). The maximum number of evaluations was 500000, and the MHC always found the end of path except for the $\ell = 13$ point for $\alpha = 4$ (in this case it found the path 28 out of 50 times). Standard deviations were relatively low with respect to the mean.

6.2.1 Binary Crossover

As a first step to constructing distance-preserving paths for one-point crossover landscapes, we can use the fact that one-point crossover between two complementary strings is isomorphic to the Hamming hypercube, and thus isomorphic to one-point mutation. This means we can construct a $DP(1)$ path in the one-point $(\alpha = 2, \oplus)$ crossover SSS for even $\ell > 0$. The fitness function that generates a unimodal long path landscape can be based on the α-ary mutation long path fitness function; e.g., if x is a string in the one-point $(\alpha = 2, \oplus)$ crossover SSS, then the fitness of x is given by **string-fitness**$(\mathcal{I}^{-1}(x, \overline{x}), \ell - 1, \alpha)$. We used GIGA on a pair of complementary strings as our crossover hill climber. Test results are shown in Figure 6.3 and are clearly exponential.

It is also possible to construct $DP(1)$ paths for the binary one-point crossover SSS, the landscape for a (possibly) non-complementary pair of binary strings under crossover. This can be done by recalling that two strings with Hamming distance h search a hypercube (possibly with self-loops) of dimension $\ell - h - 1$, and constructing a path for the $\ell - h - 1$ hypercube. This construction, however, would depend on the strings used; that is, two pairs of strings may both have Hamming distance h, but if the pairs are complementary in different locations, then they require different long path constructions.

Thus, for trivial crossover populations ($n = 2$), constructing $DP(1)$ paths for one-point crossover are easy. This is a nice result, but crossover is rarely used on trivial

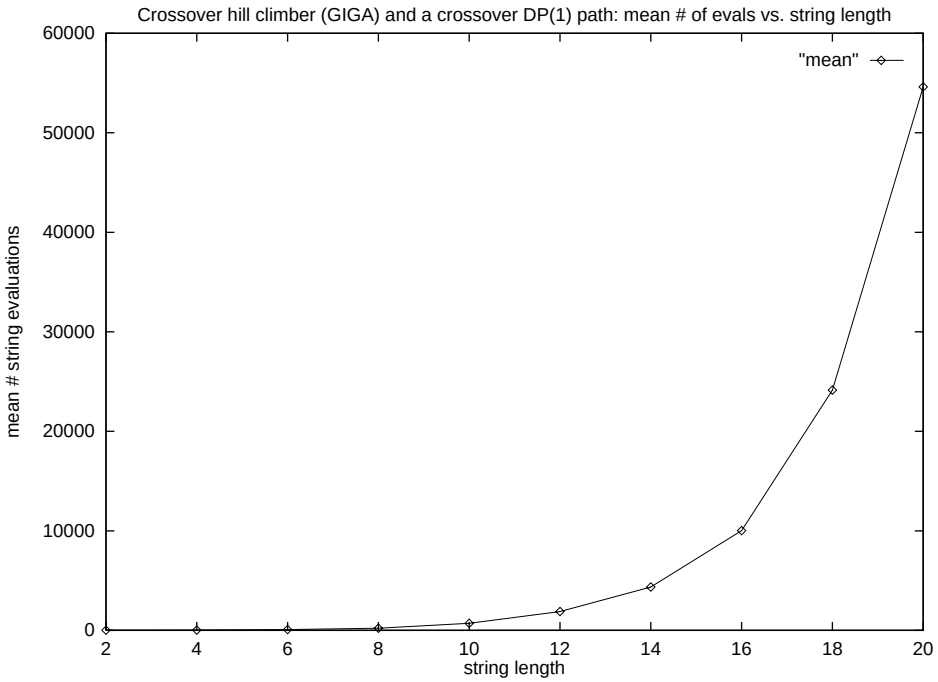


Figure 6.3: Testing the binary crossover long path landscape on minimal populations (i.e., complementary pair of strings). GIGA was used as our crossover hill climber ($n = 2$, $\alpha = 2$, 50 trials, $\mathcal{F} = 5$, elitism).

populations. Is it possible to construct a non-trivial population, crossover landscape that forces an elitist search algorithm (on this landscape) to follow a long path? Our first attempt to construct such a landscape is to use GIGA with bigger populations, and using the fitness function **string-fitness**$(\mathcal{I}^{-1}(x, \overline{x}), \ell - 1, \alpha)$ as in the $DP(1)$ path for the one-point $(\alpha = 2, \oplus)$ SSS. We term this the "naive" crossover long path landscape. This landscape potentially has exponentially long paths: e.g., on a complementary pair.

This landscape may or may not have a $DP(1)$ path, and we do not attempt to find one. It is also not unimodal. In this section we concern ourselves with the average number of fitness evaluations used by GIGA to find an optimal string, as this should reflect on the length of the path GIGA followed to get to the optimal point in the landscape[2]. Rather than worrying whether this landscape is unimodal, we instead measure the probability that GIGA does not find the optimal point (e.g., hits a false optimum). We want paths that are long on average and a landscape with few (or an insignificant number) of false optima.

GIGA's parameter's were $\mathcal{F} = 5, \alpha = 2$, elitism, 100 trials/point, and max evals=100000. Default GIGA parameters are listed in Appendix B, Table B.1. This "naive" long path crossover landscape was tested for $n = 4, 8$, and 20. We only include trials that ended in a successful run (where an optimal population was found).

[2]There are actually many optimal points, e.g., any population with an optimal member string.

The results are in Figures 6.4 and 6.5. The results are interesting, and may or may not be exponential. For $n = 20$, the plot appears exponential but levels off after $\ell = 44$, which is not shown. This levelling is likely because GIGA needed more evaluations to find the longer paths; that is, the paths were so long that more than 100000 evaluations were required to reach the end of the path. This is supported by the fact that GIGA was finding paths that used 99000+ evaluations. A solution to this would be to run GIGA until the maximum string is found or a false optimum is hit (to check this GIGA would first have to be converted into a strictly elitist hill climber; otherwise, GIGA might follow cycles). Also, the bottom plot in Figure 6.5 suggests that the probability of hitting a false optimum increases with increasing ℓ and decreases with increasing n.

We include some results from the $n = 4, \ell = 10$ tests that may help the reader get a picture of how GIGA is searching on this landscape. In the first (see Figure 6.6), GIGA (almost certainly) gets stuck on a false optimum (maximum fitness is 55). The next set of results (Figure 6.7) shows an experiment where GIGA quickly found the maximum and made large improvements (jumps) in the value of the best string found. Here is an example of population helping the search. The last set of results (Figure 6.8) is interesting because it shows GIGA making small improvements (where the fitness increases by 1), followed by a leap in fitness value, followed by yet more small improvements to an optimal point.

While these experiments are interesting, there is a potential problem that should be noted. Because GIGA is not a strict hill climber, it can wander in plateaus and thus can follow cycles. This means the average number of evaluations may not be a true reflection of the path length. However, this did not seem to be a problem in our tests.

The problem of hitting false optima may (possibly) be reduced by using a more general mating strategy in GIGA; for instance, GIGA could be altered to select non-adjacent as well as adjacent pairs to mate. This could possibly eliminate many of the false optima.

6.2.2 Some Possible Future Research Directions

Because "general" crossover hill climbers can be very complex, finding exponentially long $DP(1)$ paths appears to be very difficult. It may be possible, for some *specific* class of populations, to give an inductive $DP(1)$ construction that is exponentially long; however, it would be much harder finding a construction for general populations, of the sort that are likely to appear initially in a GA. Different hill climbers would also require different constructions.

$DP(1)$ paths for "general" hill climbers can be generated using a brute force method: e.g., start with some initial population, and follow a $DP(1)$ path. This is not a practical algorithm for large ℓ, and the $DP(1)$ paths that are generated need not be exponential in length. Indeed, some algorithms may not even have exponentially long $DP(1)$ paths on larger populations.

Because of this apparent difficulty, we focus on possible research directions for further exploring the long path problem.

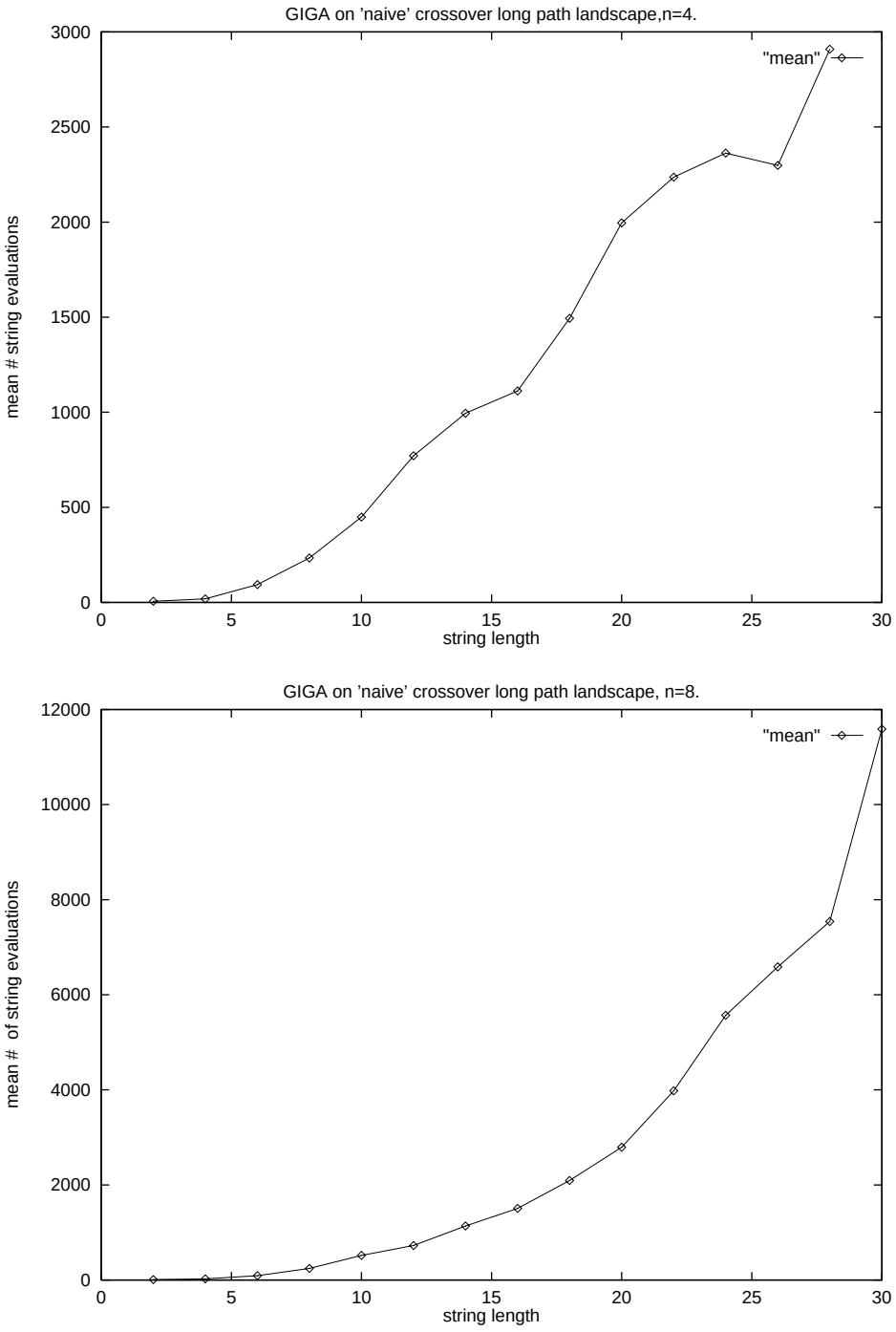


Figure 6.4: Testing the "naive" long path crossover landscape using GIGA w/ elitism. $n = 4$ (top), $n = 8$ (bottom). Standard deviations were high, generally close to the mean in size.

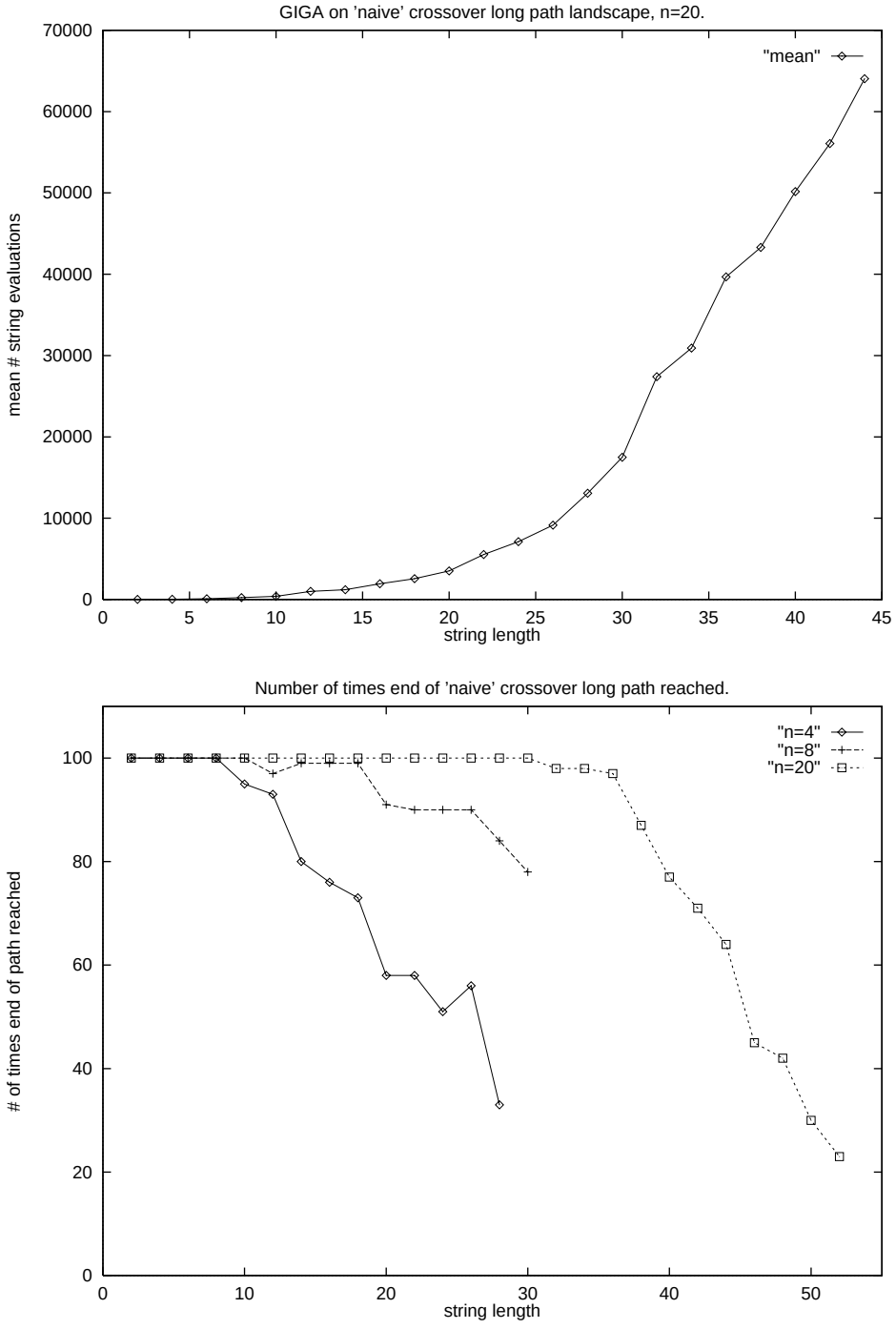


Figure 6.5: Testing the the "naive" long path crossover landscape (top), and number of times optimal point reached for $n = 4, 8$, and 20 (bottom).

```
Experiment Number 11:
Matings    Evaluations     Maximum
      0              4      5.000000
      1             14      39.000000
      2             24      40.000000
      6             64      40.000000
      9             94      41.000000
     13            134      54.000000
 100000         100004      54.000000 (Final Results)
```

Figure 6.6: Example run on "naive" crossover long path landscape. $n = 4, \ell = 10$. This experiment demonstrates GIGA probably getting stuck on a false optimum. (Optimum is 55.)

```
Experiment Number 2:
Matings    Evaluations     Maximum
      0              4      7.000000
      2             24      7.000000
      4             44      11.000000
      5             54      27.000000
      6             64      30.000000
     17            174      31.000000
     18            184      32.000000
     20            204      48.000000
     23            234      55.000000
 100000            234      55.000000 (Final Results)
```

Figure 6.7: Example run on "naive" crossover long path landscape ($n = 4, \ell = 10$). This experiment is an example where GIGA quickly finds the maximum string (Optimum is 55).

```
Experiment Number 13:
Matings    Evaluations     Maximum
      0              4      4.000000
      1             14      24.000000
      2             24      25.000000
     15            154      26.000000
     22            224      27.000000
     25            254      28.000000
     40            404      29.000000
     41            414      30.000000
     42            424      31.000000
     43            434      32.000000
     44            444      44.000000
     45            454      45.000000
     47            474      46.000000
     49            494      47.000000
     52            524      48.000000
     56            564      49.000000
     57            574      50.000000
     64            644      51.000000
     76            764      52.000000
     83            834      53.000000
     85            854      54.000000
     88            884      55.000000
 100000            884      55.000000 (Final Results)
```

Figure 6.8: Example run on "naive" crossover long path landscape ($n = 4, \ell = 10$). This experiment shows a longer path; i.e., GIGA makes small improvements.

Binary Crossover Long Paths

While it seems difficult to construct $DP(1)$ paths for general populations, it may be possible to construct landscapes that are *virtually* $DP(1)$ paths; that is, a hill climber on one of the landscapes may be forced to follow an exponentially long path most or all of the time for most populations. This was the goal taken when constructing the "naive" crossover landscape.

On the "naive" crossover landscape we wanted to demonstrate exponential behaviour in the average case, but we can probably get better long path landscapes if we only consider a worst case analysis. That is, we generate a long path landscape for each initial population.

There are many ways this could be done. One obvious method is to pick the first two strings of the population and construct a long path on their complementary bits. The path is created so that these strings are initially on the path. The value of a string is 0 if it is not in the path and not in the initial population, 1 if it is in the initial population and not on the path, and 2 plus its position in the path, otherwise. For large enough ℓ, it seems unlikely that crossover between any strings but the first two will produce a fitness greater than zero. There will be populations in which this is not true, even populations in which two strings not on the path can be crossed (once) to reach the optimal point; however, for most reasonably-sized populations, this sort of behaviour should be rare, because the path is exponentially small with respect to the size of the representation space (2^ℓ). A similar method could also be used for α-ary one-point crossover.

α-ary Crossover Long Paths

In this subsubsection we consider using a crossover hill climber similar to GIGA, except that there is no sorting of any kind, implicit or explicit; that is, the first character of each string is "rooted" in place.

Exponential $DP(1)$ paths are easy to derive for one-point (α,$*$) crossover. To do this just use $\mathcal{J}^{-1}$ and the α-ary mutation long path fitness function, as was done for the binary case in the previous subsection. An example $DP(1)$ path for $\ell = 4, \alpha = 3$ under one-point (α,$*$) crossover (rotational crossover) is:

0000	0001	0012	0120	0122	0100	0211	0212
1111	1112	1120	1201	1200	1211	1022	1020
2222	2220	2201	2012	2011	2022	2100	2101

$DP(1)$ paths can also be constructed for the α-ary one-point crossover SSS by constructing a $DP(1)$ path on the Hamming hypercube of dimension $\ell - h - 1$, where h is the Hamming distance between the two α-ary strings in this search space structure.

There are several ways long path landscapes could be constructed for α-ary one-point crossover, and we give two possible methods. In the first, we could use populations of size α generated by random rotation, and let the fitness of a string be the position of the string in the one-point (α,$*$) crossover SSS long path. Any string

not in this path would have fitness of 0. For example, the strings 0000, 1111, and 2222 would all have fitness 1. The optimal strings 0212, 1020, and 2101 would have a fitness of 8.

Using one-point crossover on this population may force an elitist crossover hill climber to follow a long path (there are definitely exponentially long paths since one-point crossover can simulate one-point (α,$*$) crossover). However, a crossover search mating only adjacent pairs may get stuck on a false optima. The following is an example of this. (The number above each population is the maximum value of any string in the population.)

1	2	2	3
0000	0001	0001	0001
1111	1110	1112	1120
2222	2222	2220	2212

If the first and last strings could be mated, then this would not be a false optima. Choosing non-adjacent pairs as well as adjacent pairs would also eliminate this problem.

If a crossover search algorithm would have to simulate one-point (α,$*$) crossover in lock-step, then this would be a $DP(1)$ path. However, this is not the case. To see this consider the following example:

1	2	3	6	6	6	6	7	8
0000	0001	0022	0201	0110	0111	0100	0022	0022
1111	1110	1110	1110	1201	1200	1211	1211	1210
2222	2222	2201	2022	2022	2022	2022	2100	2101

Since crossover can take short-cuts, this is not a $DP(1)$ path and there may be false optima. However, it may also be that, while there are many paths to the optimal point, each path is exponential in length, or exponential in length on average.

The other type of long path landscape parallels the "naive" crossover landscape for binary strings. That is, we could use $\mathcal{J}^{-1}()$ and the fitness function for the α-ary long path mutation landscape on population sizes greater than α.

6.3 Conclusion

In this chapter we discussed the long path problem for α-ary mutation and α-ary crossover. We constructed exponentially long $DP(1)$ paths for one-point α-ary mutation, and created a strictly increasing landscape for one-point α-ary mutation such that the fitness of points not on the $DP(1)$ path are sloped towards the start of the path. Test results show that a mutation hill climber starting at a random vertex (chosen uniformly) takes exponential time on average to reach the end of the path. Thus this landscape is exponentially hard in both the worst and average cases.

We then created an exponentially long $DP(1)$ path for crossover between a complementary pair of binary strings, and constructed a long path landscape for crossover. A crossover hill climber took exponential time on average to reach the end of the path.

The fitness function for the crossover long path between complementary strings was used to create a "naive" crossover long path class of landscapes for crossover on non-minimal populations. The results are interesting—GIGA appears to follow exponentially long paths on average—but non-conclusive.

Methods for making long paths for α-ary one-point crossover were also discussed, but most of this is left for future work. Hyper-order crossovers and hyper-order mutations may also be used to construct interesting, potential long path landscapes.

Chapter 7

Schizophrenic Functions

In this chapter we create the schizophrenic function. This function has two classes of optima, and is constructed in such a way that searches using mutation are expected to find one optima class, searches using crossover the other. That is, it is a function that should be able to discriminate between searches that use mutation or crossover. There is a long history of discriminating functions in the GA literature. Some examples include the Royal Road functions [18], the long path problems [29, 30, 31, 10], and others [8].

We expect that the schizophrenic function will have several uses. Researchers can use it to compare how well their genetic algorithms use crossover and mutation and to study what factors affect crossover and mutation. We use the schizophrenic function for both purposes.

There are two main sections to this chapter. In the first, we define the schizophrenic function in the binary case. The work here is related to the work by Culberson [5, 8]. In the next section, we define an α-ary schizophrenic function. For both sections, we run various test cases to offer support for (and, it turns out, some evidence contrary to) our original expectations. All in all, we feel that the schizophrenic function is a useful but imperfect tool.

7.1 The Binary Schizophrenic Function

Culberson [8] argues that traditional genetic algorithms are more heavily reliant upon mutation than crossover and do not use crossover well (e.g., converged populations make crossover ineffective as a search operator), and several papers in the literature support this view [46]. We use the schizophrenic function to provide additional evidence for this. Analyzing the test results also offers insight into how population and crossover can interact in GIGA.

We test the schizophrenic function on four different probabilistic search algorithms (see Appendix B): a one-point mutation hill climber (MHC), a traditional genetic algorithm (TGA), the gene invariant genetic algorithm (GIGA), and an extension to GIGA (NQ-GIGA); thus testing the notion that the schizophrenic function can discriminate between searches that use crossover well (GIGA,NQ-GIGA) and those

that do not use crossover (MHC) or do not use it well (TGA). Our tests will focus on GIGA and its extension, NQ-GIGA, and by doing so we analyze how population and crossover interact in GIGA.

Although we only define and test one schizophrenic function, there are many classes of schizophrenic functions, and many functions can become the basis of a schizophrenic class. Schizophrenic functions could also be designed to discriminate between genetic operators other than crossover and mutation, or to discriminate more than two operators, say mutation, crossover, and inversion. They can also be used to explore the effects of multiple levels (or iterations) of Gray coding.

7.1.1 Function Transformations Using Isomorphisms

Recall that $\mathcal{I}$ is given by:

$$\begin{aligned} \mathcal{I}(x) &= (a, \bar{a}) \\ a_i &= \begin{cases} 0 & \text{if } i = 1 \\ x_{i-1} \oplus a_{i-1} & 1 < i \leq \ell + 1 \end{cases} \end{aligned}$$

and is an isomorphism between the landscapes induced by *one-point binary mutation* on a string of length ℓ and *one-point binary crossover* on two complementary strings of length $\ell + 1$ [8].

This isomorphism can be used to transform functions that have certain properties with respect to a mutation-based search to functions that have those same properties with respect to a crossover-based search on a complementary pair of strings. If, for instance, a mutation-based search finds a function f hard, we can construct an equally hard function f' for crossover using $\mathcal{I}$. We can go the other direction using its inverse. Culberson [8] uses this isomorphism to create functions that discriminate between mutation- and crossover-based searches.

We stress that this isomorphism is between mutation on a single string and crossover on a pair of complementary strings; when the population size is not minimal (>2 for crossover, >1 for mutation) the isomorphism can "break." For example, a hard function for mutation transformed into a hard function for crossover on a complementary pair may not be hard for crossover on populations greater than two.

7.1.2 Definition of the Binary Schizophrenic Function

The ones-counting (or ones-max, or unitation) function is

$$f_u(x) = \sum_{i=1}^{\ell} x_i$$

and the transitions-counting function is

$$f_t(y) = \sum_{i=2}^{\ell} y_i \oplus y_{i-1}$$

where x and y are both binary strings of length ℓ.

The ones-counting function is easy for one-point mutation hill climbers, because the landscape is strictly increasing, and the longest (increasing) path is of length ℓ. For similar reasons, k-point mutation for small k and uniform mutation for small $p_m, p_m \approx 1/\ell$, should also find the ones-counting function easy. Note that $f_t(x) = f_t(\bar{x}) = f_u(\mathcal{I}^{-1}(x, \bar{x})), \ell \geq 2$, which means that one-point crossover on a pair of complementary strings will find the transitions-counting function as easy as one-point mutation finds the ones-counting function [8].

The ones-counting function is easy for one-point mutation but difficult for one-point crossover [8], as crossover has exponential in h false optima on the one-point crossover SSS landscape given by the ones-counting function, where h is the Hamming distance of the two strings being crossed. These false optima appear to make the problem hard for crossover. (Two-point crossover does not have these false optima, since two adjacent crossovers can mimic a mutation.) The transitions-counting function is easy for crossover but hard for mutation, because there are large plateaus in the one-point mutation SSS landscape for the transitions function, and doing mutation at a higher rate makes the problem even harder.

When the populations are non-minimal, crossover can find the ones-counting function relatively easy. For example, in GIGA [6] high-fit ones-substrings became concentrated at the bottom of the population, the average Hamming distance of these strings was reduced, and the search became easy. In this case bigger population searches did not echo the minimal population searches. However, the transitions-counting function is still easy for bigger populations [8].

Using the transitions- and ones-counting functions we can construct a binary schizophrenic function:

$$\text{bSchizo}(x) = \begin{cases} \sum_{i=3}^{\ell} x_i \oplus x_{i-1}, \text{ if } x_1 = 0 \\ \sum_{i=3}^{\ell} x_i, \text{ otherwise} \end{cases}$$

It takes a binary string x of length ℓ, $\ell \geq 3$. There are two possible optima when $x_1 = 1, 1011 \cdots 1$ and $111 \cdots 1$, and two when $x_1 = 0, 0010101 \cdots$ and $0101010 \cdots$. The four optima each have the value of $\ell - 2$, and both halves of the domain have the same quantity of functional values. In effect, the domain has been split between two different functions; one that is easy for crossover and hard for mutation, and one that is easy for mutation and hard for crossover (on minimal populations). We expect that a crossover-based search will have a difficult time finding a mutation optima even on non-minimal populations, because the search algorithm will have a hard time concentrating ones-strings since there will be equally fit transitions-strings interspersed between them.

We conjecture that searches using crossover will find the crossover half of the landscape easier than the mutation half, and so will usually find one of the crossover optima (the optima where $x_1 = 0$) before they find the mutation optima. In the same way, mutation-based searches will find the mutation half of the landscape easier to climb, and will usually find one of the mutation optima before a crossover optima.

GIGA: ($n = 2, \alpha = 2$, 1000 trials, $\mathcal{F}$=4, non-elitism)

MAX EVALS	ℓ	# mutation optima	# crossover optima	no optima
8000	20	6	994	0
80000	40	0	998	2

MHC: ($n = 1, \alpha = 2$, 1000 trials, $\mathcal{F}$=4, non-elitism)

MAX EVALS	ℓ	# mutation optima	# crossover optima	no optima
8000	20	665	335	0
40000	40	511	21	468
80000	40	595	31	374

Table 7.1: Testing the binary schizophrenic function: minimal populations, GIGA (top) and MHC (bottom).

7.1.3 Testing the Binary Schizophrenic Function

When testing the binary schizophrenic function we split the tests into those that use minimal and non-minimal populations. We do this for two reasons:

1. Non-minimal populations "break" the mutation-crossover isomorphism—results on minimal populations may not apply to larger populations.

2. Differences between the minimal and non-minimal population test results can offer insight into how population and crossover interact in GAs.

Minimal Populations

To test the conjecture that a search using mutation will find a mutation optimum and a search using crossover will find a crossover optimum, we tested GIGA on the binary schizophrenic function on a population of two complementary binary strings. Since the strings are complementary, it is possible for crossover to find any length ℓ string. For these experiments we modified GIGA to record whether the first optimum found was a mutation or crossover optimum. Our definition of best pair was the maximum fitness value of either of the strings in the pair. GIGA also has an option for *explicit* sorting, which we did not use. For testing the mutation half of the conjecture, we used our mutation hill climber, MHC, with one-point mutation on a population of one string.

The results for GIGA and the MHC are summarized in Table 7.1. These results support our conjecture, at least for the minimal populations that we used. However, the results are clearly not symmetric. There are at least two reasonable explanations for this.

The first explanation is that the mutation string could be in either the mutation- or crossover-half of the landscape but not both, while crossover always had a string in

GIGA ($n = 20, \alpha = 2$, 500 trials, $\mathcal{F}$=4, elitism)

MAX EVALS	ℓ	# mutation optima	# crossover optima	no optima
50000	20	122	378	0
60000	40	15	485	0
70000	60	1	496	3

GIGA ($n = 20, \alpha = 2$, 500 trials, $\mathcal{F}$=4, non-elitism)

MAX EVALS	ℓ	# mutation optima	# crossover optima	no optima
50000	20	171	329	0
60000	40	317	27	156
70000	60	14	0	486

Table 7.2: GIGA on the binary schizophrenic function: non-minimal populations, elitism (top) and non-elitism (bottom).

each of the mutation- and crossover-halves of the landscape (non-symmetric switch). A symmetric switch could be designed by using the first two bits of the string, rather than the single bit currently used as the switch. If the bits were a transition (e.g., 01 or 10) the string would be in the crossover-half of the landscape; otherwise, it would be in the mutation-half of the landscape. This means crossover would be able to activate the switch. This switch is not perfectly symmetric, however, since crossover only has one crossover point that activates the switch, while mutation can mutate either the first or second bits.

The second explanation is that, while crossover finds the crossover-half of the landscape as easy as mutation finds the mutation-half of the landscape, crossover may find the mutation-half of the landscape harder than mutation finds its crossover-half of the landscape. (The functions are not symmetric.)

Non-minimal Populations: GIGA

For non-minimal populations we first used GIGA with elitism. Both types of optima can now occur in the population, but we only kept track of the first occurrence. The results are in Table 7.2 (top) and seem to support our conjecture that crossover has a harder time finding optimal mutation strings than optimal crossover strings. This seems to be because the mutation strings cannot be concentrated together since there are both highly fit crossover- and mutation-strings interspersed in the bottom of the array. Thus a crossover search is forced to work at the level of the one-point crossover SSS.

However, if elitism is not used, then the results are very different. See Table 7.2 (bottom). For string lengths of 20, crossover finds the crossover half of the domain easier, while for strings of length 40 crossover finds the mutation half easier.

We offer a possible explanation for these results. With both elitism and non-

elitism, highly fit strings tend to be shuffled towards the bottom of the array. These high-fit strings include strings containing mostly ones (mutation strings) and strings where the number of ones and zeroes are roughly equal (crossover strings). This means there will be fewer zeroes in the bottom of the array. Generally, crossing highly fit mutation and crossover strings together will tend to produce strings with lower fitness, and so elitism ensures that there is little exchange between these strings. With non-elitism, however, these strings can be crossed, and this is what *can* make a non-elitist search better on a mutation string. To see this, consider a sub-string of a high-fit mutation string, say 111101111. If it is crossed just before the zero and just after, then there is a probability of about 3/4 that those two crossovers will appear as a mutation (if we assume that fit mutation and crossover strings are equally likely). With a highly fit crossover string requiring a zero in a certain position, there is only a probability of about 1/4 that two adjacent crossovers will produce a zero at that position. There can also be a shortage of zeroes in certain character positions near the bottom of the array, which can add further difficulty for finding crossover optima.

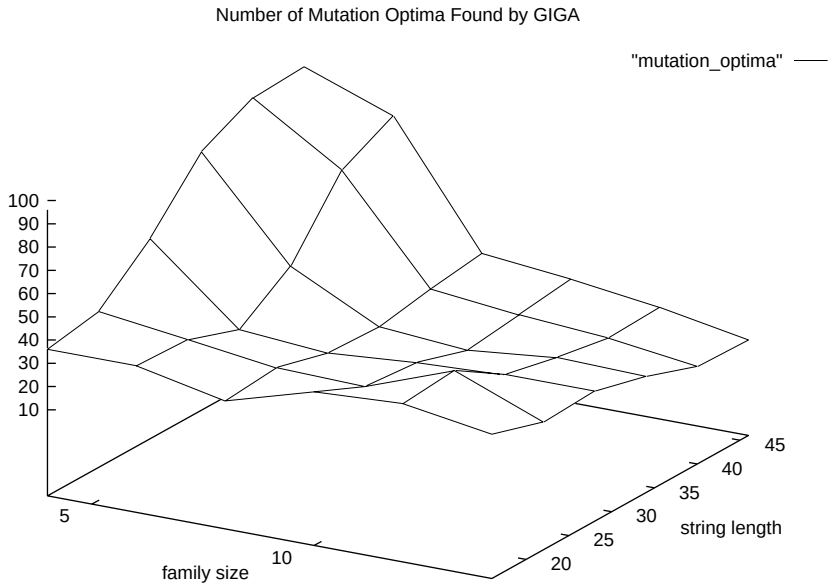
This explanation also seems to be supported by tests with GIGA where we varied family size and string length (see Figure 7.1). Notice that GIGA finds the most mutation optima when $\mathcal{F}$ is small relative to ℓ. Increasing the family size tends to make GIGA find the crossover optima more frequently, because increasing the family size mimics elitism, and the larger the family, the better elitism is mimicked. This mimicry is imperfect: if no improvement can be made, e.g., a false optimum, then a pair of children with value lower than their parents must be chosen. This is important when two highly-fit crossover and mutation strings are being crossed, since the larger the family, the higher the probability that a child pair will have been crossed at a crossover point near ℓ. This happens because crossing two such strings near the middle or front will produce strings with much lower fitness than strings crossed near the end.

Non-Minimal Populations: TGA and NQ-GIGA

We use GAC [47] as our representative TGA. It uses proportional fitness to select which individuals reproduce. We used one-point crossover at varying crossover rates and uniform mutation at varying mutation rates. We modified GAC to record which optima type was found first as well as the average number of evaluations to find the first optima. Each experiment ran for a maximum of $10,000$ evaluations. See Figure 7.2. The results clearly show that GAC finds the mutation optima much more frequently than the crossover optima, regardless of the crossover and mutation rate.

Consider the plot of the number of mutation optima found. Mutation dominates the search, since as the mutation rate increases, the number of mutation optima found decreases. Changing crossover rates has little consistent effect on this plot other than making the search slightly easier for high mutation rates. Thus mutation dominates the search of the optima that are, by far, found most frequently.

Finally, we wanted to see if GIGA could be modified to use both crossover and mutation effectively. We did this by encoding two extra parameters into GIGA: a mutation and a crossover rate. We refer to this modified GIGA as NQ-GIGA (Not



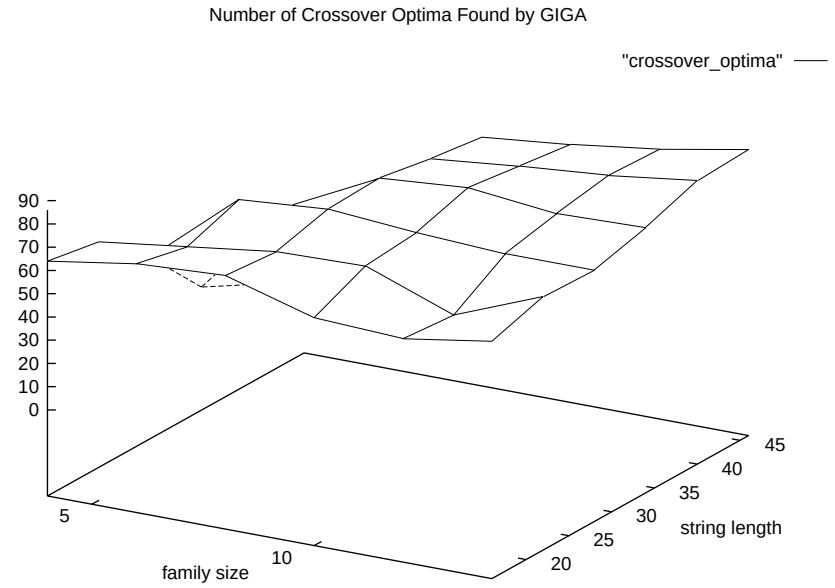


Figure 7.1: GIGA on bSchizo(x): varying $\mathcal{F}$ & ℓ ($n = 20, \alpha = 2$, 100 trials, non-elitism).

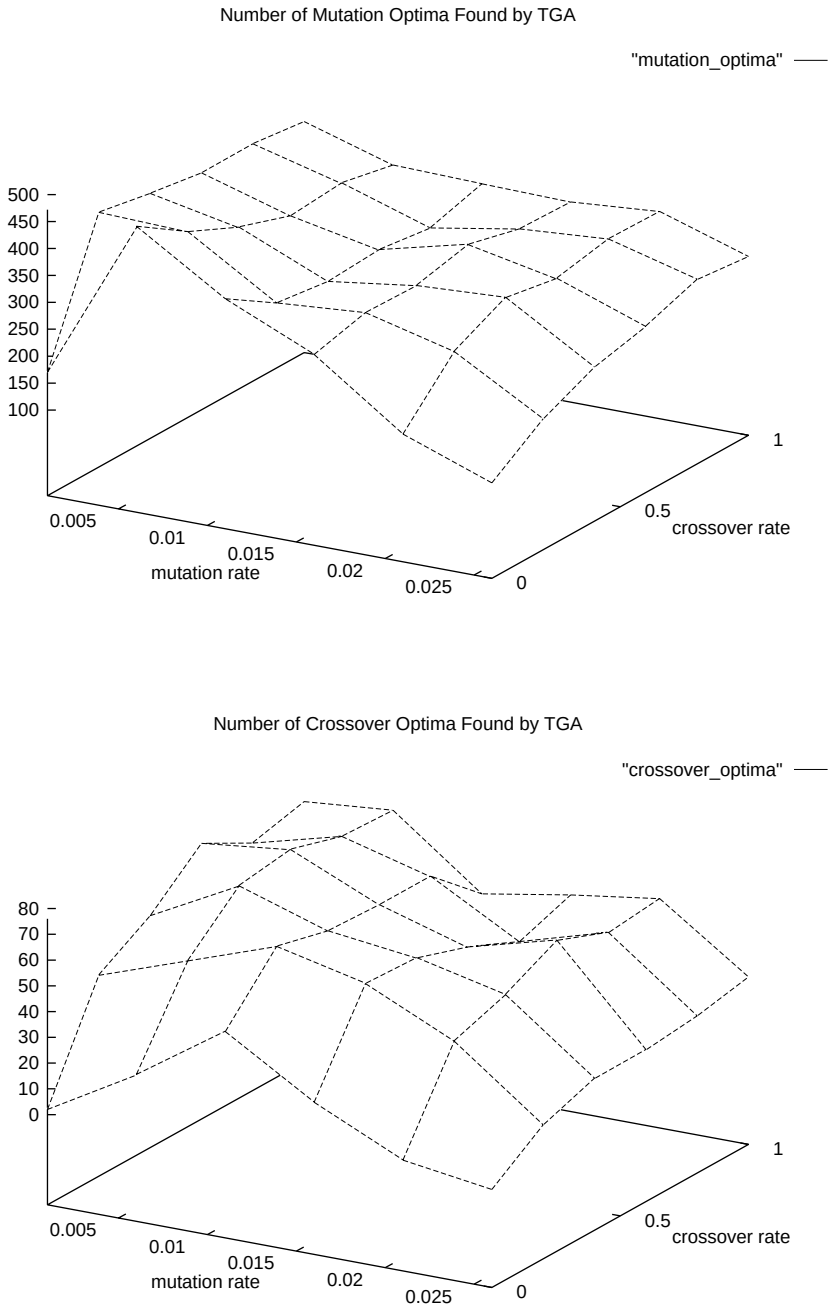


Figure 7.2: GAC($n = 50, \alpha = 2, \ell = 30$, 500 trials) on bSchizo(x): varying mutation and crossover rates.

Quite GIGA). NQ-GIGA uses the same defaults as GIGA. The maximum number of evaluations is $10,000$, as with GAC. See Figure 7.3.

These plots show that increasing the mutation rate increases the number of mutation optima found, while increasing the crossover rate increases the number of crossover optima found. This provides further evidence that (with elitism, at least) the schizophrenic function can discriminate between mutation and crossover, even when used together.

7.2 A Generalized Schizophrenic Function

In this section we use the generalized isomorphism $\mathcal{J}$ to construct a schizophrenic function for one-point mutation and one-point crossover on α-ary strings, $\alpha \geq 2$.

7.2.1 The Summation Function

We will construct our schizophrenic function based on the summation function, $f_\Sigma(x)$, where

$$f_\Sigma(x) = \sum_{i=1}^{\ell} x_i$$

This function is easy for one-point mutation for the same reason that ones-counting is easy for one-point binary mutation. When $\alpha = 2$, $f_\Sigma(x)$ is just the ones-counting function.

The summation function for one-point rotational crossover will have exponential false optima by a construction similar to that used in [8]. For example, the strings

55455
00500
11011
22122
33233

represent a false optimum, since any rotational crossover will produce a set of strings whose maximum value is lower than before the crossover. While rotational crossover may be of interest mathematically, there is little incentive in constructing a schizophrenic function for it since it is never used in the GA community. Instead, we wish to construct a schizophrenic function that can discriminate between one-point mutation and one-point crossover between two α-ary strings. This should be possible by noting that the summation function has exponential false optima for one-point crossover. For example, the following strings represent a false optimum

22122
11011

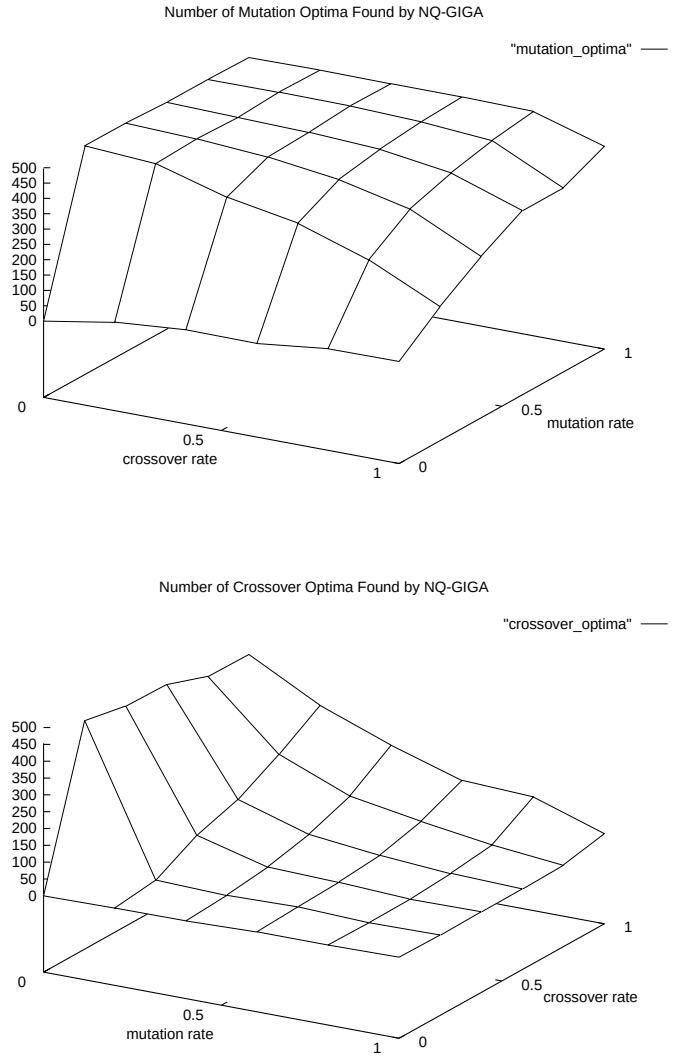


Figure 7.3: NQ-GIGA($n = 30$, $\ell = 30$, $\alpha = 2$, 500 trials, $\mathcal{F} = 4$, elitism) on bSchizo(x): varying mutation and crossover rates.

since their maximum value will decrease after a crossover.

To construct the generalized schizophrenic function, we use $\mathcal{J}$ to transform the summation function into a function that is easy for rotational crossover. That is, if the population is $a, < a >_1, \cdots, < a >_{\alpha-1}$, then the fitness of a population is given by $f_\Sigma(\mathcal{J}^{-1}(a, < a >_1, \cdots, < a >_{\alpha-1}))$. The fitness of the population can also be given by $f_s(< a >_i) = \sum_{j=2}^{\ell} a_j \ominus a_{j-1}$ for any i. Using $f_s()$, the *shift* function, with one-point α-ary crossover will hopefully generate a landscape easy for crossover. Inspection shows that the shift function has large plateaus on the one-point α-ary mutation SSS.

7.2.2 A Generalized Schizophrenic Function

The generalized schizophrenic function can be constructed in much the same way as the binary schizophrenic function was created. We just split the domain into two halves, and then apply a variant of the summation function to one of the halves and the transformed summation function, $f_s()$, to the other half. Our new definition becomes:

$$\text{Schizo}(x, \alpha) = \begin{cases} \sum_{i=3}^{\ell} x_i \ominus x_{i-1}, \text{ if } 0 \leq x_1 < \lfloor \frac{\alpha}{2} \rfloor \\ \sum_{i=3}^{\ell} x_i, \text{ if } \lfloor \frac{\alpha}{2} \rfloor \leq x_1 < 2\lfloor \frac{\alpha}{2} \rfloor \\ 0, \text{ otherwise} \end{cases}$$

This generalized definition is compatible with the binary definition.

When α is odd, the domain cannot be split into two equal halves, since x_1 can range from 0 to $\alpha - 1$. To get around this (when α is odd) we give any string with $x_1 = \alpha - 1$ a value of zero, and the rest of the domain can then be split evenly.

7.2.3 Testing the Generalized Schizophrenic Function

To test the generalized schizophrenic function, we chose $\alpha = 4$. We used GIGA and our MHC to do the tests. The MHC used a single 4-ary string, and a thousand experiments were done for each case. The results are listed in Table 7.3 (top), and seem very comparable to the binary MHC's results.

In our tests with GIGA, we had to ensure that every possible character could occur in every possible string location. To do this, we used at least 4 strings in GIGA's population. We only did 500 experiments for each case. The results are included in Table 7.3 (bottom). GIGA found a crossover optima much more frequently than a mutation optima. However, the results are not similar (i.e., found an optima less often) to the binary schizophrenic case. This is likely because we were not using rotational crossover, and thus we cannot expect one-point crossover to find the shift function as easy as one-point rotational crossover would find it.

We did tests with GIGA on populations of size 20, with and without elitism. See Table 7.4. The results are very comparable to those that we did with the binary schizophrenic function on bigger populations.

We also tested the generalized schizophrenic function with NQ-GIGA while varying the mutation and crossover rates. The maximum number of evaluations was

MHC: ($n = 1, \alpha = 4$, 1000 trials, $\mathcal{F}$=4, non-elitism)

MAX EVALS	ℓ	# mutation optima	# crossover optima	no optima
40000	12	929	71	0
60000	16	977	22	1
80000	20	951	7	42
100000	30	31	0	969

GIGA: ($n = 4, \alpha = 4$, 500 trials, $\mathcal{F}$=4, non-elitism)

MAX EVALS	ℓ	# mutation optima	# crossover optima	no optima
40000	12	35	426	39
60000	16	1	97	402
80000	20	0	8	492

Table 7.3: Testing the α-ary schizophrenic function on minimal populations with $\alpha = 4$: MHC (top), and GIGA (bottom).

GIGA: ($n = 20, \alpha = 4$, 500 trials, $\mathcal{F}$=4, elitism)

MAX EVALS	ℓ	# mutation optima	# crossover optima	no optima
40000	12	109	384	7
60000	16	65	405	30
80000	20	17	381	102

GIGA: ($n = 20, \alpha = 4$, 500 trials, $\mathcal{F}$=4, non-elitism)

MAX EVALS	ℓ	# mutation optima	# crossover optima	no optima
40000	12	363	137	0
60000	16	471	26	3
80000	20	459	1	40

Table 7.4: GIGA on the schizophrenic function: non-minimal populations, elitism (top), and non-elitism (bottom).

10,000. See Figure 7.4 for graphs of number of mutation optima and crossover optima found. Again, the results are comparable to those in the binary case.

7.3 Conclusion

The schizophrenic function appears able to discriminate between mutation and crossover in certain situations (elitism) but not in others (non-elitism with small family size). This is probably because the schizophrenic function is biased so that more ones than zeroes are concentrated towards the bottom of the population array, and that non-elitism can take advantage of this. We do not expect that this (or any) schizophrenic function can always discriminate between mutation and crossover based searches: for example, a crossover-based search that separated the mutation and crossover strings into two independent population pools would easily find mutation optima if it used a GIGA-like mating scheme. However, it may work on many different algorithms. We have also shown that the α-ary schizophrenic function induces landscapes with behaviours similar to its binary counterpart.

7.3.1 Future Work

It should be possible to make a better schizophrenic function, one that avoids the problem of one-point crossover with non-elitism being able to simulate one-point mutation, by noting that the majority function, $f_m(x) = |(\#1\text{s in } x) - (\#0\text{s in } x)|$, has many false optima in the one-point crossover SSS. It was also empirically shown to be hard for one-point crossover with non-minimal populations [6], and is also easy for the one-point mutation SSS. Since mutation strings now consist of strings with either mostly zeroes or mostly ones, the bias of having a concentrated number of ones near the bottom of the array would be eliminated if the schizophrenic function replaced its summation function with the majority function. Whether this is in fact an improvement needs to be tested. It should also be easy to generalize this new schizophrenic function to α-ary strings.

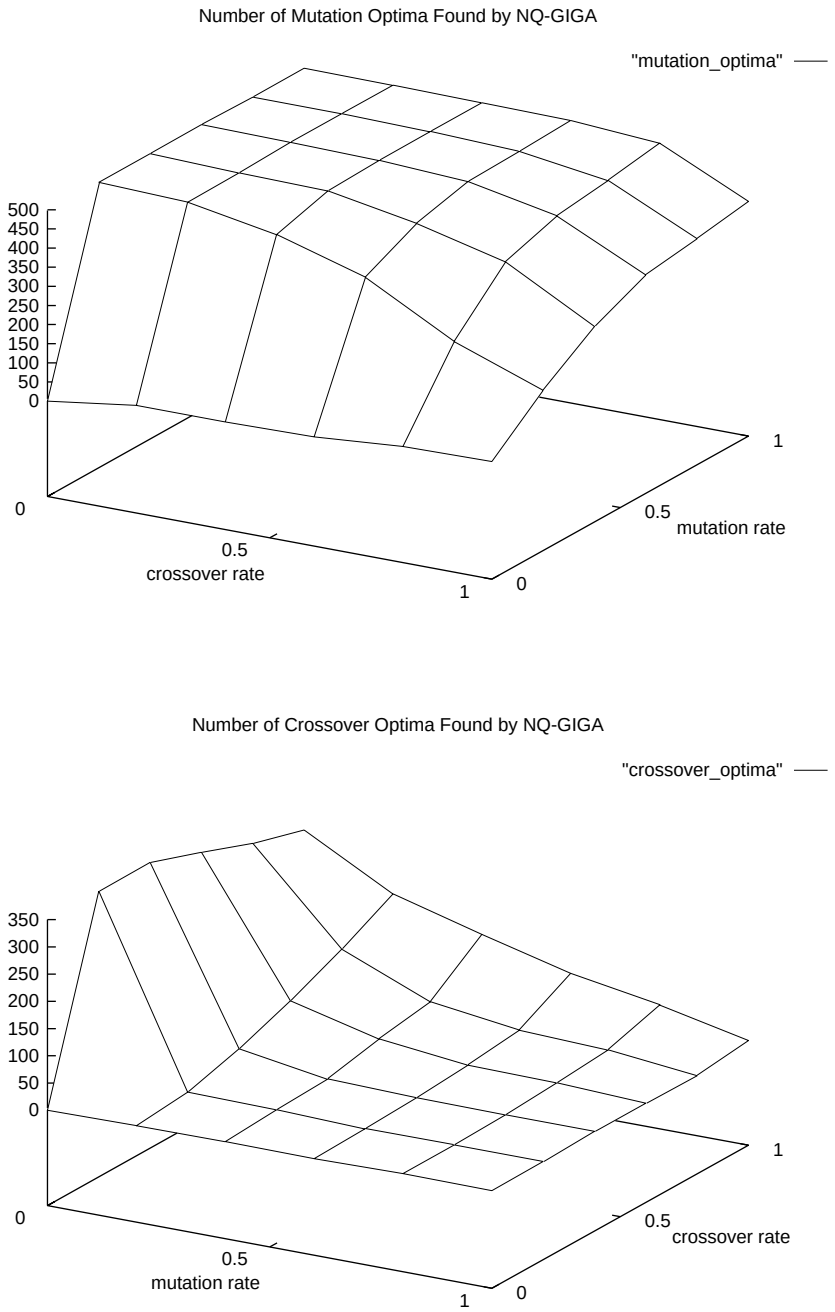


Figure 7.4: NQ-GIGA($n = 30, \ell = 16, \alpha = 4$, 500 trials/point, $\mathcal{F} = 4$, elitism) on schizophrenic function: varying mutation and crossover rates.

Chapter 8

Related Work

In this chapter we give a brief survey on landscape theory, analysis, and construction. We do not give a review of Gray codes, but reading Section 5.1 followed by Appendix A will suffice as one.

8.1 Landscapes Definitions That Include Crossover

Many papers in the GA literature use the concept of fitness landscape, but often these landscapes are the one-point mutation SSS [33], and so do not represent the search of genetic algorithms. For instance, mutation is often viewed as a local operator that moves on this landscape, while crossover warps *through* this landscape. In other words, the mutation SSS has nothing to say about crossover.

Only recently have models of landscapes been proposed that include crossover. Culberson proposes a rigorous landscape model [6, 8] that can explicitly model the search of GAs, where each population is a vertex in the graph and the *algorithm* via its operators defines edges in the landscape. He also notes that these landscapes will be too complex in general, and so simplifications may need to be made. For instance, he analyzes both the binary mutation SSSs, and crossover landscapes for a complementary pair of binary strings.

Jones [33] also introduces a model of landscapes that includes crossover. He defines a landscape $\mathcal{L}$ as the five-tuple $(\mathcal{R}, \phi, f, \mathcal{F}, >_{\mathcal{F}})$, where $\mathcal{R}$ is the representation, ϕ is an operator (possibly a composition of operators), f is the fitness function $f : \textbf{multisetof}(\mathcal{R}) \rightarrow \mathcal{F}$, and $\mathcal{F}$ is a set and $>_{\mathcal{F}}$ is a partial order over $\mathcal{F}$. Jones also offers a simplified view of the search of (traditional) GAs: a GA moves on several landscapes (a mutation landscape, a crossover landscape, and a selection landscape).

Jones's and Culberson's landscapes are nearly equivalent in that in both can be interpreted as representing each population as a vertex and operators induce edges in the graph. Jones includes edge probabilities in his definition, however.

A variant landscape based on hypergraphs has been introduced by Gitchoff and Wagner [21]. Rather than having each vertex corresponding to a population, each vertex is a single string, and edges are generalized edges (e.g., a generalized edge between two strings a and b consists of the set of all strings that can be formed by

crossover between a and b). Gitchoff and Wagner show that the strings in any generalized edge form a cycle when strings with a Hamming distance of one are connected, and are "*also connected in the corresponding hypercube* of the point mutation space," and suggest that because of this recombination spaces have the same metric (Hamming distance) as mutation SSSs. However, in the one-point mutation SSS, Hamming distance defines distance between two strings, while the Hamming distance for recombination spaces (as used by Gitchoff and Wagner) are just abstract quantities, with no relation to distance in the landscape. Regardless, hypergraphs are an interesting abstraction that may be of interest.

In this thesis we use landscapes where each population is a vertex, and operators induce edges in the graph, because this approach seems more intuitive. It can also be rigorous, in that the search can be modelled completely.

Simplifications are often required when studying landscapes, because GAs are very complex algorithms that induce very complex landscapes. However, simplified landscapes are not the landscapes searched by GAs, but only abstractions. These landscapes may relate to the search of GAs, but any analysis of simplified landscapes should be supported empirically. Sometimes simplified landscapes are not enough. For example, Culberson [8] notes that ones-counting is hard (exponentially many false optima) for crossover on the one-point crossover SSS. However, ones-counting is not hard on larger populations (for GIGA). In this case, a population-based landscape analysis that takes into account how GIGA works is required.

Even if the simplified landscape is adequate, it will still be beneficial to maintain an overlying view of the complete landscape as searched by the GA. Otherwise, key concepts may be missed or misunderstood.

8.2 Various Methods Used in Analyzing Landscapes

There are several ways a particular landscape or class of landscapes can be studied. One is to measure statistics based on a landscape and the other is to analyze a landscape theoretically.

Two main types of statistical measures have been used to analyze GA-induced landscapes: 1. fitness correlation of genetic operators, and 2. random walk correlations. The first attempts to measure the correlation between the fitness of parents and their children, the assumption being that if high-fit parents produce high-fit children, then GAs will perform well. This is explored by Manderick, *et al.* [37], Mathias and Whitley [38], and Dzubera and Whitley [12] who measure the correlation between randomly chosen parents. Grefenstette [25] calls this a *static* measure and notes that high-valued parents may not be sampled, and that the sample may not reflect the populations seen by a GA. To remedy this, he introduces a *dynamic* measure as well; that is, he measures the correlation between parents and children, where the parents have actually been created by the GA.

Another statistic is based on a random walk: do a random walk, and compute the correlation between the fitness at time t and time $t + i$. This can be used to give an estimate of how "rugged" (how many false optima are in the landscape) a

landscape is. This approach has been taken by Manderick, *et al.* [37], Hordijk [28], and Stadler [48]. Manderick, *et al.*, however, use random mutations which means they use the landscape of a hill climber, but apply this analysis to GAs. This is problematic. Crossover can do random walks if two parents produce two children rather than a single child, so random walk analysis is not limited to mutation SSSs. However, there are smooth landscapes that are hard [29], and rugged landscapes that are easy [31]. These statistical measures may be useful in analysis, but they should be used cautiously. A further difficulty with these approaches is that it is possible for the same landscape to vary in difficulty (exponentially) with different navigation strategies [10].

Another interesting statistical approach has been taken by Jones [35], where he measures the correlation between the fitness of a string and the distance (Hamming) of that string to the optimal. (Jones notes that Hamming distance is not the real distance metric of GAs, but is a first approximation.) He found that landscapes with a strong negative correlation tend to be easy for GAs, while ones with a strong positive correlation tend to be hard. It is interesting that Hamming distance, the distance for mutation, proved such a good measure of problem difficulty for GAs; this further supports the claims of Culberson [8] that traditional GAs do not use crossover well but rely on mutation for search.

Jones [32] offers another approach to analyzing problems called "reverse hill climbing." In reverse hill climbing, the basin of attraction (hill) around a point for a strictly elitist (in practical situations) hill climber can be traversed and measured. He gives a procedure that computes the size of this hill and the probability that the hill is reached from any point. This can be a powerful tool to analyze problem difficulty, once a global (or local) optima is found.

A theoretical approach can also be used to study landscapes. That is, it may be possible to derive bounds on a particular landscape, or prove some expected level of performance. In "On Searching α-ary Hypercubes" [10] certain characteristics of hypercubes are explored both theoretically and experimentally. For example, we show that the α-ary hypercube has a maximum of n/α peaks. Stadler [48] relates points of graph theory to landscapes. Chapters 2-5 and Appendix A of this thesis are theoretical analyses of landscapes (Appendix A explores Gray codes, but this can be related to Hamiltonian paths and cycles in multary-based graphs).

8.3 Discriminating Functions, and Interesting Landscape Constructions

Another approach to analyzing GAs is landscape construction. For instance, there are many functions that are designed to test certain ideas about how genetic operators work. These *discriminating functions* may be easy, in theory, for one genetic operator while hard for another. A landscape construction can also be used to exhibit examples of easy or difficult problems, or to demonstrate counter-intuitive behaviour.

Many of these discriminating functions have been discussed elsewhere in this the-

sis, so we keep this overview short. Culberson [8] uses a binary mutation-crossover isomorphism to convert hard functions for mutation into hard functions (on a complementary pair) for crossover, and vice versa. The schizophrenic function is related to this work, and can be seen as an attempt at constructing a discriminating function for mutation and crossover.

The Royal Road functions [18, 19, 40] were designed to be easy for crossover but hard for mutation, although a mutation hill climber outperformed a traditional genetic algorithm [19]. Interestingly, an idealized genetic algorithm was later proven to be faster than the hill climber by a linear factor, on average. This also demonstrated the point that an operator may have a certain potential, but that the algorithm may not use the potential of an operator. The Royal Roads are also an example of high-level crossover landscape analysis in GAs.

Horn, *et al.* [29, 30, 31] introduces the long path landscapes, unimodal but exponentially difficult landscapes for elitist mutation hill climbers. These functions discriminate between mutation and crossover, since mutation takes exponential time to find the optimal point, while crossover (with large populations) has much less trouble. These landscapes are discussed further in Chapter 6, in which we discuss long path landscapes for α-ary mutation and crossover. Long paths have also been used to discriminate between different types of navigation strategy [10].

Another interesting landscape is the maximally multimodal landscapes of Horn and Goldberg [31]. These landscapes are constructed on the mutation SSS and have the maximum number of false optima ($2^{\ell-1}$), but are easy for many non-elitist hill climbers. This shows that while many false optima may make a landscape difficult, this is not always the case.

Chapter 9

Conclusion

In this thesis we studied several α-ary landscapes. We used the landscape paradigm because it can be rigorous, yet is easy to simplify to a manageable level of complexity.

We looked at several simplified landscapes (search space structures) induced by various forms of α-ary mutation and α-ary crossover on minimal populations. These search space structures are related to α-ary hypercubes.

Using an isomorphism between α-ary crossover and α-ary mutation, we showed that crossover is at least as powerful as mutation, since one-point crossover can simulate mutation. This simulation also suggests that crossover is more powerful than mutation, in some sense, in that it is more general (larger neighbourhoods). This addresses the crossover-mutation debate. A special case of this simulation leads to the notion of iterating an α-ary Gray code. In Chapter 5 we prove an upper bound on the maximum number of unique landscapes produced when a landscape is iterated using this Gray code.

We explored the long path problem in Chapter 6 for crossover and α-ary mutation. We developed exponentially long distance-preserving paths for α-ary mutation and crossover on a complementary pair. Some work was done on extending these long paths to crossover on larger populations, and the results are interesting if preliminary.

We also introduced the schizophrenic function, a function that can sometimes discriminate between mutation and crossover based searches. Analyzing the test results on the schizophrenic function proved instructive in understanding some of the complex ways in which population-based searches can differ from minimal population searches.

Finally, in Appendix A, we gave an algorithm for generating the α-ary Gray code of Sharma and Khanna [43] in constant amortized time per code word. This Gray code is generalized to a multary Gray code, and a simple generalization of our algorithm generates this multary code in constant amortized time per code word.

Bibliography

[1] Barasch, Linda S., S. Lakshmivarahan, and Sudarshan K. Dhall. (1989). "Generalized Gray Codes and Their Properties." In *Mathematics for Large Scale Computing, lecture notes in Pure and Applied Mathematics,* J.C. Diaz (editor), vol. 120, pp. 203-216. New York: Marcel Dekker.

[2] Bitner, James R., Gideon Ehrlich, and Edward M. Reingold. (1976). "Efficient Generation of the Binary Reflected Gray Code and its Applications." *Comm. ACM,* vol. 19, no. 9, pp. 517-521.

[3] Bloch, Norman J. (1987). "Abstract Algebra with Applications." Englewood Cliffs, New Jersey: Prentice-Hall.

[4] Brassard, Gilles, and Paul Bratley. (1988). "Algorithmics: Theory & Practice." Englewood Cliffs, New Jersey: Prentice-Hall.

[5] Culberson, Joseph C. (1992). "GIGA program description and operation." Technical Report TR 92-06, University of Alberta Department of Computing Science. Available by anonymous ftp. Site: ftp ftp.cs.ualberta.ca. Directory: pub/TechReports/1992/TR92-06.

[6] Culberson, Joseph C. (1992). "Genetic invariance: a new paradigm for genetic algorithm design." Technical Report TR 92-02, University of Alberta Department of Computing Science. Available by anonymous ftp. Site: ftp ftp.cs.ualberta.ca. Directory: pub/TechReports/1992/TR92-02.

[7] Culberson, Joseph C. (1993). "Holland's Royal Road and GIGA." In *Genetic Algorithms Digest.* Volume 7 : Issue 23

[8] Culberson, Joseph C. (1995). "Mutation-Crossover Isomorphisms and the Construction of Discriminating Functions." *Evolutionary Computation*, vol. 2, no. 3, pp. 279-311.

[9] Culberson, Joseph C. (1996). "On the futility of blind search." Technical Report TR96-18, University of Alberta Department of Computing Science. Available by anonymous ftp. Site: ftp ftp.cs.ualberta.ca. Directory: pub/TechReports/1996/TR96-18.

[10] Culberson, Joseph C., and Jonathan Lichtner. "On Searching α-ary Hypercubes and Related Graphs." In *Foundations of Genetic Algorithms 4,* Richard K. Belew and Michael Vose (editors), to appear.

[11] Davis, Lawrence. (1991). "Bit-Climbing, Representational Bias, and Test Suite Design." In *Proceedings of the Fourth International Conference on Genetic Algorithms*, R.K. Belew and L.B. Booker (editors), pp. 18-23. San Mateo, CA: Morgan Kaufmann.

[12] Dzubera, John, and Darrell Whitley. (1994). "Advanced Correlation Analysis of Operators for the Travelling Salesman Problem." In *Parallel Problem Solving from Nature–PPSN III, International Conf. on Evolutionary Computation, Proceedings.* volume 866 of Lecture Notes in Computer Science, Y. Davidor, H.P. Schwefel, and R. Männer, editors, pp. 68-77. Berlin, Germany: Springer-Verlag.

[13] Eshelman, Larry J. (1991). "The CHC Adaptive Search Algorithm: How to Have Safe Search When Engaging in Nontraditional Genetic Recombination." In *Foundations of Genetic Algorithms*, Gregory J.E. Rawlins, editor, pp. 265-283. San Mateo, CA: Morgan Kaufmann.

[14] Eshelman, Larry J., and J. David Schaffer. (1993). "Crossover's Niche." In *Proceedings of the Fifth International Conference on Genetic Algorithms*, S. Forrest, editor, pp. 9-14. San Mateo, CA: Morgan Kaufmann.

[15] Field, Paul. (1997). "A Multary Theory for Genetic Algorithms: Unifying Binary and Nonbinary Problem Representations" Ph.D. Thesis, University of London, London.

[16] Floyd, Robert W., and Richard Beigel. (1994). "The Language of Machines: An Introduction to Computability and Formal Languages." New York: Computer Science Press.

[17] Fogel, D.B., and J.W. Atmar. (1990). "Comparing Genetic Operators with Gaussian Mutations in Simulated Evolutionary Processes Using Linear Systems." *Biological Cybernetics*, vol. 63, pp. 111-114.

[18] Forrest, Stephanie, John H. Holland, and Melanie Mitchell. (1991). "The Royal Road for Genetic Algorithms: Fitness Landscapes and GA Performance." In *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, F.J. Varela and P. Bourgine, editors, pp. 245-254. Cambridge, MA: MIT Press.

[19] Forrest, Stephanie, and Melanie Mitchell. (1992). "Relative Building-Block Fitness and the Building Block Hypothesis." In *Foundations of Genetic Algorithms 2*, L. Darrell Whitley, editor, pp. 109-126. San Mateo, CA: Morgan Kaufmann.

[20] Gilbert, E.N. (1958). "Gray Codes and Paths on the n-Cube." *Bell Sys. Tech. Journal*, vol. 37.

[21] Gitchoff, P., and G.P. Wagner. (1996). "Recombination Induced Hypergraphs: A New Approach to Mutation- Recombination Isomorphism." *Complexity* (In Press). Center for Computational Ecology preprint 33, Yale, http://peaplant.biology.yale.edu:8001/.

[22] Goldberg, D. E. (1989). "Genetic Algorithms in Search, Optimization and Machine Learning." Reading, MA: Addison Wesley.

[23] Gray, Frank. (1953). *Pulse code communications*, U.S. Patent 2632058.

[24] Grefenstette, John J. (1992). "Deception Considered Harmful." In *Foundations of Genetic Algorithms 2*, L. Darrell Whitley, editor, pp. 75-92. San Mateo, CA: Morgan Kaufmann.

[25] Grefenstette, John J. "Predictive Models Using Fitness Distributions of Genetic Algorithms." In *Foundations of Genetic Algorithms 3*, L. Darrell Whitley and Michael D. Vose, editors, pp. 139-162. San Mateo, CA: Morgan Kaufmann, 1995.

[26] Harary, Frank, John P. Hayes, and Horng-Jyh Wu. (1988). "A survey of the theory of hypercube graphs." *Computers and Mathematics with Applications,* vol. 15, no. 4, pp. 277-289.

[27] Holland, J. (1975). "Adaption in Natural and Artificial Systems." Ann Arbor, Michigan: University of Michigan Press.

[28] Hordijk, Wim. (1995). "A Measure of Landscapes." Technical Report SFI-TR-95-05-049, The Santa Fe Institute, Santa Fe, New Mexico.

[29] Horn, Jeffrey, David E. Goldberg, and Kalyanmoy Deb. (1994). "Long Path Problems." *Parallel Problem Solving from Nature–PPSN III, International Conf. on Evolutionary Computation, Proceedings.* volume 866 of Lecture Notes in Computer Science, Y. Davidor, H.P. Schwefel, and R. Männer, editors, pp. 149-158. Berlin, Germany: Springer-Verlag.

[30] Horn, Jeffrey. (1995). "Genetic algorithms, problem difficulty and the modality of the fitness landscapes." Master's thesis, University of Illinois, Urbana-Champaign.

[31] Horn, Jeffrey, and David E. Goldberg. (1995). "Genetic algorithm difficulty and the modality of fitness landscapes." In *Foundations of Genetic Algorithms 3*, L. Darrell Whitley and Michael D. Vose, editors, pp. 243-269. San Mateo, CA: Morgan Kaufmann.

[32] Jones, Terry, and Gregory J.E. Rawlins. (1993). "Reverse Hillclimbing, Genetic Algorithms and the Busy Beaver Problem." In *Proceedings of the Fifth International Conference on Genetic Algorithms*, S. Forrest, editor, pp. 70-75. San Mateo, CA: Morgan Kaufmann.

[33] Jones, Terry. (1995). *Evolutionary Algorithms, Fitness Landscapes and Search.* Ph.D. thesis, University of New Mexico, Albuquerque, NM.

[34] Jones, Terry. (1995). "Crossover, Macromutation, and Population-based Search." In *Proceedings of the Sixth International Conference on Genetic Algorithms*, L.J. Eshelman, editor, pp. 73-80. San Mateo, CA: Morgan Kaufmann.

[35] Jones, Terry, and Stephanie Forrest. (1995). "Fitness Distance Correlation as a Measure of Problem Difficulty for Genetic Algorithms." In *Proceedings of the Sixth International Conference on Genetic Algorithms*, L.J. Eshelman, editor, pp. 184-192. San Mateo, CA: Morgan Kaufmann.

[36] Kernighan, Brian W., and Dennis M. Ritchie. (1988). "The C Programming Language," 2nd edition. Engelwood Cliffs, New Jersey: Prentice Hall.

[37] Manderick, Bernard, Mark de Weger, and Piet Spiessens. (1991). "The Genetic Algorithm and the Structure of the Fitness Landscape." In *Proceedings of the Fourth International Conference on Genetic Algorithms*, R.K. Belew and L.B. Booker, editors, pp. 143-150. San Mateo, CA: Morgan Kaufmann.

[38] Mathias, Keith, and Darrell Whitley. (1992). "Genetic operators, the fitness landscape and the travelling salesman problem." In *Parallel Problem Solving From Nature, 2*, R. Männer and B. Manderick, editors, pp. 219-228. Elsevier Science Publishers B.V.

[39] Mitchell, Melanie. (1996). "An Introduction to Genetic Algorithms." Cambridge, Massachusetts: The MIT Press.

[40] Mitchell, Melanie, and John H. Holland. "When Will a Genetic Algorithm Outperform Hill Climbing." To appear in *Advances in Neural Information Processing Systems 6*, J.D. Cowan, G. Tesauro, and J. Alspector, editors. Morgan Kaufmann.

[41] Mühlenbein, Heinz. (1992). "How genetic algorithms really work I: Mutation and Hillclimbing." In *Parallel Problem Solving From Nature, 2*, R. Männer and B. Manderick, editors, pp. 15-25. Elsevier Science Publishers B.V.

[42] Rawlins, G.J.E. (1991). Introduction. In *Foundations of Genetic Algorithms*, Gregory J.E. Rawlins, editor, pp. 265-283. San Mateo, CA: Morgan Kaufmann.

[43] Sharma, Bhu Dev, and Ravinder Kumar Khanna. (1978). "On m-ary Gray codes." *Information Sciences,* vol. 15, no. 1, pp. 31-43.

[44] Sharma, Bhu Dev, and Ravinder Kumar Khanna. (1979). "Integer Characterization binary and m-ary Gray codes." *Journal of Combinatorics, Information and System Sciences,* vol. 4, no. 3, pp. 227-236.

[45] Reingold, Edward M., Jurg Nievergelt, and Narsingh Deo. (1977). *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall.

[46] Spears, William M. (1992). "Crossover or mutation." In *Foundations of Genetic Algorithms 2*, L. Darrell Whitley, editor, pp. 221-237. San Mateo, CA: Morgan Kaufmann.

[47] Spears, William M. GAC. C source code is available from the "Genetic Algorithms Archive" at http://www.aic.nrl.navy.mil:80/galist/

[48] Stadler, Peter F. (1995). "Towards a Theory of Landscapes." Technical Report SFI-TR-95-03-030, The Santa Fe Institute, Santa Fe, New Mexico.

[49] Syswerda, Gilbert. (1989). "Uniform Crossover in Genetic Algorithms." In *Proceedings of the Third International Conference on Genetic Algorithms*, J.D. Schaffer, editor, pp. 2-9. San Mateo, CA: Morgan Kaufmann.

[50] Whitley, D. (1989). "The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best." In *Proceedings of the Third International Conference on Genetic Algorithms*, J.D. Schaffer, editor, pp. 116-121. San Mateo, CA: Morgan Kaufmann.

[51] Williamson, S. Gill. (1985). "Combinatorics for Computer Science." Rockville, Maryland: Computer Science Press.

[52] Wolpert, D., and W. Macready. (1996). "No free lunch theorems for search." Technical Report SFI-TR-95-02-010, The Santa Fe Institute, Santa Fe, New Mexico. ftp://ftp.santafe.edu/pub/wgm/.

Appendix A

Proof of Gray Code, and an Efficient Algorithm to Generate it in Sequence

In this appendix we prove that $\mathcal{G}(\alpha, \ell)$ really is a Gray code. This proof will form the basis of an algorithm that can generate the Gray code in sequence in $\Theta(\alpha^\ell)$ time (amortized constant time per codeword). This code and algorithm will be further generalized in Section A.3. See Section 5.1 for a definition and overview of Gray codes.

Bitner, *et al.* [2] describe several uses of being able to list a Gray code efficiently. First, an ℓ-bit string can represent a set. Listing all the bit strings will list all possible sets. In certain codes (e.g., the base α, dimension ℓ integers) adjacent codewords may differ in as many as ℓ bits. If the cost of adding or deleting elements to the set is high, then we would prefer a listing that only changes one element at a time: a Gray code. Changing one element may also simplify algorithms that use these sets. Other uses they describe are in listing compositions of integers and listing all permutations of a multiset.

A.1 Proof of Gray Code

Recall that

$$\begin{aligned} \mathcal{K}(x) &= (g) \\ g_i &= \begin{cases} x_1 & \text{if } i = 1 \\ x_i \ominus x_{i-1} & 1 < i \leq \ell \end{cases} \end{aligned}$$

is an isomorphism between the natural numbers of length ℓ, base α and $\mathcal{G}(\alpha, \ell)$ [43].

First, we note that $\mathcal{K}$ is one-one and onto (this is easily proven). Then, to prove $\mathcal{G}(\alpha, \ell)$ is a cyclic Gray code, we need only show that $\mathcal{G}_x(\alpha, \ell)$ and $\mathcal{G}_{(x+1) \bmod \alpha^\ell}(\alpha, \ell)$ differ in only one character.

THEOREM **A.1.1** *$\mathcal{G}(\alpha, \ell), \alpha \geq 2, \ell \geq 1$, is a Gray code. Further, if $\mathcal{G}_x(\alpha, \ell)$ and $\mathcal{G}_{(x+1) \bmod \alpha^\ell}(\alpha, \ell)$ differ at character j, then $\mathcal{G}_{(x+1) \bmod \alpha^\ell, j}(\alpha, \ell) = \mathcal{G}_{x,j}(\alpha, \ell) \oplus 1$.*

Proof Outline: Take successive α-ary, length ℓ numbers x and x' where $x' = x + 1$. If $\mathcal{K}(x) = g$ and $\mathcal{K}(x') = g'$, then show that g and g' differ in only one character location, i.e., there exists an i such that $g_i \neq g'_i$ and for all $j \neq i$, $g_j = g'_j$. Also, it will be shown that $g'_i = g_i \oplus 1$.

Proof:
(Theorem A.1.1)

There are two cases:

1. $\exists$ an $i, 1 \leq i \leq \ell$, such that $x_i < \alpha - 1$ and $\forall j > i$, $x_j = \alpha - 1$
2. $\forall i, 1 \leq i \leq \ell$, $x_i = \alpha - 1$ (Cyclic case)

CASE 1: There $\exists i, 1 \leq i \leq \ell$, such that $x_i < \alpha - 1$ and $\forall j > i$, $x_j = \alpha - 1$. This implies that $x' = x_1 \ldots x_{i-1}(x_i + 1)0 \ldots 0$.

Let $\mathcal{K}(x) = g = g_1 g_2 \ldots g_\ell$. Simple induction shows that $g'_1 \ldots g'_{i-1} = g_1 \ldots g_{i-1}$, if $i \neq 1$.

If $i = 1$ then

$$\begin{aligned} g'_i &= x'_i \\ &= x_i \oplus 1 \\ &= g_i \oplus 1 \end{aligned}$$

else

$$\begin{aligned} g'_i &= x'_i \ominus x'_{i-1} \\ &= (x_i \oplus 1) \ominus x_{i-1} \\ &= (x_i \ominus x_{i-1}) \oplus 1 \\ &= g_i \oplus 1 \end{aligned}$$

Now show that $g'_j = g_j$, for $i + 1 \leq j \leq \ell$.

$$\begin{aligned} g'_j &= x'_j \ominus x'_{j-1} \\ &= x_j \oplus 1 \ominus x_{j-1} \ominus 1 \\ &= x_j \ominus x_{j-1} \\ &= g_j \end{aligned}$$

Hence, we have shown that g' only differs from g in one character location, and hence case 1.

CASE 2 (Cyclical): We know $x = (\alpha - 1) \ldots (\alpha - 1)$ and $x' = 0 \ldots 0$. Then $g_1 = \alpha - 1 \neq g'_1 = 0$ and for any j such that $2 \leq j \leq \ell$, $g_j = (\alpha - 1) \ominus (\alpha - 1) = 0$ and $g_i = 0 \ominus 0 = 0$. This implies that $g_i = g'_i$, and hence case 2.

Since $\mathcal{K}$ is one-one and onto, case 1 shows that $\mathcal{G}(\alpha, \ell)$ is a Gray code, and case 2 shows that it is cyclic. Further, this proof also shows that $\mathcal{K}(\mathcal{N}_i(\alpha, \ell)) = \mathcal{G}_i(\alpha, \ell)$.

■

A.2 The Transition Sequence $\mathcal{T}(\alpha, \ell)$

Any word of $\mathcal{G}(\alpha, \ell)$ can be generated by mapping $\mathcal{N}(\alpha, \ell)$ under $\mathcal{K}$, but this is less efficient than need be (takes $\Theta(\ell)$ time per codeword) if we want to compute each Gray code word in sequence. If this is the case, then a better way to generate the words of $\mathcal{G}(\alpha, \ell)$ is to figure out the transition sequence $\mathcal{T}(\alpha, \ell)$. The transition sequence $\mathcal{T}(\alpha, \ell)$ is just the ordered list of the positions of changing characters in $\mathcal{G}(\alpha, \ell)$. A subscript i will refer to the position of the character change between the $\mathcal{G}_{i-1}(\alpha, \ell)$ and $\mathcal{G}_i(\alpha, \ell)$. For example, $\mathcal{T}_1(\alpha, \ell)$ will be the position of the character that changes as we go from $\mathcal{G}_0(\alpha, \ell)$ to $\mathcal{G}_1(\alpha, \ell)$. However, instead of numbering the positions from left to right, as is done in the rest of this thesis, $\mathcal{T}_i(\alpha, \ell)$ will refer to the position of the changed character starting from the right and going to the left. As an example, the transition sequence $\mathcal{T}(2, 3)$ is (1,2,1,3,1,2,1). See Figure 5.1 for the Gray code given by this transition sequence (i.e., $\mathcal{G}(2, 3)$).

For $\mathcal{G}(2, \ell)$ (binary reflected Gray code) an efficient $\Theta(2^\ell)$ algorithm is given in [2] that will generate the each Gray code word in sequence. For some Gray codes with $\alpha \geq 3$ it is not always enough just to have a transition sequence; it must be determined not only which character changes, but what the character changes to. For example, given the general Gray code number 4988 ($\alpha = 10$, say) and that position 2 will change, we do not necessarily know what the 8 will change to. With $\mathcal{G}(\alpha, l)$ this is no problem; the character will change by $+1 \bmod \alpha$. In our example, if 4988 was a number in $\mathcal{G}(10, 4)$ and the next transition was position 2, then we know that the next character in $\mathcal{G}(10, 4)$ will be $49(8 \oplus 1)8 = 4998$.

It is possible to get the transition sequence $\mathcal{T}_\ell$ from $\mathcal{N}(\alpha, \ell)$ without fully processing each number in $\mathcal{N}(\alpha, \ell)$.

Lemma A.2.1 *$\mathcal{T}_i(\alpha, \ell)$ is the position of the first non-zero character (starting from the right) in $\mathcal{N}(\alpha, \ell)$.*

Proof:
By looking at the proof of Theorem A.1.1, we can see that the $\mathcal{T}_i(\alpha, \ell)$ is the position of the first non-zero character (starting from the right) in $\mathcal{N}_i(\alpha, \ell), 1 \leq i \leq \alpha^\ell - 1$.
■

We can use this lemma to make an efficient algorithm that generates the code words of $\mathcal{G}(\alpha, \ell)$ in sequence in $\Theta(\alpha^\ell)$. See Figure A.1 for this algorithm. This algorithm is efficient, since the inner while loop iterates in constant amortized time (although it can be as bad as $\Theta(\ell)$ time for producing any one codeword).

Lemma A.2.2 *Algorithm A.1 runs in time $\Theta(\alpha^\ell)$.*

Proof:
(Lemma A.2.2)

To show this, we need only show that the while loop takes constant time, on average. Let the integer $i, 0 < i < \alpha^\ell - 1$ represent the ith string in $\mathcal{G}(\alpha, \ell)$. Then

```
void GenerateGray(α, ℓ) {

        int g[1..ℓ]; /* holds Gray code word */
        int n[1..ℓ]; /* holds α-ary representation of naturals */
        int i;


        /* Initialize g[] and n[] */
        for (i = 1; i ≤ ℓ; i++) {
                g[i]=0;
                n[i]=0;
        }


        /*
        * Generate Gray code: to do this add one to n[]; the
        * number of carries plus one will be the next transition
        * position.
        */
        while (1) {
                i = 1;
                while (++n[ℓ − i + 1]== α) {
                        n[ℓ − i + 1]= 0;
                        i++;
                        if (i > ℓ) goto finished;
                }
                g[ℓ − i + 1] = (g[ℓ − i + 1]+1) mod α;
        };

        finished:

}
```

Figure A.1: An algorithm for generating $\mathcal{G}(\alpha, \ell)$ code words in $\Theta(\alpha^\ell)$ time. Notice that the index $\ell - i + 1$ can be replaced with just i, if x_1 becomes the least significant character rather than the most significant.

the average time spent in the while loop is

$$\frac{\sum_{i=0}^{\alpha^{\ell-1}} \ell_i}{\alpha^\ell}$$

where ℓ_i is the number of carries plus one, when one is added to $\mathcal{N}_i(\alpha, \ell)$. It can be shown inductively that this is equivalent to

$$(\frac{1}{\alpha})^{\ell-1}\ell + \frac{\alpha-1}{\alpha}\sum_{i=1}^{\ell-1}(\frac{1}{\alpha})^{i-1}i \tag{A.1}$$

which is less than

$$\sum_{i=1}^{\ell}(\frac{1}{\alpha})^{i-1}i \tag{A.2}$$

We now want to show that the series given by Equation A.2 is bounded by a constant. To do this, we will use the fact that $\sum_{i=0}^{k} a^i = \frac{a^{k+1}-1}{a-1}$. Let $x = 1/\alpha$. Then

$$\begin{aligned}
\sum_{i=1}^{\ell}(\frac{1}{\alpha})^{i-1}i &= \sum_{i=1}^{\ell} x^{i-1}i = \sum_{i=0}^{\ell} x^{i-1}i \\
&= \frac{d}{dx}\sum_{i=0}^{\ell}\int x^{i-1}i \\
&= \frac{d}{dx}\sum_{i=0}^{\ell} x^i \\
&= \frac{d}{dx}\{\frac{x^{\ell+1}-1}{x-1}\} \\
&= \frac{\ell/\alpha - \ell - 1}{\alpha^\ell(1/\alpha-1)^2} + \frac{1}{(1/\alpha-1)^2}
\end{aligned}$$

Because $\frac{\ell/\alpha-\ell-1}{\alpha^\ell(1/\alpha-1)^2}$ is always negative, the inner while loop does a constant amount of work on average.

■

Thus the while loop will do an amortized upper bound of $(\frac{\alpha}{\alpha-1})^2$ iterations. This upper bound is high since we dropped a factor of $\frac{\alpha-1}{\alpha}$ from $\ell-1$ terms of Equation A.1 to get Equation A.2. A closer approximation to the true number of iterations, amortized, for large ℓ is $\frac{\alpha-1}{\alpha}(\frac{\alpha}{\alpha-1})^2 = \frac{\alpha}{\alpha-1}$. Both the upper bound and the approximation decrease with α. This algorithm seems quite comparable to the one presented by Bitner, *et al.* [2] in terms of overall speed and simplicity, although their algorithm is constant time per codeword while ours is only constant time amortized. However, our algorithm easily generalizes to the α-ary Gray code of Sharma and Khanna [43]. Further, it is easy generalized to work on multary codes (next section).

It is also possible to define the transition sequence $\mathcal{T}_\ell$ for $\mathcal{G}(\alpha, \ell)$ recursively, in terms of α, ℓ, and $\mathcal{T}_{\ell-1}$. The transition sequence is given by $\mathcal{T}_1 = \overbrace{1, 1, \ldots, 1}^{\alpha-1 \text{ times}}$, and $\mathcal{T}_\ell = \overbrace{[\mathcal{T}_{\ell-1}\ell][\mathcal{T}_{\ell-1}\ell]\cdots[\mathcal{T}_{\ell-1}\ell]}^{\alpha-1 \text{ times}}\mathcal{T}_{\ell-1}$. This is also done by Sharma and Khanna, although our work is done independently. Further, Sharma and Khanna's definition is slightly different and is inconsistent: in the base case ($\ell = 1$), their transition sequence consists of $\alpha - 1$ ones, and thus their transition sequence carries the first element of the Gray code to the last; however, when $\ell > 1$, their transition sequence is cyclic, i.e., it carries the first Gray code element cyclically through the code to itself.

As a final note, $\mathcal{K}$ and $\mathcal{K}^{-1}$ can be modified to work on α-ary numbers with positions starting at the right and going to the left. Such a definition is given by Sharma and Khanna [43].

A.3 An Extension to $\mathcal{G}(\alpha, \ell)$ and Another Generation Algorithm

Let $\mathcal{N}_{mult}(\alpha_1, \alpha_2, \cdots, \alpha_\ell)$ represent the code where the first character (from the left) can have one of α_1 characters, the second can have one of α_2 characters, and so on. The order of this code is defined by x *is before* y (x and y are two codewords) iff **before**(x, y), where **before**$(x_i \cdots x_\ell, y_i \cdots y_\ell)$=$x_i < y_i$ or ($x_i = y_i$ and **before**$(x_{i+1} \cdots x_\ell, y_{i+1} \cdots y_\ell)$). The basis is **before**$(x_i, y_i) = x_i < y_i$. This is just the natural ordering. We call a code where each character can have a different α a multary code, as done by Field [15]. Multary codes can represent multisets: there are ℓ element types i, $i = 1, 2, \cdots, \ell$, and each x_i in the codeword x denotes the number of times element i is represented in the multiset, for a maximum of $\alpha_i - 1$. Listing $\mathcal{N}_{mult}(\alpha_1, \alpha_2, \cdots, \alpha_\ell)$ gives all the possible multisets where each element i can be represented up to $\alpha_i - 1$ times. However, as with a binary or α-ary natural ordering, two consecutive codewords can have a Hamming distance greater than one.

It is possible to get a generalized Gray code, $\mathcal{G}_{mult}(\alpha_1, \alpha_2, \cdots, \alpha_\ell)$ for these multary codes. One such possible code is a generalization of the α-ary Gray code of Sharma and Khanna [43]. To see this, modify $\mathcal{K}$ to be

$$\begin{aligned} \mathcal{K}(x) &= (g) \\ g_i &= \begin{cases} x_1 & \text{if } i = 1 \\ (x_i - x_{i-1}) \bmod \alpha_i & 1 < i \leq \ell \end{cases} \end{aligned}$$

and $\mathcal{K}^{-1}$ to be

$$\begin{aligned} \mathcal{K}^{-1}(g) &= (x) \\ x_i &= \begin{cases} g_1 & \text{if } i = 1 \\ (g_i + x_{i-1}) \bmod \alpha_i & 1 < i \leq \ell \end{cases} \end{aligned}$$

We assume the generalized definition throughout the rest of this chapter. $\mathcal{K}$ is an isomorphism between $\mathcal{N}_{mult}(\alpha_1, \alpha_2, \cdots, \alpha_\ell)$ and $\mathcal{G}_{mult}(\alpha_1, \alpha_2, \cdots, \alpha_\ell)$.

To show $\mathcal{K}$ is one-one, pick two strings x and x' such that $x \neq x'$. Then x and x' must differ in some first character, say $x_i \neq x'_i$. Let $g = \mathcal{K}(x)$ and $g' = \mathcal{K}(x')$, and assume $g' = g$. It is easy to prove that $g_i \neq g'_i$, a contradiction. To show $\mathcal{K}$ is onto pick an arbritrary g. We must show that there is a (legal) x such that $g = \mathcal{K}(x)$. This is can be done by setting $g_1 = x_1$ and progressing inductively.

The proof that $\mathcal{G}_{mult}(\alpha_1, \alpha_2, \cdots, \alpha_\ell)$ is a Gray code is virtually identical to the proof of Theorem A.1.1, except that the cyclic case does not hold in general. However, if g and g' are two adjacent words in $\mathcal{G}_{mult}(\alpha_1, \alpha_2, \cdots, \alpha_\ell)$ such that g is before g' and $g_i \neq g'_i$, then $g'_i = (g_i + 1) \bmod \alpha_i$.

We can even generalize Algorithm A.1 to produce these Gray code words in constant amortized time per code word. The generalizations are quite simple (e.g., the procedure takes an array of $\alpha[1 \cdots \ell]$). The code generation algorithm will be especially efficient if $\alpha_i \leq \alpha_j$ for $i \leq j$.

A "loop-free" algorithm is given by Williamson [51, p. 111-112] that also generates a (different) multary Gray code. However, our algorithm seems simpler, both in implementation and documentation, and our algorithm also seems comparable in terms of efficiency.

Appendix B

Description Of Search Algorithms Used in This Thesis

Gene Invariant Genetic Algorithm (GIGA)

GIGA is a genetic algorithm that uses only crossover. It works by maintaining a population of n strings in an array, and iteratively picking two adjacent strings to *mate*. A mating involves generating $\mathcal{F}$ (family size) children pairs by applying crossover to the parents, and then replacing the parents with the best pair of children (non-elitism), or the best pair including the children and the parents (elitism).

When a pair is chosen to replace the parents, the lower valued string is inserted above the higher valued string; over time, the population becomes roughly sorted, high value strings near the bottom of the array, low value strings near the top. We call this *implicit* sorting. GIGA also has an option to *explicitly* sort the population by fitness value after each mating, but this was never used.

Using only crossover ensures that the set of characters in any column of the matrix remains invariant, preventing convergence in the usual sense. Since mutation is not used, search can only be achieved through propagation of sub-strings through the population.

GIGA has several parameters, and we used several default parameter settings throughout this thesis (see Table B.1). *Unbiased adjacent selection* means that, for each mating, the (adjacent) parents are chosen uniformly from the $n-1$ possible pairs

Unbiased adjacent selection.
Random rotation.
Maximum of pair.
One-point crossover.
No explicit sorting of population.
No gray encoding.
Epsilon = 0.0 (e.g., not used).

Table B.1: Default GIGA parameter settings used throughout this thesis.

```
Create initial population
While (current # evals < some maximum number of evals) {
        Select a pair of parents to mate.
        Produce a family of offspring pairs using crossover.
        Select the best offspring pair.
        Replace parents with best pair.
}
```

Table B.2: Outline of GIGA

of parents.

Random rotation generates the first string of the population randomly. It then produces the next $\alpha - 1$ strings by adding i mod α to each character of the original string, for the next i strings, $i = 1, \cdots, \alpha - 1$. This process is then repeated as long as strings still need to be generated. For $\alpha = 4$ and $n = 6$, if our initial string is 10023, then we would get the strings 21130, 32201, and 03312. At this point the fifth string would be created randomly, say 30121, and the sixth and final string would be 01232. When $n = 2$ and $\alpha = 2$, random rotation generates a complementary pair of strings. If $n \geq \alpha$, then random rotation ensures that, for a given character position, every possible character occurs at that position.

Maximum of pair means that the fitness of a pair of strings is the maximum fitness of the strings in the pair. While GIGA can be seen as sifting sub-strings through the population, this definition of best pair means it can also be viewed as a hill climber, even for population sizes greater than two, because the family actually represents $\mathcal{F}$ different populations. Here the fitness of a population is not always equal to the maximal fitness of the strings within the population, but depends on all the strings within the population.

For a more in-depth explanation of GIGA, the reader may wish to see GIGA's documentation [6]. Our version of GIGA only differs from this version in that it halts after a maximum number of fitness evaluations, rather than a maximum number of matings. See Table B.2 for an outline of the GIGA program. This table is based on a table given by Culberson [6].

Mutation Hill Climber (MHC)

Our mutation hill climber uses one-point mutation on a single α-ary string. The population initially consists of a single randomly generated string. See Table B.3 for an outline of MHC.

MHC copies the parent string to $\mathcal{F}$ children strings, and then applies one-point mutation to each child. With non-elitism, the fittest child replaces the parent string; with elitism, the fittest child replaces the parent string only if its fitness is greater than or equal to its parent's fitness. As with GIGA, each such generation is referred to as a mating.

There are other types of mutation hill climbers; MHC was used because its search

```
Randomly initialize parent string.
While (current # evals < some maximum number of evals) {
        Produce a family of offspring strings using one-point mutation.
        Select the best offspring string.
        Replace parent string with best string.
}
```

Table B.3: Outline of MHC

is isomorphic to GIGA's when GIGA uses random rotation, $\alpha = 2$, and $n = 2$.

Not Quite-GIGA (NQ-GIGA)

NQ-GIGA (Not Quite GIGA) is the same as GIGA, except that one-point mutation has been added to the algorithm. This was done by encoding two extra parameters into GIGA: a mutation and a crossover rate. Given a pair that had been selected, the crossover rate is the probability that the pair will be crossed. The mutation rate is the probability that each string in the pair undergoes one-point mutation. NQ-GIGA uses the same defaults (see Table B.1) as GIGA.

Traditional Genetic Algorithm (TGA)

A traditional genetic algorithm (like the simple GA, but slightly less general) generates an initial n member population, where each string is generated randomly. This population is evolved by creating a new population from the old, by selecting pairs of strings from the old population and applying uniform mutation (where each character is mutated with probability p_m) and crossing these two strings with some probability p_c to produce one or two children strings which are then added to the new population. This step is repeated until the new population is generated. The new population replaces the old. Going from an old population to a new population is called a *generation*. Strings are chosen stochastically for mating; the higher the fitness of a string, the greater its chance of being chosen for mating.

There are many possible "traditional genetic algorithms," and we use GAC [47] as our representative TGA. It uses proportional fitness to select which individuals reproduce. Proportional fitness means that an individual x is expected to be mated $f(x)/\mu$ times, where μ is the average fitness of the old population. The crossover rate is the probability that a selected pair will be crossed. One-point crossover is used.

Because populations are small (small sample size) and high-fit strings can be involved in several matings in the next generation's production while low-fit strings may not be mated at all (loss of diversity), populations tend to rapidly become converged when proportional fitness is used. If a population converges too quickly (premature convergence), so that the average Hamming distance between each string is very small (i.e., all strings are highly similar), then crossover can become ineffective. This problem is well-known in the GA community, and there is evidence [8, 46] that mutation is the driving force behind traditional GAs for this very reason.