GENETIC ALGORITHMS AS A COMPUTATIONAL TOOL FOR DESIGN

Sushil J. Louis

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University

August 1993

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

Doctoral Committee

Gregory J. E. Rawlins, Ph.D.
(Principal Advisor)

Christopher T. Haynes, Ph.D.

Douglas R. Hofstadter, Ph.D.

August 4, 1993

Esther Thelen, Ph.D.

DEDICATION

This thesis is dedicated to my wife and family.

Acknowledgements

A thesis like this is never possible without the help, encouragement, motivation, and influence of a large number of people. Greg Rawlins, my advisor, taught me many things, but most importantly, how to do research and how to write well. Greg was an inexhaustible storehouse of knowledge, insight and help on just about any subject. His influence pervades this thesis and I am deeply indebted to him for being my advisor, teacher and friend. Thank you.

My thesis committee members influenced the direction of my work and the writing of my thesis in their own ways. Chris Haynes imbued in me an appreciation for programming languages and operating systems. Douglas Hofstadter has a unique perspective on many areas including computer science, artificial intelligence, geometry, and creativity, to name a few. Design balances aesthetics with pragmatics, but the need to strive for aesthetic satisfaction and an eye for detail are part of his influence. Esther Thelen solidified my interest in non-linear systems and chaos theory during a class in (perhaps surprisingly) the psychology department. My continuing interest in genetic algorithm applications in biological and social systems owes much to her.

Other than my committee members I would like to thank John Gero at the University of Sydney, for inviting me to Sydney and introducing me to the architectural, structural, and civil engineering view of design and genetic algorithms. I thank Sourav Kundu, also at Sydney, for many discussions on the subject. Closer to home, I thank David Leake for interesting me in case–based reasoning. Special thanks to Gary McGraw and Rich Wyckoff for their input, work, comments and arguments. Thanks to David Goldberg for encouraging and commenting on my ideas. Thanks to Phil Bradford and Terry Jones for interesting discussions on genetic algorithms and commenting on papers. Jon Mills commented on an earlier draft of this dissertation.

The administrative and systems staff in the computer science department went out of their way to make life easy, thank you very much.

My family encouraged, supported and motivated me to continue during the ups and downs leading to this thesis. Finally and most importantly, I would like to thank my wife who listened patiently to all my ideas, read all my papers, gave me moral support, love and the motivation to finish.

Abstract

Design is an ubiquitous activity embracing most of engineering and architecture. Because design is so pervasive, any research that leads to improvements in design processes or products can have great impact. Current efforts at capturing the design process in a computational framework do not pay heed to the evolutionary aspect of prototype creation and ongoing refinement. Further, in poorly-understood domains where expert knowledge or previous experience is lacking, current systems do not perform well.

Genetic algorithms are stochastic parallel search algorithms that model natural selection, the process of evolution. Over time natural selection has produced a wide range of robust structures (life forms) that efficiently perform a broad range of functions. The success of natural selection on earth provides an existence proof of the viability of an evolutionary process as a model for design.

This thesis lays the foundations for the use of genetic algorithms in helping to attack a well-defined and important subset of design. It maps genetic algorithms onto the design process; defines appropriate representation criteria to take advantage of the nature of the problem; specifies methods of analyzing genetic-algorithm-generated designs; and places bounds on the time complexity of the task. Scalable examples

from circuit design, floorplanning and function optimization are used to demonstrate, illustrate and ground these results.

Contents

List of Tables

List of Figures

1

Introduction

> Everyone designs who devises courses of action aimed at changing existing situations into preferred ones.
>
> –H. A. Simon. *The Sciences of the Artificial*, MIT Press, 1969.

Design is a fundamental, purposeful, pervasive, and ubiquitous activity. As Simon pointed out, anyone who formulates programs of action to change an existing state to a preferred one is solving design problems (Simon, 1969). Planning, scheduling, circuit design, VLSI layout, architecture, and most branches of engineering are, or can be cast as, design problems. Any useful tool that helps the design process would be widely applicable. Current tools are based on a knowledge intensive view of the design process. Techniques like expert systems and case-based reasoning, apart from problems with the acquisition of knowledge, suffer from a basic flaw: they fail on new and/or poorly-understood application domains, where there is insufficient knowledge to proceed.

Genetic algorithms (GAs) are randomized parallel search algorithms that model natural selection, the process of evolution. Over time, natural selection has produced a wide range of robust structures (life forms) that efficiently perform a broad range of functions. The success of natural selection on earth provides an existence proof of the viability of an evolutionary process as a model for design. Like natural selection, GAs are a robust search method requiring little information to search effectively in large, poorly-understood spaces.

In this context, the central claim of the thesis is

> *Genetic algorithms provide a computational tool for the design problem: "Given a function and a target technology to work within, design an artifact that performs the function subject to constraints."*

Although genetic algorithms have been used in engineering problems, there are reasons why designers are reluctant to embrace GAs. Representation problems are one reason familiar to artificial intelligence researchers. There was no bound on the time to convergence until recently (Louis and Rawlins, 1993b). This thesis identifies and overcomes four basic problems in establishing genetic algorithms as a computational tool for design.

The current chapter expands on the definition of design, the nature of design problems, and casts design as search through a state space. After identifying the questions addressed in this thesis, the chapter ends by providing an overview of the structure of this dissertation.

1.1 Design

Design's pervasive nature makes it difficult to find a concise, comprehensive, and well-accepted definition. Attempts at defining design usually reflect the bias of the author and range from the mundane to the theoretical. Dasgupta's book provides some historical as well as modern attempts at such definitions (Dasgupta, 1991). More recently, Chris Tong and Duvvuru Sriram advance a fairly comprehensive definition and describe design as "the process of constructing a description of an artifact that satisfies a (possibly informal) functional specification, meets certain performance criteria and resource limitations, is realizable in a given target technology, and satisfies criteria such as simplicity, testability, manufacturability, reusability, etc." (Tong and Sriram, 1992b).

The various definitions agree, however, that design is concerned with the mapping of a specified function onto a structure or description of a structure.[1] Usually, the designed structure also satisfies performance, resource, and other pragmatic constraints. In addition, most design theorists agree that the goal of design is to purposefully initiate change in some aspect of the world (Simon, 1969; Tong and Sriram, 1992b). Unlike theories of the natural sciences, design is concerned with construction of artifacts and the purposeful effecting of change.

The main reason for the difficulty of design tasks lies in the complexity of the mapping between function and structure. The specified function may be very complex and/or it may have to be realized using complex arrangements of a large number of interacting parts. Typically, the behavior of each component is well known, and the difficulty lies in predicting how an assemblage of such components will behave.

[1] In the rest of the thesis, structure and description of structure are taken as equivalent unless otherwise stated.

Additional non-functional criteria and constraints complicate the design problem even further.

The type of structure synthesized can be used to distinguish design tasks (Tong and Sriram, 1992b). In *structure synthesis*, the design is constructed from the composition of primitive parts into a structure that performs a specified function. Constructing a parity-checker circuit from logic gates is an example of this type of task. This may not be optimization, but rather, as noted by Simon, it may be a *satisficing* task, in that a designer searches for a circuit that checks parity, not an optimal circuit (Simon, 1969).

In *structure-configuration* tasks, the design is a configuration of parts of pre-determined type and connectors of pre-determined type. For example, in floorplanning, the problem is to design a configuration of rooms and doors to fit in a given area, along with the values of room and door parameters.

Lastly, in *parameter-instantiation* tasks, the problem is to find the values of parameters such that a particular design can be instantiated.

In all the cases above, optimization plays a role, since there is usually a cost associated with each design. Minimizing this cost can be part of the original design task or it may be a separate task.

1.2 Design and Search

Casting design as a search problem, a formulation with a well-established heritage in artificial intelligence research, a design process searches through a state space of possible structures for one or more structures that perform a specified function. Points in the state space represent candidate designs, and one or more of these points defines a goal state representing a design that meets specifications.

In search there is a tradeoff between *flexibility* — applicability in different domains, and *speed* — the number of candidate designs searched through before finding the correct one. Current design systems tend to depend on domain-specific knowledge and favor speed over flexibility. Most are so brittle that they fail completely when domain knowledge is scarce. Genetic algorithms, on the other hand, can make do with little domain knowledge, make few assumptions about the search space, and use domain-independent operators for generating candidate points in a state space.

1.2.1 Search

If there is sufficient knowledge about a problem domain, there is no need to search for a solution. However, if there is insufficient knowledge about a domain to arrive directly at a solution, then if there is at least enough information to delineate an area within which a solution is expected to be found, a search method can be used to hunt for a solution within the delineated area.

This thesis is concerned with design problems that do not lend themselves to a direct solution but that can be cast as search problems within a space of possible solutions.

Search algorithms typically have two components corresponding to the tradeoff between speed and flexibility. They balance *exploration* of the search space with *exploitation* of areas of the space. Exploration points out new areas to search in, while exploitation concentrates search in a particular area. The need for a balance arises because unchecked exploration leads to wasted time in unpromising areas, while severe exploitation may miss the correct solution by concentrating on too small an area. That is, too much exploration means too much time, and too much exploitation means that the correct solution may not be found. On the other hand, if there is not enough exploration, the correct solution may not be generated while insufficient exploitation may take too long. Striking a good balance between exploration and exploitation is therefore critical for a search algorithm.

Wasting time in unpromising areas of the search space may seem a small price to pay for finding the correct solution. Unfortunately, the size of the search spaces that arise in design domains is often so large that a human lifetime is short compared to the time needed by the fastest supercomputers to look at a significant fraction of the space. This is why random and/or exhaustive search (RES) is infeasible: it explores too much. On the other hand, RES works equally well across all search spaces, since it makes the fewest assumptions about exploitable properties of the search space. It assumes that a minimal amount of knowledge is available — enough to know when to stop. Put another way, RES needs and uses minimal information about the search space to work equally well across a variety of spaces.

At the other extreme are algorithms that do no exploration and have knowledge available to solve the problem quickly and directly. These algorithms make strong assumptions about the search space and sufficient exploitable information is available to avoid searching. However, because these strong assumptions do not usually hold across domains (search spaces), these algorithms work only on the particular space they were designed for and do miserably on others.

In between these extremes of random and/or exhaustive search and no search, every other search algorithm makes assumptions of varying kinds about search spaces. These assumptions correspond to knowledge about the space and may be correct, incorrect, or misleading, implicit or explicit, and known or unknown, when used for searching a particular space. This knowledge is exploited to guide exploration, speeding up search by generating a search bias manifested in the set of generated solutions.

Genetic algorithms belong to a class of algorithms, called blind search algorithms, which make the assumption that there is enough knowledge to compare two solutions and tell which is better (Rawlins, 1991). Genetic algorithms are the search method of choice for this dissertation.

1.3 Genetic Algorithms and Search

A Genetic Algorithm (GA) is a randomized parallel search method modeled on evolution. GAs are being applied to a variety of problems and becoming an important tool in machine learning and function optimization. They have already been used in the preliminary design of aircraft engine turbines and, more recently, in protein structure prediction (Powell et al., 1989; Le Grand and Merz Jr., 1993). Goldberg's book gives a list of application areas (Goldberg, 1989b).

1.3.1 Intuition

Intuitively, genetic algorithms can be understood in terms of differences with other search methods that can be applied in the same situation. Consider the problem of finding the correct combination for opening a thirty-digit combination lock (designing a key). One possible algorithm is random and/or exhaustive search. In this case the only information used by the algorithm is whether or not the currently generated combination is the correct one. With this algorithm the chances of success — finding the correct combination — are better than 1/2 only after the generation of $10^{30}/2$ combinations. This is a very large number. In fact, if RES generates a combination every nanosecond, it would take more than the age of the earth before success is more than 50% probable.

If, in addition to whether the current combination is correct, more information is available, a search algorithm could use this information to increase efficiency. Hill-climbing or gradient-descent algorithms, when given information about which of two generated combinations is "closer" to the correct combination, make use of this additional, *directional* information to speed up the search. The hill-climbing analogy considers the search space as a landscape through which a search algorithm moves towards the highest point, where height corresponds to "closeness" to the optimum. However, a hill-climber can be trapped on a hill that is not a global optimum but a local optimum. In other words, if the search landscape is rugged with a lot of hills (local optima), the algorithm could climb the nearest hill and find that any further movement decreases height and thus remain trapped on this hill, whereas the highest point (global optimum) is actually on another taller hill.

A population of hill-climbers is more robust. Trapping a large number of hill-climbers, working independently on the same problem, is less probable than trapping just one hill-climber.

Genetic algorithms are more than a population of hill-climbers. In addition to having a population of individuals that hill-climb, genetic algorithms allow the exchange of information among individual hill-climbers in generating combinations that are ever closer to the correct one. Genetic algorithms exemplify the maxim: "Two heads are better than one" and can be considered a population of information-exchanging hill-climbers.

1.3.2 Formalism

When used in design, a GA encodes a candidate design in a binary string[2]. A randomly generated set of such strings forms the initial population from which the GA starts it search. Three basic genetic operators: selection, crossover, and mutation guide this search. The genetic search process is iterative: evaluating, selecting, and recombining strings in the population during each iteration (generation) until reaching some termination condition. The basic algorithm, where $P(t)$ is the population of strings at generation t, is given below.

$t = 0$
initialize $P(t)$
evaluate $P(t)$
while (termination condition not satisfied) **do**
begin
    select $P(t+1)$ **from** $P(t)$
    recombine $P(t+1)$
    evaluate $P(t+1)$
    $t = t + 1$
end

Evaluation of each string is based on a fitness function that is problem-dependent. It determines which of two candidate solutions is better (figure 1.1). This corresponds

[2]For clarity, this thesis only considers binary encodings. The same ideas can be extended to higher-cardinality alphabets or real numbers.

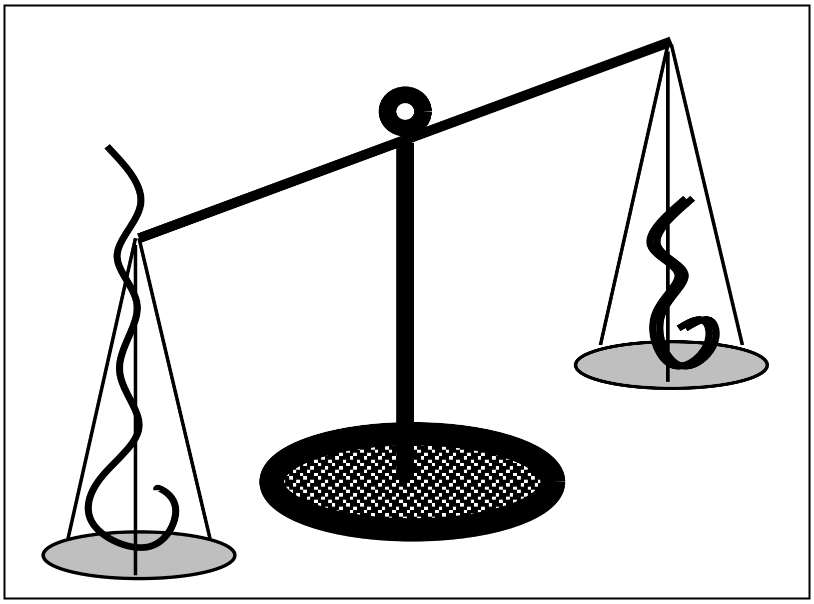

Figure 1.1: Evaluating individuals determines which is better.

to the environmental determination of survivability in natural selection. Selection of a string, which represents a point in the search space, depends on the string's fitness relative to that of other strings in the population. It probabilistically culls from the population those points that have relatively low fitness (figure 1.2). Mutation and crossover imitate sexual reproduction. Mutation, as in natural systems, is a very low probability operator and just flips a specific bit. Crossover in contrast is applied with high probability. It is a randomized yet structured operator that allows information exchange between points. Simple crossover is implemented by choosing a random point in the selected pair of strings and exchanging the substrings defined by that point. Figure 1.3 shows how crossover mixes information from two parent strings, producing offspring made up of parts from both parents. Note that this operator which does no table lookups or backtracking, is very efficient because of its simplicity.

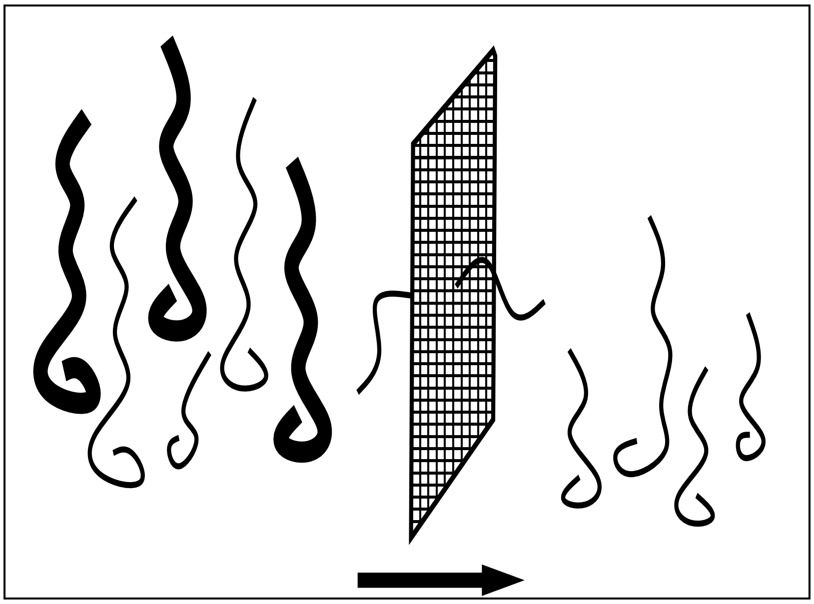

Figure 1.2: Selection acts as a sieve, selecting better individuals to continue.

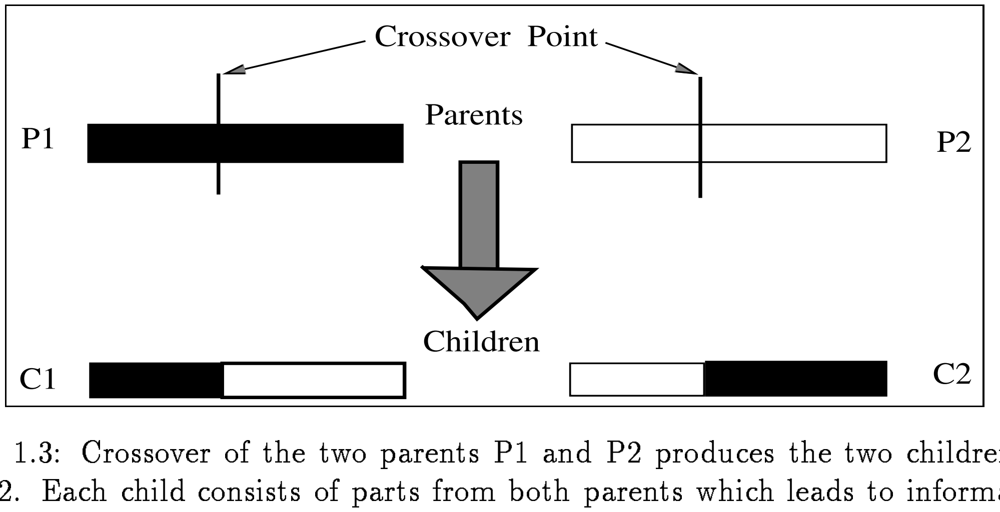

Figure 1.3: Crossover of the two parents P1 and P2 produces the two children C1 and C2. Each child consists of parts from both parents which leads to information exchange.

Selection probabilistically filters out designs that perform poorly, choosing high performance designs to concentrate on or *exploit*. Crossover and mutation, through string operations, generate new designs for *exploration*. Thus genetic algorithms only require two elements: 1) an encoding of candidate structures (solutions), and 2) a method of evaluating the relative performance of candidate structures. Given an initial population of elements, GAs use the feedback from the evaluation process to select fitter designs, generating new designs through recombination of parts of selected designs, eventually converging to a population of high performance designs. Since GAs only require feedback on the relative performance of candidate designs to guide search, they are applicable in any design task that can provide this information. In other words, given a function to perform, a genetic algorithm only needs to know which of a pair of candidate designs is "closer" to performing the function. And even this information need not be perfect but can be subject to noise and misdirection to some extent. This thesis is only concerned with such design problems.

However, there are unresolved problems in using genetic algorithms for design. First, although genetic algorithms allow sharing information among hill-climbers, are there search spaces where this sharing leads to better performance? Second, since genetic algorithms work with an encoding of the problem, what kind of biases can be expected to be introduced by the encoding and can the genetic algorithm surmount unfavorable encoding biases? Third, genetic algorithms should be applied in knowledge lean domains where the solutions that are found tend to be difficult to understand. The question is, what kind of analysis is possible of genetic algorithm designs? Finally, given a problem, how long should a designer run a genetic algorithm on it?

The dissertation addresses each of these problems in turn, using design examples from circuit design, floorplanning and function optimization.

1.4 Related Work

Although design encompasses a lot of fields, this thesis is most concerned with work that has been done at the nexus of engineering, architecture, and artificial intelligence. Perhaps the closest piece of research connected to this dissertation is in using genetic algorithms to design computer programs.

John Koza uses genetic algorithms to evolve programs in the LISP programming language, illustrating the use of genetic algorithms in the task of program design (Koza, 1993). The work presented in this thesis is however, qualitatively different. This dissertation develops tools and methods, and addresses questions and problems, that can arise in using genetic algorithms in *any* applicable design task and thus attempts to address more foundational issues.

Knowledge-based systems for design depend on domain knowledge to accomplish design tasks. A description of the state of the art in engineering design can be found in (Tong and Sriram, 1992a). Systems that use a combination of techniques also exist. Such integrated design environments use a variety of tools to structure and attack design problems. "Engineous" is a system that includes genetic algorithms in its toolbox to uncover unconventional designs when needed (Tong et al., 1992; Powell et al., 1989). The work in this thesis should help improve the construction, use and applicability of such genetic algorithm tools.

Innovation and creativity are often associated with design processes. Chapter 2 explains in what sense genetic algorithms can be considered innovative. Other approaches to innovative engineering design can be found in (Mostow et al., 1992; Huhns and Acosta, 1992; Sycara and Navinchandra, 1992; Goel and Chandresekaran, 1992;

Ulrich and Seering, 1992; Tong, 1992). These approaches include case-based reasoning, structural mutation, and innovation through the combination of multiple knowledge sources. Lenat's work on heuristics and meta-heuristics can also be considered as an innovative approach to the design of effective strategies (Lenat, 1983).

Modeling the creative aspects of a design process is addressed in (Gero, 1992; McGraw, 1992). The work in (Gero, 1992) seeks to define and model the creative aspects of a design task. Letter Spirit, the project described in (McGraw, 1992), attempts to model central aspects of human high-level perception and creativity on a computer, in the domain of artistic letter-design. This dissertation does not address creativity.

1.5 Structure of this Thesis

Chapter 2 introduces natural selection and genetic algorithm theory, fleshing out the problems identified above. Although search is the model used in this thesis, the chapter also illustrates the close mapping between genetic algorithms and the well-established *Analysis, Synthesis, Evaluation* model of the design process. Then, the innovativeness of genetic algorithms and issues in evaluating and encoding problems are discussed. The chapter ends by explaining how to represent a problem for a genetic algorithm using an example from floorplanning in architectural design.

Chapter 3 compares genetic and other search algorithms. It defines a class of search spaces which are easy for genetic algorithms and hard for stochastic hill-climbers. Since recombination is the crucial difference between genetic and other search algorithms, the results provide insight into the kind of spaces where recombination is necessary. This chapter helps answer the question of when to expect genetic algorithms to outperform other algorithms and thus when to use them to greatest effect. An earlier version of the work reported in this chapter has appeared in (Louis and Rawlins, 1993a).

In chapter 4 the dissertation shows how problem encoding can bias genetic search. This leads into the important issue of problem encoding for genetic algorithms in the design domain. A bad encoding can cause a GA to flounder, to be misled, or both. Modifying the classical genetic algorithm to mitigate these problems sheds light on the nature of good encodings from the perspective of genetic algorithms and from the differing perspective of design. This chapter uses examples from combinational circuit design to illustrate the results. Parts of this chapter have appeared earlier in (Louis and Rawlins, 1991).

Chapter 5 is concerned with analyzing GA generated designs. The importance of design analysis lies in being able to explain and subsequently modify designs in principled ways. Since genetic algorithms are designed for working in poorly understood spaces, knowledge about the space needs to be acquired, stored and processed for useful analysis. This chapter shows how to apply case-based reasoning tools to acquire knowledge about a design space to explain solutions generated by a GA. Much of this information is implicit in the processing done by the genetic algorithm. The case-based reasoning tools extract and process information supplied by the trajectory of a genetic algorithm through the search space. This organized information in the case-base can be used to explain how a solution evolved and to help identify its building blocks. The resulting knowledge base can also be used to tune a genetic algorithm for that specific domain. Examples from function optimization and circuit design clarify the results. This work has appeared in (Louis et al., 1993).

Chapter 6 considers computational complexity, a crucial element in any computational tool and in any design model where a designer must specify time limits for a design task. Since evolution is driven by diversity or variation in a gene pool, the average hamming distance is used as a diversity metric to derive bounds on the time convergence of genetic algorithms. Analysis of a *flat* domain provides worst case time complexity for static functions. Further, employing linearly computable runtime information provides tighter bounds on the time beyond which progress is unlikely on arbitrary static functions. As a by-product, this analysis also gives qualitative bounds by predicting average fitness, the quality of an average solution at convergence. Examples from the DeJong test suite of problems are used to provide supporting empirical evidence. The part of this chapter on worst case time complexity has appeared in (Louis and Rawlins, 1993b).

The last chapter draws together the threads of representation, analysis, and complexity paving the way toward establishing genetic algorithms as a computational tool for design. Questions and possible avenues of research brought to light by this dissertation are addressed at the end.

2

Genetic Algorithms and Design

I have called this principle, by which each slight variation, if useful, is preserved, by the term of Natural Selection.

–Charles Darwin. *On the Origin of Species,* 1859.

This chapter introduces natural selection, genetic algorithm theory, and shows the correspondence between genetic algorithms and a classical design model. This sets the stage for describing the innovative powers of a genetic algorithm and elaborating on encoding and evaluation of designs for genetic algorithms. Finally, an example from floorplanning in architecture illustrates these issues.

2.1 Natural Selection

In its most general form, natural selection means the differential survival of entities (Dawkins, 1986). Some entities live and others die. For this to happen there must be a population of entities capable of reproduction. Natural selection prunes this population according to the criterion of survivability (fitness). It acts as a sieve. In sexually reproducing species the unit of natural selection is the *gene*. The values of a gene are its *alleles*. However, sieving by itself is not capable of producing designs like those of the life forms on this planet in any reasonable amount of time. Natural selection needs reproducing entities to feed the results of one sieving process on to the next, and so on. It is the continuing *cycle* of reproduction and selection (the sieving process) that is responsible for the diversity of life on earth. The existence of finite resources leads to competition for these resources and survival of those entities that have a competitive advantage. A life form, or *phenotype*, is a survival machine built by a set of genes (a *genotype*) to create a competitive advantage over other forms. Although it is the phenotypes that are directly competing for survival, it is the genotypes that get selected on the basis of this competition for further consideration. Evolution can be thought of as a search process. It searches a very large space of genotypes producing structures that are efficient at carrying out functions desirable for survival in their environment. For example the search space for the ϕX174 viral genotype is of the order of 4^{5400}, and this is one of the smallest life forms. In 1975 John Holland defined a *Genetic Algorithm*, a search process based on natural selection, now used as a tool for searching the large, poorly-understood spaces that arise in many application areas of science and engineering (Holland, 1975).

2.2 Genetic Algorithm Theory

A genetic algorithm uses the mechanics of natural selection to guide search. It seems natural to use a genetic algorithm to design artifacts, since the paradigm on which they are based is so successful at it. However, as already indicated, there are difficulties and it is helpful to understand the theory behind genetic algorithms before considering the difficulties in detail. The algorithm was introduced in chapter 1, this chapter considers genetic algorithm theory.

Crossover causes genotypes to be cut and spliced. This means that instead of considering the fate of individual strings in analyzing a genetic algorithm, the substrings created and manipulated by crossover must be considered. These substrings define regions of the search space and are called schemas. More formally, a schema is a template that identifies a subset of strings with similarities at certain string positions. Holland's *schema theorem* is fundamental to the theory of genetic algorithms. For example consider binary strings of length 6. The schema 1**0*1 describes the set of all strings of length 6 with 1*s* at positions 1 and 6 and a 0 at position 4. The "*" denotes a "don't care" symbol which means that positions 2, 3 and 5 can be either a 1 or a 0. Although this thesis only considers a binary alphabet, the notation can be extended to non-binary alphabets. The *order* of a schema is defined as the number of fixed positions in the template, while the *defining length* is the distance between the first and last fixed positions. The order of 1**0*1 is 3 and its defining length is 5. The *fitness* of a schema is the average fitness of all strings matching the schema.

A search algorithm balances the need for exploration — to avoid local optima, with exploitation — to converge on the optima. Genetic algorithms dynamically balance exploration versus exploitation through the recombination and selection operators respectively. With the operators as defined earlier, the schema theorem proves that relatively short, low-order, above average schema are expected to get an exponentially increasing number of trials or copies in subsequent generations (Goldberg, 1989b). Mathematically

$$m(h,t+1) \geq \frac{m(h,t)f(h)}{\overline{f}_t}\left[1 - P_c\frac{\delta(h)}{l-1} - O(h)P_m\right] \tag{2.1}$$

Here $m(h,t)$ is the expected number of schemas h at generations t, $f(h)$ is the fitness of schema h and $\overline{f}_t$ is the average fitness at generation t. The genotype length is l, $\delta(h)$ is the defining length and $O(h)$ the order of schema h. P_c and P_m are the probabilities of crossover and mutation respectively.

The schema theorem leads to a hypothesis about the way genetic algorithms work, the building block hypothesis (bbh):

> A genetic algorithm seeks near-optimal performance through the juxtaposition of short, low-order, high performance schemas or *building blocks* (Goldberg, 1989).

This means that genetic algorithms exploit syntactic similarities in the genotype as long as the building blocks (short, high-performance schemas) lead to near-optima. The emphasis on short schemas has important consequences for encoding a problem for genetic search and is detailed in the next chapter.

2.3 Models of Design

Since the study of design is still in a pre-scientific stage, models provide a limited but useful abstraction, less ambitious than theories. That is, a model provides an abstraction of a subset of the phenomenon under consideration. Search is the overarching model used in this thesis and is well-established in both artificial intelligence and design (Simon, 1969). More specifically genetic algorithms are used as a tool to attack the subset of design problems stated in Chapter 1; those that have an evaluation[1] function and can be encoded to generate candidate designs. This section

[1]The evaluation function is also called the fitness function from the analogy with biological fitness

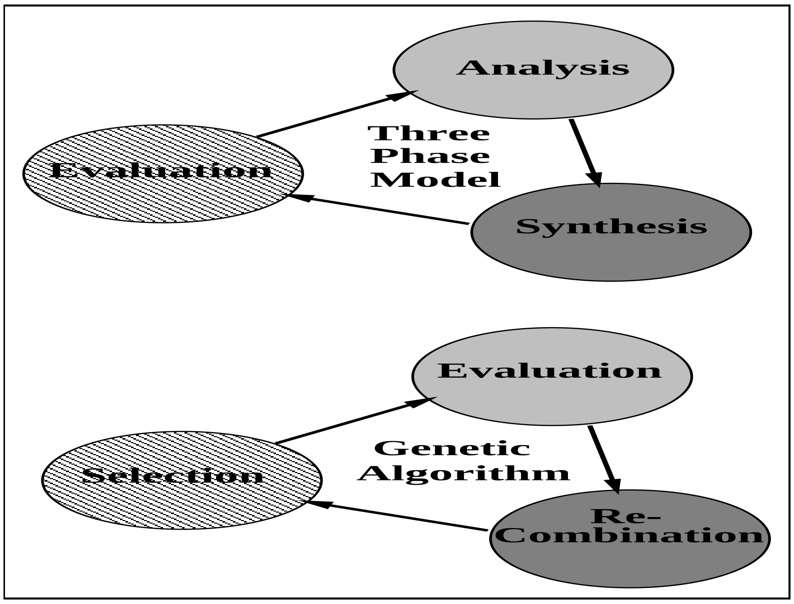

Figure 2.1: The one-to-one correspondence between the three phase model and the genetic algorithm.

maps genetic algorithms onto an old and well-established model of design.

One model that is widely accepted and agreed upon is the three phase model of Asimow (Coyne et al., 1990). The three phases are *analysis, synthesis* and *evaluation* and a designer cycles through these phases until reaching a satisfactory design. Analysis prepares the problem, producing an explicit statement of goals. In GAs, this corresponds to defining the problem-dependent evaluation function and choosing an encoding. Synthesis finds or generates plausible solutions as carried out by crossover and mutation in genetic algorithms. Evaluation in the three phase model judges the validity of candidate solutions and selects among them — selection in genetic algorithms. The mapping is not an exact one but the linkage between the two serves to provide a good starting point since the three phase model has already been mapped to others (Coyne et al., 1990). Figure 2.1 shows the correspondence between genetic algorithms and the three phase model.

The interesting part for design is how new candidate designs are generated. In a genetic algorithm, the search bias is defined by the encoding and the genetic operators of selection, crossover and mutation. These operators make no assumptions about the continuity, differentiability or linearity of the space. Selection highlights promising areas of the space, judging candidate designs on the basis of the evaluation function. Mutation when combined with selection results in a stochastic hill-climber or gradient-descent algorithm. Crossover mixes information from many different areas of the search space and, when paired with selection, allows the combining of good partial solutions into better complete solutions. Before describing the role of the encoding and the evaluation function, this thesis establishes the innovative power of these operators.

2.4 Innovation

Gero and Jo, among others (Gero and Jo, 1991; Tong and Sriram, 1992b), define two types of design activity: 1) routine or parametric design and 2) non-routine design. They further divide non-routine design into innovative and creative design. Innovativeness is characterized by unusual *combinations* of design variables or features while creativity introduces *new* design variables. These definitions and GA theory are used to show how GAs are innovative.

The building block hypothesis states that GAs work by putting together good building blocks. Innovative designers combine concepts or ideas that worked well in certain contexts with other ideas that worked well in other contexts to form new, possibly better ideas which help to perform the current task. In the same way genetic algorithms through the operations of selection and crossover combine many different high-performance schemas (or building blocks) to form new solutions. Mathematically, the schema theorem gives the growth rate of high-performance schemas. As long as the disruption of these schemas due to crossover and mutation (the last two terms in equation 2.1) is sufficiently small, the necessary number of building blocks are expected to be present. Most crossover and mutation operators satisfy this requirement. Holland's second law of genetic algorithms also quantifies the probability that good schemas will combine through the randomized action of crossover. Suppose m building blocks are needed to form a particular solution and each building block has at a least P representatives in the current population of size N. Repeated crossing over with a low disruption crossover operator implies that no less than $(P/N)^m$ individuals have all the required m building blocks together. Thus selection combined with crossover allows building blocks to grow and innovatively combine into design solutions.

2.5 Representing Designs for GAs

In its simplest form, using a genetic algorithm for design is like playing with a Meccano set (a child's construction kit). Given some low level primitives, the task is to put them together so that they perform a certain function. The genotype is a particular specification of the type and arrangement of the primitives while the phenotype is the designed artifact as a whole. The genotype can be thought of as a program which details how to build the phenotypic artifact. Using this simple analogy, a GA used for design can be considered a manipulator. It manipulates low-level "tools," playing with their arrangements, until it finds the required structure. Encoding therefore consists of finding a set of low-level tools for the GA to manipulate. The number of possible codings to choose from may be very large and it is fortunate that GAs are robust and work quite well on arbitrarily chosen encodings. However in addition, the following two principles are often used (Goldberg, 1989).

1. The principle of meaningful building blocks

2. The principle of minimal alphabets

In the context of design, the first principle asks the user to select codings such that the building blocks of the underlying design are small and relatively unrelated to building blocks at other positions. The second principle states that the user should select the smallest alphabet that permits an expression of the design so that the number of exploitable schemas is maximized.

However, in the kind of spaces often encountered in design, a designer may not know whether either principle of GA coding is being followed. The mapping from genotype to phenotype is now much more complex. To get an idea of this complexity, compare the structure of an eye (a design phenotype) with a point in the search space (a phenotype in function optimization) Interdependence in phenotypic space may not be reflected in the genotype, unless it is very carefully encoded, and may cause a GA to be mislead because it violates the first principle. Epistasis, or interrelationships among components, in phenotypic structures therefore plays an important part in determining the suitability of an encoding for design.

At first glance, the second principle may seem to be much simpler to follow. Since digital computers ultimately represent all objects in bits and bytes, the problem reduces to making sure that genetic operators work on this binary representation. In fact, the efficiency of bit operations in digital computers is motivation for using a binary representation for genotypes. However, the problem is to arrange for such an encoding to simultaneously follow the first principle. Our intuition about design spaces may not translate into an understanding of the binary encoded spaces that GAs thrive on. Chapter 4 gives an example of this type of problem, suggests a solution, and presents favorable empirical evidence in support.

2.6 Evaluating Designs for GAs

Evaluating a design measures performance relative to predetermined criteria. If the criteria are well defined, a scalar or vector value can be assigned to a candidate design and compared with others. On the other hand if the problem is ill-structured and the criteria are not well defined, precision may decrease, leading to fuzzy quantization. This may knock some paradigms out of contention, but GAs are remarkably resilient. When using a distributed selection mechanism, such as tournament selection, all that is needed from the evaluation function is a partial ordering on the generated population of design alternatives. If the performance criteria are ill-defined, an interacting designer can bring aesthetic and/or other judgments into the evaluation process and need only compare pairs of candidate solutions indicating which is better.

For example, consider the problem of constructing a composite of a criminal's face by an eyewitness to the crime. Caldwell and Johnston describe a system that uses a genetic algorithm to rapidly search through a search space containing over 34 billion possible facial composites (Caldwell and Johnston, 1991). The eyewitness interactively and subjectively establishes the measure of fitness used to guide search. The algorithm combines building blocks made up of five basic features. The type and position of the forehead, the eyes and their separation, and the shape and position of nose, mouth and chin, respectively. In their example, just 10 generations suffice to generate a composite that is remarkably similar to the criminal.

The genetic algorithm is also quite resistant to noise in the evaluation process and noisy judgments are processed in useful ways (DeJong, 1975). In all cases, once a partial ordering is available, selection can prune the search and concentrate on a computationally tractable subset of the space.

2.7 Example Problem: Floorplanning

This section uses a floorplanning problem (Radford and Gero, 1988) to ground the issues discussed so far. Consider the apartment floorplan shown in figure 2.2. The problem is to find the length and width of each room that will minimize the cost of the apartment, subject to some constraints. The cost for each room in this case is the area except for the cost of the kitchen and bathroom, whose costs are twice their respective areas. The constraints are:

	Length		Width		Area		
Room	Min	Max	Min	Max	Min	Max	Proportion
Living	8	20	8	20	120	300	1.5
Kitchen	6	18	6	18	50	120	any
Bath	5.5	5.5	8.5	8.5			
Hall	5.5	5.5	3.5	6	19	72	any
Bed1	10	17	10	17	100	180	1.5
Bed2	9	20	9	20	100	180	1.5
Bed3	8	18	8	18	100	180	1.5

Further, there must be a space — 3.0 units — for a doorway in the walls connecting bed2 and bed3 to the hall, and all rooms are rectangular (as is the entire plan).

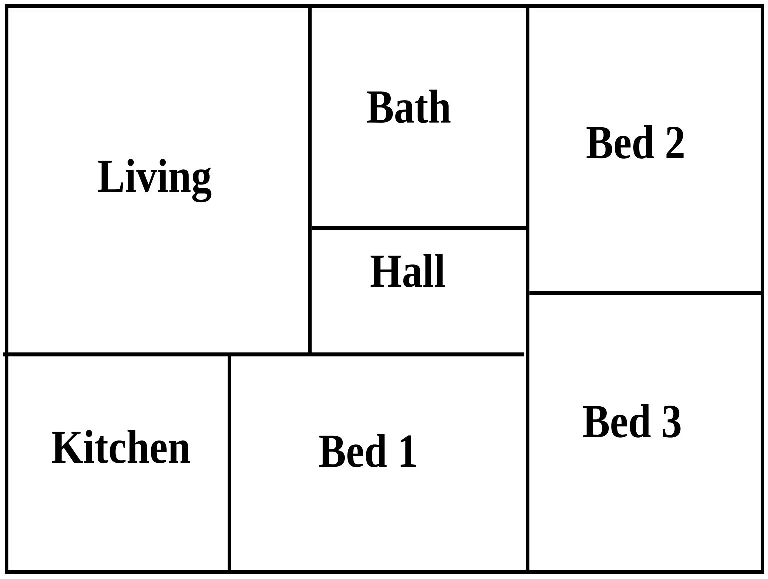

Figure 2.2: Dimensionless floor plan of a three bedroom apartment

This design problem is a nonlinear optimization problem and can be formulated mathematically as follows. Let length be the horizontal dimension and width the vertical dimension, then the variables are given in the table below.

Room	Length/Width	Label
Living	length	X_1
Living	width	X_2
Kitchen	length	X_3
Kitchen	width	X_4
Bed1	length	X_5
Bed2	length	X_6
Bed2	width	X_7
Bed3	width	X_8

Although at first glance there appear to be fourteen variables, one each for the length and width of each room, *analysis* of the problem reveals that some of the variables can be computed from others. For example the width of the hall is the width of the living room minus the width of the bathroom.

The design task is to minimize the cost of the apartment, that is, minimize

$$\begin{aligned} F = X_1X_2 + 2X_3X_4 + 93.5\,(\text{cost of bath}) + \\ 5.5(X_2 - 8.5) + X_4X_5 + X_6X_7 + \\ X_6X_8 \end{aligned}$$

subject to the following constraints:

$$\text{Living} \quad \begin{aligned} X_1, X_2 &\leq 20 \\ X_1, X_2 &\geq 8 \\ X_1X_2 &\leq 300 \\ X_1X_2 &\geq 120 \\ X_1/X_2 &\leq 1.5 \end{aligned}$$

$$\text{Kitchen} \quad \begin{aligned} X_3, X_4 &\leq 18 \\ X_1, X_2 &\geq 6 \\ X_1X_2 &\leq 120 \\ X_1X_2 &\geq 50 \end{aligned}$$

$$\text{Hall} \quad \begin{aligned} 5.5(X_2 - 8.5) &\leq 72 \\ 5.5(X_2 - 8.5) &\geq 19 \end{aligned}$$

$$\text{Bed1}\quad \begin{array}{lcl} X_5, X_4 & \leq & 17 \\ X_5, X_4 & \geq & 10 \\ X_5 X_4 & \leq & 180 \\ X_5 X_4 & \geq & 100 \\ X_5/X_4 & \leq & 1.5 \end{array}$$

$$\text{Bed2}\quad \begin{array}{lcl} X_6, X_7 & \leq & 20 \\ X_6, X_7 & \geq & 9 \\ X_6 X_7 & \leq & 180 \\ X_6 X_7 & \geq & 100 \\ X_6/X_7 & \leq & 1.5 \end{array}$$

$$\text{Bed3}\quad \begin{array}{lcl} X_8, X_6 & \leq & 18 \\ X_8, X_6 & \geq & 8 \\ X_8 X_6 & \leq & 180 \\ X_8 X_6 & \geq & 100 \\ X_6/X_8 & \leq & 1.5 \end{array}$$

$$\text{Doorways in hall}\quad \begin{array}{lcl} X_7 - 8.5 & \geq & 3 \\ X_8 - X_4 & \geq & 3 \end{array}$$

In addition the whole plan is rectangular therefore:

$$\text{Wall alignments} \quad \begin{array}{rcccl} X_1 + 5.5 & = & X_3 + X_5 & & \\ 8.5 + (X_2 - 8.5) + X_4 & = & X_7 + X_8 & = & X_2 + X_4 \end{array}$$

To use a genetic algorithm on this problem these constraints must be incorporated in the encoding or the objective function F, and the design variables ($X_1 \ldots X_8$) have to be represented in a form suitable for a GA. That is, each variable must be encoded in a bit string and the concatenated string of all variables forms the genotype. Since the minimum and maximum value of each variable is given, only the range (the difference between the maximum and minimum values) needs to be represented. The maximum range is 12 and as an accuracy of about 0.5 units is sufficient, 11 bits are used to encode the variables with a range of 12 and 10 bits for the rest. 11 bits can be used to represent values from 0 to 2048 and must be mapped to the values from 0 to 12, thus if y_1 represents the binary encoded value of the first variable's (X_1's) range, the length of the living room can be computed through

$$X_1 = \mathrm{Min}(X_1) + \frac{100.0 \times 12.0}{y_1}$$

where $\mathrm{Min}(X_1) = 8.0$, the minimum length of the living room. The genetic algorithm therefore codes for the range from 0 through 12 in increments of 12.0/20.48. Increasing the number of bits for y_1 by 1 would result in a precision of 12/40.96. In this way the constraints on the maximum and minimum lengths and widths of rooms are assimilated in the encoding.

Violation of constraints on area, proportion, door and wall alignments take the form of penalties which increase the total cost. Thus the function to be minimized

— the objective or *evaluation* function — becomes:

$$
\begin{aligned}
F = X_1X_2 + 2X_3X_4 + 93.5\,(\text{cost of bath}) + \\
5.5(X_2 - 8.5) + X_4X_5 + X_6X_7 + \\
X_6X_8 + \text{penalties}
\end{aligned}
$$

A genetic algorithm can now work on the encoded design problem. Starting with an initial random population of solutions encoded as described above, selection *evaluates* candidate solutions, while crossover and mutation *synthesize* new solutions. The minimum cost found by sequential linear programming, an optimization technique that first converts the problem to a linear one and then solves it, is 715.98 units and this solution violates some of the constraints. In experiments with a population size of 30, crossover probability of 0.9, and a mutation probability of 0.01, genetic algorithms find solutions that have costs from 689.30 to 752.92 with various degrees and types of constraint violations. This allows an engineer or architect to choose from among the solutions, produced in less then a minute on a work station.

2.8 Summary

This chapter has illustrated how genetic algorithms model the design process. They mirror the three stage model of analysis, synthesis and evaluation searching through a large space of possible designs for near-optimal structures. Noisy environments, perhaps a side effect of ill-structured problems, do not cause major problems. In interactive systems where human designers can help evaluate ill-defined criteria (beauty, elegance etc.), GAs only need pairwise comparisons of candidate solutions and do not need other more time consuming evaluations. Finally, a floorplanning problem was used to illustrate the issues involved in analyzing and encoding a problem for a genetic algorithm.

Nevertheless, there are problems. Usually, domain information resides in the evaluation function, but performance of a GA depends on the suitability of the encoding. That is, whether it follows the two principles of GA encodings. Domain knowledge also plays a large part in deciding if a particular encoding is appropriate. But given the same problem and encoding, when is a genetic algorithm expected to perform better than other blind search algorithms? Next, whatever the underlying knowledge-base and its representation, there is first the problem of encoding this domain knowledge for a GA. Third, although GAs may be expected to invent new designs, analysis of why the design works and what are its strong and weak points is difficult. Finally, the time to convergence — the time that a GA should run — must be bounded before a *computational* tool can be advanced.

3

Comparing Genetic and Other Search Algorithms

'Life is very strange' said Jeremy. 'Compared to what?' replied the spider.

–Anon

This chapter compares genetic algorithms with other search algorithms. It defines a class of search spaces which are easy for genetic algorithms and hard for stochastic hill-climbers. The defined spaces require genetic recombination for successful search and are partially deceptive. A deceptive space for a search algorithm leads the algorithm *away* from the global optimum and *towards* a local optimum. Problems where tradeoffs need to be made subsume spaces with these properties. Results comparing a genetic algorithm without crossover against one with two-point crossover support these claims. Further, a genetic algorithm using pareto optimality for selection outperforms both. In problems with more than one criterion to be optimized, pareto optimality provides a way to compare solutions without the need to combine the criteria into a single fitness measure. These results provide insight into the kind of spaces where recombination is necessary, suggesting further study of properties of such spaces, and when to expect genetic algorithms to outperform others.

3.1 The Importance of Recombination

Recombination plays a central role in genetic algorithms and constitutes one of the major differences between GAs and other applicable algorithms. Therefore, the answer to the question of whether a search space is particularly suited to a genetic algorithm hinges on the power of recombination in generating an appropriate search bias. This chapter defines one class of spaces that requires genetic recombination for successful search. The class includes problems where a particular kind of tradeoff exists and is exemplified by design problems in engineering and architecture.

Recent work on GA-easy and "royal-road" functions suggests that it is not easy to find spaces where GAs emerge a clear winner when compared with a stochastic hill-climber (SHC) or a population of stochastic hill-climbers (Wilson, 1991; Mitchell and Forrest, 1993). The work suggests that it is not enough that optimal lower order schemas combine to form optimal higher order schemas. Hill-climbers often solve such problems in a smaller number of function evaluations than a genetic algorithm. Instead, research on deception provides insight into the kinds of spaces that are *deceptive* (Goldberg, 1989a) for SHCs or a population of SHCs and that are relatively easy for genetic algorithms. Identifying properties of spaces that are easy for genetic algorithms and hard for stochastic hill-climbers should provide insight into the inner workings of a GA and aid in the practical selection of search algorithms.

The next section defines the model of a stochastic hill-climber used in this chapter.

Then a problem that is partially deceptive and needs recombination to overcome deception is designed. Spaces with structure similar to the problem defined in section three may be found in multicriteria optimization. Pareto optimality, used to compare multicriteria solutions, is defined in section four and used in selecting candidates for recombination in a genetic algorithm. Empirical evidence in section five compares performance on the problem type defined in section three.

3.2 GAs, Hill-climbers and Recombination

The model of a classical genetic algorithm in this chapter assumes proportional selection, two-point crossover and a mutation operator that complements a bit at a randomly selected position. Proportional selection selects a candidate for reproduction with probability proportional to the fitness of the candidate. That is, if f_i is the fitness of individual i, its probability of selection is

$$\frac{f_i}{\sum\limits_{i=0}^{i=N} f_i}$$

where N is the population size. Two point crossover is like one-point crossover (figure 1.3), except that *two* points are randomly chosen and the substring defined by those points exchanged to produce offspring. For experimental purposes a population of stochastic hill-climbers is a genetic algorithm without crossover. In this chapter "hill-climber" and "stochastic hill-climber" are used interchangeably and problems are cast as maximization problems in hamming space, eliding the issue of encoding.

A genetic algorithm without crossover and only mutation can be likened to an *Evolution Strategy* (Bäck et al., 1991), albeit a simple one with a nonvarying mutation rate and random bias. A GA without crossover is a stochastic hill-climber and, depending on the rate of mutation, can make jumps of varying sizes through the search space. Being population-based and stochastic, it makes fewer assumptions about the search space, and as such is more robust and harder to deceive than the bit-setting optimization and steepest-ascent optimization algorithms defined in Wilson's paper (Wilson, 1991). A bit-setting optimization algorithm assumes that a point in the search space is represented by a bit string S of length l. The algorithm flips each bit, remembering the settings that led to improvements. The solution S' consists of the better settings of each bit. Steepest-ascent optimization accepts S' as the starting point for iterating bit-setting optimization. This continues until no improvements can be made.

Although there is debate in genetics as to the evolutionary viability of crossover in nature, it suffices to note that crossover does exist in biological systems. In genetic algorithms, Holland's schema theorem emphasizes the importance of recombination and provides insight into the mechanics of a GA (Holland, 1975). Recombination is the key difference between the genetic algorithm and the hill-climbers defined above. Recombination allows *large, structured* jumps in the search space. In other words, it facilitates combination of low-order building blocks to form higher-order building blocks, if they exist. The type of combinations facilitated depends on the crossover operator and population distribution. In a random population of strings (encoded points), large jumps in the search space are possible. However, the size of jumps due to crossover in a GA depends on the average hamming distance between members of the population (hamming average) or the degree of convergence of the population. The hamming distance between two binary strings of equal length is the number of corresponding positions that differ. For example, the hamming distance between 1011 and 0010 is 2 since the bits in the first and last positions differ. As the population converges, the usual and corresponding decrease in hamming average of the population leads to a decrease in maximum jump size and exploration of a correspondingly smaller neighborhood. But this kind of neighborhood exploration can be carried out by a GA with mutation only, that is, a stochastic hill-climber.

During the early stages of GA processing, large jumps, combinations of building blocks that are hamming dissimilar, are more probable since the hamming distance between possible mates is higher. The GA should search effectively in spaces where such combinations of building blocks lead toward higher-order optimal schemas. But at this early stage, not only is the variance in estimated fitnesses of schemas high, but there may be enough diversity in a population of hill-climbers to stumble on the correct higher-order schema. Both the order of early building blocks and diversity of the population need to be considered in analyzing recombination.

The discussion above implies that for recombination to be effective and hill-climbing ineffective, a GA needs to be able to make large, *useful*, structured jumps through recombination even after partial convergence.

3.3 A Hill-climbing Deceptive Problem

Consider the function shown in figure 3.1. For this four-bit function, the global optimum is at 1111, but there are two hill-climbing deceptive optima at 0011 and 1100, the global optimum is 2 bits away from the suboptima. Figure 3.2 gives another rough, but perhaps more intuitive, view of a space that should be easy for a GA. Broadening the base of the peak in figure 3.2 makes the function easier on hill-climbers and genetic algorithms alike. The function in figure 3.1 can be specified for an l length string as follows: let x_1 be the number of ones on the right half of the string ($l/2$ bits), x_2 be

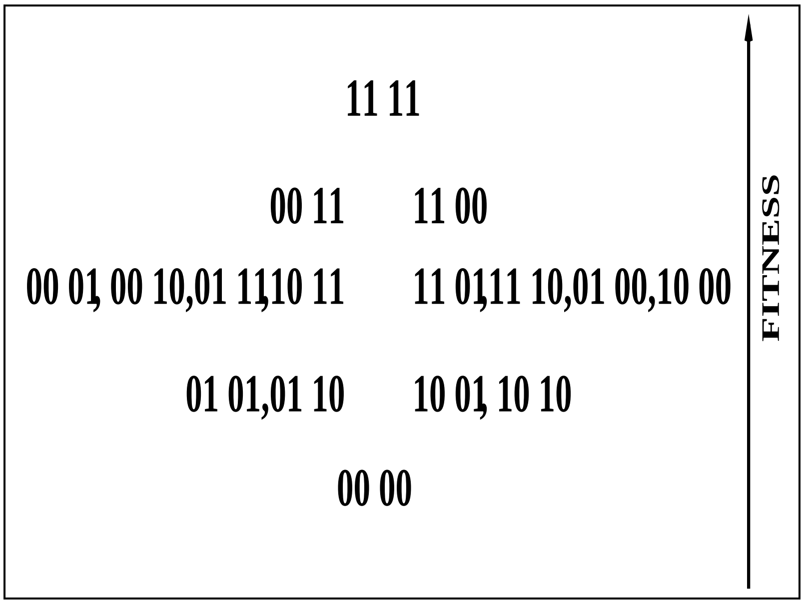

Figure 3.1: A hill-climbing deceptive function that should be easy for GAs

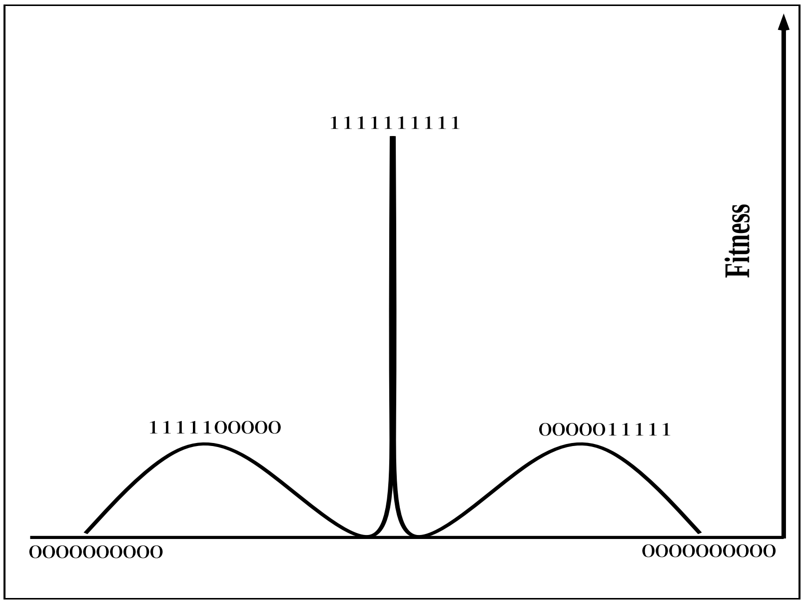

Figure 3.2: An alternate view of a hill-climbing deceptive space that should be easy for GAs (10 bits)

the number of ones on the left half of the string. Using | | to denote the absolute-value function, and p the size of the base of the peak,

$$
\begin{array}{ll}
\text{if} & \\
& (x_1 + x_2) \geq (l - p) \\
\text{then} & \\
& \text{fitness(string)} = (x_1 x_2)^2 \\
\text{else} & \\
& \text{fitness(string)} = \mid 2x_2 - x_1 \mid + \mid 2x_1 - x_2 \mid
\end{array}
\tag{3.1}
$$

Intuitively, counting from one side (left or right), fitness increases as the number of 1's increases until the halfway point of the string, after this point, fitness decreases as the number of 1's increases until reaching the area near the global optimum. Here the fitness increases sharply, peaking at the global optimum. The squaring of the values near the peak tries to ensure that once found by a classical genetic algorithm, the peak will not be lost easily.

For sufficiently large l this type of function is deceptive for a SHC or population of SHCs since for a string of length l, once a stochastic hill-climber reaches a deceptive optimum, it will have to simultaneously set all the remaining $l/2$ (approximating to within p bits) bits to 1 to attain the global optimum. Setting fewer than $l/2$ bits decreases fitness. The probability of simultaneously setting $l/2$ bits to 1 decreases exponentially with l, thus making it hard on a SHC or a population of SHCs for sufficiently large l. Using this chapter's model of a stochastic hill-climber with mutation probability p_m, after reaching the suboptima, the probability of setting $l/2$ bits is: the probability that $l/2$ incorrect bits are flipped, multiplied by the probability that the correctly set bits are not flipped

$$(p_m)^{l/2}(1-p_m)^{l/2}$$

Now consider the power of recombination. Once a genetic algorithm reaches the hill-climbing deceptive optima, crossover can produce the global optimum in one operation, *assuming* that the population contains representatives of both suboptima. Genetic operators that are conducive to maintaining stable subpopulations at the peaks of a multimodal function are called niching operators (see (Deb and Goldberg, 1989) for a survey of niching operators). Niching operators can be used to provide representatives at the suboptima as long as these operators do not entail mating restrictions which may then not allow the algorithm to find the global optimum.

Next, consider whether this type of space may be found in normal application domains. In many areas of engineering, a problem is usually stated in terms of multiobjective (or multicriteria) optimization, and the function above can be easily and naturally stated as a two-objective optimization problem. This dissertation suggests that multicriteria optimization problems may provide fertile ground for finding search spaces in which genetic algorithms do better than other blind search algorithms. However, multicriteria optimization raises a point. The issue for the algorithms presented earlier is that, in general, it is difficult to combine multiple measures into a single fitness in a reasonable way without making assumptions about the relationships among the criteria. Unfortunately, a single fitness measure is the usual feedback for genetic algorithms and for the niching methods in (Deb and Goldberg, 1989). For example, consider designing a beam section where the objectives are to maximize the moment of inertia and minimize the perimeter. Minimizing the perimeter usually decreases the moment of inertia and maximizing the moment of inertia increases the perimeter, resulting in a tradeoff (Gero et al., 1993). Combining the two criteria into a single measure makes no sense unless the relationship between the two is already known. Fortunately, the concept of *pareto optimality* allows us to do this reasonably and, as will be seen, provides a ready-made niching mechanism, killing two birds with one stone.

3.4 Pareto Optimality and Selection

Pareto optimality operates on the principle of nondominance of solutions. An n criteria solution s_1 with values $(c_1, c_2, \ldots, c_n)$ dominates s_2 with values $(d_1, d_2, \ldots, d_n)$ if

$$\forall_i (c_i \geq d_i) \text{ AND } \exists_i (c_i > d_i)$$

The set of pareto optimal solutions, the pareto-optimal set, is the set of non-dominated solutions. Figure 3.3 shows the pareto-optimal set for a two-criteria problem. An axis represents values for a particular criterion. The coordinates of each point (solution) denote the values of their respective criterion. X's in the figure represent dominated points. The O's represent non-dominated points that belong in the pareto-optimal set.

Considering the maximal value along each criteria to be one of our hill-climbing deceptive optima, there is a correspondence between the function defined in section 2 and multicriteria optimization problems. If a central peak (global optimum) exists, as in figure 3.2, then recombination should find it, if both suboptima are represented in the population. Incorporating pareto optimality into the selection mechanism of a genetic algorithm facilitates the maintenance of stable subpopulations representing the suboptima.

A variant of binary tournament selection is used to incorporate pareto optimality in the genetic algorithm. When using this variant of binary tournament selection, the algorithm selects two individuals at random from the current population, mates them, and produces the pareto-optimal set of the parents and offspring. Two random individuals from this pareto-optimal set form part of the next population. The procedure repeats until the new population fills up, thus becoming the next current population. This method with a tournament size of 2 is computationally inexpensive since only 4 solutions compete at a time.

The genetic algorithm with pareto-optimal selection (pareto-optimal GA) is used to attack the problem in section 3.3. First, the problem is split into two, corresponding to the two additive terms in equation 3.1. Cast as a 2 criteria optimization problem, the first criterion is

$$\begin{array}{l} \text{if} \\ \qquad (x_1 + x_2) \geq (l - p) \\ \text{then} \\ \qquad c_1(\text{string}) = x_1 x_2 \\ \text{else} \\ \qquad c_1(\text{string}) = \mid 2x_1 - x_2 \mid \end{array} \tag{3.2}$$

where x_1 stands for the number of 1's in one half of the string and x_2 the number of

1's in the other half of the string. The second criterion c_2 then becomes

$$
\begin{aligned}
&\text{if} \\
&\qquad (x_1 + x_2) \geq (l - p) \\
&\text{then} \\
&\qquad c_2(\text{string}) = x_1 x_2 \\
&\text{else} \\
&\qquad c_2(\text{string}) = \mid 2x_2 - x_1 \mid
\end{aligned}
\tag{3.3}
$$

The point of decomposing the original problem into a two-criteria problem is not to suggest that functions in general can be decomposed to form multicriteria problems. Rather, the argument is that many design problems are best posed as multicriteria optimization problems and thus provide search spaces that may be easy for genetic algorithms.

3.5 Results

This section compares the number of times the optimum is found by a GA without crossover, a classical GA with two-point crossover (CGA) and a pareto-optimal GA with two-point crossover. The population size in all experiments was 30 and the results were averaged over 11 runs of the algorithm with different random seeds. The GA with mutation ran with mutation probability from 0.01 to 0.10 in increments of 0.01 and the best results used. The probability of crossover was 0.8 and the mutation rate 0.02 for the other two algorithms.

Figures 3.4 and 3.5 compare the maximum fitness over time for the three algorithms. The algorithms ran for 200 generations or 6000 function evaluations. The

Length	Width	PGA	CGA	PSHC
20	1	100 %	55 %	45 %
30	2	100 %	27 %	18 %
40	3	73 %	0 %	0 %
50	4	45 %	0%	0%

Table 3.1: Comparing the number of times the optimum was found in 200 generations for narrow peaks.

string length was 30, the parameter p in the definition of the function was set to 2 to produce figure 3.4 and to 4 to produce figure 3.5. The difference in performance of the pareto-optimal GA provides empirical support for the importance of recombination and the niching effect of pareto-optimal selection. As expected the width of the peak affects performance, with wider peaks allowing the GA with two-point crossover to perform significantly better than the stochastic hill-climber. The plot for the pareto-optimal GA shows the average fitness for only one of the two criteria, hence the low (about 1/2 the value of the other two plots) initial values.

Table 3.1 compares the percentage of times the algorithms found the optima for strings of length 20, 30, 40 and 50 for peaks of width 1, 2, 3 and 4. In the table, peak size is given in column 2, PGA is the pareto-optimal GA while CGA is the GA with two-point crossover and PSHC is the GA with no crossover — a population of stochastic hill-climbers. Table 3.2 depicts the same information but for wider peaks.

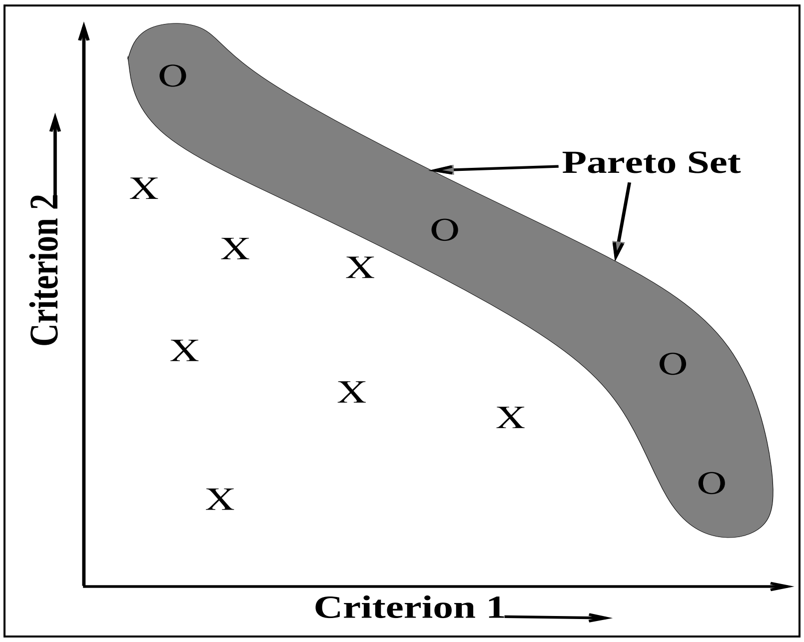

Figure 3.3: A graphical representation of solution points on a two-criteria optimization problem. The points within the shaded area represent the pareto optimal set of solutions.

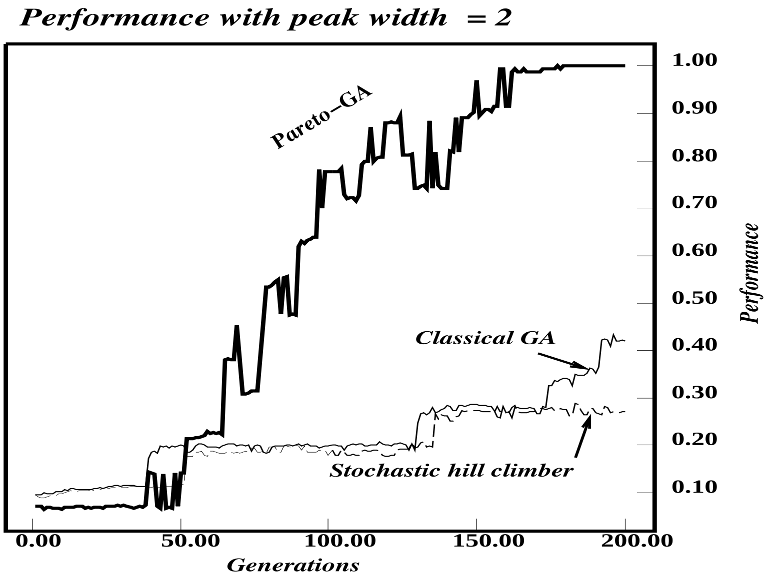

Figure 3.4: Performance comparison for string length 30. The peak width is 2.

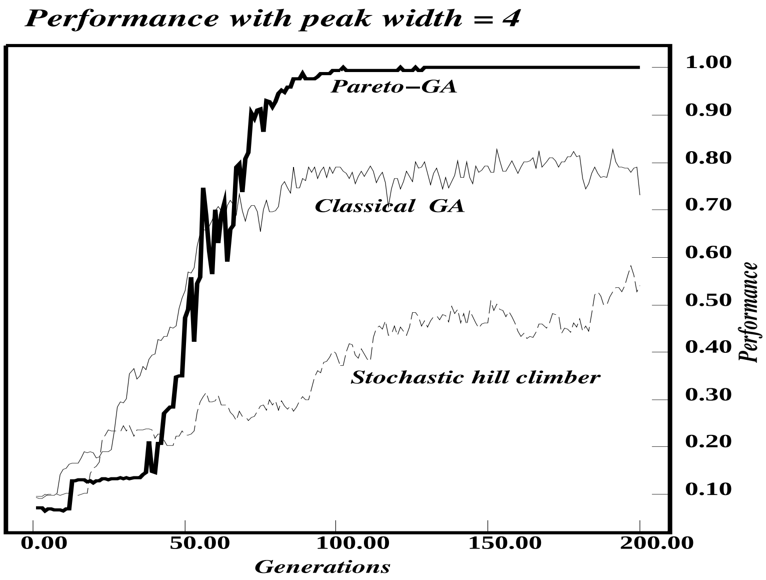

Figure 3.5: Performance comparison for string length 30 but with peak width 4.

For table 3.1, statistical testing using a χ^2 test of significance for comparing proportions indicates that the PGA is significantly better at a confidence level greater than 95% for the length-20 problem and a significance level greater than 99% for the others. The difference in performance of the CGA and PSHC is not statistically significant for any of the problems in this table. Table 3.3 shows the significance and confidence levels for the data in table 3.2. Comparing performance using date for the wider peaks given in table 3.2, the χ^2 test indicates that the differences in performance between the PGA and CGA and between the CGA and PSHC on the 30 bit problem are statistically significant. The confidence level is greater than 75%. On the 40 and 50 bit problems the pareto-optimal GA significantly outperforms the other algorithms.

These results strongly indicate that a pareto-optimal GA is more likely to find the optimum on problems of this type, followed by a GA with crossover. The stochastic hill-climber appears least likely to find the optimum and in this case had less than

Length	Width	PGA	CGA	PSHC
20	2	100 %	91 %	100 %
30	4	100 %	82 %	55 %
40	6	100 %	64 %	0 %
50	8	82 %	0%	0%

Table 3.2: Comparing the number of times the optimum was found in 200 generations for broader peaks.

Length	Width	PGA vs CGA Significant?	CGA vs PSHC Significant?
20	2	No	No %
30	4	Yes (75%)	Yes (75%)
40	6	Yes (95%)	Yes (95%)
50	8	Yes (99%)	No

Table 3.3: Statistical Significance of differences in performance according to the χ^2 test of significance. Confidence levels are in parenthesis.

a 50% chance, even on the smaller length-20 problem when the peak was narrow. Wider peaks help the hill-climber for small problems (length 20, peak width 2) but as the problem size increases, the hill-climber's performance deteriorates to become worse than even a classical GA. In all cases, the pareto-optimal GA does as well as, or better than, the other two algorithms.

3.6 Discussion

A preliminary analysis assumes that the algorithms need to reach the local optima to be able to jump to the global optimum. For the function type defined in this chapter, as the length of the deceptive basin between a local optimum and the global optimum decreases, hill-climbers should perform better. A break-even point for a function shaped like the one in figure 3.2 occurs when the chances of reaching the global optimum from one of the local optima become greater than 1/2. From the earlier analysis the probability that a stochastic hill-climber reaches the global optimum is

$$(p_m)^{l/2}(1 - p_m)^{l/2}$$

where p_m is the probability of mutation and l the length of the string. Let d stand for the length of the deceptive basin ($l/2$ is the length of the deceptive basin above). Figure 3.6 is a copy of figure 3.2 but with the deceptive basin size d clearly marked. Plotting the function for various values of p_m and d results in figure 3.7.

The plot (figure 3.7) indicates that the length of the deceptive basin plays a major part in determining success for a stochastic hill-climber. Only for small values

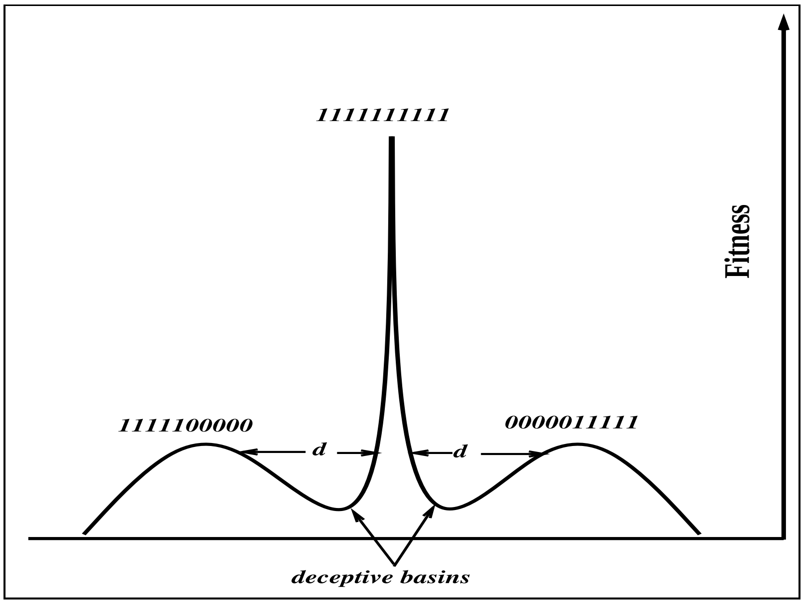

Figure 3.6: The shape easier for GAs and harder for hill-climbers, d denotes the size of the deceptive basin.

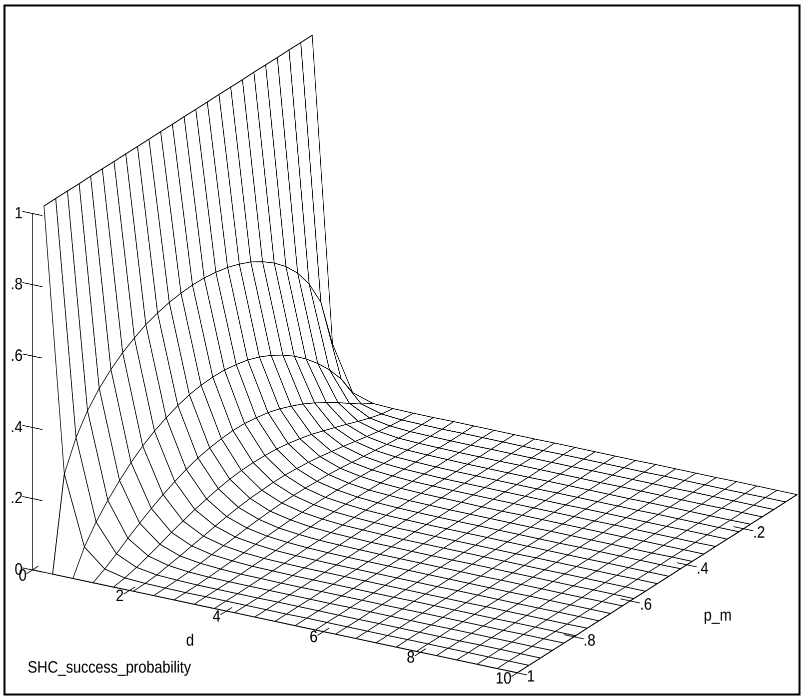

Figure 3.7: The vertical axis indicates the probability of finding the global optimum in terms of mutation rate (p_m) and length of deceptive basin (d).

of d are the chances of finding the optimum appreciable over a range of mutation values. For a genetic algorithm using one-point crossover, the probability that the global optimum is found assuming equally fit local optima is roughly the probability that the mates chosen represent the two local optima, multiplied by the probability that crossover produces the global optimum. Assuming that the genetic algorithm population only contains representatives of the local optima in equal proportions, the probability of finding the global optimum is $\frac{1}{2l}$. This simplified analysis assumes that the algorithms need to reach the local optima to be able to jump to the global optimum. The assumption is not strictly true. Analysis becomes much more complex, depending on the detailed behavior of the objective function, when disallowing this assumption. However, macroscopic behavior, like the importance of deceptive basin length (d) is predictable.

3.7 Summary

Design problems are often formulated as multiobjective or multicriteria optimization problems. In many cases such problems involve tradeoffs among possibly conflicting criteria. Spaces where recombination of two or more single criterion optima leads toward a global optimum, with a deceptive basin in between to trap hill-climbers, are well suited for genetic algorithms especially when using pareto optimality in their selection process. Pareto-optimal selection appears to provide the necessary niches until recombination produces the global optimum. This chapter defined a problem in hamming space with these properties and empirically compared the performance of a classical GA with two-point crossover to that of a stochastic hill-climber (a GA without crossover), and a GA with pareto-optimal selection. The GA with pareto-optimal selection always found the optimum more often than the others, while the classical

GA usually did better than the stochastic hill-climber. This evidence suggests a closer look at the spaces defined in this chapter for function encoding combinations that are relatively easy for genetic algorithms and hard for hill-climbing methods.

Pareto-optimal selection also eliminates the need to combine disparate criteria into a single fitness as is usual in genetic algorithms. The method described in this chapter is distributable and computationally less expensive compared to other suggested methods for incorporating pareto optimality (Goldberg, 1989b). Exploring niche formation with pareto-optimal selection remains an interesting area for further research.

This chapter did not consider encodings since all problems were cast as optimization problems in hamming space. However, encoding a problem for a genetic algorithm is an important issue, with the biases generated by an encoding playing a significant part in performance. The next chapter attacks this problem.

4

Genetic Algorithm Encodings

Each type of knowledge needs some form of "representation" and a body of skills adapted to using that style of representation.

– Marvin Minsky. *Society of Mind.* 1985

This chapter describes the assumptions behind the principle of meaningful building blocks and defines a new crossover operator that relaxes these assumptions. The new operator makes choosing a GA encoding easier, paving the way for applying GAs in design. The chapter uses design problems from combinational circuit design (designing adders and parity checkers) to illustrate the results.

4.1 Search Bias

The search bias during genetic search depends on: the problem, the structure of the encoded search space for the problem, and the genetic operators of selection, crossover, and mutation. For every problem there are a large number of possible encodings. It is often possible to follow the principle of minimal alphabets when choosing an encoding for a genetic algorithm, but simultaneously following the principle of meaningful building blocks can be much harder. This is because our intuition about the structure of the problem space may not translate well in the binary-encoded spaces that genetic algorithms thrive on.

There are two possible ways of tackling the problem of coding design for meaningful building blocks.

1. Search through possible encodings for a good one while searching for a solution.
2. Expand the number of good encodings by increasing the types of building blocks considered *meaningful*.

The first choice uses reordering operators, like inversion, that change the encoding while searching for the solution and is discussed next. The rest of the chapter motivates, develops and illustrates the second approach.

4.1.1 Inversion

Inversion rearranges the bits in a string allowing linked bits to move close together (see figure 4.1). Inversion occurs in nature and serves a similar function. Genes and their alleles are linked if their expression is dependent on one another. Tight linkage is established when linked alleles are close together and such alleles are called co-adapted alleles. Epistasis is the term used to describe the degree of linkage among alleles. In genetic algorithms, inversion is implemented by changing the encoding to carry along a tag which identifies the position of a bit in the string (Goldberg, 1989b).

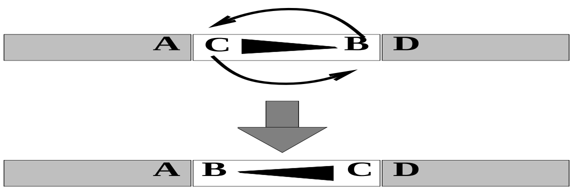

Figure 4.1: Inversion. Notice that after inversion "A" is close to "B"

With the tags specifying position, it is now possible to cut and splice parts of a string allowing bits to migrate and come together. Inversion-like reordering operators have been implemented by Goldberg and others (Goldberg and Lingle, 1985; Smith, 1985) with some good results.

The problem with using inversion and inversion-like operators is the decrease in computational feasibility. If l is the length of a string, inversion increases the search space from 2^l to $2^l!$. Natural selection has geological time scales to work with and therefore inversion is sufficient to generate tight linkage. Designers do not have this amount of time or the resources available to nature.

Instead of using reordering operators, this thesis expands the number of encodings that a genetic algorithm finds useful by increasing the types of building bocks considered meaningful by a genetic algorithm.

4.1.2 Disruption and Crossover

The principle of meaningful building blocks (stated in chapter 2, section 2.5) arises mainly from the search biases generated by the crossover operator. To see why, consider the probability of disruption of a schema. Let H be a schema, $\delta(H)$ its defining length and $O(H)$ its order. Then the probability that the crossover point falls within the schema is $\delta(H)/(l-1)$ where l is the length of the string containing the schema. This is the second term in equation 2.1 and is the probability that crossover will disrupt the schema. Since the probability of disruption is proportional to the defining length, schemas of long defining length tend to be disrupted more often than their shorter counterparts. Consequently, the principle of meaningful building blocks tries to ensure that a chosen encoding does not have long building blocks. However, the mapping from genotype to phenotype is in general much more complex in design, hence it may not be known whether the encoding follows the principle of meaningful building blocks even when the underlying domain is relatively well understood. This means that schemas of arbitrary defining length may need to be preserved and the bias towards short schemas becomes a liability.

One way of combating the problem of disruption of long schemas in the encoding of design problems is to remove the bias toward short schemas and allow low order schemas of arbitrary defining length to bias search in useful directions.

Previous approaches to this problem used a new crossover operator like punctuated crossover or uniform crossover. Punctuated crossover relies on a binary mask, carried along as part of the genotype, in which a '1' identifies a crossover point. Masks, being part of the genotypic string, change through crossover and mutation. Experimental results with punctuated crossover did not conclusively prove the usefulness of this operator or whether these masks adapt to an encoding (Schaeffer and Morishima, 1987; Schaeffer and Morishima, 1988).

Uniform crossover exchanges every corresponding pair of bits with a probability of 0.5. The probability of disruption of a schema is now proportional to the order of the schema and independent of defining length. Experimental results with uniform crossover suggest that this property may be useful in some problems (Syswerda, 1989). However, in design problems the idea is *not* to disrupt a highly fit schema whatever its defining length.

A second point needs to be made. Natural selection works by biasing search in any *direction* that shows the slightest improvement in survivability. This directional information is implicit in the number of competing alleles that exist in a population, and in nature, cannot be stored anywhere. Classical genetic algorithms and their operators, mimicking natural selection, are also bound by these constraints.

In summary, designing an encoding is complex and something of an art. To solve the problem of bad taste on the part of GA programmers, or at least to give them more leeway, this thesis lets the GA exploit encodings that are less constrained. The following sections describe and use a masked crossover operator to remove the bias toward short schemas and to make use of explicitly stored directional information to efficiently bias search.

4.2 Crossover

The assumption that short, low-order schemas are needed to solve a problem is dependent on the encoding and the crossover operator. A strategy that identifies highly fit schemas or building blocks of arbitrary defining length would expand the set of possible encodings with which a GA could work, leading perhaps to a lower aesthetic threshold for GA programmers.

The operator that introduces bias due to the encoding in genetic search is crossover. Various alternatives to the classical crossover operators have been proposed and studied (Schaeffer et al., 1991; Syswerda, 1989). Syswerda's paper concludes that there are no clear winners, but when in doubt, use uniform crossover. It is clear, however, that for all the operators studied, some do better on certain problems and worse on others, indicating that the biases introduced are problem/encoding dependent. An adaptive crossover operator that adapts to the encoding, being able to exploit information unused by classical crossover operators, may be better than classical crossover operators. The cost to be paid for adaptive crossover lies in the added complexity of the operator and the paradoxical point that it can be more easily misled. The more information that an algorithm looks at, the more its trajectory can be affected by bad leads. A genetic algorithm using classical crossover is less likely to be misled than a genetic algorithm using adaptive crossover since classical crossover ignores information used by adaptive crossover and is thus immune to misdirection in this information.

Simple non-adaptive crossover, the classical crossover operator, consists of choosing a point in the string and following the procedure outlined below:

for i **from** 1 **to** crossover-point
begin
  Copy gene **to** child1[i] **from** parent1[i]
  Copy gene **to** child2[i] **from** parent2[i]
end
for i **from** (crossover-point + 1) **to** Chromosome-length
begin
  Copy gene **to** child1[i] **from** parent2[i]
  Copy gene **to** child2[i] **from** parent1[i]
end

One point crossover, and variants thereof, limit the number of encodings that a genetic algorithm can exploit because they assume that contiguous genes form building blocks. With adaptive crossover, this constraint should be relaxed.

4.3 Crossover Bias Modification

For the search process to exploit *any* information in the encoding, the crossover operator needs to be able to preserve highly fit schemas, no matter what their length. This suggests a masking scheme, in which the positions making up highly fit schemas are identified, tagged and preserved. Masks can be propagated in two ways.

1. Consider the masks as an extra set of genes and use the usual genetic operators on them.
2. Use special *mask functions* which redirect bias in search.

When a child is produced, the masks used to produce it may be modified depending on how well the child does relative to the parents. Initial masks can be generated randomly, although making them dependent on the initial chromosome set using any available domain knowledge will probably be more useful.

4.4 Masked Crossover

This dissertation defines an operator that directly makes use of the relative fitness of the children, with respect to their parents, to guide crossover. The *relative* fitness of the children indicates the desirability of proceeding in a particular search direction. The use of this information is not limited to our operator, and can be used in classical GAs with minor modifications (or example, one way to improve traditional crossover

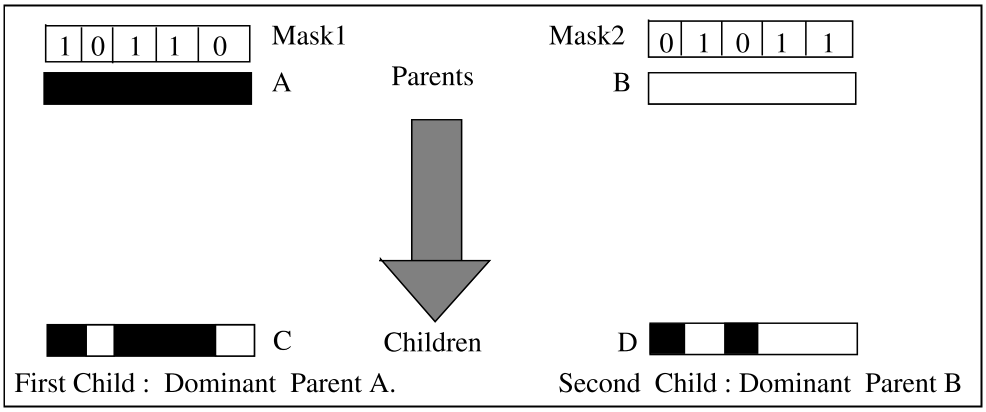

Figure 4.2: Masked crossover. The bits that are exchanged depend on the masks. This allows preservation of schemas of arbitrary defining length

operators is to keep a sorted list of previous crossover points that produced highly fit children).

Masked crossover (MX) uses binary masks to direct crossover. Let A and B be the two parent strings, and let C and D be the two children produced. $Mask1$ and $Mask2$ are a binary mask pair, where $Mask1$ is associated with A and $Mask2$ with B. A subscript indicates a bit position in a string. Masked crossover is shown in figure 4.2 and defined below:

copy A **to** C **and** B **to** D
for i **from** 1 **to** string-length
begin
  if $Mask2_i = 1$ **and** $Mask1_i = 0$
    copy the i^{th} bit **from** B **to** C
  if $Mask1_i = 1$ **and** $Mask2_i = 0$
    copy the i^{th} bit **from** A **to** D
end

First, it is easy to see that traditional crossover operators are special cases of MX. One-point crossover (figure 1.3) can be implemented by simply setting the first n bits ($n <$ length of string) of $M1$ to 1 and the rest to 0, $M2$ being the complement of $M1$. For uniform crossover, the i^{th} bit of $M1$ is decided by a fair coin toss, $M2$ again being the complement. This allows the disruptiveness of MX to range from that of one point to that of uniform crossover.

Masked crossover tries to preserve schemas identified by the masks. Let A be called the dominant parent with respect to C, and B the dominant parent with respect to D. This follows from the definition of masked crossover since during production of C, when corresponding bits of A and B are the same, the bit from A is copied to C.

4.4.1 Masks

Intuitively, 1's in the mask (probabilistically) signify bits participating in highly fit schemas. MX preserves A's schemas in C while adding some schemas from B at those positions that A has not fixed. A similar process produces D. Search biasing is done by changing masks in succeeding generations. Instead of using genetic operators on masks, this thesis uses a set of rules that operate bitwise on parent masks to control future mask settings. Since crossover is controlled by masks, using meta-masks to control mask string crossover then leads to meta-meta masks and so on. Using rules for mask propagation avoids this problem. Choosing the rule to be used is dependent on the fitness of the child relative to that of its parents.

Three types of children can be defined:

The *Good* Child: has fitness higher than that of both parents.

Case	**Rule**
Both good	MF_{gg}
Both bad	MF_{bb}
Both average	MF_{aa}
One good, one bad	MF_{gb}
One good, one average	MF_{ga}
One average, one bad	MF_{ab}

Figure 4.3: Six ways of pairing children and their associated mask functions/rules (MF).

The *Average* Child: has fitness between that of the parents.

The *Bad* Child: has fitness lower than that of both parents, or equal to one or both parents.

With two children produced by each crossover and three types of children, there are a total of 3^2 or nine possibilities, with associated interpretations and possible actions on the masks. However, since the order of choosing children does not matter, the number of cases falls to six (see figure 4.3).

4.4.2 Rules for Mask Propagation

This section specifies rules for mask propagation. In each case a child's mask is a copy of the dominant parent's except for the changes the rules allow. The underlying premise guiding the rules is that when a child is less fit than its dominant parent, the recessive parent contributed bits deleterious to its fitness. Encouraging search in the area defined by these loci, the idea is to search in areas close to one parent with information from the other parent providing some guidance.[1] A mask mutation

[1]Note that in MX, this is done without regard to defining length.

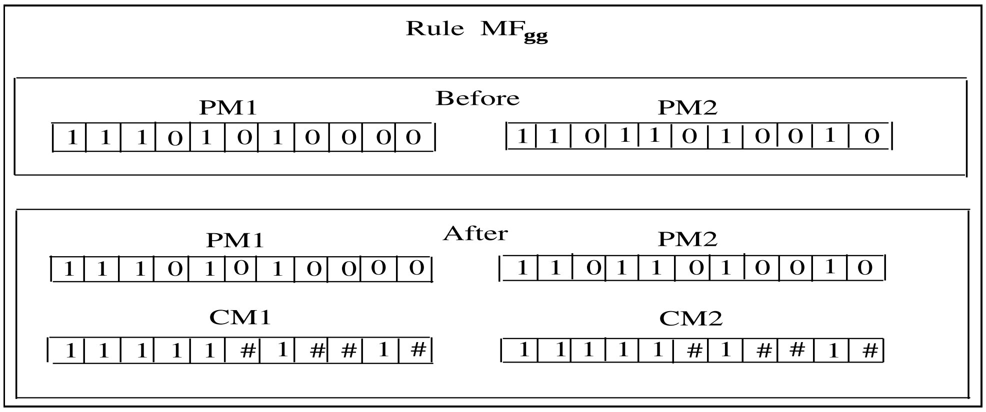

Figure 4.4: Mask rule MF_{gg}: Example of mask propagation when both C1 and C2 are good

operator that flips a mask bit with low probability is assumed to act during mask propagation. To illustrate a mask function and to give some intuition, the mask function for MF_{gg}, when both children are good, is described below. Appendix A contains the complete set of mask rules used in this chapter.

Let $P1$ and $P2$ be the two parents, $PM1$ and $PM2$ their respective masks. Similarly, $C1$ and $C2$ are the two children with masks $CM1$ and $CM2$. The modifications to masks depend on the relative ordering of $P1$, $P2$, $C1$ and $C2$. In this section's figure, the "#" represents positions decided by tossing a coin.

1. MF_{gg}:

   Case: Both children are good.

   Summary: Very encouraging behavior and as such is reflected in the mask settings below and in figure 4.4. 1's in the masks probabilistically identify highly fit schemas. The parents' masks are OR'd to produce the children's masks, ensuring preservation of the contributions from both parents.

   Action:

- $CM1$: OR the masks of $PM1$ and $PM2$. If there are any 0's left in $CM1$, toss a coin to decide their value.
- $CM2$: Same as for $CM1$.
- $PM1$: No changes except for those produced by mask mutation.
- $PM2$: Same as for $PM1$.

The mask functions in appendix A are examples from one of several different sets of possible rules, since many mask propagation rules can be defined. In fact a GA can search the space of mask rules to find a suitable set if an evaluation function can be attached to the masks. This may be overkill, since the number of rules is usually small enough that simpler methods will suffice.

With mask propagation through mask rules, directional information is explicitly stored in the masks and used by the crossover operator to bias search. The main features of the masked crossover operator are then, storage and use of directional information, and independence from defining length of schemas. Think of masked crossover as a good compromise between the disruptiveness of uniform crossover and the bias toward short schemas of classical crossover.

Masked crossover presents a problem when using classical selection procedures. The classical strategy of allowing the children produced to replace the original population will not allow a genetic algorithm using masked crossover to converge. Masks will tend to disrupt the best individuals while searching for promising directions to explore because of the nature of the rules guiding mask propagation. Therefore, the selection procedure used, is a modification of the CHC selection strategy, in which, if the population size is N, the children produced double the population to $2N$. From this population of $2N$, the N best individuals are chosen for further consideration (Eshelman, 1991). This *elitist* selection strategy is used to enhance convergence. Another problem which may occur is that although MX preserves schemas of arbitrary defining length, the fitness information itself may be misleading. Such problems are called *deceptive*. When fitness information is misleading, a GA using masked crossover can be expected to perform worse than a GA using crossover operators that do not use such information. This is borne out by results from the adder problem.

A performance difference due to selection should be expected since fitness information biases CHC selection more than traditional selection. The elitist selection strategy used here results in a stronger focus on exploitation of the search space at the cost of reduced exploration. Less exploration implies a reduced emphasis on crossover (crossover explores the space). Crossover's smaller role leads to decreased epistatic effects since epistasis only influences crossover. Whatever the crossover operator used, the CHC selection strategy should increase performance, but again at the cost of increasing susceptibility to deception. Results presented in the next section illustrate the effects of selection strategy.

A Designer Genetic Algorithm (DGA) therefore differs from a classical genetic algorithm both in the crossover operator (masked crossover) and in the selection strategy (elitist) used.

4.5 Results

A designer genetic algorithm's performance is compared with that of a classical GA on the adder and parity problems. In all experiments, the population is made up of 30 genotypes. The probability of crossover is 0.7 and the probability of mutation is 0.04. These numbers were found to be optimal through a series of experiments using various population sizes and probabilities. The graphs in this section plot maximum

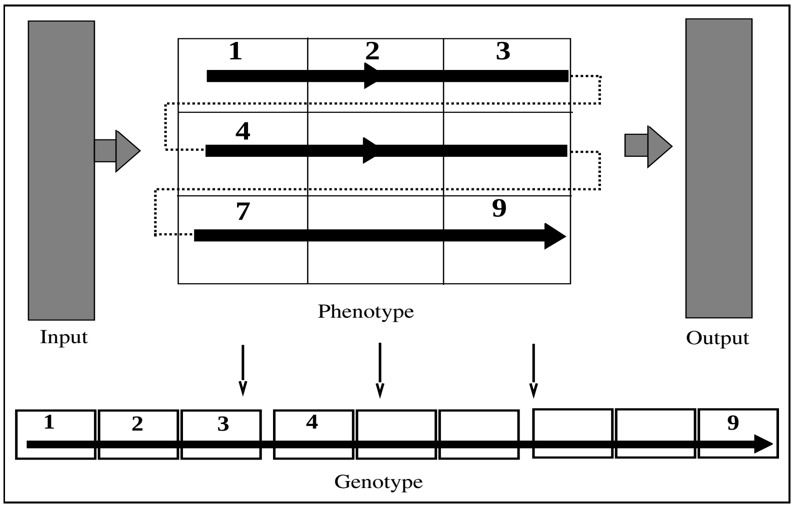

Figure 4.5: A mapping from a two-dimensional phenotypic structure (circuit) to position in a one-dimensional genotype.

and average fitnesses over ten runs.

Each genotype is a bit string that maps to a two-dimensional structure (phenotype) embodying a circuit as shown in figures 4.5 and 4.6. 3 bits are needed to represent 8 possible gates. A gate has two inputs and one output. Considering the phenotype as a two dimensional array of gates S, a gate $S_{i,j}$, gets its first input from $S_{i,j-1}$ and its second, from one of $S_{i+1,j-1}$ or $S_{i-1,j-1}$ as shown in figure 4.6. An additional bit associated with each gate encodes this choice. If the gate is in the first or last rows, the row number for the second input is calculated modulo the number of rows. The gates in the first column, $S_{i,0}$ receive the input to the circuit. Connecting wires are simply gates that transfer one of their inputs to their output. The other gates are AND, OR (inclusive OR), NOT and XOR (exclusive OR).

The fitness of a genotype is determined by evaluating the associated phenotypic structure that specifies a circuit. If the number of bits is n, the circuit is tested on the 2^n possible combinations of n bits. The fitness function returns the sum of the correct responses. This sum is maximized by the algorithm. For the 4-bit parity

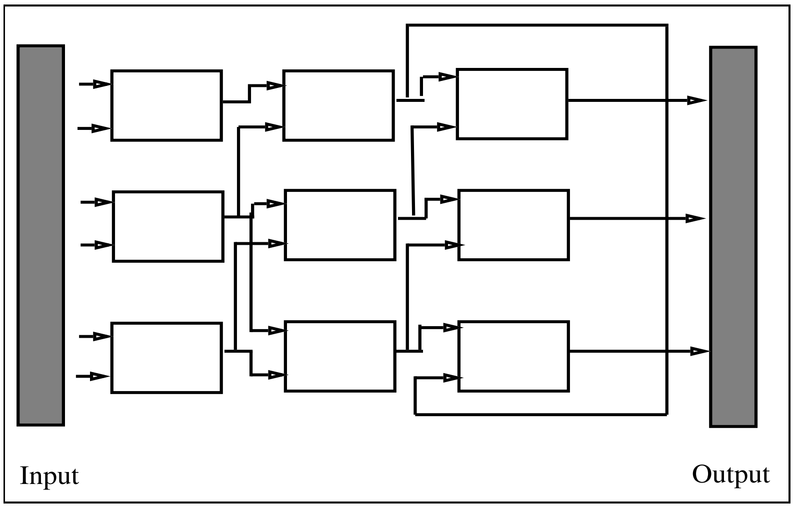

Figure 4.6: A gate in a two-dimensional template gets its second input from either one of two gates in the previous column.

checker, the binary numbers 0 to 15 are inputs to the decoded circuit. If the circuit is correct, its fitness is 16, which means that the circuit correctly finds the parity of all the 16 possible 4-bit numbers. For the adder problem the input is a set of 2, n-bit numbers. The output is an $n+1$ bit sum. For a 2-bit adder, the maximum fitness would be $3 * 2^4$ or 48. Note that in contrast to optimization problems, the maximum fitness is known — the algorithm searches for a structure (circuit) that achieves this maximum fitness.

When comparing the performance of a classical GA using elitist selection with a DGA on a 2-bit adder problem, the graphs in figures 4.7 and 4.8 show that the classical GA does better, although the difference is not great. This is not very encouraging. However, implementing the carry bit makes the solution space of the adder problem deceptive. A problem is deceptive to a GA when highly fit low-order schemas lead away from highly fit schemas of higher order. As explained earlier, since MX uses fitness information to bias search, it is more easily misled than traditional crossover.

Although a problem is deceptive, it does not mean that no solutions can be found.

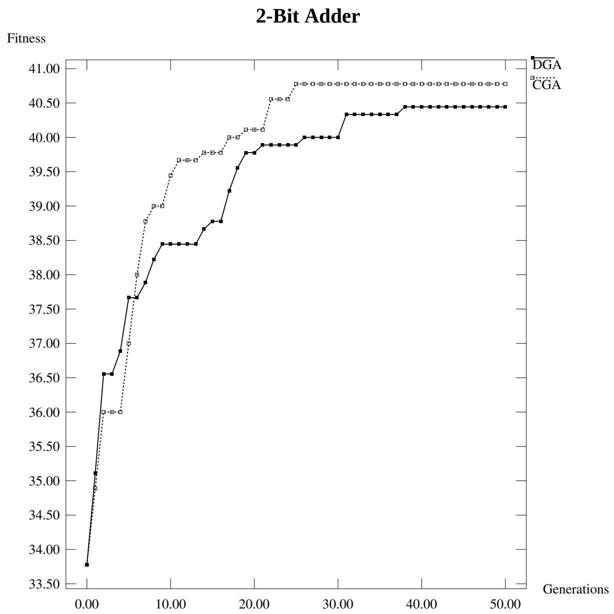

Figure 4.7: Performance comparison of maximum fitness per generation of a classical GA versus a DGA on a 2-bit adder.

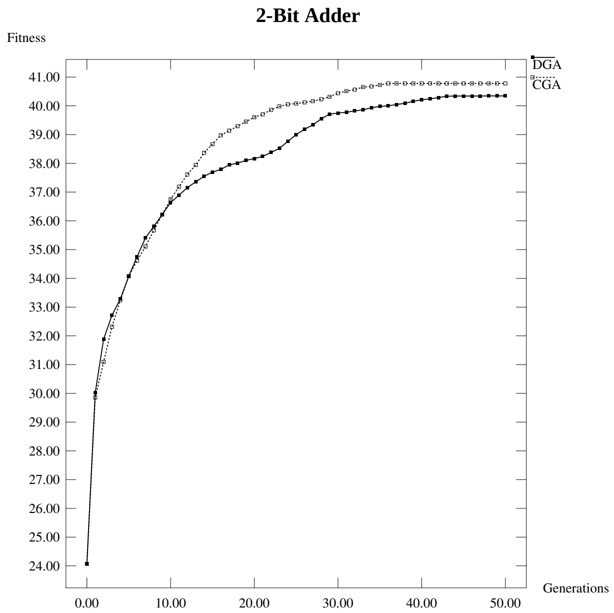

Figure 4.8: Performance comparison of average fitness per generation of a classical GA versus a DGA on a 2-bit adder.

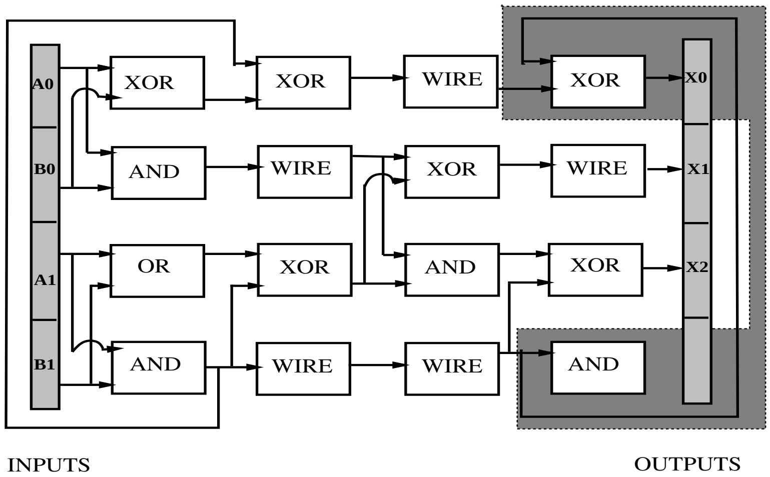

Figure 4.9: A 2-bit adder designed by a designer genetic algorithm.

Figures 4.9 and 4.10 show solutions to the 2-bit adder problem found by a designer genetic algorithm and classical genetic algorithm. As wire gates ignore their second input, only one input is shown for such gates. The gate in the third row and third column (S_{33}) is shown unconnected because it does not affect the output.

In figures 4.9 and 4.10 the shaded regions represent highly fit schemas. Notice that the DGA preserves a schema of much longer length than the CGA. Additionally the figures illustrate the difficulty of understanding why the circuit works. The problem of opacity of GA-generated solutions is handled in the next chapter.

Now consider the parity problem. The encoding described in figure 4.5 will violate the principle of meaningful building blocks with regard to the solution to the parity problem as shown in figure 4.11. Since diagonal elements of S (the 2-D phenotype) are further apart in the one-dimensional genotypic string, any good subsolutions (highly fit, low-order schemas) found will tend to be disrupted by traditional crossover. In other words, gates which are close together in two–dimensional (phenotype) space may be far apart in one-dimensional (genotype) space, causing problems for classical

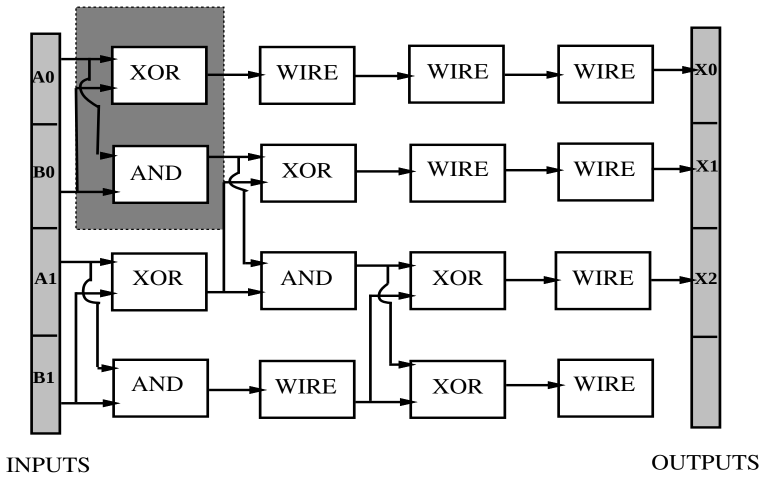

Figure 4.10: A 2-bit adder designed by a classical genetic algorithm.

GAs. MX, however, will find and preserve these subsolutions as its performance is independent of defining length. To observe performance under these conditions, the experiments restrict the number of gate types available to the GA to three and do not allow a choice of input (the second input is now always from the next row, modulo the number of rows). Although this reduces the size of the search space, traditional crossover disrupts low-order schemas and therefore performs worse than the DGA. Figure 4.12 and figure 4.13 show this for a 4-bit parity checker. (In cases where there were no restrictions the performance of both GAs was comparable.)

As expected, the difference in performance gets larger as the problem is scaled in size. Figure 4.14 and figure 4.15 compare the maximum and average fitness performance on a 5-bit problem. In the 5-bit experiments the choice of gates was still restricted to the same three as in the previous example. When all possible gates are allowed, the performance difference is less, and is due to the large increase in the number of possible solutions and therefore a lesser degree of violation of the meaningful building block principle (see figures 4.19 and 4.20). However, as the number of solutions in a given space decreases, masked crossover does better than traditional

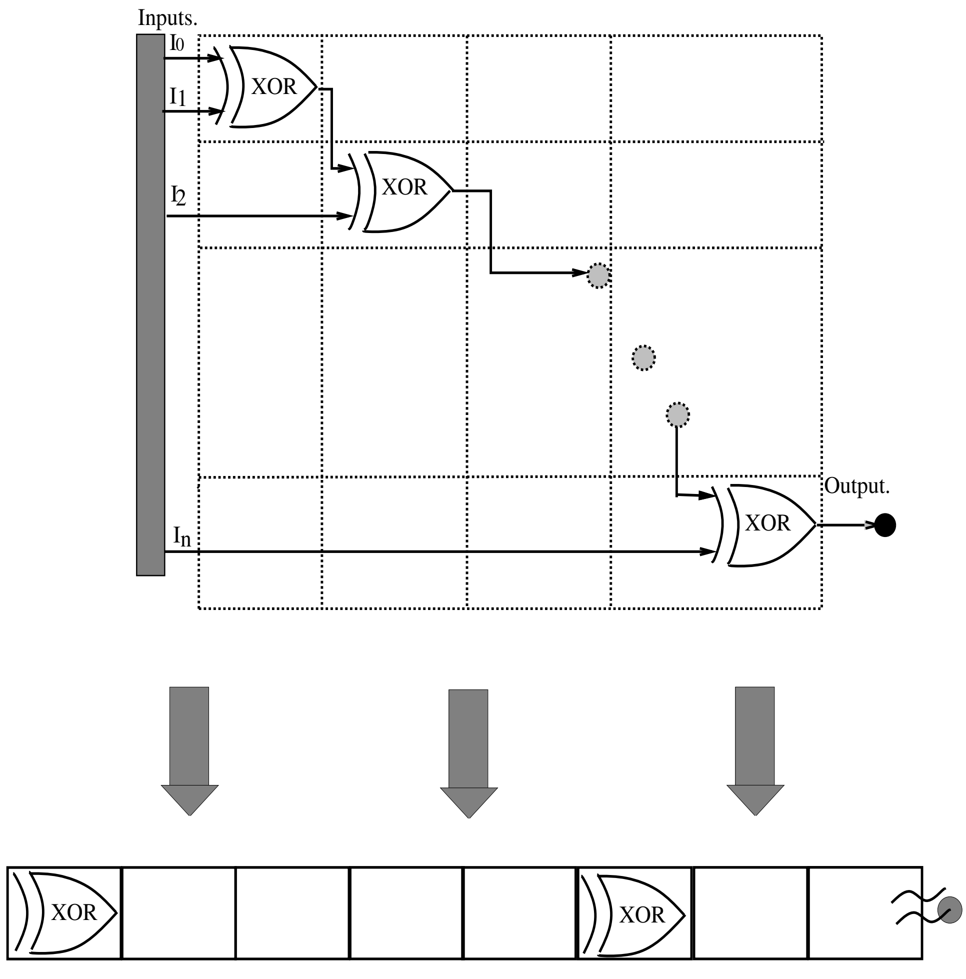

Figure 4.11: The XOR gates that are close together (diagonally) in the two–dimensional grid will be far apart when mapped to a one–dimensional genotype

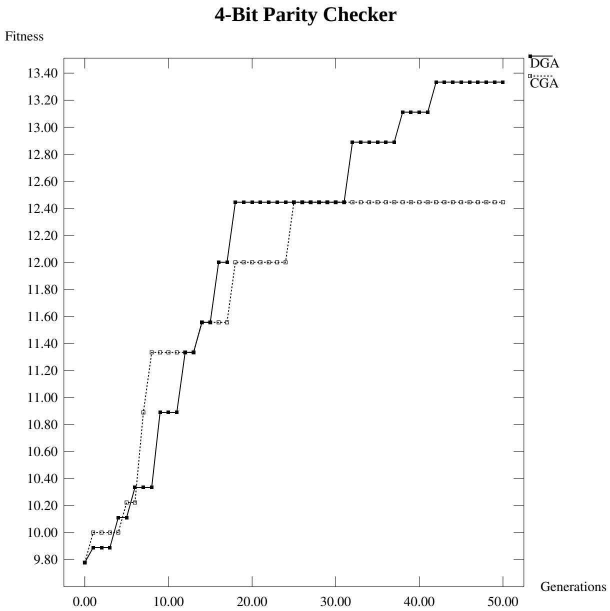

Figure 4.12: Performance comparison of maximum fitness per generation of a classical GA versus a DGA on a 4-bit parity checker.

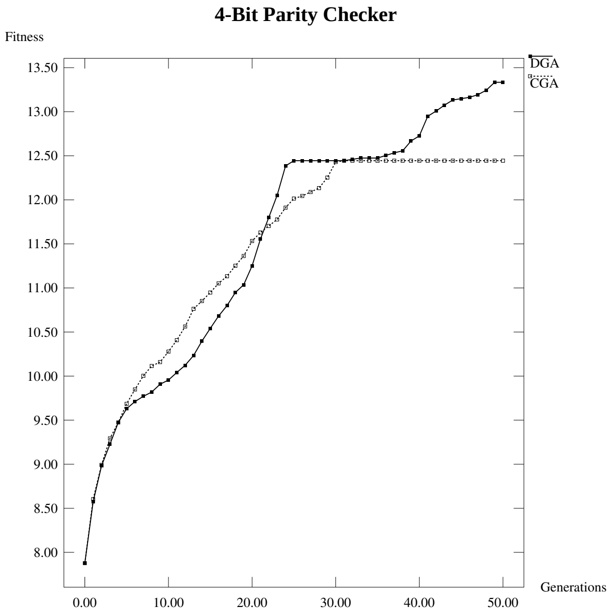

Figure 4.13: Performance comparison of average fitness per generation of a classical GA versus a DGA on a 4-bit parity checker.

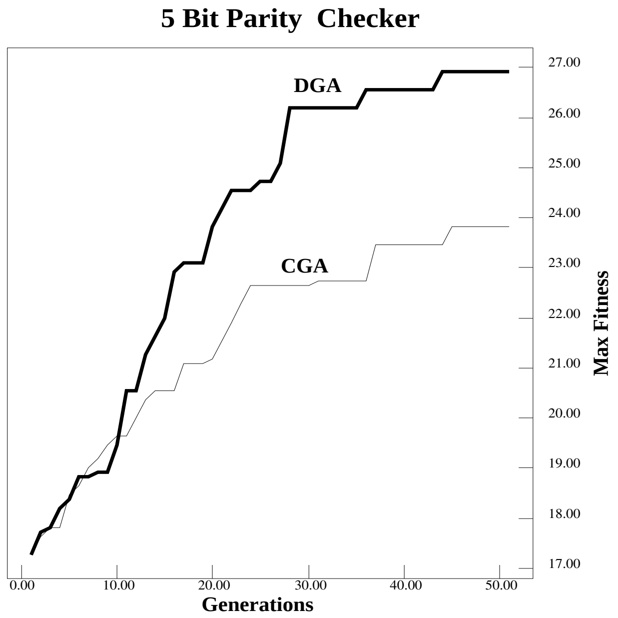

Figure 4.14: Performance comparison of maximum fitness per generation of a classical GA versus a DGA on a 5-bit parity checker.

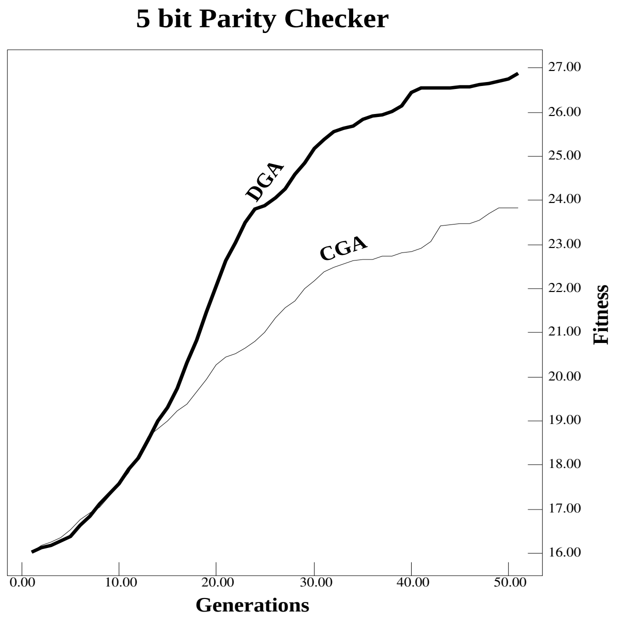

Figure 4.15: Performance comparison of average fitness per generation of a classical GA versus a DGA on a 5-bit parity checker.

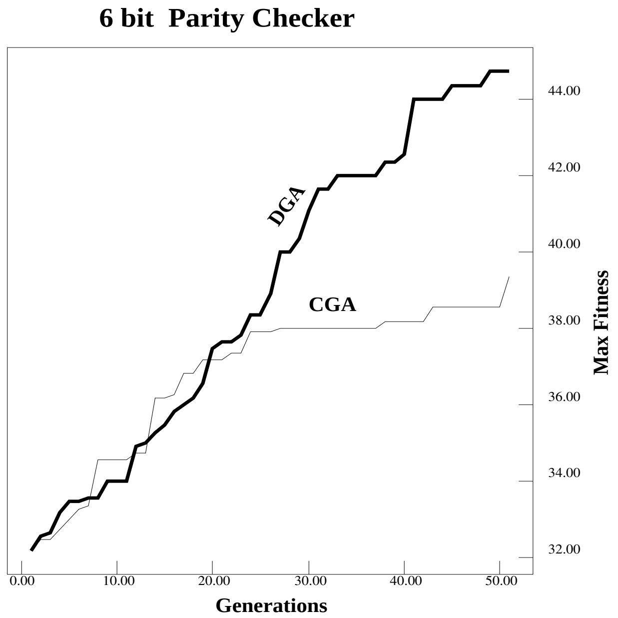

Figure 4.16: Performance comparison of maximum fitness per generation of a classical GA versus a DGA on a 6-bit parity checker.

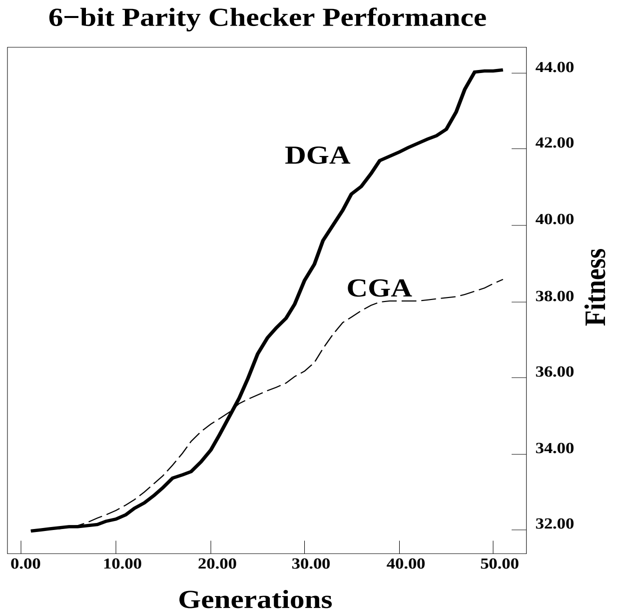

Figure 4.17: Performance comparison of average fitness per generation of a classical GA versus a DGA on a 6-bit parity checker.

crossover because it uses differential information about child fitness to bias search independent of schema defining length. Hypothesis testing using the student's t-test on the experimental data from the five-bit parity checker proves that the difference in performance is significant at a confidence level greater than 95%.

On the 6-bit parity checker problem, the performance difference is still larger. Figures 4.16 and 4.17 display the maximum and average fitnesses. Once again the difference in performance is statistically significant using the student's t-test at a confidence level greater than 95%. Figure 4.18 shows performance at the scale of the fitness units (vertical axis) in the 5-bit case. The 5-bit and 6-bit performance plots are displayed together to make the change in the performance difference more apparent. This provides evidence that as the length of high fitness schemas grows, the performance difference gets larger.

In the comparisons above the effect of selection was not addressed. Figures 4.19 and 4.20 compare the performance of: 1) a GA using traditional crossover and selection, 2) a GA using traditional crossover and elitist selection, and 3) a DGA on a 5-bit parity problem. The same parameter set as in the previous examples is used although setting the number of gate types to six, increasing the number of possible solutions. This was done in the hope of coaxing better performance from the GA using traditional selection and crossover. The figures clearly show the importance of selection strategy.

The next two figures show examples of correct circuits for the 4-bit parity problem. A DGA produced the circuit in figure 4.21 and a classical genetic algorithm produced the circuit in figure 4.22. The unconnected gates in the last column do not contribute to the output, therefore their connections have been left out. Both algorithms had the full complement of gates available.

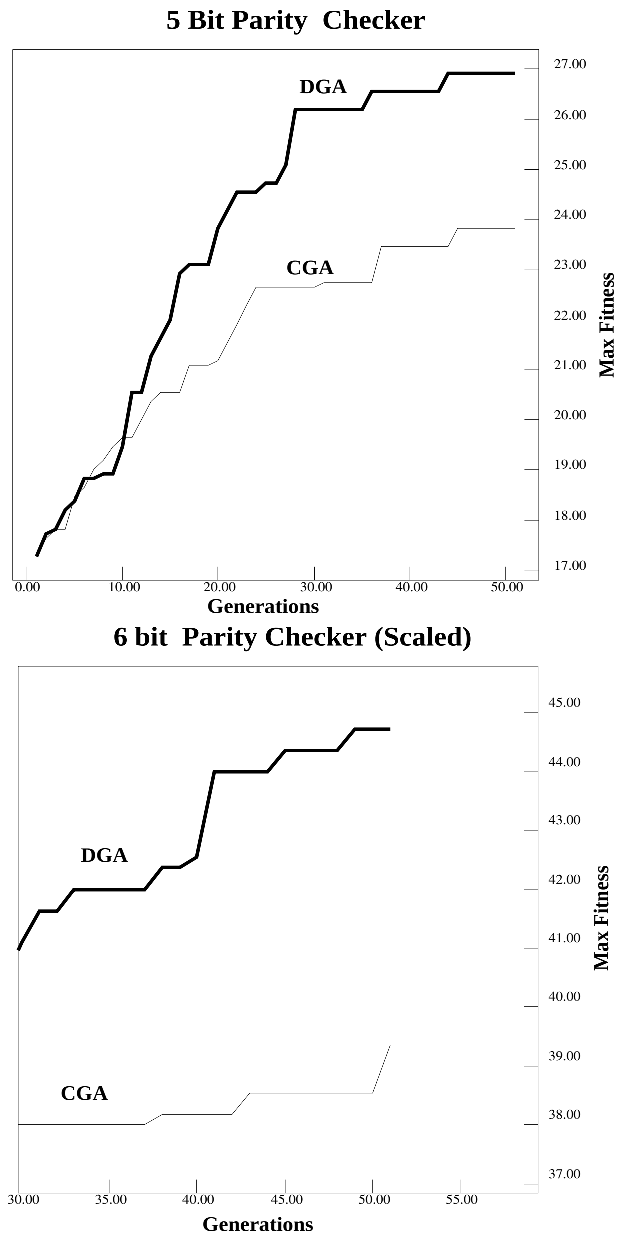

Figure 4.18: Change in performance difference as the length of high fitness schemas (proportional to problem size) increases. Comparing a 5-bit and 6-bit parity checker performance difference. The vertical axis for the 6-bit plot is now to the same scale as the five bit plot. The increase in performance difference is clearer.

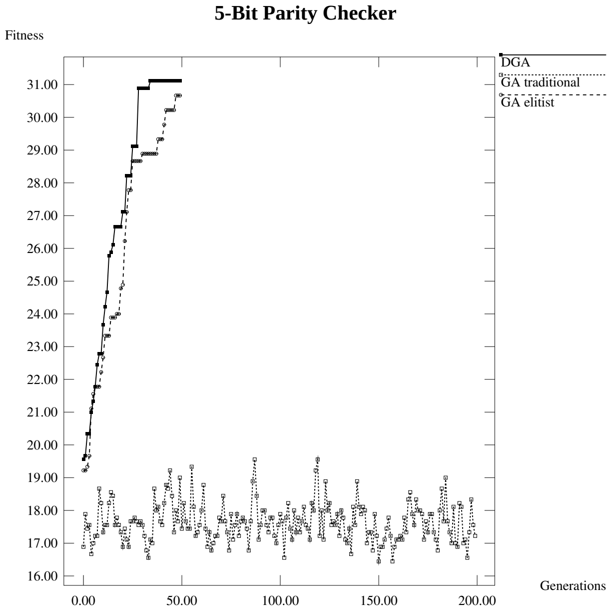

Figure 4.19: Performance comparison of maximum fitness per generation of a classical GA using traditional selection, a classical GA with elitist selection, and a DGA on a 5-bit parity checker.

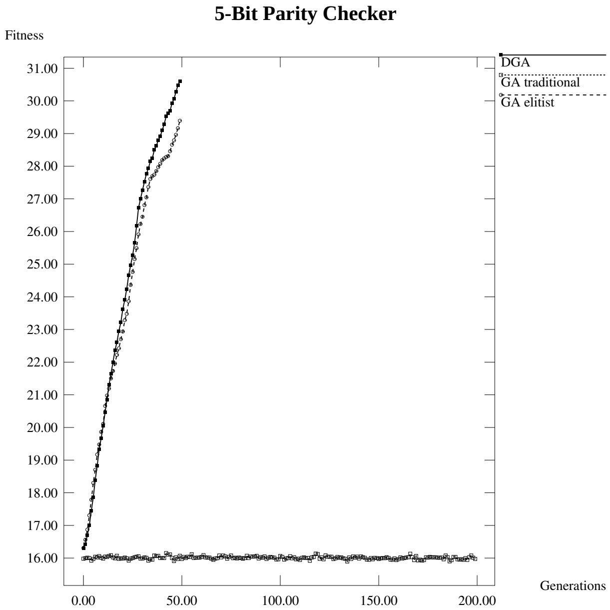

Figure 4.20: Performance comparison of average fitness per generation of a classical GA using traditional selection, a classical GA with elitist selection, and a DGA on a 5-bit parity checker.

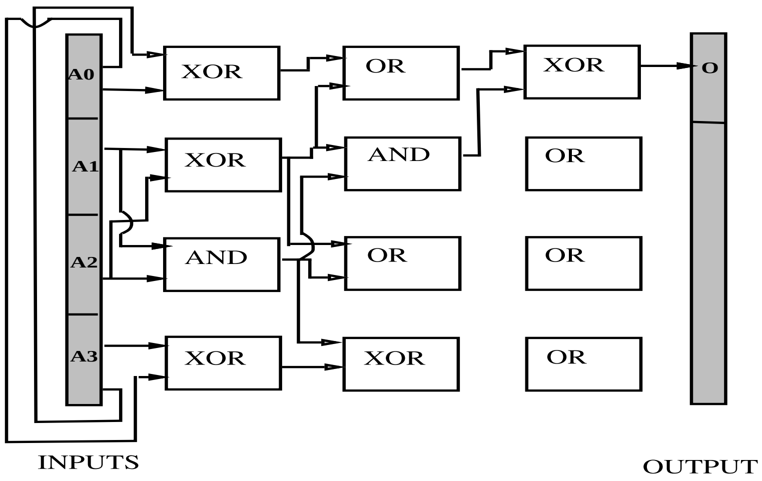

Figure 4.21: A circuit designed by a designer genetic algorithm that solves the 4-bit parity problem.

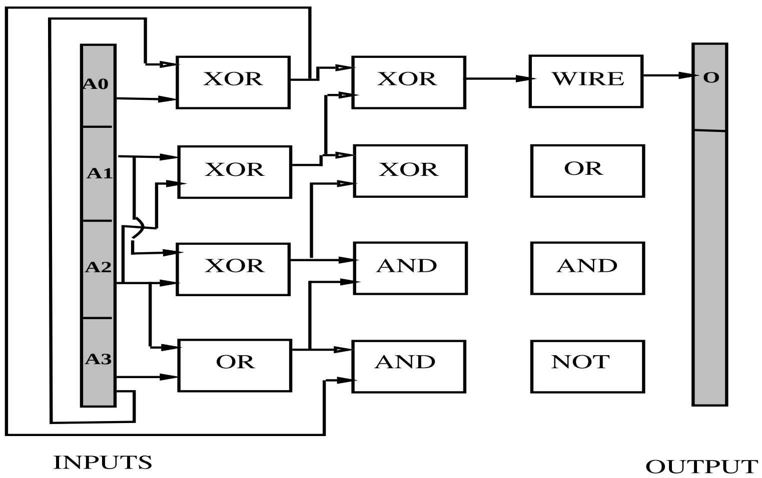

Figure 4.22: A circuit designed by a classical genetic algorithm that solves the 4-bit parity problem.

4.6 Summary

A designer genetic algorithm increases the domain of application of genetic algorithms by relaxing the emphasis on schemas of short defining length. Using masked crossover mitigates the problem of epistasis while elitist selection is crucial to good performance. This is cheaply attained since the increase in cost in using a DGA is by at most a constant factor per generation. Comparing the performance of the two GAs on a problem also gives insights about properties of the search space. If the performance difference is large, then highly fit, low-order schemas are expected to be of large defining length. Some of the circuits generated by the genetic algorithms in this chapter are not easily understandable. That is, although they can be tested for functional performance, it is not obvious how they work. In the next chapter the thesis describes a system that extracts information about the search space and uses this information to explain solutions generated by genetic algorithms.

Deception also plays a role in determining performance. Traditional crossover may do better on deceptive problems because it uses no information about search direction and is thus immune to misleading directional information.

The problem of circuit design as stated in this chapter is not an optimization problem. Since the fitness of a correct circuit, one that correctly performs the function, is known, the problem is to find a design that achieves this fitness. This is a *satisficing* task, not an optimization one. However, minimizing the number of gates, wires, or size of the circuit are constraints that are usually applied in practice.

Having alleviated the problem of choosing an encoding for a genetic algorithm, this thesis now considers the problem of understanding solutions generated by a genetic algorithm.

5

Understanding Genetic Algorithm Solutions

> [The great supercomputer, asked what is the answer to] the great problem of life, the universe and everything [replied, after many years of computation] 42.
>
> – Douglas Adams, The Hitch-hiker's Guide to the Galaxy

As illustrated in the last chapter, solutions generated by genetic algorithms may be difficult to understand. There are two main reasons for this:

1. GAs are best used in poorly-understood domains, where domain knowledge is scarce, making both design and analysis difficult.

2. The stochastic nature of GA operators and the size of the space of all possible building blocks and their combinations makes finding useful building blocks

and explaining their significance difficult. This makes it hard for a designer to calculate or accept the worthiness of an evolved solution.

During the course of a GA's search through the underlying space for a design, it *implicitly* extracts and uses information about the space. The kind of information extracted and used by a genetic algorithm depends on the specific operators used and is indicated by the schema theorem and building block hypothesis. However, this knowledge is never made explicit and is discarded at the end of a GA's run. This thesis describes a system that uses tools developed for case-based reasoning and genetic algorithm theory to identify, extract, and organize this information. The organized knowledge can be used to learn about the domain, to explain how a solution evolved, to discover its building blocks, to justify why it works, and to tune the GA in various ways.

The chapter starts with a short introduction to case-based reasoning, describes the system, and clarifies the results using examples from circuit design and function optimization.

5.1 Case-Based Reasoning

Case-based reasoning (CBR) is based on the idea that reasoning and explanation can best be done with reference to prior experience, stored in memory as *cases* (Bareiss, 1991; Riesbeck and Schank, 1989). When confronted with a problem to solve, a case-based reasoner extracts the most similar case in memory and uses information from the retrieved case and any available domain information to tackle the current problem. The strategy is first to collect and index a large and varied collection of examples, and then, when presented with a new situation, to fetch and possibly manipulate the stored example that most closely resembles the new situation. Each stored case may be considered a previously evaluated data point, the nature and location of which is problem-dependent. If the distance metrics have been well designed and the case base includes sufficient variety, retrieved cases reveal useful conclusions from previous analysis. Because it may be impractical to store all prior experience in detail, CBR systems often include principled mechanisms for generalization and abstraction of cases. Note that except when comparing distinct cases, a CBR system need not include any notion of an underlying model of the domain of application, and that even during comparison one can make do without much of one. One side-effect of this approach is that the generalizations and abstractions may in fact *induce* a useful domain model. This chapter uses the basic tenet of CBR — the idea of organizing information based on "similarity" — to help explain the solutions generated by a genetic algorithm.

5.2 Genetic algorithms and CBR

GA theory tells us little about the actual path a given GA will take through the space on its way to an answer. Because of the stochastic nature of GA operators, a solution's optimality is not guaranteed and its building blocks are unknown.

Information about the building blocks is implicit in the processing done by a GA but is not documented explicitly or available to the user. Keeping track of historical regularities and trends is the key to acquiring the knowledge needed to explain what sort of solution evolved, why it works, and where it came from. Discovering building blocks by explicitly keeping track of the GA's search trajectory allows principled partitioning of the search space. Empirical regularities can be formalized and possibly put to use. The individuals in a GA's population act as a primitive memory for the GA. Genetic operators manipulate the population, usually leading the GA away from unpromising areas of the search space and towards promising ones, without the GA having to *explicitly* remember its trail through the search space. Genetic algorithms trade the use of explicit memory for speed and simplicity. This chapter uses the methods and tools developed for case-based reasoning to extract, store, index and (later) retrieve information generated by a GA. Individuals created by the GA provide a set of initial cases from which a well-structured case-base can be constructed.

Defining an appropriate index or measure of similarity among the cases is one of the first tasks in creating a case-base. The building block hypothesis implies that syntactically similar (short and low-order), high-performance individuals bias genetic search. Since genetic algorithms process syntactic similarity and fitness as stated in the building block hypothesis, this thesis uses fitness and genotype (the encoded design) as metrics for indexing the case-base. These measures provide the key to indexing the cases and conveniently solve one of the hardest of CBR problems.

CBR's model of information organization is applied to GAs as an analysis tool in order to track the history of a search. The system (described below) creates a case for each individual that the GA has evaluated. These cases are indexed on syntactic similarity and fitness. When properly formed and analyzed, this case-base can contribute to the understanding of how a solution was reached, why a solution works, and what the search space looks like. At the interruption (or termination) of a GA run, the case-base allows the interpretation and subsequent modification of discovered solutions.

In addition, during the course of a run, as the system digests more information, it can generate hypotheses about the space. These hypotheses, which take the form of new individuals injected into the GA's population, can be tracked and evaluated, testing the validity of the hypotheses. Given proper hypothesis formation, the system can guide evolution more directly towards convergence. False hypotheses may provide evidence of deception in the current search space.

5.3 The System

The system is made up of a genetic algorithm which runs in tandem with an analysis module (see figure 5.1).

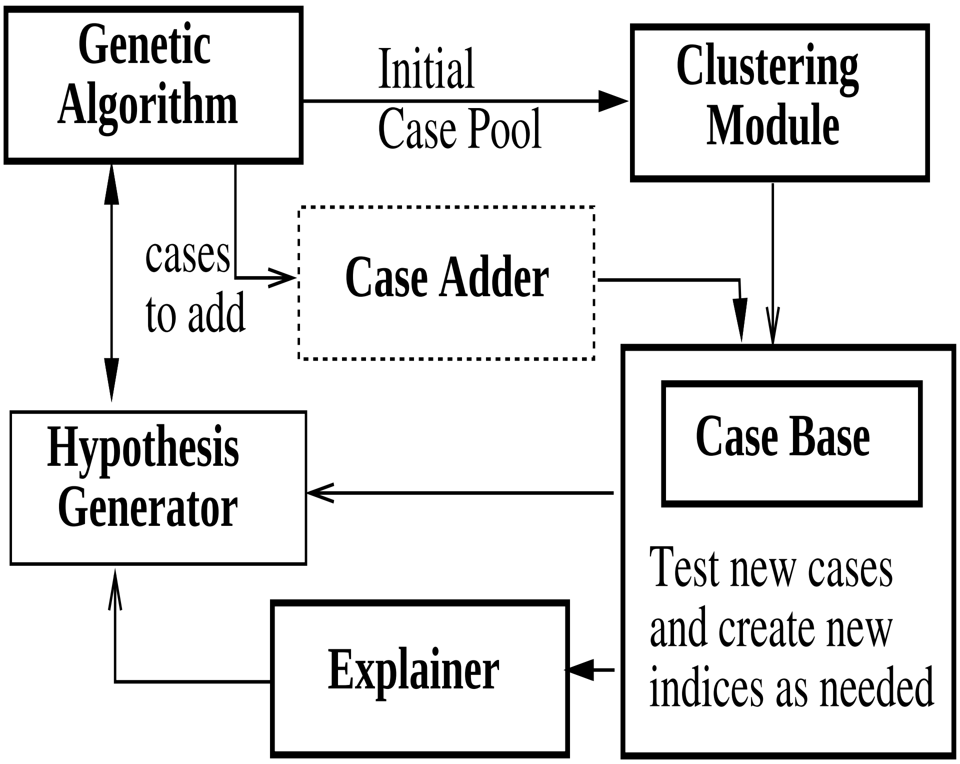

Figure 5.1: Schematic diagram of the system

The genetic algorithm records data for each individual in the population as it is created and evaluated. This data includes a fitness measure, the genotype, and chronological data, as well as some information on the individual's parents. This collection of data is the *initial case data.* Though normally discarded by the time an individual is replaced, all of the case data collected is usually contained in the genetic algorithm's population at some point and is easy to extract.

After a sufficient number of individuals have been created over a number of generations, the initial case data is sent to a clustering program. A hierarchical clustering program clusters the individuals based on both the fitness and the alleles of the genotype. This clustering constructs a binary tree in which each leaf includes the data of a specific individual. The binary tree structure provides an index for the initial case-base. Figure 5.2 shows a small part of one such tree. The numbers at the leaves of the tree correspond to the case number (an identification number) of an individual created by the GA. Internal nodes are denoted with a "$*$". An abstract case is computed for each internal node based on the information contained in the leaves and nodes beneath it. The final case-base includes: 1) cases corresponding directly to GA individuals (at the leaves) and 2) more abstract cases made up of information generalized from the leaves.

Although a number of clustering techniques and clustering sets are possible (for example, fitness and genotype data versus fitness alone), standard hierarchical clustering on fitness and genotype produces a coherent initial index for the case-base. The fact that fitness and genotype matter the most is predicted by the building block hypothesis and borne out experimentally. After an index has been created, several easy computations are recursively performed to flesh out the abstract cases (these computations are explained in Section 5.4).

The hypothesis generator runs in tandem with the GA. Using information from the

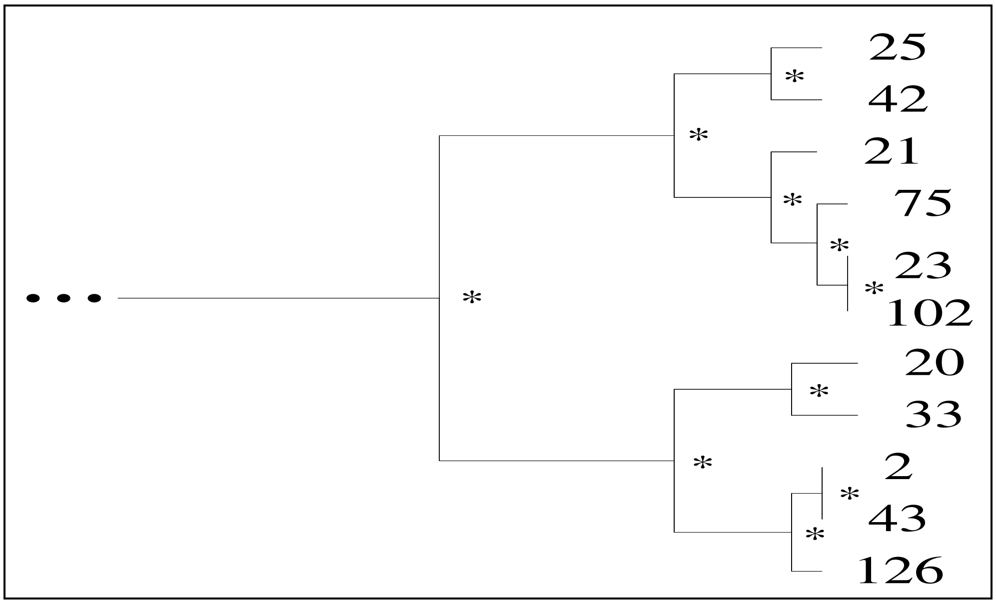

Figure 5.2: A small subsection of a typical binary tree created by the clustering algorithm. Individuals are represented by the numbers at the leaves. Internal nodes where abstract cases go are denoted with a "*". The binary tree provides an index for the case-base.

the case-base, it engineers individuals for injection into the population. Information about the measured performance of such individuals relative to the remainder of the population can also be fed back into the hypothesis module for analysis. The success or failure of generated hypotheses can be used in decisions regarding the validity of the current case-base.

At the end of a run, a well-developed case-base can be used to explain how the GA came up with its answer, and why that answer is strong. The system can explain something about where the winning answer came from, which of its building blocks are the strongest, and which the weakest. This sort of data allows any future search of the space to be fine-tuned. It also allows *post facto* analysis. Much of the knowledge that is usually discarded by a GA is made available in processed form for future manipulation.

5.4 The Case-Base

Clustering the first few hundred individuals using genotype and fitness values produces a case-base. As explained above, clustering determines the indexing scheme of the case-base. All abstract cases, which are indexed at the internal nodes of the tree, must be recursively computed from the leaves up. Although much of the initial case data saved during the GA run is not used by the clustering algorithm, it is added to the cases in the case-base.

Cases include the following information:

- Case number
- Distance from the root of the tree to the level of the case
- Schema for the case
- Schema order
- Average fitness
- Weight: Number of leaves (individuals) below
- Generation information: the earliest and latest leaf occurrence as well as the average in the subtree
- Additive schema: Bitwise sum of alleles of the genotypes of all the leaves below

Using the information in the case-base, the system produces a report regarding the GA run so far. Using fitness as a metric, computing the top-ranked or bottom-ranked n cases is easy, as is finding the top-ranked or bottom-ranked n case schemas. These schemas can be combined to make new schemas by applying standard schema creation rules. For example, the two schemas **110**10 and **100***0 combine to the new schema **1*0***0. The resulting schemas show (among other things) which alleles are important in the genotype and which ones are not. Since order, longevity, and weight information is available, it is straightforward to find the top few cases with given sets of these characteristics.

It is also useful to calculate not only the schema that includes the subset of leaf cases, but one that records in a more democratic fashion the relative frequency of allele settings for the positions which are not absolutely fixed. Since the normal schema-producing rules result in a "$*$" wherever there is disagreement about an allele among leaves, schemas near the top of the tree tend to be filled with "$*$"s. To avoid this information loss, the system generates an *additive schema* by computing the bitwise sum of the genotypes associated with the leaves below.

An *elected schema* can be computed by dividing the additive schema by the weight and squashing the results to 0, 1, or "$*$" using thresholds that can be changed. Elected schema information provides a more informative way of looking at upper-level schemas. Elected schemas play a large role in hypothesis generation, discussed in section 5.7.1.

A schema actually describes a convex hull for the leaves from which it was computed. That is, it describes a portion of the space within which the unaltered leaves fit. Since the normal schema-producing rules result in top-level schemas with a large number of "$*$"s and the hull tends to span too large a portion of the entire space, it is preferable to describe a smaller hull, one for which the average hamming distance from an actual leaf to the hull surface is less than some predetermined constant. In other words, the leaves deviate from their closest encompassed relative by only a small amount on average. The elected schema describes this smaller hull.

Generation information can be used to determine when each part of the search space covered by the GA was considered. This information is very useful in detecting premature convergence, finding local minima, and computing the relative longevity of a subspace. The search space itself can be carved into approximately equally populated subsections by chopping the tree at nodes having a certain maximum weight. Analysis of these subsections across dimensions of fitness and longevity sheds light on the search space.

All of this processed information is available on–line. If desired, any case can be displayed. Also available is the tree information in graphical form with case numbers in their proper locations. The graphics are expandable so that subtrees can be thoroughly investigated. The "report" is meant for user consumption and usually raises some questions which can be answered by pulling more information from the system.

It is important to emphasize that the system is an interactive *tool* for use in explaining GA-generated solutions. Although it automates the information extraction and processing to some extent, human perception is a critical ingredient in the final analysis. However, without the reasonable organization and pre-processing that this system provides, the large amount of data created during a GA run is impossible to digest even for an experienced GA researcher.

5.5 An Example and Methodology

Several experiments in function optimization and circuit design show that clustering techniques, if properly applied, provide a strong case-base, yielding useful data about the GA run. These results are reported in more detail in (Louis et al., 1993). A simple example ($f(x) = x$) in function optimization introduces the system and provides a methodology for analyzing the information in the report. Next, a *design* example from a genetic-algorithm-generated circuit for a 5-bit parity checker supports the methodology and results obtained through the system.

5.5.1 A Simple Example

The tree structure resulting from clustering defines a schema hierarchy (figures 5.3 and 5.4). Schemas of lower order are found near the top of the tree. Schema order gets higher and higher towards the leaves, which, with all alleles fixed, have the maximum possible order. Figure 5.3 depicts the index tree of an entire case-base for maximizing a simple function: $f(x) = x$. Some salient features of the space are immediately recognizable on the graph (when one can zoom in and look at cases). These features are marked in figure 5.3. Figure 5.4 shows an enlargement of the small boxed area in figure 5.3 (labeled "Area of Detail"). In this figure, the positions of both initial cases (individuals) to the right of the figure and abstract cases at the internal nodes are represented by their schemas. Schemas closer to the root of the tree are more general (that is, of lower order) than those towards the leaves.

The case-base was created by clustering the 400 GA individuals from the first 20 generations (which resulted in a binary tree) and then computing the abstract cases at the internal nodes. The resulting case-base had 799 cases, 400 of which came directly from the GA. The salient features in the search space, like the importance of the sign bit, can be identified and marked using the report mentioned above in conjunction with the index-tree graph. Clustering partitions the individuals into sets sharing common features, namely specific allele values (syntactic similarity) and similar fitness. In this case, all the low-fitness cases are clumped together toward the top of the graph. The high-fitness cases are at the bottom. A sign bit was used in the

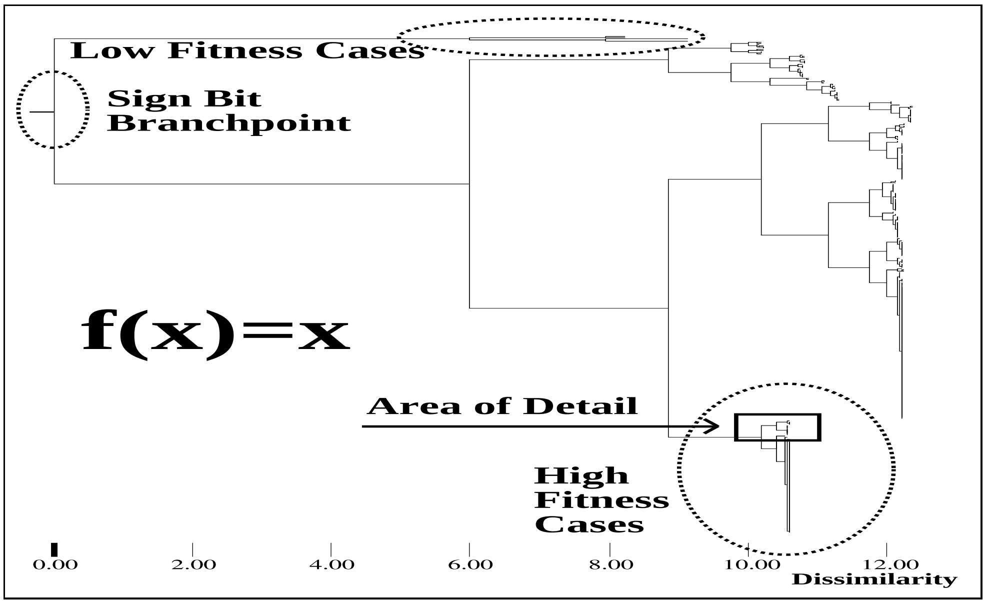

Figure 5.3: Tree structure of the cases. Nodes represent abstracted cases.

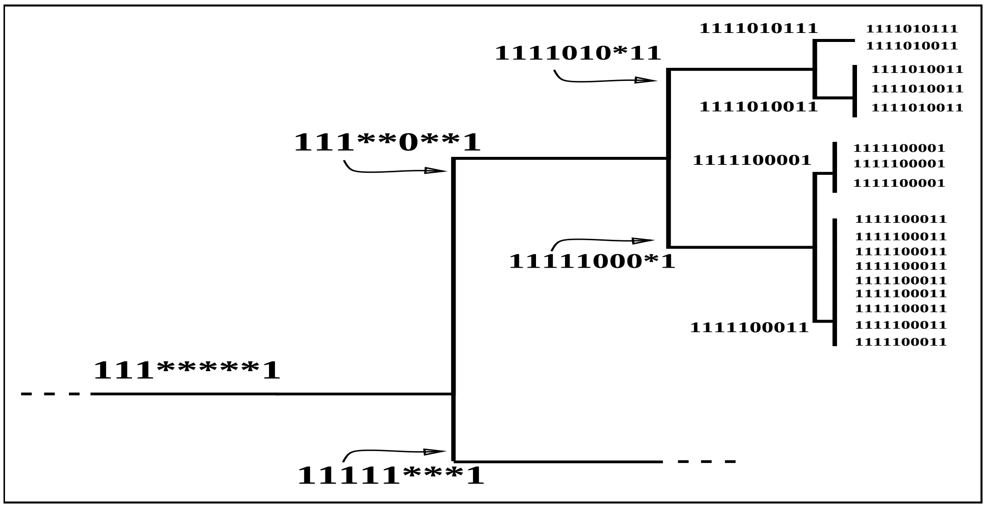

Figure 5.4: A closer look at the clustered cases reveals nested schemas.

genotype (values ranged from -511 to 511). The sign-bit branching point is shown in figure 5.3 as well. Not surprisingly, all the negative numbers are clustered together above the sign-bit branchpoint, and very little time was spent in that portion (half) of the search space by the GA. This information was discovered by zooming in on the subsections of the graph and noting the obvious common features while consulting the on–line case-base.

Case zero is located at the root of the tree. Actual output from a query shows case-information:

```
> (show-case 0)
+------------------------------------------------------------+
| Case: 0    | Distance: 0    Weight: 400   Fitness: 8441.44  |
|------------+ Order: 0       Schema: **********             |
| Additive schema: (396 333 170 283 338 330 20 72 332 392)     |
| Longevity: 19    (lo:1  avg:10.5  hi:20)                    |
| Left subtree: 1                            Right subtree: 396 |
+-------------------------------------------------------------
```

Since case 0 is located at the root of the tree, its distance from the root is 0. There are 400 leaves below it, making its weight 400. Fitness is calculated by averaging the fitnesses of the cases below. The case schema is filled with "$*$"s, showing once again why the additive schema information is important. Leaves under case 0 spanned all generations from 1 to 20 resulting in a longevity value of 19. Case 0 has two pointers, one to case 1 and another to case 396. This reflects the binary nature of the index. Had the case been a leaf it would not have pointed to further cases.

5.5.2 Methodology

Experiments in function optimization (summarized in section 5.7), combined with genetic algorithm theory, indicate that a useful high-level description of the case-base and the search recorded within it can be generated as follows:

- Select a subset of the internal nodes which implicitly include all of the leaves, have no overlap, and are of fixed maximum size.
- Sort these nodes by increasing average birth generation (age).
- Report the available information about generation, weight, fitness, order, and elected case-schema.

5.6 Results from Circuit Design

The parity-checker problem's search space is poorly understood, discontinuous, and highly non-linear. As encoded for the GA, a 5-bit parity-checker problem results in a 2^{80} sized search space having these troublesome properties. The experiments used a genotype of length 80, and a population size of 20, and ran the GA for 25 generations. Appendix B contains a plot of the cluster tree, and an abbreviated report for the 5-bit parity-checker problem from chapter 4, along with instructions for obtaining the public domain code for clustering. The analysis below uses the report, cluster tree (figure B.1), and figure 5.5.

The GA does not solve the problem in 25 generations. In the experiments, a case-base was created by clustering the 500 individuals created by the GA and computing the 499 abstract cases (see figure B.1 in appendix B). The analysis proceeded by examining the report and following the steps outlined in the methodology section. The disjoint subtree list was then inspected. The strongest such subtree (H) was also heavily weighted and spanned many generations. In addition, the best partial solutions all fell within H. This implied that a complete solution would fit H's schema. Since the order of the schema was appreciable (38 of 80), the search could be constrained to that schema in the hope of finding a solution. One correct solution was in fact contained within H's schema.

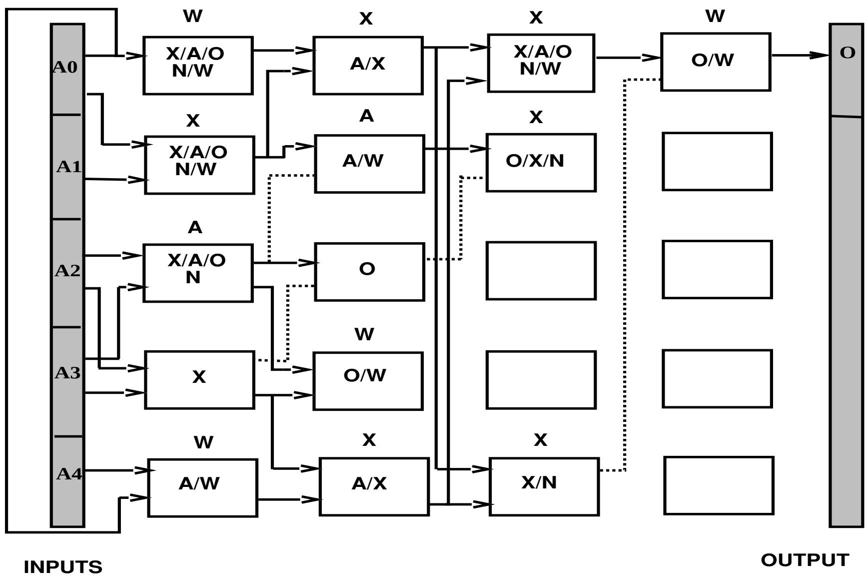

Figure 5.5: Parity circuit indicated by H, and the correct instantiation of choices.

Figure 5.5 shows the circuit indicated by the previous analysis of H. By decoding H, it was possible to discern the general structure of a solution and determine which parts of it were important. Labeled boxes represent one of the logic gates e**X**clusive-or, **A**nd, **O**r, **N**ot, and **W**ire. Since the circuit in figure 5.5 represents a schema, multiple labels on the boxes show its possible instantiations. Labels above boxes represent a correct instantiation found near the 50^{th} generation of an unconstrained run. The top row should be considered adjacent to the bottom.

Since gates form easily recognizable building blocks, the low-order schemas corresponding to boxes whose inputs are not fixed denote unimportant gates. The unlabeled boxes thus reflect unimportant dimensions of the search space, which can be safely ignored. Recall from chapter 4 that gates have two inputs, one of which is determined by the genetic algorithm. The figure and report imply that gates participating in a correct solution had already fixed their input locations by generation 25. Thus the alleles specifying input location denote an important building block. The search space size also decreases significantly, since the schemas that specify unlabelled boxes, and the already fixed positions in H at generation 25, can be ignored. In fact, the search space size decreases by approximately 15 orders of magnitude. Additionally, of the 10 schemas corresponding to boxes whose inputs are fixed, nine can specify an e**X**clusive-or. These gates are important for a parity checker. Finally, genetic algorithm theory implies that the higher the fitness of a schema, the earlier that schema is fixed (dominates the population). Therefore the gates and inputs that were fixed earlier in the search for a circuit will tend to cause greater changes in performance compared to gates that were fixed later during the search.

5.7 Results on Function Optimization and Hypothesis Generation

Results presented here are a summary of the more extensive work reported in (Louis et al., 1993). Testing on a series of optimization problems including the DeJong (DeJong, 1975) functions and a pair of deceptive problems shows that the CBR tools are useful for understanding GA solutions in many different kinds of spaces as exemplified by the test problems. These functions are given in table 5.1 and range from linear to exponential. The three functions from the DeJong test suite for genetic algorithms (DeJong, 1975) provide multimodal and discontinuous spaces.

$f(x) = c$	c is a constant (control)
$f(x) = ones(x)$	number of ones in genotype
$f(x) = x$	signed, genotype length 10
$f(x) = x^2$	signed, g-length 10
$f(x) = x^3$	signed, g-length 10
$f(x,y) = x^y$	signed, g-length 20
$f(x) = 2^x$	all positive, g-length 10
$f(x) = log(x)$	all positive, g-length 10
$f(x,y) = x^2 - y^2$	all positive, g-length 20
$f(x_i) = \sum_i^3 x_i^2$	DeJong F1, $(-5.12..5.12)$
$f(x_i) = 100(x_1^2 - x_2)^2$	DeJong F2 $(-2.048..2.048)$
$\quad + \sum_i^2 (1 - x_i)^2$	
$f(x_i) = \sum_i^5 \text{integer}(x_i)$	DeJong F3 $(-5.12..5.12)$
A pair of deceptive functions.	

Table 5.1: Test Problems.

Evidence of the GA's path through the search space can be seen. Many of the earlier subtrees may have low average fitness and low order, whereas later subtrees will have higher order if convergence was approached. Typically, later nodes have higher fitness, but as clustering is sensitive to genotype, it is possible to have a weak but young subtree.

Low-order long-lived subtrees with poor fitness are often associated with crippled individuals formed by mutation or unfortunate crossover. If there are many weak subtrees, one can infer a stronger focus on *exploration* of new parts of the space as opposed to *exploitation* of strong, higher-order schema. The schemas defined by such subtrees denote unpromising areas of the space.

High-order short-lived subtrees offer evidence of temporary concentration of search effort, which often follows discovery of a beneficial building block. Once a building block has been incorporated into the population (exploited), the search effort may move on to further exploration.

In problems where there is deception (that is, where most of the gradient in the space draws the GA *away* from the absolute global optimum but toward some local optimum), a user of this system would expect to see the GA focus first on the local maximum. If the GA manages to discover points near enough to the global maximum that they are favorably evaluated, a paradigm shift may occur. This has been evident in the case-bases.

If a user notices that a small but strong subtree was discovered but quickly lost and not surpassed, that might suggest a need for alteration of the GA parameters in favor of a more elitist selection strategy (like CHC). When there is suspicion of deception, it may be confirmed or discredited by looking at the lower levels of the quickly lost cluster. If it is discovered that within the cluster, the later and stronger subclusters had schema changes that appeared to cause relative *decline* in other contemporary clusters, then there is stronger evidence for deception. At this point a more thorough investigation of that part of the search space without much competition is justified.

5.7.1 Hypothesis Generation

Motivations for including a hypothesis generator are many. If hypothesis-produced individuals prove to be better on average than GA-produced individuals, the GA will be accelerated. They also provide evidence that produced solutions are reasonably well understood. Further support is furnished by the identification of building blocks.

Work on hypothesis generation focuses on using the principled heuristics sketched in section 5.5.2. The convex hull of each subtree selected by the method outlined there contains a very large number of possible individuals that have never been evaluated. The hypothesis module picks some specific individuals from within the hull of the most promising subtree and injects them for participation and evaluation within the GA population. Only a small number of generated individuals are injected in a given generation.

Since individuals that fit in the reduced hull of the *elected schema* may be considered more prototypical of the subtree than those that do not, the elected schema provides a better template for the generator than does the regular schema. Hypotheses are therefore chosen from this reduced hull.

Finally, consider cases where a small but strong cluster is discovered but quickly swamped by outside evolutionary pressures. Injection of some new members of the "lost" subspace into the population may help, as the undersampled space may deserve further investigation.

In each of the solution spaces searched by the GA, hypothesis injection on average provided detectable and reproducible improvements. Even though injection was performed only twice per experiment, hamming distance to the optimal solution was decreased by more than 10% of the problem size over the baseline case. A GA converges when the average hamming distance between individuals in a population stabilizes (see chapter 6). In several instances, the solution was discovered within one generation of the injection. In experiments, the GA converged more quickly and to better solutions when hypothesis injection was used.

Hypotheses generated by CBR tools can provide principled direction to otherwise less-productive GA search as well as help in the avoidance of deception. Although it is possible to damage the search through hypothesis injection, this risk can be reduced by meta-level application of CBR techniques. In the future, the hypothesis generator should track the progress of genetically engineered individuals and update its knowledge base. Hypothesis injection should also be continuous with a small number of individuals injected every generation and their progress tracked.

5.8 Summary

The system described in this chapter provides a tool with which users can explain discovered solutions while simultaneously discovering important aspects of the search space. Tools and techniques from case-based reasoning are able to make explicit the building blocks used by a GA. Analysis of the building blocks and the path taken by a GA through a search space provides a powerful explanatory mechanism. Salient features are apparent from the structure of the graph and are important for increasing understanding of poorly-understood functions. The explanation can be used to guide post-processing in order to improve results in case of premature convergence or no convergence.

During subsequent attempts at the task of designing a parity checker, search space size could be reduced, building blocks identified, and unimportant areas of the search space avoided The importance of the e**X**clusive-or, input location choice, and the unimportance of certain positions (unlabeled boxes in figure 5.5) provides useful knowledge to a designer. The system also allows sensitivity analysis based on genetic algorithm theory. The GA theory implies that schemas discovered earlier cause greater changes in performance compared to schemas that were discovered later.

As well as providing a system for GA explanation and search space analysis, the system provides empirical evidence that the building block hypothesis is correct, at least for the functions in table 5.1. As mentioned above, the hierarchical clustering of a set of GA individuals (distributed over several generations) according to fitness and genotype leads to nested schemas. Analysis shows that the subtrees with the most fit schemas receive the most reproductive energy while the least fit schemas are culled from the population. Although theory said that clustering on fitness and genotype should yield nested schemas (as a result of the building block hypothesis), the system was nonetheless tested on several other clustering possibilities, including more generational information and parental information. Clustering on fitness and genotype was by far the most successful technique. The nested schemas that result from such a clustering provide a coherent index for the case-base.

Although the time complexity of the fitness function may be known, the problem in this thesis is to put bounds on the number of generations until convergence. A computational tool must be able to bound time complexity. The next chapter addresses this issue.

6

Predicting time to convergence for GAs

The best theory is inspired by practice,
The best practice is guided by theory.

– Donald Knuth, IFIP 1992 address.

It is difficult to predict a genetic algorithm's behavior on an arbitrary problem. Combining genetic algorithm theory with practice, this chapter uses the average hamming distance as a syntactic metric to derive bounds on the time convergence of genetic algorithms. Analysis of a *flat* function provides worst case time complexity for static functions. Further, employing linearly computable runtime information, this thesis provides bounds on the time beyond which progress is unlikely on arbitrary static functions. As a byproduct, the analysis also provides qualitative bounds by predicting average fitness.

6.1 A Measure of Variation

Natural selection uses diversity in a population to produce adaptation. Ignoring the effects of mutation for the present, if there is no diversity there is nothing for natural selection to work on. Since GAs mirror natural selection, this chapter applies the same principles and uses a measure of diversity for estimating time to stagnation or convergence. A GA converges when most of the population is identical, or in other words, the diversity is minimal. Using average hamming distance (hamming average) as a measure of population diversity this chapter derives bounds for the time to minimal diversity, when the genetic algorithm may be expected to make no further progress (hamming convergence).

Previous work in GA convergence by Ankenbrandt, and Goldberg and Deb focuses on a lower limit on the time to convergence to a particular allele (Ankenbrandt, 1991; Goldberg and Deb, 1991) using fitness ratios to obtain bounds on time complexity. Since GAs use syntactic information to guide their search, it seems natural to use syntactic metrics in their analysis. The analysis here, using hamming averages, can predict the time beyond which qualitative improvements in the solution are unlikely. Since the schema theorem intimately links schema proportions with fitness, bounds on average fitness follow.

The next section identifies the effect of genetic operators on the diversity metric, which is the hamming average. Deriving an upper bound on the expected time to convergence suggests syntactic remedies to the problem of premature convergence and, as a side effect, how to mitigate deception in GAs. Since crossover does not affect the hamming average, extrapolating the change in the hamming average sampled during the first few generations is used to predict the hamming average in later generations. Results on a test suite of functions indicate that accurate predictions are possible.

6.2 Genetic Algorithms and Hamming Distance

The average hamming distance of a population is the average distance between all pairs of strings in a population of size N. As each member of the population is involved in $N-1$ pairs, the sample size from which to calculate the average is:

$$\frac{N(N-1)}{2}$$

Let the length of the member strings in the population be l. The hamming average of the initial population of binary strings is well approximated by the normal distribution with mean h_0 where

$$h_0 = \frac{l}{2}$$

and standard deviation s_0 given by

$$s_0 = \frac{\sqrt{l}}{2}$$

Ignoring the effect of mutation, the hamming average of a converged population is zero (mutation increases the hamming average of a converged population by an amount $\epsilon > 0$ depending on the probability of mutation). Given that the initial hamming average is $l/2$ and the final hamming average is zero, the effects of selection and crossover determine the behavior of a genetic algorithm in the intervening time.

6.3 Crossover and Average Hamming Distance

Assuming that offspring replace their parents during a crossover, all crossover operators can be partitioned into two groups based on whether or not they change the hamming average. If one parent contributes the same alleles to *both* offspring (as in masked crossover (chapter 4)) the hamming distance between the children is less than the hamming distance between their parents. This leads to a loss of genetic material, reducing population hamming average and resulting in faster hamming convergence. The vast majority of traditional operators, like one-point, two-point, ... l-point, uniform and punctuated crossover (Spears and DeJong, 1991; Schaeffer and Morishima, 1987; Schaeffer and Morishima, 1988; Syswerda, 1989), do not affect the hamming average of a population. For these crossover operators the following lemma is proved:

Lemma 1 *Traditional crossover operators do not change the average hamming distance of a given population.*

Proof: The hamming average in generation $t+1$ is proved to be the same as the hamming average at generation t under the action of crossover alone. Assuming a binary alphabet $\{a, b\}$, express the population hamming average at generation t as the sum of hamming averages of l loci, where l is the length of the chromosome. Letting $h_{i,t}$ stand for the hamming average of the i^{th} locus results in:

$$h_t = \sum_{i=1}^{l} h_{i,t}$$

where h_{t+1} is

$$h_{t+1} = \sum_{i=1}^{l} h_{i,t+1}$$

In the absence of selection and mutation, crossover only changes the order in which to sum the contributions at each locus. That is:

$$h_{i,t} = h_{i,t+1}$$

The hamming average in the next generation is therefore

$$h_{t+1} = h_t$$

Q.E.D

Having eliminated crossover from consideration since it causes no change in the hamming average, it remains for selection to provide the force responsible for hamming convergence.

6.4 Selection and Average Hamming Distance

Unlike recombination, selection intensity depends on the properties of the search space. It is the domain-dependent or problem-dependent part of a genetic algorithm. But, independent of the domain, it is possible to obtain an upper bound on the time to convergence. Obtaining an upper bound is equivalent to assuming the worst possible search space and estimating the time required for finding a point in this space. For a static function, a space on which an algorithm can do no better than random search satisfies this criterion. The *flat* function defined by

$$f(x_i) = \text{constant}$$

contains no useful information for an algorithm searching for a particular point in the space. Thus no algorithm can do better than random search on this function. In terms of selection in genetic algorithms, the flat function assigns an equal fitness to all members of a population. There is no selection pressure to lead to a decrease in hamming average since the probability of selection is now the same for every individual. However, a simple GA without mutation loses diversity and eventually converges. An expression for the time convergence on such a function gives an upper bound on the time complexity of a genetic algorithm on any static function. The GA's convergence on the flat function is caused by random genetic drift where small random variations in allele distribution cause a GA to drift and eventually converge. The time for a particular allele to become fixed due to random genetic drift is used to derive an upper bound.

If a population of size N contains a proportion p_i of a binary allele i, then the probability that k copies of allele i are produced in the following generation is given by the binomial probability distribution

$$\begin{pmatrix} N \\ k \end{pmatrix} p_i^k (1 - p_i)^{N-k}$$

This distribution is useful in calculating the probability of a particular frequency of occurrence of allele i in subsequent generations — A classical problem in population genetics. Although the exact solution is complex, approximating the probability that allele frequency takes value p in generation t is practical. Wright's approximation for intermediate allele frequencies and population sizes, as given in (Gale, 1990), is sufficient. Let $\Phi(p,t)$ stand for the probability that allele frequency takes value p in generation t where $0 < p < 1$ then

$$\Phi(p,t) = \frac{6p_0(1-p_0)}{N} \left(1 - \frac{2}{N}\right)^t$$

The probability that an allele is fixed at generation t is

$$\mathcal{P}(t) = 1 - \Phi(p,t)$$

Applying this to a genetic algorithm, assuming a randomly instantiated population at $t = 0$,

$$\mathcal{P}(t) = 1 - \frac{6p_0(1-p_0)}{N} \left(1 - \frac{2}{N}\right)^t$$

Assuming that alleles consort independently, which is true for a flat function, the expression for the probability that all alleles are fixed at generation t for a chromosome of length l alleles, is given by

$$\mathcal{P}(t,l) = \left[1 - \frac{6p_0(1-p_0)}{N}\left(1 - \frac{2}{N}\right)^t\right]^l \tag{6.1}$$

Equation 6.1 gives the time to convergence for a genetic algorithm on a flat function and is therefore an upper bound for any static function. For example, on a 50 bit chromosome and a population size of 30, this gives an upper bound of 92% on the chance of convergence in 50 generations. Experimental results get between 92% and 73% convergence starting with an initial hamming average of $50/2 = 25$. Previous work by Goldberg and Segrest on genetic drift gives a more exact albeit more computationally taxing expression for time to convergence due to genetic drift (Goldberg and Segrest, 1987). They also include the effect of mutation in their analysis, which once again is directly applicable to our problem. In fact, since the flat function costs one assignment statement and genetic operators are computationally inexpensive, running a genetic algorithm on a flat function with mutation and crossover probabilities set to practical values can cheaply produce an upper bound.

Finally, that GAs converge due to random drift gives a theoretical basis to the often observed problem of premature convergence, when the genetic algorithm converges before high fitness schemas have sufficient time to recombine into the global optimum (Thierens and Goldberg, 1993). The next section suggests one method of alleviating this problem.

6.5 Handling Premature Convergence

Nature uses large population sizes to "solve" the premature convergence problem. This is expensive, furthermore engineers need not be restricted to nature's methods but can do some genetic engineering. Mutation seems a likely candidate, and in practice, is the usual way of maintaining diversity. However, although high mutation rates may increase diversity, its random nature raises problems. Mutation is as likely to destroy good schemas as bad ones and therefore elitist selection is used to preserve the best individuals in a population. This works quite well in practice, but is unstructured and cannot insure that all alleles are always present in the population.

Instead of increasing mutation rates, picking an individual and adding its *bit complement* to the population ensures that every allele is present and the population spans the entire encoded space of the problem. The individual to be complemented can be picked in a variety of ways depending on assumptions made about the search space. Randomly selecting the individual to be complemented makes the least number of assumptions and may be the best strategy in the absence of other information. The best or worst individual could also be selected, or probabilistic methods can be used in choosing whom to complement. Instead of complementing one individual, choosing a set of individuals allows spanning the encoded space in many directions. The most general approach is to maintain the complement of every individual in the population, doubling the population size.

The presence of complementary individuals also makes a GA more resistant to deception. Intuitively, since the optimal schema is the complement of the deceptively optimal schema, it will be repeatedly created with high probability as the GA converges to the deceptive optimum (Goldberg et al., 1989). In experiments the minimum fitness individual was replaced with either the complement of a randomly picked individual in the current population, or the complement of the current best individual. Let l_i represent the number of bits needed to represent variable i in a deceptive problem, then the deceptive functions used can be described as follows

$$\text{Deceptive}(x_i) = \sum_{i=1}^{n} \left\{ \begin{array}{ll} x_i & \text{if } x_i \neq 0 \\ 2^{l_i+2} & \text{if } x_i = 0 \end{array} \right\}$$

Dec1 and *Dec2* which are 10-bit and 20-bit deceptive problems respectively were used. Letting superscripts denote the number of bits needed for each variable, Dec1 can be described by

$$\text{Dec1: Deceptive}(x_1^2,\ x_2^8)$$

and Dec2 by

$$\text{Dec2: Deceptive}(x_1^2,\ x_2^8,\ x_3^5,\ x_4^5)$$

Figure 6.1 compares the average fitness after 100 generations of a classical GA (CGA) and a GA with complements (GAC) on a Dec1. The GAC easily outperforms the classical GA, both in average fitness and the number of times the optimum was found. In most experiments the classical GA fails to find the optimum in 11 runs of the GA with a population size of 30 in a 100 generations. Since the converged hamming average for the CGA is very small, it is not expected to be ever able to find the

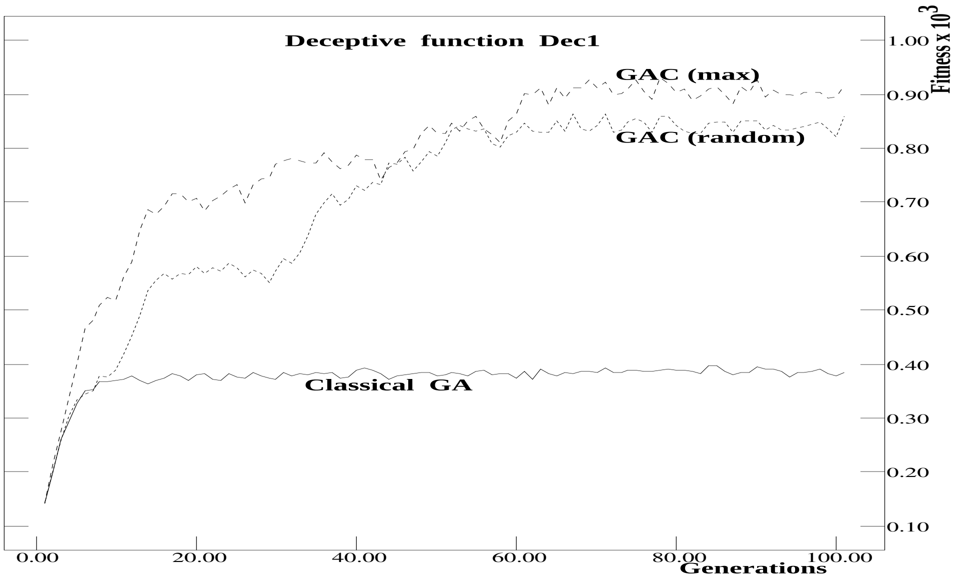

Figure 6.1: Average fitness over 100 generations of classical GA and GA with complements. GAC (max) replaces the worst individual with the complement of the current best individual. GAC (random) replaces the worst individual with the complement of a random individual in the population

global optimum. GAC on the other hand easily finds the optimum for Dec1 within a 100 generations and usually finds the optimum for Dec2 within a few hundred generations. Figures 6.2 and 6.3 show the number of times the optimum was found for Dec1 and Dec2 within a total of 100 and 1000 generations respectively. In the figures GAC (max) replaces the worst individual with the complement of the current best individual. GAC (random) replaces the worst individual with the complement of a random individual in the population.

Although this genetically engineered algorithm can mitigate certain kinds of deception, it is not a panacea and cannot guarantee an optimal solution on all problems. Messy Genetic Algorithms (Goldberg et al., 1989) can make much stronger guarantees but their computational complexity is daunting. Not surprisingly, GAC also does better than a classical GA on Ackley's One Max and no worse on the De Jong test

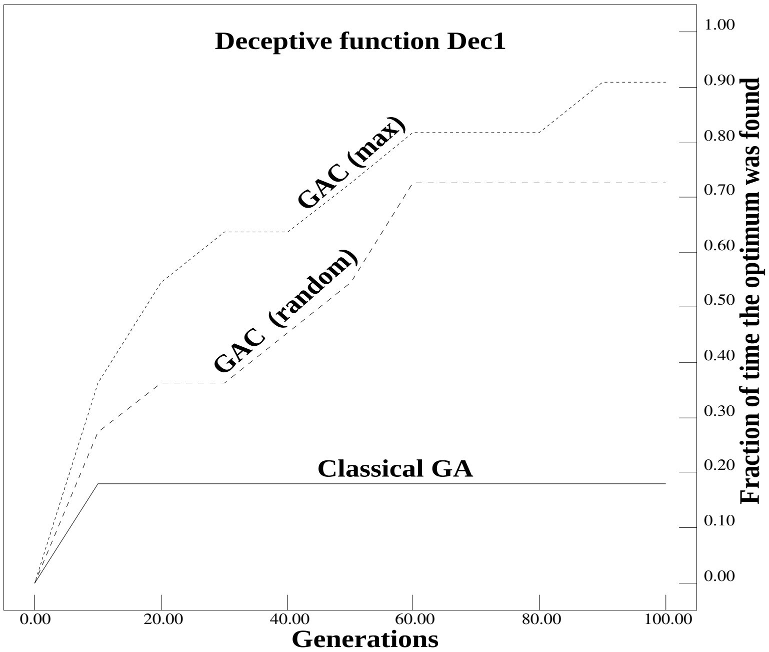

Figure 6.2: Number of times the optimum was found on **Dec1** for a classical GA compared with the same statistic for GAs with complements (GACs).

suite (Ackley, 1987; DeJong, 1975). In problems consisting of mixed deceptive and easy subproblems the GAC still does much better than the classical GA

6.6 Predicting Time To Convergence

Predicting performance on an arbitrary function is more difficult than obtaining an upper bound because of the non-linearities introduced by the selection intensity. However tracking the decrease in hamming average while a GA is working on a particular problem allows predicting the time to hamming convergence.

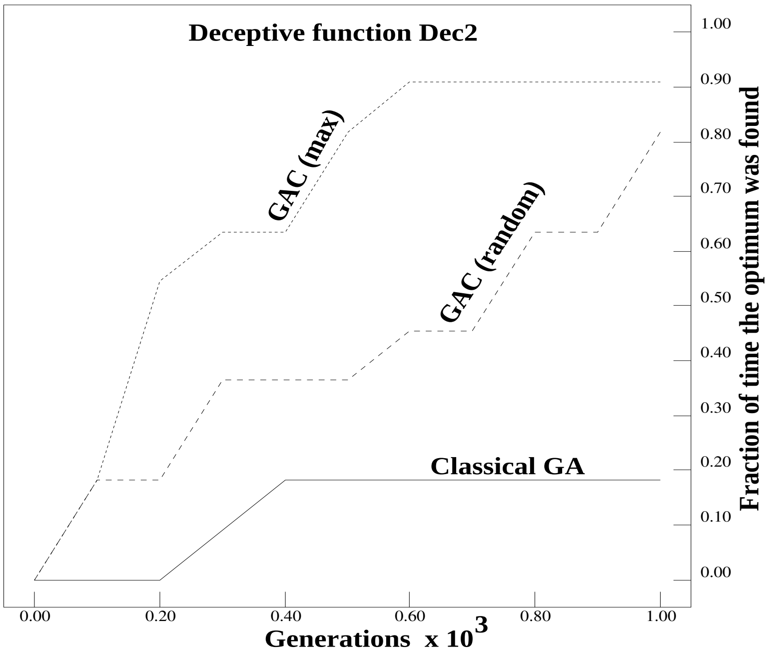

Figure 6.3: Number of times the optimum was found on **Dec2** for a classical GA compared with GAs with complements (GACs).

6.6.1 Convergence Analysis

Without mutation a GA reduces the hamming average from $l/2$ to zero. Therefore predicting the time to convergence reduces to the problem of finding the rate of decrease in hamming average. A general model for the change in hamming average per generation:

$$h_{t+1} = \mathcal{F}(h_t) \tag{6.2}$$

relates the hamming average h_t in generation t to the hamming average in the next generation. Finding $\mathcal{F}$ and solving the recurrence is enough to predict the time to convergence.

The population hamming average at generation t can be expressed as the sum of hamming averages of l loci, where l is the length of the chromosome. Letting $h_{i,t}$ stand for the hamming average of the i^{th} locus gives:

$$h_t = \sum_{i=0}^{l-1} h_{i,t} \tag{6.3}$$

Assuming a binary alphabet $\{a, b\}$, and a large population, the hamming average of a locus in the current population is well approximated by the product of the number of a's and the number of b's divided by the sample size $N(N-1)/2$, where N is the population size. More formally, letting $m(a_i, t)$ and $m(b_i, t)$ be the number of a's and b's at a locus i,

$$h_{i,t} = \frac{m(a_i, t)m(b_i, t)}{N(N-1)/2} \tag{6.4}$$

Since a and b are first-order schemas competing at locus i, their growth rates are given by the schema theorem. The expected growth rate for a schema s with fitness $f(s)$ due to selection alone is

$$m(s, t+1) = \frac{m(s,t)f(s)}{\overline{f}_t} \tag{6.5}$$

Here $\overline{f}_t$ is the population's average fitness in generation t. As only first-order schema are being considered, there is no modification needed to incorporate crossover's effects. Ignoring mutation for the present, the hamming average in generation $t+1$ at locus i can be calculated as follows. First:

$$h_{i,t+1} = \frac{m(a_i, t+1)m(b_i, t+1)}{N(N-1)/2}$$

Then using the schema theorem $h_{i,t+1}$ can be expressed in terms of $h_{i,t}$,

$$h_{i,t+1} = \frac{m(a_i, t)f(a_i)}{\overline{f}_t}\frac{m(b_i, t)f(b_i)}{\overline{f}_t}\frac{1}{N(N-1)/2}$$

Substituting using equation 6.4:

$$h_{i,t+1} = h_{i,t}\frac{f(a_i)f(b_i)}{\overline{f}_t^2} \tag{6.6}$$

The solution to this recurrence is

$$h_{i,t} = [f(a_i)f(b_i)]^t \frac{h_{i,0}}{\prod_{k=0}^{t} \overline{f}_k^2} \tag{6.7}$$

The denominator needs to be computed. This is an interesting problem not only because it helps in predicting the hamming average, but because it implies the possibility of fitness prediction. Once again concentrating on first-order schemas, consider the average fitness in generation t at locus i.

$$\overline{f}_{i,t} = \frac{f(a_i)m(a_i, t) + f(b_i)m(b_i, t)}{N}$$

But $\overline{f}_{i,t}$ also approximates the population average fitness in low epistasis problems. The approximation gets better as epistasis decreases and our estimates of first-order schema fitnesses get better. Keeping this assumption in mind and dropping the first subscript

$$\overline{f}_t = \frac{f(a_i)m(a_i,t) + f(b_i)m(b_i,t)}{N} \tag{6.8}$$

Using the schema theorem (equation 6.5) to replace the t terms on the right hand side

$$\overline{f}_t = \frac{1}{\overline{f}_{t-1}} \frac{f(a_i)^2 m(a_i,t-1) + f(b_i)^2 m(b_i,t-1)}{N}$$

Multiplying both sides by $\overline{f}_{t-1}$

$$\overline{f}_t \overline{f}_{t-1} = \frac{m(a_i,t-1)f(a_i)^2}{N} + \frac{m(b_i,t-1)f(b_i)^2}{N}$$

But by the schema theorem (equation 6.5)

$$m(a_i,t-1) = \frac{m(a_i,t-2)f(a_i)}{\overline{f}_{t-2}} \quad \text{And} \quad m(b_i,t-1) = \frac{m(b_i,t-2)f(b_i)}{\overline{f}_{t-2}}$$

Therefore

$$\overline{f}_t\overline{f}_{t-1} = \frac{m(a_i,t-2)f(a_i)^3}{N\overline{f}_{t-2}} + \frac{m(b_i,t-2)f(b_i)^3}{N\overline{f}_{t-2}}$$

Multiplying by $\overline{f}_{t-2}$

$$\overline{f}_t\overline{f}_{t-1}\overline{f}_{t-2} = \frac{m(a_i,t-2)f(a_i)^3}{N} + \frac{m(b_i,t-2)f(b_i)^3}{N}$$

In general,

$$\vdots$$

$$\prod_{k=0}^{t}\overline{f}_k = \frac{m(a_i,0)f(a_i)^{t+1}}{N} + \frac{m(b_i,0)f(b_i)^{t+1}}{N} \tag{6.9}$$

which is the needed expression. This equation's significance in fitness prediction is discussed in the next section. For the present, substituting equation 6.9 in 6.7 gives $h_{i,t}$ in terms of the initial hamming average.

$$h_{i,t} = [f(a_i)f(b_i)]^t \frac{h_{i,0}}{\left[\frac{m(a_i,0)f(a_i)^{t+1}}{N} + \frac{m(b_i,0)f(b_i)^{t+1}}{N}\right]^2}$$

For a randomly initialized population

$$m(a_i,0) = m(b_i,0) = N/2$$

Therefore

$$h_{i,t} = [f(a_i)f(b_i)]^t \frac{4h_{i,0}}{[f(a_i)^{t+1} + f(b_i)^{t+1}]^2}$$

Rearranging terms

$$h_{i,t} = \frac{4\,[f(a_i)f(b_i)]^t}{[f(a_i)^{t+1} + f(b_i)^{t+1}]^2} h_{i,0}$$

Here $h_{i,0}$ is the initial hamming average at locus i. For a randomly initialized population, the hamming average for each locus is $1/2$. Replacing $h_{i,0}$ by $1/2$

$$h_{i,t} = \frac{2\,[f(a_i)f(b_i)]^t}{[f(a_i)^{t+1} + f(b_i)^{t+1}]^2}$$

Summing over i from 0 to $l-1$ gives the population hamming average in generation t

$$h_t = \sum_{i=0}^{l-1} h_{i,t} = 2\sum_{i=0}^{l-1} \frac{[f(a_i)f(b_i)]^t}{[f(a_i)^{t+1} + f(b_i)^{t+1}]^2}$$

the expression for the hamming average in generation t, depends only on the fitnesses of alleles in the chromosome.

$$h_t = 2\sum_{i=0}^{l-1} \frac{[f(a_i)f(b_i)]^t}{[f(a_i)^{t+1} + f(b_i)^{t+1}]^2} \tag{6.10}$$

6.6.2 Handling Mutation

With mutation the hamming average does not converge to zero, rather a population converges when the *rate of change* of hamming average is very low or zero. Consider mutation's effect on the number of a's and b's. Mutation flips a bit with probability p_m, changing the number of copies of a and b in subsequent generations. Without selection

$$\begin{aligned} m(a_i, t+1) &= m(a_i, t)\left[1 - p_m\right] + m(b_i, t)p_m \\ m(b_i, t+1) &= m(b_i, t)\left[1 - p_m\right] + m(a_i, t)p_m \end{aligned}$$

since the a's become b's and vice-versa. Incorporating selection

$$\begin{aligned} m(a_i, t+1) &= \frac{f(a_i)m(a_i, t)\left[1 - p_m\right] + f(b_i)m(b_i, t)p_m}{\overline{f}_t} \\ m(b_i, t+1) &= \frac{f(b_i)m(b_i, t)\left[1 - p_m\right] + f(a_i)m(a_i, t)p_m}{\overline{f}_t} \end{aligned} \tag{6.11}$$

The equation for hamming distance in generation t is

$$h_{i,t} = \frac{m(a_i, t)m(b_i, t)}{N(N-1)/2} \tag{6.12}$$

To solve this $m(a_i, t)$ is needed in terms of $m(a_i, 0)$. Substituting $m(b_i, t) = N - m(a_i, t)$ in equation 6.11

$$m(a_i, t) = \frac{f(a_i)m(a_i, t-1)(1 - p_m) + f(b_i)p_m\left[N - m(a_i, t-1)\right]}{\overline{f}_{t-1}}$$

Collecting terms

$$m(a_i, t) = \frac{m(a_i, t-1)\,[f(a_i)(1-p_m) - f(b_i)p_m]}{\overline{f}_{t-1}} + \frac{f(b_i)p_m N}{\overline{f}_{t-1}}$$

This is of the form

$$m(a_i, t) = \frac{c\,m(a_i, t-1)}{\overline{f}_{t-1}} + \frac{d}{\overline{f}_{t-1}}$$

where

$$c = f(a_i)(1-p_m) - f(b_i)p_m$$

$$d = f(b_i)p_m N$$

The solution to this recurrence is

$$m(a_i, t) = \frac{c^t m(a_i, 0)}{\prod_{k=0}^{t-1} \overline{f}_k} + \frac{d}{\prod_{k=0}^{t-2} \overline{f}_k} \sum_{i=0}^{t-2} c^i \prod_{k=i+1}^{t-2} \overline{f}_k$$

Before calculating the product of fitnesses, rewrite the second term as

$$m(a_i, t) = \frac{c^t m(a_i, 0)}{\prod_{k=0}^{t-1} \overline{f}_k} + \frac{d\,\overline{f}_{t-1}}{\prod_{k=0}^{t-1} \overline{f}_k} \sum_{i=0}^{t-2} c^i \prod_{k=i+1}^{t-2} \overline{f}_k \tag{6.13}$$

Similarly letting

$$u = f(b_i)(1 - p_m) - f(a_i)p_m$$

$$w = f(a_i)p_m N$$

results in

$$m(b_i,t) = \frac{u^t m(b_i,0)}{\prod_{k=0}^{t-1} \overline{f}_k} + \frac{w\,\overline{f}_{t-1}}{\prod_{k=0}^{t-1} \overline{f}_k} \sum_{i=0}^{t-2} u^i \prod_{k=i+1}^{t-2} \overline{f}_k \tag{6.14}$$

Substituting these values into equation 6.12 directly

$$h_{i,t} = \frac{\left[c^t m(a_i,0) + d\,\overline{f}_{t-1} \sum_{i=0}^{t-2} c^i \prod_{k=i+1}^{t-2} \overline{f}_k\right] \left[u^t m(b_i,0) + w\,\overline{f}_{t-1} \sum_{i=0}^{t-2} u^i \prod_{k=i+1}^{t-2} \overline{f}_k\right]}{\frac{N(N-1)}{2} \prod_{k=0}^{t-1} \overline{f}_k^2} \tag{6.15}$$

Unlike the case without mutation, it is difficult to find a simple expression for the product of fitness averages. In addition, the calculations implied by equation 6.15 are cumbersome. For practical prediction this chapter resorts to an approximate model.

6.7 Practical Prediction

The intuition needed to address the problem is that the schema theorem works both ways. If schema fitnesses are known, proportions can be predicted — if proportions are tracked, fitnesses can be predicted. Rewriting equation 6.5 as

$$f(a_i) = \frac{m(a_i, t+1)\overline{f}_t}{m(a_i, t)}$$

Keeping track of $m(a_i, t)$ over a few generations should give an estimated fitness for a_i. Mutation can be handled in the same way starting with equation 6.11.

$$\begin{aligned} \overline{f}_t m(a_i, t+1) &= f(a_i)m(a_i, t)\,[1 - p_m] + f(b_i)m(b_i, t)p_m \\ \overline{f}_t m(b_i, t+1) &= f(b_i)m(b_i, t)\,[1 - p_m] + f(a_i)m(a_i, t)p_m \end{aligned} \tag{6.16}$$

Solving for $f(b_i)$ using the second expression

$$f(b_i) = \frac{\overline{f}_t m(b_i, t+1) - f(a_i)m(a_i, t)p_m}{m(b_i, t)(1 - p_m)}$$

Substituting back in the first expression in equation 6.16 and solving for $f(a_i)$

$$\overline{f}_t m(a_i, t+1) = f(a_i)m(a_i, t)(1 - p_m) + [m(b_i, t)p_m]\left[\frac{\overline{f}_t m(b_i, t+1) - f(a_i)m(a_i, t)p_m}{m(b_i, t)(1 - p_m)}\right]$$

which leads to

$$\overline{f}_t m(a_i, t+1) = f(a_i)m(a_i, t)(1-p_m) + \frac{\overline{f}_t m(b_i, t+1)p_m}{(1-p_m)} - \frac{f(a_i)m(a_i, t)p_m^2}{(1-p_m)}$$

multiplying by $(1-p_m)$

$$\overline{f}_t m(a_i, t+1)(1-p_m) - \overline{f}_t m(b_i, t+1)p_m = f(a_i)m(a_i, t)(1-p_m)^2 - f(a_i)m(a_i, t)p_m^2$$

from which the expression for $f(a_i)$ can be computed

$$f(a_i) = \frac{\overline{f}_t m(a_i, t+1)(1-p_m) - \overline{f}_t m(b_i, t+1)p_m}{m(a_i, t)(1-2p_m)} \tag{6.17}$$

and subsequently an expression for $f(b_i)$

$$f(b_i) = \frac{\overline{f}_t m(b_i, t+1)(1-p_m) - \overline{f}_t m(a_i, t+1)p_m}{m(b_i, t)(1-2p_m)} \tag{6.18}$$

The two equations 6.17 and 6.18 above can be used to estimate first order schema fitnesses by tracking their proportions over time. Using these estimates in the expression for average fitness $\overline{f}_t$ in generation t gives

$$\overline{f}_t = \frac{f(a_i)m(a_i, t) + f(b_i)m(b_i, t)}{N} \tag{6.19}$$

where N is the population size. This results in being able to predict fitness averages. In other words, tracking first-order schema proportions allows estimation of their fitness. Next, using these first-order schema fitnesses allows predicting their future proportions from the schema theorem (equation 6.11). Finally, knowing first-order schema fitnesses and proportions allows predicting the subsequent population average fitness, which is needed for estimating future proportions and so on. Although a closed-form solution is not available for the case with mutation, the recurrences can be programmed and run on a computer for a large number of generations.

Assuming that string similarity implies fitness similarity, the variance of fitness is a measure of the nonlinearities in the genome and hence a measure of epistasis. Fortunately, competing schemas with a large fitness variance feel high selection pressure and quickly lose diversity, diminishing epistatic effects. Since a GA exponentially reduces high variances, $O(logN)$ generations should suffice to reduce most nonlinear effects.

The following methodology is therefore used for practical prediction:

- Instead of predicting $f(a_i)$ or $f(b_i)$ by averaging the contributions of all individuals which contain a_i or b_i, use equations 6.17 and 6.18 over a small number of generations to get an *adjusted fitness* value for each first-order schema.

- Start sampling allele proportions after $O(\log N)$ generations.

Using an adjusted fitness computed this way, it is possible to predict the growth rate of first-order schema and therefore the population average fitness and population hamming average in subsequent generations.

6.8 Upper and Lower Bounds

Consider the effect of newly discovered schemas on the predictions. Fresh schemas that decrease adjusted fitnesses have little effect since they are quickly eliminated by selection. When higher fitness schemas are discovered by crossover, the fitness of alleles participating in these schemas increases. This means that the predicted hamming average will be greater than or equal to the observed value giving an *upper* limit on the time to convergence. The predictions therefore provide upper bounds on the hamming average and time to convergence to that hamming average. Such an estimate is more desirable than an overly optimistic estimate: It is better to let a GA run too long than to stop it early.

The variance in adjusted fitness for an allele bounds the number of generations sampled to get an accurate estimate of the allele's fitness. This variance bounds the current fitness of schemas in which the allele participates. At a particular locus, using the allele with the least variance to predict growth gives a conservative estimate of hamming average. Low variance implies a good approximation to the current fitness of the allele and provides an upper bound on our predicted hamming average. Similarly, using the allele with greater variance gives an optimistic estimate or lower bound. These bounds should surround the predicted value of the hamming average. Results given in the following sections support this preliminary analysis.

Noisy functions and noise induced by mutation may cause less desirable, more unpredictable behavior. This problem can be solved by having a larger sample space, that is, using more generations to calculate adjusted fitnesses.

For much the same reasons above, calculating adjusted fitnesses allows predicting a *lower* limit on the fitness average because these fitnesses only incorporate the effects of current schemas. Any new schemas (created by crossover) that increase the adjusted fitness of an allele can lead to a higher than predicted fitness average. In multimodal functions, the opposite may happen. The GA could initially concentrate around one optimum and then move toward another optimum. Since the calculated adjusted fitnesses may no longer be correct and it takes a GA time to overcome the inertia created by the first optimum, the predicted fitness average may be higher than the actual fitness average.

6.8.1 Comparing Predictions

Since hamming average prediction is quite robust and a first-order approximation may be interesting enough, a linear approximation of $\mathcal{F}$ in equation 6.2 can be tried out for prediction. Assuming a quick and simple model for the case without mutation

$$h_{t+1} = a h_t$$

whose general solution is given by

$$h_t = a^t h_0 \tag{6.20}$$

This chapter compares this simple equation's predictions with the predictions generated using the analysis in section 6.7. The task is to predict the hamming average at generation 50 for the following functions:

1. Flat: $f(x_1, x_2) = 10$, with a 50 bit chromosome.
2. DeJong's five functions: $F1 \ldots F5$ (De Jong, 1975).
3. One Max: $f(X) = |X|$ where $|X|$ stands for the norm of the bit string X, for both a 50 (OneMaxI) and a 200 (OneMaxII) bit chromosome.

All problems were reduced to maximization problems. The GA population size in all experiments was 30, using proportional selection and two-point crossover. a in equation 6.20 was computed from successive hamming averages in generations 0 through 10. Table 6.1 summarizes the results. The last two columns predict lower and upper bounds from the analysis in section 6.7 (equations 6.17, 6.18, and 6.19), using proportions sampled in generations $\lceil \log 30 \rceil = 5$ through 9 only. Since the encoding for the DeJong functions used a sign bit, 5 samples were optimistically chosen as sufficient. The experimental values are averages over 11 runs with no mutation. The standard deviations over 11 runs of the GA are given in parentheses.

Surprisingly, the values predicted using the rough approximation are quite good and are within a standard deviation of the observed values. That good results come from even such a simple model clearly indicates the validity of this approach. Predicted upper and lower bounds on the upper limit include the observed values except for OneMaxI whose lower bound is greater than the observed hamming average. This implies a large variance in sampled allelic fitnesses. Intuitively, the observed fitness

Function	*Observed*		*Predicted*		
	h_0	h_{50}	h_{50} eqn 6.20	h_{50} lower	h_{50} upper
Flat	25	4.6 (2.4)	5.1 (2.2)	4.3 (1.3)	9.3 (1.1)
F1	15	2.1 (1.1)	2.2 (1.2)	2.1 (0.8)	4.6 (1.3)
F2	12	2.1 (0.9)	2.5 (1.5)	1.7 (1.0)	4.3 (1.1)
F3	25	3.5 (1.7)	3.4 (1.5)	3.2 (1.2)	7.9 (2.0)
F4	120	18.6 (8.0)	25.3 (11.5)	17.4 (6.5)	42.2 (5.7)
F5	17	2.9 (1.7)	4.2 (2.3)	2.8 (1.1)	5.7 (1.0)
OneMaxI	25	3.7 (2.3)	4.7 (3.1)	4.2 (1.0)	8.0 (2.1)
OneMaxII	100	15.8 (8.1)	16.0 (9.7)	12.8 (5.4)	29.8 (4.8)

Table 6.1: Comparing actual and predicted hamming averages (no mutation)

of an allele depends on the number of ones in the entire chromosome, leading to the observed conservative estimate in the absence of mutational noise. OneMax should be expected to create greater problems when including mutation.

Next, realizing that mutation makes the final hamming average greater than 0, another related simplified model is used to predict hamming average at convergence for the same set of problems but with the mutation rate set to 0.01. The model is given below.

$$h_{t+1} = ah_t + b \tag{6.21}$$

The general solution is

$$h_t = a^t h_0 + b\frac{1 - a^t}{1 - a}$$

In these experiments the GA ran until hamming convergence when the hamming average did not change significantly over time. Experimentally, 500 generations was sufficient for the problem set. Calculating the values of a and b in equation 6.21 using the method of least squares, the hamming average is predicted at convergence (generation 500). This is found by setting $\epsilon = h_t = h_{t+1}$ and solving for ϵ, resulting in

$$\epsilon = \frac{b}{1-a}$$

Table 6.2 compares the observed hamming average at generation 500 with the predicted values. Column three tabulates the hamming average at convergence and its standard deviation over the 11 runs. Columns four and five are the predicted hamming averages using equation 6.21 and data from generations 5 through 20 and 5 through 45 respectively. The simpler model (curve fitting) does much worse when mutation is incorporated. It requires more samples, and even predicts negative hamming averages at convergence. However, the more refined model developed in this chapter predictably needs no extra information and therefore still uses samples from generations 5 through 9 to produce the last two columns.

The predictions of the model developed in this chapter, displayed in table 6.2, are very close to the observed values. Although the hamming average after 500 generation is large for $F4$ and OneMaxII, there is no significant change even after 2000 generations. This just indicates that once a GA reaches the equilibrium hamming average, some other search method should be tried especially if the converged hamming average is large. Predicted upper and lower bounds are still good except for OneMaxII. Here, noise induced by mutation is the major culprit. The high variance

Function	*Observed*		*Predicted*			
	h_0	h_{500}	h from 25	h from 45	h lower	h upper
Flat	25	11.2 (1.9)	14.4 (6.0)*	11.4 (1.5)	7.7 (0.9)	11.2 (1.3)
F1	15	5.8 (1.1)	9.8 (10.2)	6.4 (1.3)	4.1 (0.5)	6.1(1.3)
F2	12	4.3 (1.2)	7.3 (2.4)	5.7 (1.4)	3.5 (0.9)	5.4 (1.0)
F3	25	8.9 (1.7)	8.9 (3.7)*	9.1 (2.1)	6.3 (1.1)	9.5(1.3)
F4	120	48.9 (4.7)	64.3 (14.0)*	49.3 (12.2)*	32.3 (3.9)	51.4 (4.7)
F5	17	6.8 (1.3)	11.3 (4.4)*	7.2 (2.3)	4.8 (0.7)	6.8 (1.3)
One Max	25	9.2 (1.7)	10.3 (4.0)*	9.0 (2.2)	7.3 (1.3)	11.3(1.7)
One Max	100	45.7 (5.3)	51.5 (11.4)	40.3 (13.6)	25.7 (4.6)	38.7(4.5)

Table 6.2: Comparing actual and predicted hamming averages with the probability of mutation set to 0.01, the $*$'s indicate that the negative values predicted in some runs were discarded.

in allele fitnesses suggests using a larger sample space to increase the signal to noise ratio. Sure enough, using an additional 15 generations to sample fitnesses suffices to correct the problem and give good bounds on OneMaxII.

6.8.2 Predicting Average Fitness

Since a simplified curve fitting model gave good results on hamming average prediction, equation 6.21 was used for fitness prediction as well. Table 6.3 predicts the average fitness (F) at convergence without mutation. Column two gives the initial average fitness and column three the converged average fitness in generation 50. The standard deviations of the observed and predicted average fitnesses over the 11 runs are in parentheses. The next two columns are the predicted fitness averages using equation 6.21 from fitnesses sampled in generations 5 through 15 and 5 through 25 respectively. The last two columns use equation 6.19 to provide conservative and

Function	Observed		Predicted			
	F_0	F_{500}	$F/10$	$F/20$	F lower	F upper
F1	53.6	74.5 (2.8)	72.9 (14.6)	72.8 (5.2)	71.7 (4.3)	72.8 (4.5)
F2	3462.6	3931.1 (69.1)	3856.2 (71.4)	3872.6 (65.7)	4031.0 (77.2)	4098.9 (109.0)
F3	32.4	49.3 (2.9)	43.0 (3.3)	51.0 (14.4)	40.8 (2.3)	41.9 (2.5)
F4	970.0	1055.4 (40.8)	1023.0 (31.0)	1020.1 (57.2)	1040.1 (27.5)	1055.5 (27.8)
F5	3.99651	3.99796 (0.0)	3.9972 (0.0)	3.99679 (0.0)	4.2 (0.0)	4.3 (0.0)
One Max	24.6	33.5 (2.3)	36.0 (12.2)	30.6 (11.2)	29.8 (1.2)	30.3 (1.3)
One Max	99.9	110.9 (5.4)	124.8 (59.8)	109.8 (5.9)	109.5 (3.2)	111.5 (2.7)

Table 6.3: Comparing actual and predicted fitness averages (no mutation)

optimistic bounds on the average fitness. As stated earlier, both bounds are expected to underestimate the observed fitness, unless the function is multimodal or has high fitness variance. Since the average fitness is a byproduct of hamming average prediction, the hamming average data from generations 5 through 10 is used for fitness prediction. The *flat* function is not considered.

The simple linear model's predictions of average fitness are are also within about one standard deviation of observed values. It is more interesting to look at the upper and lower bounds. As expected F2 and F5 which are multimodal, have higher predicted fitnesses. Other predictions serve as conservative and optimistic lower bounds.

Table 6.4 summarizes results on the same set of problems with mutation probability set to 0.01. Mutation's effect forces good predictions to use even larger data samples for the simpler linear model. The upper and lower bounds again fall out of the hamming average data for generations 5 through 9 with mutation.

Function	Observed		Predicted			
	F_0	F_{500}	$F/20$	$F/45$	F lower	F upper
F1	53.6	75.0 (1.8)	71.2 (3.8)	73.4 (2.6)	70.8 (4.2)	72.4 (4.5)
F2	3462.6	3919.4 (66.1)	3842.6 (43.3)	3866.8 (39.9)	3959.4 (115.0)	4029.0 (142.0)
F3	32.4	49.6 (1.9)	55.3 (21.6)	46.9 (2.6)	40.4(2.9)	41.5 (2.9)
F4	970.0	1058.6 (24.5)	1243.3 (615.8)	1077.0 (68.6)	1031.1 (22.4)	1052.7 (23.5)
F5	3.996	3.997 (0.00)	3.996 (0.00)	3.997 (0.00)	4.20 (0.04)	4.25 (0.06)
One Max	24.6	35.2 (1.6)	31.0 (4.5)	34.8 (5.0)	28.8 (1.1)	29.3 (1.2)
One Max	99.9	110.7 (3.6)	107.4 (3.6)	108.8 (5.2)	108.9 (2.9)	111.2 (3.0)

Table 6.4: Comparing actual and predicted fitness averages with the probability of mutation set to 0.01

These preliminary results support this chapter's model of genetic algorithm behavior. Although curve fitting has its uses in providing a very rough approximation, it cannot predict the effects of changes in GA parameters and does badly in the presence of mutation. Using the analysis in section 6.7 it is easy to predict the time to convergence on a problem. This is just the time at which the change in hamming average is insignificant. Knowing available computing resources fixes the hamming average below which exhaustive search of the remaining space can be undertaken, and therefore the time needed to deliver on an application. In practice, instrumenting a GA to keep estimates of allele fitness and proportions should reduce the time spent experimenting with various parameter settings and let the developer know the computing resources needed. If the problem encoding is highly epistatic or otherwise undesirable from a prediction point of view, just keeping track of the hamming average and rate of change of hamming average indicates the progress of a genetic algorithm on the problem. In addition, incorrect predictions from the theory indicate that the problem encoding exhibits these undesirable properties. This is useful information about the problem space and can be used in changing the GAs parameters or the problem's encoding.

6.9 Summary

Analyzing a GA is complicated because of its dependence on the application domain and our lack of understanding of the mapping between our view of the application domain and the GA's view of the same domain. The GA's view of the domain is implicit in the chosen encoding. However, the overall behavior is predictable and can be deduced from sampling suitable metrics.

Analysis of the syntactic information in the encoding gives a surprising amount of knowledge. This chapter derived an upper bound on the time complexity from considering the effects of drift on allele frequency. Then, from a model of the effect of selection and mutation on the behavior of genetic algorithms, the analysis was able to approximate two solution characteristics at convergence for static search spaces.

1. Bounds on an upper limit to the average hamming distance of a population at convergence. In addition to providing time limits, the average hamming distance of the population also denotes the remaining amount of work.
2. The amount of work possible by the GA, indicated by the average fitness at convergence.

Combining fitness prediction with hamming average prediction indicates how much progress is possible with a GA along with bounds on how much remaining work can be done. The latter is especially important when including mutation. The predicted average fitness indicates how much progress is possible, but it is the predicted hamming average that denotes the remaining work.

This dissertation has helped develop the foundations for the use of genetic algorithms as a computational tool for design. The last chapter summarizes these developments, discusses their limitations and makes explicit the questions arising from the research.

7

Discussion and Conclusions

> There must be a beginning of any great matter, but the continuing unto the end until it be thoroughly finished yields the true glory.
>
> Sir Francis Drake.

This thesis' main objective — to establish genetic algorithms as a computational tool for design — was achieved by solving four problems considered to be hindering the applicability of genetic algorithms in design. Design was cast as a search through a state space of possible designs, and genetic algorithms were introduced as the search method of choice and contrasted with other applicable search techniques. The mapping between genetic algorithms and the three phase model of design was established. Subsequently, four open problems lying on the path to achieving this objective were identified and overcome. This chapter summarizes the problems and their solutions, discusses results and limitations, and points out areas for future work raised by this dissertation.

7.1 Four Contributions

Searching in poorly–understood domains is a somewhat unpredictable proposition. There are few guarantees about whether a solution will be found, and if found, what its quality will be. The underlying assumptions made by a search algorithm determine its speed, flexibility, and whether it will in fact find a solution. If an acceptable solution is found, it can be used. However, when a solution is not found, what can the search algorithm's behavior tell us about the search space? This is an important question because 1) any information about a poorly understood space can be made use of in subsequent investigations of the problem and 2) not using available information is, to say the least, wasteful. In design, extracting and using domain information is especially important since sensitivity analysis and post-processing also depend on knowledge gleaned during the course of search. This point of view forms the backdrop for discussing the results in this thesis.

The four problems identified in establishing genetic algorithms as a computational tool for design were:

1. Determining where genetic algorithms are better.

2. Surmounting encoding biases in genetic search.

3. Analyzing genetic-algorithm-generated solutions.

4. Computational complexity of genetic algorithms.

This thesis overcame each of the four problems above and illustrated the solutions with examples from circuit design, function optimization and floorplanning. The next four sections review the issues, results and limitations of this work.

7.2 Where GAs are better

Once the mapping between genetic algorithms and the three–phase model of design was established, the question became "Why should a designer use a genetic algorithm rather than a hill–climber?" A related question is "What algorithm should a designer use on the problem at hand?" Chapter 3 answered these questions by defining search spaces in which genetic algorithms perform better, and suggested that such spaces may be found in design problems formulated as multiobjective or multicriteria optimization problems. In many cases, such problems involve tradeoffs among possibly conflicting criteria. Spaces where recombination of two or more single-criterion optima leads toward a global optimum, with a deceptive basin in-between to trap hill-climbers, were found to be well suited for genetic algorithms, especially ones that use pareto optimality in their selection process (see figure 7.1). Pareto-optimal selection maintains the necessary niches until recombination produces the global optimum. The dissertation defined a problem in hamming space with these properties (figure 7.1) and empirically compared performance on the problem for three algorithms:

1. A classical GA with two-point crossover.

2. A stochastic hill-climber (GA without crossover).

3. A GA with pareto optimal selection and two-point crossover.

The GA with pareto-optimal selection always found the optimum more often than the others, while the classical GA did better than the stochastic hill-climber.

Pareto-optimal selection also eliminates the need to combine disparate criteria into a single fitness as is usual in genetic algorithms. The method described in this chapter was distributable and computationally less expensive compared to other suggested methods for incorporating pareto optimality (Goldberg, 1989b).

When choosing a search algorithm to attack a particular problem, current choices include simulated annealing (a stochastic hill–climber) and genetic algorithms with and without crossover. Most other methods are variations on these algorithms (Ackley, 1987; Mahfoud and Goldberg, 1992). Simulated annealing (SA) is a stochastic hill–climber inspired through an analogy with the cooling of metals (Kirkpatrick et al., 1983). It starts with a high *temperature* T and a randomly generated initial solution i with a fitness F_i. To find a solution that minimizes F, the SA generates a new solution near i and accepts this new solution j as the current solution if the fitness of the new solution, F_j, is less than F_i. If F_j is not less than F_i then j may still be accepted as the current solution with a probability depending on T. As the temperature decreases during the course of a run, the probability of accepting j when F_j is not less than F_i decreases. Initially, at high temperatures almost any change is accepted and the algorithm stresses exploration over exploitation of the search space. At lower temperatures the probability of accepting worse solutions is low and the algorithm stresses exploitation over exploration and converges on a solution.

The differences between a genetic algorithm with no crossover (SHC) and a simulated annealer are 1) the existence of a cooling schedule (the rate of decrease of temperature), 2) the fact that simulated annealers are not population-based, and 3) the convergence behavior, which is usually very different. Typical convergence behavior for both algorithms on a maximization problem is shown in figure 7.2.

In general, choosing among blind search algorithms is not easy. The assumptions underlying the algorithms' behavior, the structure of the search space, domain knowledge, available resources, and personal preference affect this choice. As shown in chapter 3, if there are structured local optima leading towards a global optimum with deceptive basins in between, a genetic algorithm outperforms other algorithms. Since spaces with such properties may be found in multicriteria optimization (a large number of design problems are multicriteria optimization problems), choosing genetic algorithms to attack these problems makes sense.

The convergence behavior also provides a means of choosing between simulated annealing and genetic algorithms. GAs quickly improve the quality of their solutions, while initially, a simulated annealer only slowly improves quality. Thus, to get a quick solution, a designer may choose a genetic algorithm over a simulated annealer.

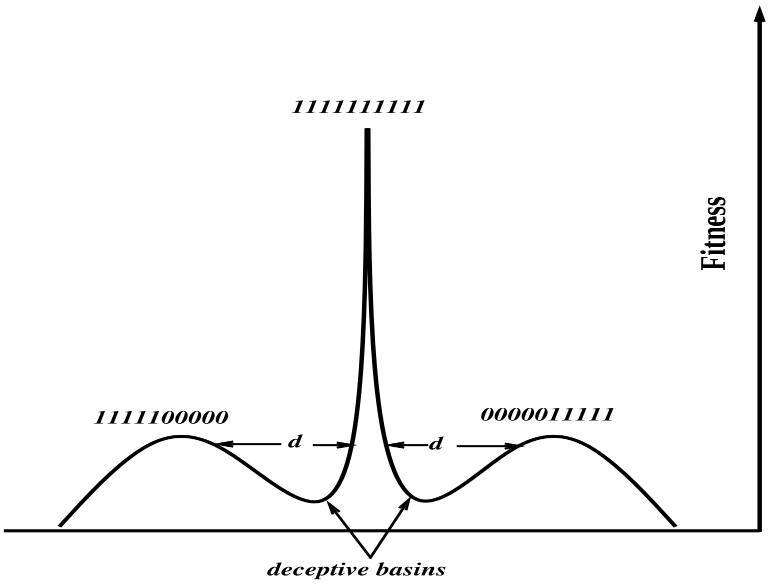

Figure 7.1: The type of spaces easy for a genetic algorithm and hard for a stochastic hill–climber

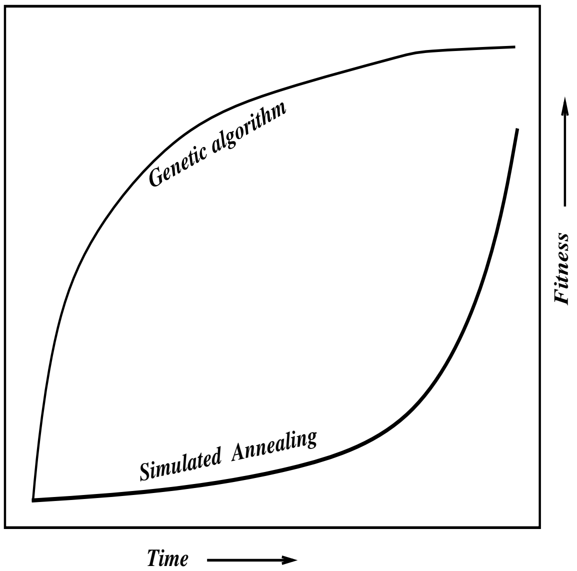

Figure 7.2: Comparing convergence behavior of a simulated annealer and a genetic algorithm

7.2.1 Limitations and Future Work

Although genetic algorithms outperform stochastic hill–climbers in the kind of spaces described in the chapter, a stronger approach would identify and analyze the properties of a whole range of spaces; from those that are clearly GA-*easy* and stochastic hill–climbing hard, to spaces that are *break-even*, where GAs and stochastic hill–climbers are expected to perform similarly. Unimodal spaces clearly confer an advantage to hill–climbers. This thesis identifies one kind of GA-easy spaces, especially relevant to design. A more general theory that includes schema fitness and proportions is an open problem and a matter for further research. Another interesting research problem lies in exploring niche formation with pareto-optimal selection. Stability of niches, cross-breeding among niches, and the role of deception need to be analyzed.

7.3 Mitigating Encoding Problems

Once a designer has decided to use a genetic algorithm, there comes the task of encoding a design problem. The biases generated by an encoding and surmounting unfavorable biases formed the subject of chapter 4. In design, epistatic encodings make a GA's preference for schemas of short defining length a liability. A designer genetic algorithm (DGA) that used a masked crossover operator to identify and preserve high-fitness schemas of arbitrary defining length was introduced to alleviate these problems. Elitist selection was also introduced, both to mesh with masked crossover's properties and to increase performance. A designer genetic algorithm increased the domain of application of genetic algorithms by relaxing the emphasis on schemas of short defining length with an increase in cost of only a constant factor per generation. Comparing the performance of a classical genetic algorithm and a designer genetic algorithm also provided information about the space. If the DGA did better, it meant that highly-fit, low-order schemas were mostly of large defining length and/or the encoded problem was not deceptive. On the other hand, if the classical genetic algorithm did better, it meant that the encoding followed the principle of meaningful building blocks and/or the problem was partially deceptive. The example used to illustrate these results, combinational circuit design, was a *satisficing* problem. The maximum fitness achievable was known, and the problem was to find a structure that implemented the specified function.

This chapter illustrated the satisficing nature of some design tasks and provided a tool to mitigate problems in representing designs for genetic algorithms. When designing an encoding, following the principle of meaningful building blocks ensures that epistatic effects are minimized and a genetic algorithm finds smooth sailing. If this principle cannot be followed (lack of domain knowledge, for example), designer genetic algorithms can help. DGAs may solve the problem, or at least provide additional information about the current encoding of the problem. However, designing a good encoding is still something of an art. Masked crossover reduces one encoding problem but there are others that this thesis does not handle. For example, this thesis only considers fixed-length strings and representations that do not produce non-viable offspring. Goldberg's book indicates examples of such problems and how their encodings are handled (Goldberg, 1989b).

7.3.1 Limitations and Future Work

Encoding a problem for a genetic algorithm is something of an art. Masked crossover is an option that lowers the aesthetic threshold for genetic algorithm programmers but is more easily misled on deceptive problems. The heuristics used for mask propagation are rather simple, and masks are not evaluated independently of their associated genotypes. This thesis has opted for simplicity and domain independence rather than convergence speed and algorithmic complexity. However, tracking and evaluating mask performance and incorporating any information gleaned from this can lead to better mask rules and finer control of search direction. Structure-synthesis problems in two or three dimensions can also be tackled by developing and using two-dimensional and three-dimensional genotypes and recombination operators. With these representations and operators, schemas of short defining "area" or "volume" can be preserved without masks by exchanging contiguous areas or convex volumes during crossover. Investigating two-dimensional and three-dimensional genotypes and operators in structure synthesis tasks is an unexplored area with much potential.

7.4 Analyzing Genetic Algorithm Solutions

The results of chapter 4 illustrated the difficulty of explaining genetic algorithm generated solutions. Solutions, if found, are already difficult to understand in poorly-understood domains, using a blind search algorithm to generate solutions makes understanding such solutions that much more difficult. Typically, the algorithm is started on a problem, runs for a preset number of generations/iterations or until some termination condition is met, and the current best solution or solutions examined. The examination provides a ranking of current solutions, if there are more than one, but little else. Why is this solution good? What happens when this substructure or parameter is changed by a little? Is the response linear or non–linear? Are there any crucial parts that combined to make this solution strong? These questions and others are difficult to answer mainly because the algorithm's trajectory through the search space was not tracked. The sequence of generated solutions (history) provides information on what the algorithm considers important in coming up with the current solution set. When combined with knowledge about a particular algorithm's assumptions about search spaces, the algorithms history can furnish a designer with information needed to answer questions about a solution.

The system described in this chapter provides a tool with which designers can explain solutions discovered by genetic algorithms. A module which uses case–based reasoning techniques, is able to organize and make explicit the building blocks used by a GA. Analysis of the building blocks and the path taken by a GA through a search space provides a powerful mechanism for explaining the solutions generated by a GA. Results of the explanation can be used to guide post-processing in order to improve results in case of premature convergence or no convergence. Injecting genetically engineered individuals created through such analysis also improves performance.

As well as providing a system for explaining GA-generated solutions and search-space analysis, the system provides empirical evidence that the building block hypothesis is correct. The hierarchical clustering of a set of GA individuals (distributed over several generations) according to fitness and genotype leads to nested schemas.

Analysis shows that the subtrees with the most fit schemas receive the most reproductive energy while the least fit schemas are culled from the population. Although the building block hypothesis implies that clustering on fitness and genotype should yield nested schemas, the system was nonetheless tested on several other clustering possibilities, including more generational information and parental information. Clustering on fitness and genotype was by far the most successful technique. The nested schemas that result from such a clustering provided a coherent index for the case–base.

7.4.1 Limitations and Future Work

Chapter 5 made a start at identifying, extracting, organizing, and storing domain knowledge garnered while searching in a poorly-understood domain. During the course of future searches, this knowledge can be used and added to, tuning the algorithm for this particular space. As more and more knowledge gets added, the search can use this information to increase speed. The system as a whole gravitates towards the domain dependent (inflexible) but fast end of the systems spectrum. In other words, this thesis proposes another use for search methods. Genetic or other blind search algorithms can be used as explorers of poorly understood domains, with the side effect of inducing a domain model during the course of search. Initially, in such a system, there would be little knowledge and much searching. With increasing knowledge, in subsequent searches, the population of a genetic algorithm would be seeded with individuals hypothesized to be close to the required solution. When hypothesized solutions *are* the required solutions (say more than 50% of the time), the system can be said to have induced a domain model.

Instrumentation for tracking individuals injected by the hypothesis module, a better user interface, and more support for the case-based explanation module need to be added before induction (learning) of domain models can become automated. The work in this thesis only considers case-based systems. Rule-based systems that induce domain models with genetic search already exist and are called *classifier systems* (Goldberg, 1989b; Holland, 1975).

7.5 Computational Complexity

Last but not least is the problem of computational complexity. In simulated annealing, the annealing schedule defines the number of iterations until termination. In genetic algorithms there was no such upper limit on the time to convergence; in fact, convergence was not well defined. In this chapter, the dissertation advanced a definition of convergence and bounded the time to convergence for genetic algorithms. In addition, using bit complements of population members was identified as a way of maintaining diversity and guarding against deception.

Using the average hamming distance between every pair of individuals in a population (hamming average) as a measure of diversity, a genetic algorithm converges when the change in hamming average is insignificant. Since crossover does not change the hamming average, selection and mutation must be responsible for reducing the hamming average. An upper bound on the time to convergence of a genetic algorithm was therefore provided by a genetic algorithm's performance on a *flat* function — that is, a function whose fitness landscape is absolutely flat and all individuals in the GA's population have identical fitness. Mutation rate being fixed, any other function must provide greater selection pressure and thus must converge in a time less than the time taken on a flat function with an identical genotype length. Genetic algorithms converge on the flat function due to *genetic drift*. This chapter derived an upper bound on the time complexity from considering the effects of drift on allele frequency. Then, from a model of the effect of selection and mutation on the behavior of genetic algorithms, the analysis was able to approximate two solution characteristics at convergence for static search spaces.

1. Bounds on an upper limit to the average hamming distance of a population at convergence. In addition to providing time limits, the average hamming distance of the population also denotes the remaining amount of work.

2. The amount of work possible by the GA, indicated by the average fitness at convergence.

Combining fitness prediction with hamming average prediction indicates how much progress is possible with a GA along with bounds on how much remaining work can be done. The latter is especially important when including mutation. The predicted average fitness indicates how much progress is possible, but it is the predicted hamming average that denotes the remaining work.

In satisficing problems the maximum fitness is known and the structure that attains this fitness needs to be generated. When the maximum fitness is known, the predictability of average fitness at convergence indicates whether the genetic algorithm will succeed, and if so, how long it will take.

7.5.1 Limitations and Future Work

The upper limit on time to convergence is an upper limit for all functions of a particular genotypic length. The analysis in this thesis that approximates convergence time by tracking first-order schema proportions assumes low epistatic encodings, and will fail on highly epistatic ones. The other point of view is that epistatic encodings can be detected through this analysis (useful information for a programmer). Finally, since the rate of convergence depends greatly on the selection pressure, it may be possible to tell whether a function is linear, polynomial, or exponential from observing the rate of decrease of hamming average. Finding out broad properties of a space by instrumenting a genetic algorithm can be a very useful area for research.

7.6 Conclusions

Design is ubiquitous. Design tasks range from spacecraft design to planning the day. This thesis helps establish genetic algorithms as a computational tool for design. Those problems that can be cast as search problems in poorly-understood domains belong to the set attacked in this thesis. In poorly-understood domains, genetic algorithms can find solutions to problems in structure synthesis, structure configuration and parameter instantiation. The thesis introduced genetic algorithms, compared and contrasted them with other search algorithms, and explained how to encode problems for GAs. The biases inherent in genetic search were made explicit and designer genetic algorithms were defined to help surmount unfavorable biases. Solutions generated in poorly-understood domains are hard to explain. Using ideas from case-based reasoning, the system described in chapter 5 tracked, organized, and stored information during the course of genetic search. This information was used to explain genetic algorithm solutions and help in sensitivity analysis. Finally, the time to convergence of a genetic algorithm was bounded.

The thesis settles why, when and how to use genetic algorithms for design, provides tools to explain and analyze genetic algorithm designs, and bounds the time complexity of the task, helping establish genetic algorithms as a computational tool for design.

7.7 Further Directions

The work reported here was concerned with laying a firm foundation for exploiting the potential of genetic algorithms in design tasks. However, designer–client interactions usually consist of a cycle of design and modification to design, converging on an acceptable and realizable design. During this cycle the specifications of the design may change, or the tools available to realize the design may change. Modeling these dynamic processes is an open field of research. For example, designing a user interface that over time adapts to a user or group of users is an application area which, with the growth of interest in user interface design, is both interesting and perhaps profitable.

With increasingly powerful computers becoming available, applications in the biological sciences are growing. Genetic algorithms have been used to model aspects of the human immune system (Smith et al., 1993) and for predicting the three-dimensional structure of proteins (Le Grand and Merz Jr., 1993). The problem of molecular design using computer simulations to provide a measure of the behavior of designed molecules is still open.

Studying complex adaptive systems seeks to understand the underlying commonalities in economies, ecologies, immune systems, and political systems. This field is of particular significance because of the need to predict the effect of changes in our interconnected and interdependent world. Increasing computing power and the availability of accessible electronic data have made the task of simulating and validating complex systems more feasible. Mathematical modeling has often proved inadequate because of the highly non-linear nature of these systems, and direct physical experiments are usually infeasible. Genetic-algorithm-based adaptive systems appear well suited for modeling and controlling complex systems whose aggregate behavior stems from the interaction of constituent parts. For example, one interesting problem is to model the effects of government policy on the state of a resource. The policy's intended effect and its actual implementation after percolating through several layers of society are difficult to reconcile. Modeling such a complex system is a first step toward effective policy making. A genetic-algorithm-based classifier system using feedback on the state of the resource may be able to learn effective strategies to manage the resource optimally.

The number of possible applications is growing. With continued growth in the speed and size of computer systems, larger and more complex problems will be amenable to genetic search. The work reported in this dissertation will hopefully help lay the foundations for the growth of genetic algorithms in design.

References

Ackley, D. A. (1987). *A Connectionist Machine for Genetic Hillclimbing*. Kluwer Academic Publishers.

Bäck, T., Hoffmeister, F., and Schwefel, H. (1991). A survey of evolution strategies. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 1–10. Morgan Kauffman, San Mateo, CA.

Bareiss, R., editor (1991). *Proceedings of the Case-Based Reasoning Workshop*. Morgan Kauffman, Inc.

Caldwell, C. and Johnston, V. S. (1991). Tracking a criminal suspect through "Face-Space" with a genetic algorithm. In *Proceedings of Fourth International Conference on Genetic Algorithms*, pages 416–421. Morgan Kauffman.

Coyne, R. D., Rosenman, M. A., Radford, A. D., Balachandran, M., and Gero, J. S. (1990). *Knowledge-Based Design Systems*. Addison-Wesley, New York.

Dasgupta, S. (1991). *Design Theory and Computer Science*. Cambridge University Press, Cambridge, New York.

Dawkins, R. (1986). *The Blind Watchmaker*. Longman.

Deb, K. and Goldberg, D. E. (1989). An investigation of niche and species formation in genetic function optimization. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 42–50. Morgan Kauffman, San Mateo, CA.

DeJong, K. A. (1975). *An Analysis of the Behavior of a class of Genetic Adaptive Systems.* PhD thesis, University of Michigan, Ann Arbour. Department of Computer and Communication Sciences.

Eshelman, L. J. (1991). The chc adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. In Rawlins, G. J. E., editor, *Foundations of Genetic Algorithms-1*, pages 265–283. Morgan Kauffman.

Gale, J. S. (1990). *Theoretical Population Genetics.* Unwin Hyman, London.

Gero, J. S., editor (1992). *Second Round Table Conference on Computational Models of Creative Design.*

Gero, J. S. and Jo, J. H. (1991). Design mutation as a computational process. Working Paper, Design Computing Unit, Department of Architectural and Design Science, University of Sydney, NSW 2006, Australia.

Gero, J. S., Louis, S. J., and Kundu, S. (1993). Evolutionary learning of novel grammars for design improvement. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, to appear.

Goel, A. and Chandresekaran, B. (1992). Case-based design: A task analysis. In Tong, C. and Sriram, D., editors, *Artificial Intelligence in Engineering Design, Vol II*, pages 165–184. Academic Press, Inc.

Goldberg, D. E. (1989a). Genetic algorithms and walsh functions: Part 2, deception and its analysis. *Complex Systems*, 3:153–171.

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

Goldberg, D. E., Korb, B., and Deb, K. (1989). Messy genetic algorithms: Motivation, analysis, and first results. Technical Report TCGA Report No. 89002, The Clearinghouse for Genetic Algorithms, University of Alabama, Tuscaloosa.

Goldberg, D. E. and Lingle, R. (1985). Alleles, loci and the traveling salesman problem. In *Proceedings of an International Conference on Genetic Algorithms*, pages 10–19. Morgan Kauffman.

Goldberg, D. E. and Segrest, P. (1987). Finite markov chain analysis of genetic algorithms. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 1–8. Lawrence Erlbaum Associates.

Holland, J. (1975). *Adaptation In Natural and Artificial Systems.* The University of Michigan Press, Ann Arbour.

Huhns, M. and Acosta, R. (1992). Argo: An analogical reasoning system for solving design problems. In Tong, C. and Sriram, D., editors, *Artificial Intelligence in Engineering Design, Vol II*, pages 105–144. Academic Press, Inc.

Kirkpatrick, S., Gelatt, C. D., and Vechi, M. P. (1983). Optimization by simulated annealing. *Science*, 220(4598):671–680.

Koza, J. R. (1993). *Genetic Programming.* MIT Press.

Le Grand, S. M. and Merz Jr., K. M. (1993). The application of the genetic algorithm to the minimization of potential energy functions. *Journal of Global Optimization*, 3:49–66.

Lenat, D. B. (1983). The role of heuristics in learning by discovery: Three case studies. In Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., editors, *Machine Learning, An Artificial Intelligence Approach*, pages 243–306. Morgan Kauffman.

Louis, S. J., McGraw, G., and Wyckoff, R. (1993). Case-based reasoning assisted explanation of genetic algorithm results. *Journal of Experimental and Theoretical Artificial Intelligence*, 5:21–37.

Louis, S. J. and Rawlins, G. J. E. (1991). Designer genetic algorithms: Genetic algorithms in structure design. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 53–60. Morgan Kauffman, San Mateo, CA.

Louis, S. J. and Rawlins, G. J. E. (1993a). Pareto optimality, ga-easiness and deception. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 118–123. Morgan Kauffman, San Mateo, CA.

Louis, S. J. and Rawlins, G. J. E. (1993b). Syntactic analysis of convergence in genetic algorithms. In Whitley, L. D., editor, *Foundations of Genetic Algorithms - 2*, pages 141–152. Morgan Kauffman, San Mateo, CA.

Mahfoud, S. W. and Goldberg, D. E. (1992). Parallel recombinative simulated annealing: A genetic algorithm. Technical Report IlliGAL Report No. 92002, The Illinois Genetic Algorithm Laboratory, Department of General Engineering, University of Illinois at Urbana-Champaign.

McGraw, G. (1992). Letter spirit: Recognition and creation of letterforms based on fluid concepts. Technical Report 61, Center for Research on Concepts and Cognition, Indiana University.

Mitchell, M. and Forrest, S. (1993). Relative building-block fitness and the building-block hypothesis. In Whitley, L. D., editor, *Foundations of Genetic Algorithms - 2*, pages 109–126. Morgan Kauffman, San Mateo, CA.

Mostow, J., Barley, M., and Weinrich, T. (1992). Automated reuse of design plans in bogart. In Tong, C. and Sriram, D., editors, *Artificial Intelligence in Engineering Design, Vol II*, pages 57–104. Academic Press, Inc.

Powell, D. J., Tong, S. S., and Skolnik, M. M. (1989). Engeneous domain independent machine learning for design optimization. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 151–159. Morgan Kauffman.

Radford, A. D. and Gero, J. S. (1988). *Design by Optimization in Architecture, Building, and Construction.* Van Nostrand Reinhold Company, New York.

Rawlins, G. J. E. (1991). Introduction. In Rawlins, G. J. E., editor, *Foundations of Genetic Algorithms-1*, pages 1–12. Morgan Kauffman.

Riesbeck, C. K. and Schank, R. C. (1989). *Inside Case-Based Reasoning.* Lawrence Erlbaum Associates, Cambridge, MA.

Schaeffer, D. J., Eshelman, L. J., and Offut, D. (1991). Spurious correlations and premature convergence in genetic algorithms. In Rawlins, G. J. E., editor, *Foundations of Genetic Algorithms-1*, pages 102–114. Morgan Kauffman.

Schaeffer, D. J. and Morishima, A. (1987). An adaptive crossover distribution mechanism for genetic algorithms. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 36–40. Lawrence Erlbaum Associates.

Schaeffer, D. J. and Morishima, A. (1988). Adaptive knowledge representation: A content sensitive recombination mechanism for genetic algorithms. *International Journal of Intelligent Systems*, 3:229–246.

Simon, H. (1969). *Sciences of the Artificial.* MIT Press, Cambridge, MA.

Smith, D. (1985). Bin packing with adaptive search. In *Proceedings of an International Conference on Genetic Algorithms*, pages 202–206. Morgan Kauffman.

Smith, R. E., Forrest, S., and Perelson, A. S. (1993). Population diversity in immune system models: Implications for genetic search. In Whitley, L. D., editor, *Foundations of Genetic Algorithms - 2*, pages 153–166. Morgan Kauffman, San Mateo, CA.

Spears, W. M. and DeJong, K. A. (1991). An analysis of multi-point crossover. In Rawlins, G. J. E., editor, *Foundations of Genetic Algorithms-1*, pages 301–315. Morgan Kauffman.

Sycara, K. and Navinchandra, D. (1992). Retrieval strategies in case-based design system. In Tong, C. and Sriram, D., editors, *Artificial Intelligence in Engineering Design, Vol II*, pages 145–164. Academic Press, Inc.

Syswerda, G. (1989). Uniform crossover in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2–8. Morgan Kauffman.

Thierens, D. and Goldberg, D. E. (1993). Mixing in genetic algorithms. In *Proceedings of Fifth International Conference on Genetic Algorithms*, pages 38–45. Morgan Kauffman.

Tong, C. (1992). Using exploratory design to cope with problem complexity. In Tong, C. and Sriram, D., editors, *Artificial Intelligence in Engineering Design, Vol II*, pages 287–332. Academic Press, Inc.

Tong, C. and Sriram, D., editors (1992a). *Artificial Intelligence in Engineering Design.* Academic Press, Inc.

Tong, C. and Sriram, D. (1992b). Introduction. In Tong, C. and Sriram, D., editors, *Artificial Intelligence in Engineering Design*, pages 1–53. Academic Press, Inc.

Tong, S. S., Powell, D., and Cornett, D. (1992). Engineous: A unified method for design automation, optimization, and integration. In Tong, C. and Sriram, D., editors, *Artificial Intelligence in Engineering Design, Vol III*, pages 235–254. Academic Press, Inc.

Ulrich, K. and Seering, W. (1992). Function sharing in mechanical design. In Tong, C. and Sriram, D., editors, *Artificial Intelligence in Engineering Design, Vol II*, pages 185–214. Academic Press, Inc.

Wilson, S. W. (1991). Ga-easy does not imply steepest-ascent optimizable. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 85–89. Morgan Kauffman, San Mateo, CA.

A

Complete Mask Rules

1. MF_{gg}:

   Case: Both children are good.

   Summary: Very encouraging behavior and as such is reflected in the mask settings below and in figure A.1. The parents' masks are OR'd to produce the children's masks, ensuring preservation of the contributions from both parents.

   Action:

   – $CM1$: OR the masks of $PM1$ and $PM2$. If there are any 0's left in $CM1$, toss a coin to decide their value.

   – $CM2$: Same as for $CM1$.

   – $PM1$: No changes except for those produced by mask mutation.

   – $PM2$: Same as for $PM1$.

2. MF_{bb}:

   Case: Both children are bad.

   Summary: Discouraging behavior and must be guarded against in future. Each parent contributed bits that were detrimental to the children's fitness.

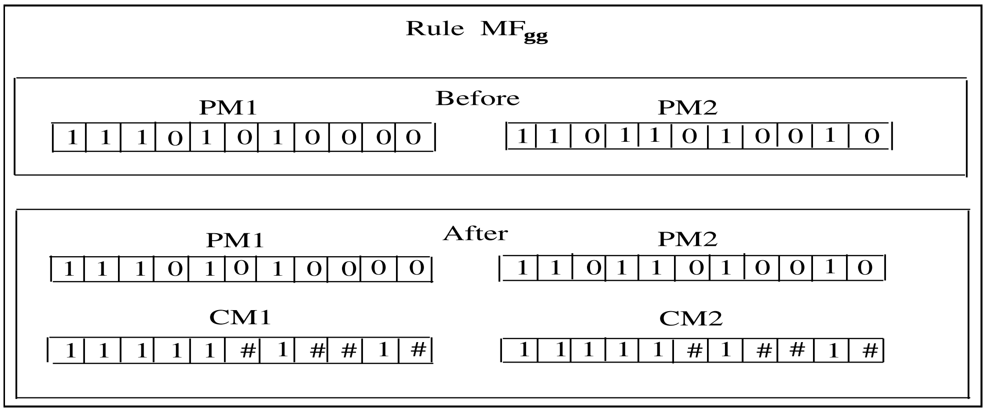

Figure A.1: Mask rule MF_{gg}: Example of mask propagation when both C1 and C2 are good

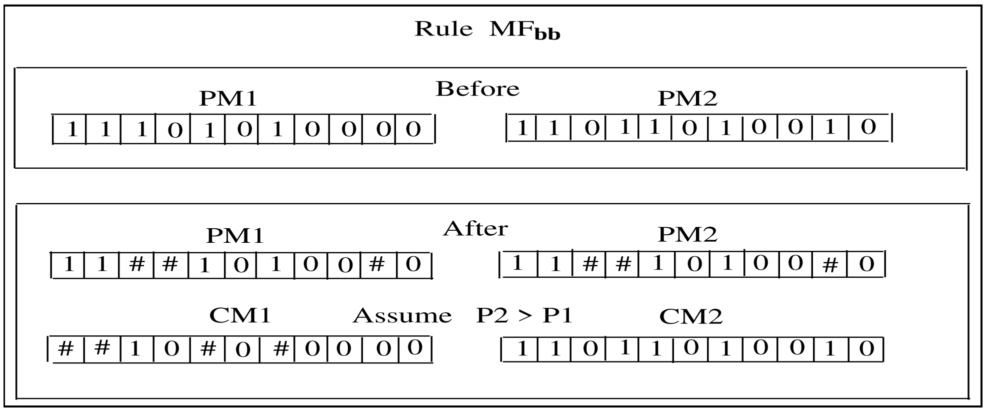

Figure A.2: Mask rule MF_{bb}: Example of mask propagation when both C1 and C2 are bad.

MX has not set up the parents' masks correctly. Changes are given below and shown in figure A.2.

Action:

- $CM1$: This mask should reflect the undesirability of the current search direction. Contributions from $P2$ were detrimental, therefore $CM1$ should search in the area of $P2$'s contribution which is specified by the loci where $PM1_i$ is 0 and $PM2_i$ is 1. Set these loci in $CM1_i$ to 0. If $P2 > P1$ in fitness, toss a coin to set the bits of $CM1$ at those locations where both $PM1_i$ and $PM2_i$ are 1.

- $CM2$: A similar rule applies to $CM2$.
- $PM1$: $P2$'s contribution to $C1$ led to a bad child. The $C1$ positions copied from $P2$ need to be explored in $P1$. These loci are those for which $PM1_i$ was 0 and $PM2_i$ was 1. Therefore set these loci in $PM1_i$ by tossing a coin. In addition, $PM1$ specified loci that were detrimental to $C2$. Therefore when $PM1_i$ is 1 and $PM2_i$ is 0, set these locations by tossing a coin.
- $PM2$: A similar rule applies for $PM2$.

3. MF_{ab}:

Case: One child is bad while the other is average. Assume that $C1$ is bad and $C2$ is average.

Summary: There are two sub-cases. If $C2$'s fitness is less than that of $P2$, it means that $P1$'s contribution was deleterious, and that $PM2$ needs to be augmented with 1's at those positions that were copied from $P1$ to $C2$ (see figure A.3). If $C2$'s fitness is less than that of $P1$, $P1$'s contribution increased $C2$'s fitness, so preserve $P1$'s contribution in $C2$. The masks are shown in figure A.4.

Action:

- When $\mathbf{C1 < P1 < C2 < P2}$:
  * $CM1$: Very similar to MF_{bb}'s $CM1$. This mask should reflect the undesirability of the current search direction. Contributions from $P2$ were detrimental, therefore $CM1$ should search in the area of $P2$'s contribution which is specified by the loci where $PM1_i$ is 0 and $PM2_i$ is 1. Set these loci in $CM1_i$ to 0. Since $P2 > P1$ in

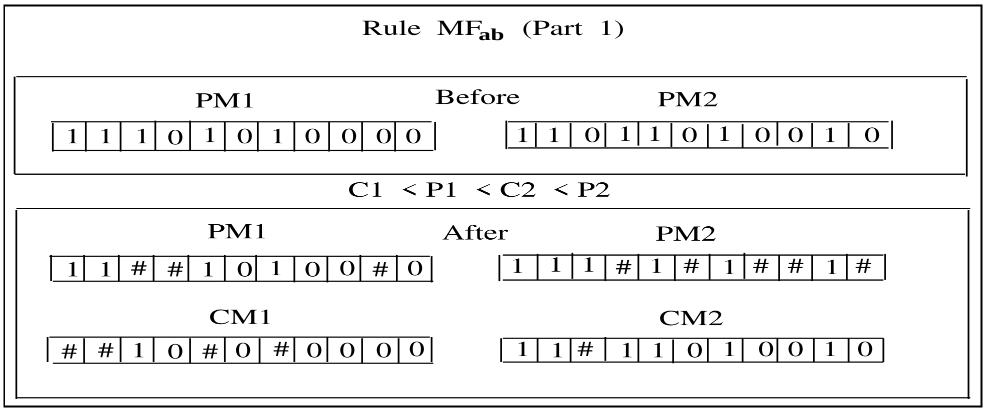

Figure A.3: Mask rule MF_{ab}, Part 1: Example of mask propagation when C1 is bad and C2 is average. $(C1 < P1 < C2 < P2)$

fitness, toss a coin to set the bits of $CM1$ at those locations where both $PM1_i$ and $PM2_i$ are 1.

* $CM2$: As $P1$'s contribution decreased $C2$'s fitness, set $CM2_i$ by tossing a coin when $PM1_i$ is 1 and $PM2_i$ is 0, searching around $P1$'s contribution.

* $PM1$: $P2$'s contribution to $C1$ led to a bad child. The $C1$ positions copied from $P2$ need to be explored in $P1$. These loci are those for which $PM1_i$ is 0 and $PM2_i$ is 1. Therefore set these loci in $PM1_i$ by tossing a coin. In addition, $PM1$ specified loci that were detrimental to $C2$. Therefore when $PM1_i$ is 1 and $PM2_i$ is 0, again set these locations by tossing a coin.

* $PM2$: Since $P1$'s contribution led to a decrease in fitness, set $PM2_i$ to 1 for those loci for which $PM1_i$ is 1 and $PM2_i$ is 0. To help fix positions currently 0, toss a coin to fix those loci in $PM2_i$ for which both $PM1_i$ and $PM2_i$ are 0.

– When **C1 < P2 < C2 < P1:**

* $CM1$: Same as the action for $CM1$ when $C1 < P1 < C2 < P2$.

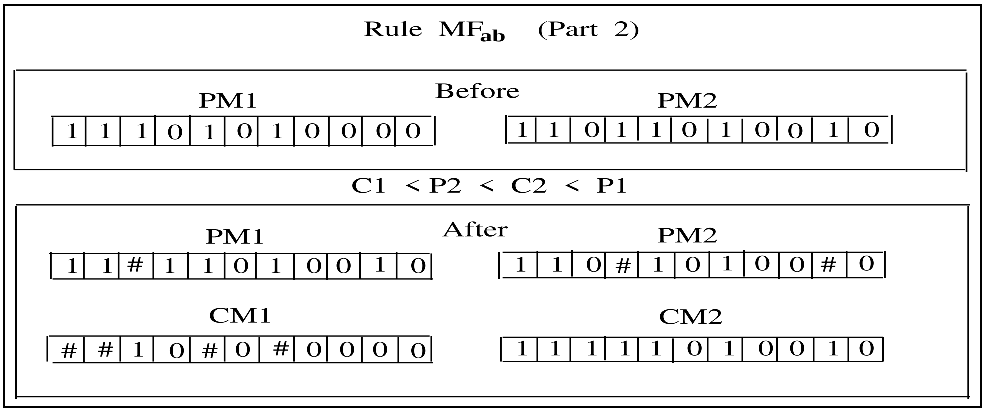

Figure A.4: Mask rule MF_{ab}, Part 2: Example of mask propagation when C1 is bad and C2 is average. $(C1 < P2 < C2 < P1)$

Except that as $P1 > P2$, no action need be taken when both parent masks have a 1 at some position.

- $CM2$: To preserve $P1$'s contribution, when $PM1_i$ is 1 and $PM2_i$ is 0 set $CM2_i$ to 1.
- $PM1$: Those bits that contributed to $C1$ but not those that helped $C2$ need to be modified. Therefore for those loci where $PM1_i$ is 1 and $PM2_i$ is 0, toss a coin to decide $PM1_i$. $P2$'s contribution was detrimental so when $PM2_i$ is 1 and $PM1_i$ is 0 set $PM1_i$ to 1.
- $PM2$: Set $PM2_i$ by tossing a coin for those loci that contributed to $C1$.

4. MF_{aa}:

- **Case:** Both children are average.
- **Summary:** Once again there are two sub-cases depending on whether a child is better or worse than its dominating parent. Assume that $C1$ is better than $P1$ implying that $C2$ is worse than $P2$. The complementary

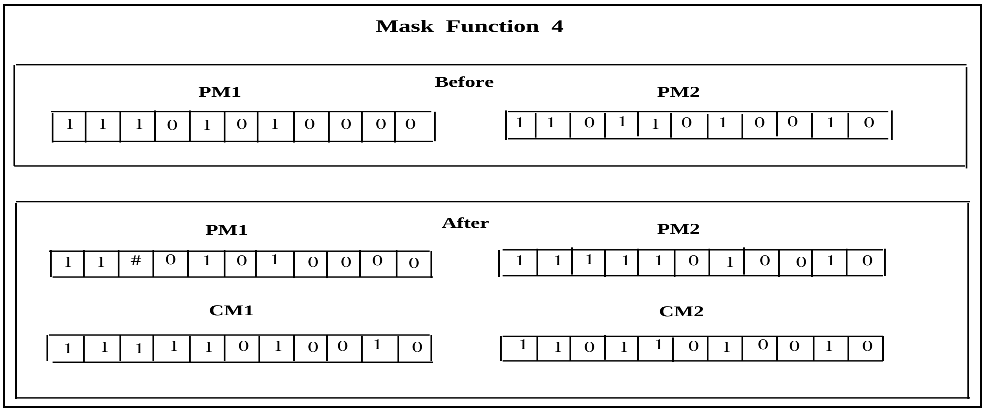

Figure A.5: Mask rule MF_{aa}, Example of mask propagation when both C1 and C2 are average.

case when $C1$ is worse than $P1$ and $C2$ is better than $P2$ is treated the same way by substituting $C1$ for $C2$ and $P1$ for $P2$. The masks are given in figure A.5.

- **Action:**
  - $CM1$: This mask should reflect the desirability of the current search direction since $P2$ contributions lead to $C1$'s increased fitness. OR the masks of $PM1$ and $PM2$, if there are any zeero's left in $CM1$ toss a coin to decide their value.
  - $CM2$: Contributions from $P1$ were detrimental. Set the loci that specify $P1$ contribution to $C2$ to 0.
  - $PM1$: $P1$'s contributions to $C2$ were detrimental, set these loci by a coin toss.
  - $PM2$: The positions that $P1$ contributed to $C2$ decreased $C2$'s fitness, set these loci to 1.

5. MF_{ga}:

- **Case:** One child is good and the other average.

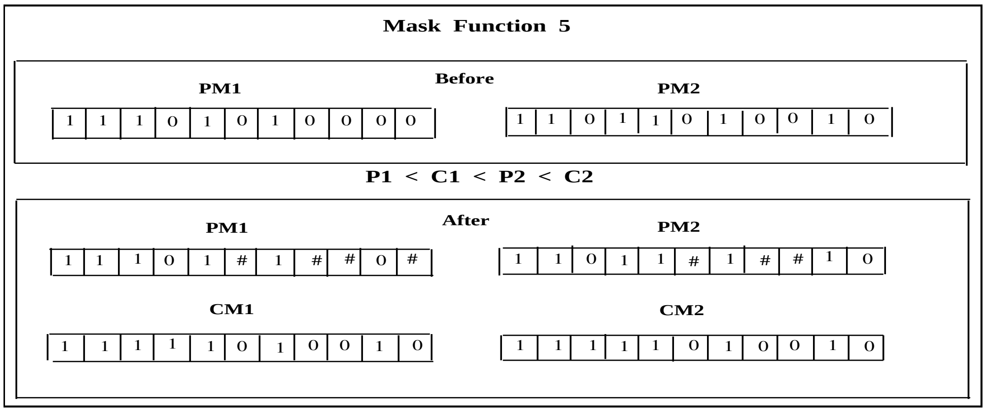

Figure A.6: Mask rule MF_{ga}, Part 1: Example of Mask propagation when C1 is average and C2 is good. ($P1 < C1 < P2 < C2$)

- **Summary:** There are two sub-cases depending on the relative ordering of the parents and children. If $P1 < C1 < P2 < C2$ is behavior to be encouraged. The other case stresses the importance of $P1$.
- **Action:**
  - When $P1 < C1 < P2 < C2$: This is also encouraging behavior and is depicted in figure A.6.
    * $CM1$: Same as for MF_{gg}'s $CM1$.
    * $CM2$: Same as for $CM1$ above.
    * $PM1$: Since both children are better than their prospective parents flip a coin to decide positions that are 0 in both $PM1$ and $PM2$
    * $PM2$: Same as for $PM1$ above .
  - **P2 $<$ C1 $<$ P1 $<$ C2** Clearly $P1$'s contribution is vital to reproduction. The mask settings describe this below and in figure A.7.

    * $CM1$: $P2$'s contribution decreased $C1$'s fitness. To encourage search in this area, when $PM2_i$ is 1 and $PM1_i$ is 0 set $CM1_i$ to

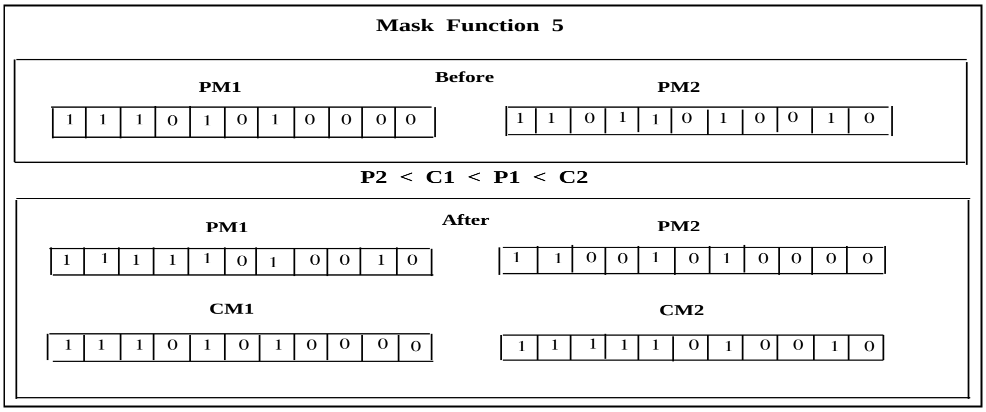

Figure A.7: Mask rule MF_{ga}, Part 2: Example of Mask propagation when C1 is average and C2 is good. ($P2 < C1 < P1 < C2$)

0.

- $CM2$: $P1$'s contribution clearly contributed positively to $C2$'s fitness. Set $CM2_i$ to 1 when $PM1_i$ is 1 and $PM2_i$ is 0.
- $PM1$: Since $P2$'s contribution decreased the fitness of $C1$, $PM1_i$ should be 1 at those loci where $PM2_i$ is 1 and $PM1_i$ is currently 0.
- $PM2$: Since $P2$'s contribution to $C1$ was negative, set $PM2_i$ to 0 at those loci where $PM1_i$ is 0 and $PM2_i$ is currently 1.

6. MF_{gb}:

7. **Case:** One child is good and the other bad. Assume $C1$ is bad and $C2$ is good.

8. **Summary:** This should try to preserve $C2$'s schemas, while the relative fitnesses of $P1$ and $P2$ drive the settings of $PM1$ and $PM2$.

9. **Action:**

- $CM1$: Same as MF_{bb}'s CM1.

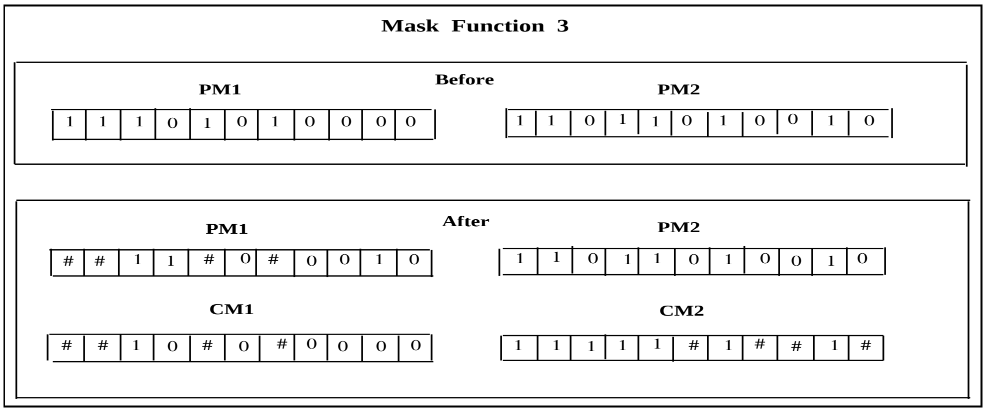

Figure A.8: Mask rule MF_{gb}: Example of Mask propagation when C1 is bad and C2 is good

- $CM2$: When $PM1_i$ is 1 and $PM2_i$ is 0, set $CM2_i$ to 1. In addition, whenever both parent masks are 0 use a random coin toss to set the corresponding position in $CM2_i$.
- $PM1$: $P2$'s contribution decreased $C1$'s fitness. Set $PM1_i$ to 1 when $PM1_i$ is 0 and $PM2_i$ is 1. If $P1 < P2$ in fitness, to further improve $PM1$, flip a coin to set positions where both $PM1_i$ and $PM2_i$ are 1.
- $PM2$: No changes except for mask mutation.

B

Data for the 5-Bit Parity Checker

B.1 Cluster Tree

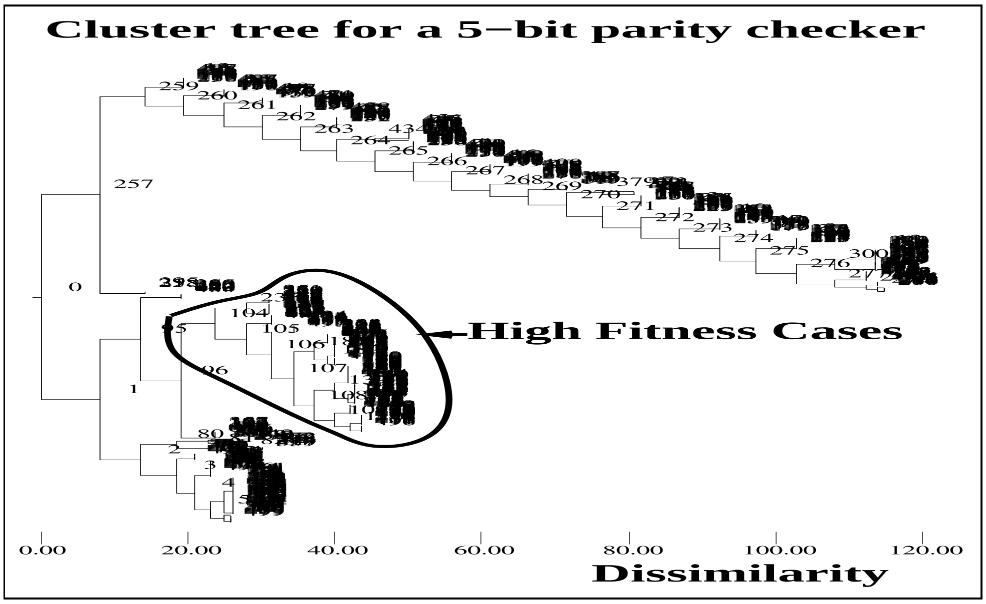

Figure B.1: Tree structure of the cases for a 5-bit parity checker

The labels represent the case number of the cases in the tree. Because of the number of individuals that are very similar in fitness and genotype, the labels at the leaves overlap causing the dark areas. The utility "xgraph" however allows zooming in on areas, expanding the level of detail that can be made visible to a user and clearing the image.

B.2 Report for 5-bit Parity Checker

This section contains part of the report generated after creating cases for the 5 bit parity checker. The report is too large and cumbersome to include in its entirety. Vertical dots (⋮) indicate material that was not included.

```
Chez Scheme Transcript [Fri Feb  7 20:23:12 1992]

¿¿ (process-cases 'circ2 100)
Loading casevec "/usr2/gem/circuit/circ2-c2c.cvec"
length 499
(
.
.
.
————-Average age of schema - distance less than 5:
NO   DST  Schema                                                                             ORD WT  LON lw  av  hi Fitness
259  2    ******************************************************************************** 0   241 14  0   5   14  16.09
260  3    ******************************************************************************** 0   229 14  0   5   14  16.1
257  1    ******************************************************************************** 0   243 15  0   5   15  16.12
261  4    ******************************************************************************** 0   219 14  0   5   14  16.1
262  5    ******************************************************************************** 0   211 14  0   5   14  16.1
0    0    ******************************************************************************** 0   500 24  0   12  24  19.94
236  5    ****00110100*0000100000110*101111111000110011110100*00000*0001100111010111101110 92  17  11  6   12  17  20.0
81   4    0***0011***0*0*0*010001110*1111001111101*00001000010*1*111**1001101*0100*11*1*00 78  10  4   12  14  16  19.4
80   3    0****011***0*0*0*010001110*1*11001111*01*0000*000010*1*111***0*1*01****0*11*1**0 65  16  4   12  14  16  19.62
90   4    00001011101000001010001110010*1100111100100000000000101101110000110011101011101110 98  6   3   13  14  16  20.0
82   5    0*11001101000010001000111001111001111101*0000100001011*1111*1001101*0100011*1*00 91  7   3   13  14  16  20.0
104  4    ****0011***0*00001*000001*0*10*111*11000*10*1*1*0100*00000****1*00*1*0101*1*1*** 65  149 19  5   17  24  23.54
96   3    *****011***0*000***000**1********1*11*00*10*1*1*0100*00000****1*00*1*0101*1*1*** 54  157 19  5   17  24  23.57
95   2    *****011***0*000***000**1********1*11*0**10*1*1*0100*00000****1*00*1*0101*1*1*** 53  163 19  5   17  24  23.73
1    1    *****011***0*0*0***000**1********1*11*****0*****0**0************0*****0**1*1***  31  257 19  5   17  24  23.56
5    5    000010111010000010100011100111100111110100001*000101101*100100110110100*1101*00  95  51  16  8   17  24  24.0
2    2    0****011***0*0*0*010001110*1*110*1111**1*0000**00010*1*1*1***0*1*01****0*11*1**0 58  94  16  8   17  24  23.26
105  5    ****0011***0*00001*000001*0*10*111*11000*10*1*1*0100*00000****1*00*1*0101*1*1*** 65  132 19  5   18  24  24.0
4    4    00001011101000001010001110011110*11111*1000001*000101101*100100110110100*1101*00 91  70  16  8   18  24  24.0
3    3    00001011101000001010001110011110*11111*1000001*000101101*1*0100110110100*1101*00 87  78  16  8   18  24  24.0
```

———-Cases within depth 5:

NO	DST	Schema	ORD	WT	Fitness
0	0	**	0	500	19.94
1	1	*****011***0*0*0***000**1********1*11*****0*****0**0*************0*****0**1*1***	31	257	23.56
257	1	**	0	243	16.12
2	2	0****011***0*0*0*010001110*1*110*1111**1*0000**00010*1*1*1***0*1*01****0*11*1**0	58	94	23.26
95	2	*****011***0*000***000**1********1*11*0**10*1*1*0100*00000****1*00*1*0101*1*1***	53	163	23.73
258	2	11000001010111110101010111111000001111010000010000101101110010011011110001101000	()	2	20.0
259	2	**	0	241	16.09
3	3	00001011101000001010001110011110*11111*1000001*000101101*1*0100110110100*1101*00	87	78	24.0
80	3	0****011***0*0*0*010001110*1*11001111*01*0000*000010*1*111***0*1*01****0*11*1**0	65	16	19.62
96	3	*****011***0*000***000**1********1*11*00*10*1*1*0100*00000****1*00*1*0101*1*1***	54	157	23.57
252	3	11000011010000000010000001101101101111001110011110100100000000001000011010101110 10	()	6	28.0

.
.
.

———-Additive schemata - depth less than 5 range, (0.6 0.4):

NO	DST	Add Schema	ORD	WT	LON	lw	av	hi	Fitness
259	2	***000**1**1******0********111*****10**0****0**1*010******000****01**1*0**11**0*	0	241	14	0	5	14	16.09
260	3	***000**1***0*1***0****1***11******101*0****0**1***0**0*1*00******10*1*0**11**0*	0	229	14	0	5	14	16.1
257	1	***000**1**1******0********11****0*101*0****0**1*010******00*****01**1*0**11**0*	0	243	15	0	5	15	16.12
261	4	***000**1**10*1**10*****0**1******1101*0****0**1*0*0**0*1*000****010*1*0**11**0*	0	219	14	0	5	14	16.1
262	5	***000**1**1011**10*****0**11*****11*1*0*******1***0*10*1*00*****010*1*0**11**0*	0	211	14	0	5	14	16.1
0	0	00**00111**00000***000**1*011*1**1111*00**00*1*10**01*0****0*011*011**10111*1**0	0	500	24	0	12	24	19.94
236	5	00110011010010000100000011001011111111000110011110100100000000001100111010111011 10	92	17	11	6	12	17	20.0
81	4	00110011010000100010001110011110011111010000010000101101111010011011010001101000	78	10	4	12	14	16	19.4
80	3	00**0011***000*0*01000111001111001111101000001000010110111*0100110110100*1101*00	65	16	4	12	14	16	19.62
90	4	00001011101000001010001110011110011110010000000000101101110000110011101011101110	98	6	3	13	14	16	20.0

.
.
.

———-Additive schemata - depth less than 5 range, (0.8 0.2):

NO	DST	Add Schema	ORD	WT	LON	lw	av	hi	Fitness
259	2	****0***1***	0	241	14	0	5	14	16.09
260	3	****0***1***	0	229	14	0	5	14	16.1
257	1	****0***1***	0	243	15	0	5	15	16.12
261	4	****0***1**************************1**	0	219	14	0	5	14	16.1
262	5	****0******************************1**	0	211	14	0	5	14	16.1
0	0	*****0*****************************1***************0*******0*****0*****0**1*****	0	500	24	0	12	24	19.94
236	5	****00110100*00001000000110*1011111111000110011110100*00000*000110011101011101110	92	17	11	6	12	17	20.0
81	4	0***0011***0*0*0*010001110*1111001111101*00001000010*1*111**1001101*0100*11*1*00	78	10	4	12	14	16	19.4
80	3	00***011***000*0*01000111001111001111*0100000*000010110111*0*0*1*011***0*1101**0	65	16	4	12	14	16	19.62
90	4	0000101110100000101000111001*1100111100100000000000101101110000110011101011101110	98	6	3	13	14	16	20.0

.
.
.

()

¿¿ (transcript-off

)

B.2.1 The Schema extracted from the Report

The schema selected by the methodolgy in chapter 5 is given below. This schema, case number 95 has weight 163, one of the longest lives, 19, and is in the region where the genetic algorithm's solutions have high fitness (the bounded area in figure B.1). The schema's fitness is 23.75, which can be compared to fitness of the best solution found so far at 28.0.

```
*****011***0*000***000**1********1*11*0**10*1*1*0100*00000****1*00*1*0101*1*1***
```

B.3 Obtaining Clustering Code

The cluster code can be obtained through anonymous ftp from the University of California, Berkeley. Follow the instructions below to obtain the program.

```
cluster is available via anonymous ftp from icsi-ftp.berkeley.edu
(128.32.201.55).  To get it use FTP as follows:

% ftp icsi-ftp.berkeley.edu
Connected to icsic.Berkeley.EDU.
220 icsi-ftp (icsic) FTP server (Version 5.60 local) ready.
Name (icsic.Berkeley.EDU:stolcke): anonymous
Password (icsic.Berkeley.EDU:anonymous):
331 Guest login ok, send ident as password.
```

```
230 Guest login Ok, access restrictions apply.
ftp> cd pub/ai
250 CWD command successful.
ftp> binary
200 Type set to I.
ftp> get cluster-2.2.tar.Z
200 PORT command successful.
150 Opening BINARY mode data connection for cluster-2.2.tar.Z (15531 bytes).
226 Transfer complete.
15531 bytes received in 0.08 seconds (1.9e+02 Kbytes/s)
ftp> quit
```

Curriculum Vitae

Sushil J. Louis was born on March 23, 1964 in Tiruchirapalli, India. He attended the University of Delhi where he obtained a Bachelor of Science (Honors) degree in Physics in 1983. Turning to computer science, he obtained a Master of Computer Applications degree from the same university in 1986. He obtained admission to Indiana University's graduate program in Computer Science in the fall of 1986, and came to Bloomington. During the course of his graduate studies in computer science he was financially supported by the Indiana University Computer Science Department as an Associate Instructor. He completed his Doctor of Philosophy degree in Computer Science in 1993.