Fernando Miguel Pais da Graça Lobo

THE PARAMETER-LESS GENETIC ALGORITHM: RATIONAL AND AUTOMATED PARAMETER SELECTION FOR SIMPLIFIED GENETIC ALGORITHM OPERATION

Dissertação apresentada para obtenção do Grau de Doutor em Engenharia do Ambiente,
na especialidade Sistemas Naturais e suas Tensões, pela Universidade Nova de Lisboa,
Faculdade de Ciências e Tecnologia.

Lisboa

2000

ABSTRACT

Genetic algorithms (GAs) have been used to solve difficult optimization problems in a number of fields. One of the advantages of these algorithms is that they operate well even in domains where little is known, thus giving the GA the flavor of a general purpose problem solver. However, in order to solve a problem with the GA, the user usually has to specify a number of parameters that have little to do with the user's problem, and have more to do with the way the GA operates. This dissertation presents a technique that greatly simplifies the GA operation by relieving the user from having to set these parameters. Instead, the parameters are set automatically by the algorithm itself. The validity of the approach is illustrated with artificial problems often used to test GA techniques, and also with a simplified version of a network expansion problem.

RESUMO

Os algoritmos genéticos (AGs) têm sido utilizados na resolução de problemas difíceis de optimização. Uma das vantagens destes algoritmos consiste no facto de eles exibirem boa performance em domínios para os quais não existe muito conhecimento. Esta característica faz com que o algoritmo genético seja um método geral para a resolução de problemas. No entanto, para se resolver um problema com um AG, o utilizador geralmente necessita de específicar uma série de parâmetros que pouco ou nada têm a ver com o problema em si, e que têm mais a ver com o modo de funcionamento interno do próprio AG. Esta dissertação apresenta uma técnica que descobre automáticamente o valor para os parâmetros do algoritmo, simplificando assim a sua operação. A validação do método é feita utilizando problemas artifíciais que são normalmente usados para testar técnicas de AGs, e tambem numa versão simplificada de um problema de expansão de redes.

Acknowledgments

I would like to thank a number of people for their help and support during my dissertation work.

Professor António Câmara for accepting me as his student and for being an open-minded advisor. During my doctoral studies I spent more time abroad than in Lisbon, but he never put any obstacles to that, on the contrary, he encouraged me to go and he let me find my own way.

Professor David E. Goldberg for his guidance and for inviting me several times to his laboratory at the University of Illinois at Urbana-Champaign. All I know about genetic algorithms, I learned from him and from his students. In addition to that, Professor Goldberg taught me to be more organized, taught me to write better, taught me to explain things better, and taught me that hard work pays off. I feel very fortunate to have had the opportunity to work with him.

Georges Harik for his help and for his friendship. It was a pleasure to be sitting in the jacuzzi of his house in California and hear him talk about genetic algorithms, mathematics, and computers; his enthusiasm is contagious. The work contained herein is, in large part, a refinement and application of Georges's brilliant ideas.

Professor Kalyanmoy Deb for his help and friendship. I learned much from working with him, both in the summer of 1997 in Urbana, and during the three wonderful months that I spent in his laboratory at the Indian Institute of Technology in Kanpur.

Júlia Seixas, Luís Correia, Luís Marcelino Ferreira, António Câmara, David Goldberg, and Georges Harik, for accepting to be in the comitee of my dissertation.

My colleagues and friends both at the University of Illinois at Urbana-Champaign and at Universidade Nova de Lisboa: Hillol Kargupta, Jeff Horn, Brad Miller, Erick Cantú-Paz, Ângela Pereira, Yuji Sakamoto, Martin Pelikan, Martin Butz, Dimitri Knjazew, Franz Rothlauf, Tommaso Torelli, Luca Bertini, Tadashi Ogitsu, Mario Medina, Marcia Muñoz, Paula Braga, Fábio Kon, Kiwako Ito, Consuelo Waight, Jacqueline Navarro, Claudia Rodrigues, Inês Moraes, Paulo Ferreira, Ivo Souza, Carlos Lobo, José Miguel Remédio, Pedro Gonçalves, Eduardo Dias, Teresa Romão, Nelson Neves, Inês Sousa, and Ana Pinheiro, for their company and friendship.

Conceição Capêlo, Telma Lourenço, and Donna Eiskamp, for always helping me with administrative duties.

Finally I thank my parents, and brothers and sisters.

This research work was sponsored by Junta Nacional de Investigação Ciêntífica e Tecnológica, Portugal, through a doctoral scholarship PRAXIS BD4020/94 and also by the portuguese electrical utility company, Electricidade de Portugal. During the year of 1999, it was also sponsored by Fundação Luso Americana para o Desenvolvimento.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation is about genetic algorithms (GAs), search and optimization algorithms inspired in principles of natural selection and evolution. They were invented by Holland (1975) and have been applied in a number of different fields including problems in engineering, medicine, finance, and even in the arts. But one thing that stands out from the current literature is that these algorithms seem to require quite a bit of expertise in order to make them work well for a particular application. The expertise is needed because users generally don't know how to decide among the various codings and operators, as well as on deciding on a good set of parameter values for the genetic algorithm. Under this state of affairs, it is no surprise that most users have no other choice other than doing ad-hoc experimentation with a variety of codings, operators, and parameter settings. In the end, instead of being a robust and an easy-to-use method, genetic algorithms as implemented today require a lot of tuning and parameter fiddling.

This situation is unfortunate for the user and it seems that in order to have success with the GA, he needs to have a substantial knowledge of GA techniques; he needs to be a GA expert. But most users are not (and shouldn't be required to be) experts in the field of genetic algorithms. An analogy comes handy here. When people use an electrical appliance, say a toaster, they don't have to know about Ohm's Law or about the internal workings of the toaster. Yet, people use toasters everyday, and most of them have never heard about Ohm's Law [1]. Likewise, with genetic algorithms, users shouldn't need to worry about population sizes, crossover probabilities, and other GA internals. Yet, people should be able to use GAs, even if they don't understand much about how they work. Unfortunately, genetic algorithms are not as easy to use as toasters are.

1.1 Goal of this dissertation

The goal of this dissertation is to make genetic algorithms easier to use. One way of achieving this goal is to develop techniques that simplify the task of setting the parameters of the GA. The work contained herein shows how it is possible to so in a rational and automated way. With an efficient and easy-to-use GA in the practitioners's hands, it is not difficult to envision a worldwide boom in real-world GA-based applications across a variety of domains.

1.2 Scope of this dissertation

This is a dissertation in environmental engineering, and the reader might wonder what do genetic algorithms have to do with it. Put it in another way, is the study of genetic algorithms relevant for the field of environmental engineering? The answer is yes.

Genetic algorithms have been applied in a variety of problem areas, and the field of environmental engineering is no exception. Examples of application are pipe network optimization (Simpson, Dandy, & Murphy, 1994), facility location (Pereira, 1998), pollution source apportionment (Cartwright, 1997), and ecological design of products (Shu, Wal-

[1] This analogy is taken from Johnson (1997) in the context of software design and used here in the context of genetic algorithm usability.

lace, & Flowers, 1996), to name a few. But the scope of this dissertation is much more general than that. The tools to be explored in this work are relevant for almost any field that one can think of, and the field of environmental engineering is just a particular case.

Next, we give a road-map to the dissertation by explaining the contents of each chapter.

1.3 Organization

The dissertation is organized as follows. Chapter 1 sets the stage for the whole dissertation by pointing out to some of the pitfalls of current genetic algorithm practice. Following that, it states the goal or thesis of the dissertation, and outlines the contents of the remaining chapters.

Chapter 2 is a review of genetic algorithms. It includes its operation and basic theory, and also some of the key aspects of its current theory. This theory is centered around the notion of a building block. These studies are motivated by the desire of building better GAs, algorithms that can solve difficult problems quickly, accurately, and reliably. It is therefore a theory that is guided by practical matters. The last part of the chapter focuses on current GA practice, and describes some of the difficulties encountered by users when applying the technology. It also describes some of the things that users typically do to get around these difficulties in order to solve their practical problems.

Chapter 3 reviews the most important research efforts towards understanding the relationship among the various GA parameters, how they affect performance, and attempts to eliminate some of them. They include both empirical and theoretical studies, and some of the work span over to a GA-related class of algorithms called *Evolution Strategies*.

Chapter 4 explains the development of a genetic algorithm that has no parameters. The development of the algorithm takes into account several aspects of the theory of GAs, including previous research work on population sizing, the schema theorem, building block mixing, and genetic drift. With the resulting parameter-less GA, the task of setting the GA parameters is taken out from the user and is handled by the algorithm itself. This is important because users generally don't know how to set the GA parameters and oftentimes set them wrong, resulting in either a waste of computational resources or in poor quality solutions at the end of the GA run. Computer simulations with a simple genetic algorithm demonstrate the validity of the technique.

Chapter 5 illustrates that the method described in chapter 4 is not tied up with a specific genetic algorithm implementation and that it can be used with any kind of GA. More important, the chapter shows that parameter settings is one thing, and codings and operators is another thing. It then moves on and describes one of the most advanced linkage learning algorithms, the extended compact genetic algorithm (ECGA), which was recently developed by Harik (1999). It then shows the parameter-less technique coupled together with the ECGA, resulting in an algorithm that not only relieves the user from setting the GA parameters, it also relieves the user from choosing an appropriate coding and operators. Again, computer experiments demonstrate the validity of the parameter-less technique.

Chapter 6 shows how the tools developed and explored in the dissertation can be transferred to real-world problems. For illustration purposes, it uses a network expansion problem, something that utility companies are often faced with. The example is used as a case study but similar design principles may be applied to other kind of problems as well. Indeed, the chapter can be seen as a guide for genetic algorithm practitioners. The chapter finishes off by defining an empirical measure of problem difficulty, something that can be used to compare the easiness or hardness of real-world problems.

Chapter 7 summarizes the major contributions of the dissertation, gives suggestions for future research, gives suggestions for the genetic algorithm practitioner, and describes how the state of knowledge has changed with the work reported herein.

Finally, appendix A gives a theoretical and empirical analysis of the time complexity of genetic algorithms on problems with exponentially scaled building blocks. Although the research reported in the appendix is a bit different from the main flow of the dissertation, it does have important connections as it is related to population sizing and run duration.

Chapter 2

Overview of genetic algorithms

2.1 Introduction

A genetic algorithm is a search procedure inspired by principles from natural selection and genetics. It is often used as an optimization method to solve problems where little is known about the objective function. The operation of the genetic algorithm is quite simple. It starts with a population of random individuals, each corresponding to a particular candidate solution to the problem to be solved. Then, the best individuals survive, mate, and create offspring, originating a new population of individuals. This process is repeated a number of times, and typically leads to better and better individuals.

This chapter is a review of these search algorithms. It starts with the operation and the basic theory of the genetic algorithm, and then moves on to the key aspects of current genetic algorithm theory. This theory is centered around the notion of a building block. We will be talking about deception, population sizing studies, the role of parameters and operators, building block mixing, and linkage learning. These studies are motivated by the desire of building better GAs, algorithms that can solve difficult problems quickly, accurately, and reliably. It is therefore a theory that is guided by practical matters. The last part of the chapter focuses on current GA practice, and describes some of the difficulties encountered by users when applying the technology. It also describes some of the things that users typically do to get around these difficulties in order to solve their practical problems. The chapter concludes by saying that there is a gap between theory and practice, and that this gap should be narrowed.

2.2 Genetic algorithm operation

This section describes the operation of a simple genetic algorithm (Holland, 1975), (Goldberg, 1989a). The exposition uses a step-oriented style and is written from an application perspective. The steps of applying a GA are:

1. choose an encoding
2. choose a fitness function
3. choose operators
4. choose parameters
5. choose initialization method and stopping criteria

Encoding

The application of a genetic algorithm to a problem starts with the encoding. The encoding specifies a mapping that transforms a possible solution to the problem into a structure containing a collection of decision variables that are relevant to the problem at hand. A particular solution to the problem can then be represented by a specific assignment of values to the decision variables. The set of all possible solutions is called the *search space*, and a particular solution represents a point in that search space. In practice, these structures can be represented in various forms, including among others, strings, trees, and graphs. There are also a variety of possible values that can be assigned to the decision variables, including binary, k-ary, real, and permutation values.

Traditionally, genetic algorithms have used mostly string structures containing binary decision variables. We will be assuming this representation for the purpose of illustrating the basic operation and theory of the GA. The terminology used in GAs is borrowed from real genetics. The structure that encodes a solution is called a *chromosome* or *individual*. A decision variable is called a *gene* and its value is called an *allele*.

Fitness function

Coming up with an encoding is the first thing that a genetic algorithm user has to do. The next step is to specify a function that can assign a score to any possible solution or structure. The score is a numerical value that indicates how well the particular solution solves the problem [1]. Using a biological metaphor, the score is the *fitness* of the individual solution. It represents how well the individual adapts to the environment. In this case, the environment is the search space. The task of the GA is to discover solutions that have high fitness values among the set of all possible solutions.

Operators

Once the encoding and the fitness function are specified, the user has to choose selection and genetic operators to evolve new solutions to the problem being solved.

The selection operator simulates the "survival-of-the-fittest". There are various mechanisms to implement this operator, and the idea is to give preference to better individuals. Selection replicates individuals with high fitness values and removes individuals with low

[1] The assignment of a score to a solution is the most common case but it is not absolutely necessary for the application of a GA. What is necessary is that given any two solutions, the user is able to tell which of the two is the better one.

Individual	Fitness
A	8
B	3
C	5
D	7

Individual	Fitness
A	8
D	7
A	8
C	5

Figure 2.1: The selection operator on a population of 4 individuals.

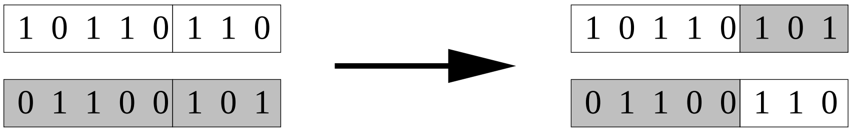


Figure 2.2: Single point crossover.

fitness values. Figure 2.1 illustrates the selection operator on a population of 4 individuals. If selection was the only operator available, no new solutions would ever be created. In order to explore new solutions, the GA relies on two variation operators: crossover and mutation.

Crossover works by pairing members of the population and mixing pieces of one solution with pieces of another solution. The original pair of individuals are called the parents, and the resulting pair of individuals are called the children. This operator is typically applied with a high probability and is responsible for most of the search performed by the GA. Figure 2.2 illustrates a commonly used crossover operator: *single-point* crossover. Note however, that there are many types of crossover operators. The key idea is to exchange pieces from one solution with pieces of another solution. Mutation randomly changes the value of a decision variable. With binary coded GAs, this means changing a 0 to a 1 or vice-versa. Within genetic algorithms, mutation is usually considered a background operator that should be used with a very low probability. It is often used as a way to ensure that diversity is never lost at any gene position.

Parameters

With an encoding, a fitness function, and operators in hand, the GA is ready to enter in action. But before doing that, the user has to specify a number of parameters such as population size, selection rate, and operator probabilities. The work contained in this dissertation greatly simplifies the user's task of specifying these parameters.

Initialization method and stopping criteria

The last steps of applying a GA are the specification of an initialization method and a stopping criteria. The genetic algorithm is usually initialized with a population of random individuals, but sometimes a fraction of the population is initialized with previously known (good) solutions.

Following the initialization step, each individual is evaluated according to the user's specified fitness function. Thereafter, the GA simulates evolution on the artificial population of solutions using operators that mimic the survival-of-the-fittest and principles of natural genetics such as recombination and mutation. The process is repeated a number of times (or generations) until some specified stopping criteria is met. A number of criteria can be chosen for this purpose, including among others, a maximum number of generations or time has elapsed, some predefined fitness value is reached, or the population has converged substantially.

2.3 Basic genetic algorithm theory

This section reviews the basic theory of genetic algorithms which was developed by Holland (1975). This theory constitutes the foundations for the majority of subsequent GA research, including the advanced theoretical work that will be described in section 2.4.

Holland's theory relies heavily on the notion that a good solution can be constructed by combining good pieces, called *building blocks*, from different solutions. Holland introduced the notion of *schema* (*schemata* in plural) to analyze the effects of the GA operators on these pieces (or sub-solutions), and summarized the results in the schema theorem.

A schema is a similarity template that represents a set of solutions from the search space. In the context of binary alphabets, a schema is a string over the ternary alphabet {0,1,*}. The star symbol represents a "don't care" position. For example, schema H = 1**0* represents all strings that have a 1 in the first position and a 0 in the fourth position. Strings 10101 and 11100 are members (or instances) of schema H, but 11111 and 00100 are not. Two important definitions are the *order* and the *defining length* of a schema. The order is the number of fixed-positions (ones and zeroes) in a schema. The defining length is the distance between the two outermost fixed positions. For example, schema H = 1**0* has order 2 and defining length 3.

With the definitions of schema order and schema defining length, Holland was able to analyze the effects of the GA operators on an arbitrary schema H. He quantified mathematically how the number of representatives of a schema change when going from one generation to the next, and summarized it in the schema theorem:

$$m(H, t+1) \;\geq\; m(H, t)\, \frac{\bar{f}(H, t)}{\bar{f}(t)} \left(1 - p_c\, \frac{\delta(H)}{\ell - 1} - p_m\, o(H)\right)$$

where:

$m(H, t)$ is the number of instances of schema H at time t,

$\bar{f}(H, t)$ is the average fitness of the instances of schema H at time t,

$\bar{f}(t)$ is the average fitness of the population at time t,

$\delta(H)$ is the defining length of schema H,

p_c is the probability of crossover,

p_m is the probability of mutation,

ℓ is the string length,

$o(H)$ is the order of schema H.

Holland's schema theorem is written assuming fitness-proportion selection, and single-point crossover. In words, the schema theorem says that selection emphasizes fit schemata, and the variation operators, destroy some of these schemata. The overall lesson of the schema theorem is that highly fit schemata that are not too disrupted by the variation operators, those that are of low order and of short defining length, tend to grow from generation to generation. A more general version of the schema theorem can be written as:

$$m(H, t+1) \;\geq\; m(H, t)\; \phi(H, t)\; [1 - \epsilon(H, t)]$$

This version of the theorem is independent of a particular choice of GA operators. It is also easier to interpret. The effect of selection is given by the reproduction ratio $\phi(H, t)$, and the effect of the variation operators is given by the disruption factor $\epsilon(H, t)$. Overall, a schema can grow or decay according to the net growth factor defined by:

$$\phi(H, t)\; [1 - \epsilon(H, t)]$$

Part of the reason why genetic algorithms are so powerful, is because of its ability to give credit to partial solutions. As an example, consider the 5-bit individual 10100. When the GA evaluates this individual solution, it is also evaluating partial solutions such as 1****, *0***, 1***0, and many other sub-solutions of 10100. The evaluation of these partial solutions or schemata, is done automatically by the GA without any extra computational expenses. This action that the GA does "behind the scenes" is called implicit parallelism, and it is one of the main reasons why genetic algorithms are so powerful. While the GA evaluates a population of N solutions, it also evaluates a much larger number of partial solutions. And it does it without spending more than N fitness function evaluations.

2.4 Current genetic algorithm theory

After Holland's pioneering work, there has been substantial improvements in the understanding of what simple GAs can and cannot do. Holland's theory of schemata suggest that GAs work by combining highly fit sub-solutions or building blocks. Unfortunately, the simple GA can only process well a small fraction of these building blocks, those that are compact [2]. For this reason, there has been extensive research in designing GAs that can effectively process building blocks, regardless of whether they are compact or not. Another important, and related, area of research has been the understanding of how GA performance is affected by the GA parameters and operators. Work along these lines, included studies on selection, population sizing models, and systematic controlled experiments for analyzing the relationship between various GA parameters.

In the remainder of this section, we dissect each of these topics. The exposition is

[2] a building block is compact when its genes are located close to each other in the chromosome.

centered around the notion of a building block and is nothing but an explanation of Goldberg's decomposition for designing competent GAs (Goldberg, Deb, & Clark, 1992).

2.4.1 Deception and building blocks

The fundamental theory of GAs say that they seek near-optimum performance by combining highly fit low-order schemata into higher-order schemata. The problem is that sometimes low-order schemata that are highly fit don't combine to form high-order schemata that are also highly fit. Goldberg introduced the notion of deception to illustrate exactly this point (Goldberg, 1987). Deceptive problems contain two things that cause difficulties to a genetic algorithm: the global solution is isolated, and information misleads the GA to sub-optimal solutions. The following example describes the minimal deceptive problem, which was introduced by Goldberg as an example of a function that can fool the GA. Consider a 5-bit problem where:

$$f(0****) > f(1****)$$
$$f(****0) > f(****1)$$

but:

$$f(0***0) < f(1***1)$$
$$f(0***1) < f(1***1)$$
$$f(1***0) < f(1***1)$$

This example illustrates a case where high-performance low-order schemata, 0**** and ****0, don't combine to form high-performance high-order schemata. In the example, the GA can be easily mislead to solutions like 0***0, when it would be desirable for the GA to have solutions of the form 1***1. Another way to look at this problem is to recognize that having a 1 in the first gene position is not good by itself. Likewise, having a 1 in the fifth position is also not good by itself. What is good is to have a 1 simultaneously in the first and fifth positions. Simply stated, strings of the form 1***1 have to be discovered all at once. We can't expect the GA to discover them by combining strings belonging to 1**** with ****1, because 1**** and ****1 have poor fitnesses and will tend to get extinct from the population quite fast. 1***1 is a building block, but 1**** and ****1 are not building blocks.

The above 2-bit deceptive problem can be extended to a k-bit deceptive problem. These k-bits correspond to k decision variables that can be coded anywhere in the chromosome. As k increases, the problem gets more and more difficult. Although difficult problems contain complex interactions among its decision variables, it is very unlikely that all the decision variables interact with each other in such a complex way. If that was the case, then the whole problem would be equivalent to finding a needle in a hay stack, a task that can only be achieved reliably by a complete enumeration of the search space. GAs are not good at finding a needle in a hay stack. GAs are good at finding solutions that can be constructed by combining sub-parts of the overall solution. Each sub-part corresponds to a small set of decision variables that do interact with each other in a complex way, perhaps corresponding to a little needle in a hay stack sub-problem. These sub-parts are the building blocks, the stuff that highly fit individuals are made of.

2.4.2 Population sizing models

A proper setting of the population size is crucial for the success of the GA. Too small and the GA will converge to sub-optimal solutions. Too large and the GA will spend unnecessary computational resources.

Theoretical studies on population sizing are based on the correct decision-making between a building block and its competitors. For example, consider the two competing schemata: 1*** and 0***. Suppose that on average, the strings belonging to 1*** have higher fitnesses than the strings belonging to 0***. Clearly, the GA should prefer strings of the form 1***, but it can sometimes prefer 0*** because the population might be too small to properly sample the competing schemata. For example, consider the onemax problem. In this problem the fitness of an individual is given by the number of bits set to 1. Consider the two individuals a and b shown below:

individual	chromosome	fitness
a	1011	3
b	0101	2

When these two individuals compete, individual a wins. However, at the level of the gene, a decision error is made on the second position. That is, selection incorrectly prefers the schema *0** to *1**. This occurs because the star symbols are a source of noise for the correct evaluation of a schema. Therefore, the population needs to have enough schema samples so that the GA can correctly decide, in a statistical sense, between 1*** and 0***, *1** and *0**, and so on. The previous example for an order-1 schema can be extended for an order-k schema or order-k building block. Then, the population needs to have enough samples to properly decide between an order-k building block and its $2^k - 1$ competitors. This turns out to be a statistical problem and has been recognized by modeled by Holland (1973), De Jong (1975), Goldberg, Deb, and Clark (1992) and by Harik et al. (1997).

Having a proper population size is essential, but there are two other things that need to be fulfilled in order to have success with the GA. First, the building blocks have to grow from generation to generation. Second, the building blocks from the different individuals eventually have to combine (or mix) in a single individual. Having the building blocks grow is a matter of obeying the schema theorem. Having proper building block mixing is more difficult and is related to the topic of linkage learning.

2.4.3 Building block mixing

The schema theorem provides a way of analyzing the disruptive effects of crossover and mutation, but says nothing about the constructive effects of these operators. Specifically, it doesn't tell how the building blocks from the different individuals are going to come together in a single individual. This topic is known as building block mixing, and was studied by Goldberg, Deb, and Thierens (1993), Thierens and Goldberg (1993), and Thierens (1995). It was an important work that confirmed what was suspected for a long time: without proper linkage, simple GAs are unable to solve problems in reasonable amounts of time.

The work on building block mixing introduced a new tool for analyzing the behavior of the GA: the control map. An example of a control map is shown in figure 2.3. The map contains a region where the GA is expected to work well. In their work, the population is sized according to a model that accounts for proper building block decision-making. The control map is then obtained by varying two GA parameters: selection rate, and crossover rate (the authors ignored mutation in their work in order to study the limits of a GA with selection and crossover alone). For a given combination of crossover rate p_c and selection rate s, the control map predicts whether the GA will be able to mix the building blocks in an optimal solution.

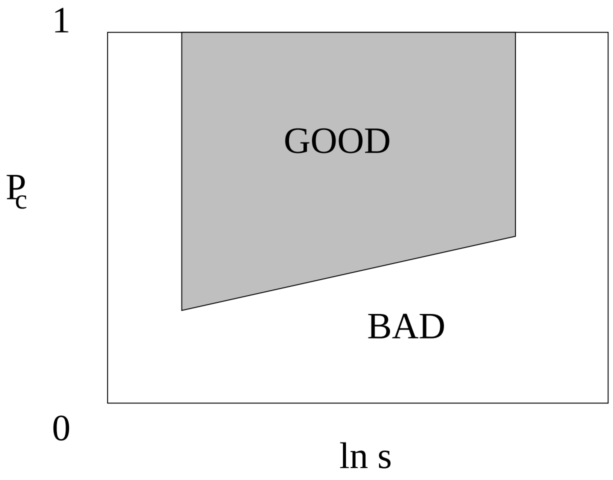


Figure 2.3: A control map for a problem with compact building blocks (Goldberg, Deb, & Thierens, 1993).

On problems where the building blocks are compact, the control maps obtained are similar to the one shown in figure 2.3. However, when the building blocks are not compact, the good region of the control map simply vanishes. That is, no combination of parameters, p_c and s, are able to solve the problem. In order to solve these problems reliably with a simple GA, the population size has to grow exponentially with the problem length. Of course this is not practical, and the performance of the simple GA in these problems becomes no better than a pure enumerative method.

Nonetheless, when the building blocks are compact, the GA is quite robust in its parameter settings. The population size is still a critical parameter, but the others don't seem to be so critical. The GA performs well for a wide range of combinations of p_c and s. The important thing is that selection can't be neither too low nor too high. Too low, and the problem suffers from the effects of genetic drift. Too high, and the GA doesn't have time to mix the building blocks. These extreme examples can be easily illustrated. At low selection pressures, the GA can't distinguish between the good and bad building blocks. At high selection pressures there's no time for mixing and the GA converges too fast. When the selection rate is not too low and not too high, the GA will work well, provided that the growth rate for building blocks expressed in the schema theorem is greater than 1.

The theoretical work on building block mixing was extremely important, and had both good and bad news. The good news is that when the building blocks are compact, the GA is quite robust regarding its parameter values, and there is a big region where the GA is expected to work well. The bad news is that when the building blocks are not compact, the simple GA exhibits poor performance. This is the reason why extensive research has been performed on linkage-learning GAs. These advanced GAs are able to detect which genes constitute building blocks and are able to propagate them as unbreakable units. The ability to do so is crucial for solving difficult problems with genetic algorithms.

2.4.4 Linkage learning

On difficult problems, decision variables interact with each other in a highly complex and non-linear way, and as a consequence, highly fit solutions can be isolated and hard to find. Within simple GAs, the problem arises when the genes that interact with each other in a complex way are located far away from each other in the chromosome. In such cases, the traditional crossover operators are not able to propagate these widely separate genes as a whole piece. Because simple GAs use fixed codings and non-adaptive operators, their performance on difficult problems becomes dependent on whether these highly interacting decision variables are coded near each other or not.

On easy problems any coding does well, but on difficult problems simple GAs rely on a good coding in order to exhibit good performance. Unfortunately, choosing a good coding is far from a trivial task, especially on difficult problems where little is known about the objective function, and about the interactions among its decision variables. Wouldn't it be nice to have a GA that works well regardless of the coding chosen by the user? Of course it would, and that is exactly what GAs that learn gene linkage do. These advanced GAs are able to autonomously detect sets of functional related genes, and are able to process them like whole blocks, like building blocks to construct super individuals.

The topic of linkage has been investigated almost from the beginning of the GA field. Work on inversion (Bagley, 1967; Frantz, 1972; Holland, 1975), messy GAs (Goldberg, Korb, & Deb, 1989; Deb, 1991; Goldberg, Deb, Kargupta, & Harik, 1993; Kargupta, 1995), gene expression messy GAs (Kargupta, 1996), and the linkage learning GA (Harik, 1997) are among the most important contributions to this topic. A detailed review of research work in linkage learning is beyond the scope of this dissertation. For the interested reader, chapter 2 of Harik's dissertation (Harik, 1997) provides a good overview of some of these research works. All of them were significant and very important, but unfortunately all of them had limitations that were hard to overcome. Overall, the competent GA envisioned by Goldberg was only partially completed. But recently, Harik (1999) presented a complete solution to the linkage learning problem. He did it by recasting the linkage learning problem in terms of probabilistic optimization. Similar attempts are also currently being developed and with very promising results (Pelikan, Goldberg, & Cantú-Paz, 1999).

With the topic of linkage we finish this section on current GA theory. At this point, we would like to remind the reader that the theoretical work that was described in this chapter is by no means the only thing that is important in GA theory. Other important works include improving the GA efficiency by means of parallelization, hybridization, function evaluation relaxation, and diversity preservation techniques. Also important is a better understanding of problem difficulty, the role of mutation versus crossover, and the topic of parameter adaptation. We now proceed to the next section, and see what is it that users typically do when applying GA technology.

2.5 Current genetic algorithm practice

During the last years, there has been a boom in the application of GA techniques in the solution of practical problems as can be seen by the large number of application papers in major evolutionary computation conferences (Bäck, 1997; Fogel et al., 1998; Koza et al., 1998; Eiben et al., 1998; Banzhaf et al., 1999). This boom in GA applications is probably due to a wide dissemination of these techniques during the past decade, and also due to the existence of more powerful computers. However, the techniques employed by users have remained nearly the same for the past 15-20 years. At first this might seem odd, but a second thought reveals that theory always need some incubation time before it is put to good use. Moreover, some of the theoretical advances are not easily transferable to a practical context. And besides all this, some of the most important theoretical work is very recent. In summary, there is a gap between GA theory and GA practice. Up to this point we have been looking at theoretical aspects of GAs. In the remainder of the chapter we shift to the practical side of GAs and review what users do when solving practical problem with GAs.

2.5.1 Coding and operators

In general, users have a good intuition regarding what constitutes a good coding and what constitutes good operators. They understand that the GA needs to combine good pieces of one solution with good pieces of other solutions, and therefore, they typically code together in the chromosome things that are somehow physically or spatially related in their problem. Other times, they design special purpose operators that are able to propagate things that have some meaning in their problem. In either case, what the user is trying to do is to respect building blocks by either choosing a good encoding, designing special purpose operators, or both. In a way, the user is somehow seeding the GA with good linkage information. This is probably the reason why simple GAs have had so much success in some applications.

The problem with this kind of strategy is that on difficult problems the interactions among the decision variables are very complex and difficult to understand, thus the the user might think that he has a good coding and good operators, when in fact he does not. Another drawback of this approach is that it is not very general. What is good for one problem, is not necessarily good for another problem. Thus instead of having a method that is widely applicable, we end up having GAs that are too specialized for a particular problem domain. One of the goals of the linkage learning GAs that we have talked about in the previous section, is to have GAs that can in principle relieve the user from having to come up with special purpose codings and operators. Instead, the GA autonomously discovers where the important information is, and automatically comes up with operators that are able to process that information in a proper way.

2.5.2 Parameter setting

Setting the parameters of the GA usually requires a lot of experimentation, and there is not an easy way to set them well for an arbitrary problem. In 1975, De Jong did for the first time a systematic study of the application of GAs to function optimization. He tested various combinations of GA parameters on a set of 5 functions, and concluded that for that set of functions, the following parameters worked well: population sizes of around 50 to 100 individuals, high probability of crossover (between 0.6 and 0.9), and low probability of mutation (between 0.01 and 0.001). After De Jong published his dissertation, these parameter settings have been adopted by many researchers and practitioners. They have been used so often, that they are sometimes referred to as *standard settings*. Unfortunately, many users have taken De Jong's result too seriously, and applied the same kind of parameter settings to solve all kind of problems. Theoretical studies have shown that this is a profound mistake.

2.5.2.1 Population size

Genetic algorithms are quite robust with some of its parameters, but the population size is a critical one, and is related to the problem's difficulty. The more difficult a problem is, the larger the population size should be used in order to reliably achieve a good solution. It should be quite intuitive that in order to solve a larger problem, the GA needs to spend more resources, and therefore the population size should grow when the problem length grows, but that's not the whole story. A typical misconception about this topic is to consider the size of the search space as the only source of problem difficulty. It's not uncommon to hear statements like: " ... my problems is very difficult, it has more than 1000 decision variables ... ". Clearly, as the problem size increases, the problem gets more difficult, but a 1000-bit problem might be much much easier than a 30-bit problem.

What makes a problem difficult is the existence of deep building blocks. Some building blocks are easy to detect by the selection operator, but others might give a very low fitness signal. It has been shown on population sizing models that the population size should be proportional to the building blocks's signal-to-noise ratios, otherwise, the GA will not be able to discriminate between the building block and its competitors. And in addition to all this, there is the mixing issue. The population sizing models, assume perfect mixing of building blocks, and that's something that does not hold in practice within simple GAs, unless the building blocks are compact.

Summarizing, setting the population size for a practical application is quite complex, and there is not an easy way to do it. Theory says that we shouldn't blindly set the population size to 50 or 100 regardless of the problem. But applying the theoretical models is also difficult because they depend on assumptions that may not hold for an arbitrary real-world problem. In practice, users generally conduct empirical tests. Typically, they start with a small population size and then they might try larger sizes. In the end, they try perhaps 5 to 10 different sizes to have a feeling of how the problems responds to different population size settings.

2.5.2.2 Selection, crossover, and mutation rates

It is not uncommon to see users applying proportionate selection schemes without doing any kind of fitness scaling. There are two main problems with such an approach: First, early in the run, a super individual might take over the whole population quite fast leaving no time for recombination to occur. Second, late in the run, the GA might stall because there is no fitness signal to distinguish between good and bad individuals. Due to this reason, proportionate selection schemes should use some kind of scaling, or for that matter, users should instead use order-based selection schemes, because they provide a constant selection pressure (usually tunable by a parameter), and there is no need for doing fitness scaling whatsoever. Surprisingly, many users don't understand this effect, and consequently, they usually say (incorrectly) that GAs have slow convergence times. Regarding crossover and mutation rates, users generally follow the standard settings, or do empirical tests to see what works best.

2.6 Summary

This chapter reviewed the basics of genetic algorithms. It started by giving a step-oriented review of GAs from an application point of view, and then moved to the most critical theoretical aspects of these algorithms. Finally, we looked at the current practice of GAs and explained what users generally do in order to solve a problem with a GA.

The differences between the theory and practice of GAs are large. The usage of advanced genetic algorithms that use adaptive operators are rarely used in practice. Likewise, theoretical studies for setting the GA parameters are rarely used, instead, users tend to do quite a lot of experimentation with parameters, codings, and operators.

A major goal of this dissertation is to narrow the existing gap between GA theory and GA practice by making some of the theoretical lessons more easily accessible to the user. One way of narrowing this gap is by automating a number of decisions that are usually left to the user, for example, setting the GA parameters, a topic to be explored in the next chapter.

Chapter 3

Parameter setting in genetic algorithms

3.1 Introduction

One of the main difficulties that a user faces when applying a genetic algorithm is on deciding on an appropriate set of parameter values. Before running the algorithm, the user has to specify a number of parameters, such as population size, selection rate, crossover probability, and mutation probability. Over the years, there has been a variety of research studies with the purpose of understanding their interactions. Several studies have shown that the GA can tolerate some variation in its parameter values without affecting much the overall performance. That's a good feature of GAs because it would be very bad if the algorithm was extremely sensible to its parameter values, especially when there are no precise methods for setting them. But even though the GA is quite robust in respect to some of its settings, it doesn't mean that the GA will work well regardless of the way it is set up. It can certainly tolerate some variation, but if a poor choice of settings is chosen, the GA will not perform well.

It is well known, both through theoretical and experimental studies, that a proper setting of the parameters depends on the difficulty of the problem being solved. Unfortunately, in general it is difficult to estimate a problem's degree of difficulty, and consequently, there are no recipes for setting them appropriately for an arbitrary problem.

Due to this reason, many users get puzzled when they are faced with the task of solving a problem with a GA. Not only they get puzzled, they also have little intuition on how to set the parameters, because it is hard to establish a relationship between the GA parameters and the solution quality of the problem. That is, the GA parameters have more to do with the GA itself, and not so much to do with the problem that the user is trying to solve.

This chapter reviews the most important research efforts towards understanding the relationship among the various GA parameters, how they affect performance, and attempts to eliminate some of them. They include both empirical and theoretical studies, and some of the work span over to a GA-related class of algorithms called *Evolution Strategies* (ES). For a better exposition of these research works, we divide them in the following three categories:

- Empirical studies
- Facetwise theoretical studies
- Parameter adaptation

Empirical studies measure GA performance on a variety of test problems by systematically varying the parameters of the GA. Facetwise theoretical studies analyze the effects of one or two parameters in isolation, and ignore on purpose the other ones. Parameter adaptation include several techniques that change or adapt the parameter values as the search progresses. In the remainder, we review each category in detail.

3.2 Empirical studies

De Jong 1975

In 1975, De Jong did a systematic study of the application of GAs to function optimization. He tested various combinations of parameters on a set of five functions, including problems that were continuous and discontinuous, convex and non-convex, unimodal and multimodal, deterministic and noisy. By doing so, De Jong included a variety of characteristics that may occur in other problem areas, and thus, his test functions could be seen as a representative of a larger class of problems. Since then, De Jong's functions have been used often by researchers to test and compare various modifications of the genetic algorithm.

In his work, De Jong used a genetic algorithm with roulette wheel selection, one-point crossover, and simple mutation. He investigated the influence of four parameters: population size, crossover probability, mutation probability, and generation gap. The last parameter allowed him to study the effect of overlapping populations (a specified fraction of the population remained intact during a generation step). He did systematic experiments varying each of the parameters. The major conclusions were that increasing the population size resulted in better long-term performance, but smaller population sizes responded faster and therefore could exhibit better initial performance. He found that mutation was necessary to restore lost alleles, but it should be kept at a low rate because otherwise the GA quickly degenerates into a random search strategy. For the crossover probability, a value of around 0.6 worked best, and a non-overlapping population model indicated better performance in general.

Overall, he observed that for his test functions, the following set of parameters gave good performance: population sizes in the range 50-100, crossover probability of 0.6, mutation probability of 0.001, generation gap of 1.0. After De Jong published his dissertation, these parameter settings have been adopted by many researchers and practitioners. They have been used so often, that they are sometimes referred to as "standard" settings.

De Jong also investigated variations off the original algorithm. Specifically, he studied the effects of the following modifications: elitism, multiple crossover points, expected value model, and crowding. He found that elitism (keeping the best individual in the population) was beneficial on unimodal but not on multimodal functions. The expected value model, a proportionate selection scheme that is less noisy than the roulette wheel scheme, gave slightly better performance. Regarding the crossover operator, increasing the number of crossover points degraded the performance. The *crowding* technique, a modification to the selection scheme that helps preserve diversity in the population, was helpful for multimodal functions.

De Jong's investigation was very important and gave practical guidelines for subsequent applications of GAs. Unfortunately, many researchers and practitioners have taken his results too strictly and applied them to solve all kind of problems. Subsequent research has shown that blindly using these so-called "standard" settings can be a serious mistake.

Schaffer et al. 1989

Recognizing that the parameter values can have a significant impact in the performance of the GA, and that a more through investigation was needed, Schaffer et al. (1989) decided to expand De Jong's work by doing a larger experimental work with more exhaustive comparisons among the different parameter combinations. In addition to the 5 test functions used by De Jong, Schaffer et al. included 5 more functions. They performed experiments with six population sizes (10, 20, 30, 50, 100, 200), ten crossover rates (0.05, 0.15, 0.25, ... , 0.95), seven mutation rates (0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1) and two crossover operators. For each combination of parameters, ten independent runs were performed. The authors observed an inverse relationship between population size and mutation rate. That is, high mutation rates were better for small population sizes, and low mutation rates were better for large population sizes.

Overall, Schaffer et al. speculated that the following parameters result in good performance: population sizes in the range 20-30, crossover rates in the range 0.75-0.95, and mutation rates in the range 0.005-0.01. The authors also speculated that an evolution scheme based solely on selection and mutation, is likely to be a powerful search algorithm, and that perhaps, the role of mutation has been underestimated in the GA community. The authors recognized that it is not clear whether their results can be generalized beyond the 10 functions included in their test suite. They said that since their problems seem to be quite insensitive to the crossover rate, and very successful with high mutation rates, that perhaps their test suite might be constituted by functions that are still somehow easy.

3.3 Facetwise theoretical studies

Because the dynamics of the GA are so complex, it is hard to study the effect of all the parameters simultaneously. Therefore, many research studies have analyzed the effect of one or two parameters in isolation, and ignored on purpose the other ones. These facetwise studies have proven to be very useful to get insights about the GA dynamics. Among the most relevant ones, are the works on selection alone (Goldberg & Deb, 1991), mutation (Mühlenbein, 1992; Bäck, 1993), population sizing (Goldberg, Deb, & Clark, 1992), (Harik et al., 1997), and control maps (Goldberg, Deb, & Thierens, 1993; Thierens

& Goldberg, 1993). We review them next.

Selection

Goldberg and Deb (1991) conducted the first theoretical study to analyze and compare different selection schemes in genetic algorithms. Their work completely isolated the selection operator by ignoring on purpose the effects of crossover and mutation. If selection is the only operator in a GA no new solutions are ever created; the repeated application of selection alone in a population of size N, eventually lead (with a high probability) to a population filled up with N copies of the best individual contained in the initial population.

Goldberg and Deb defined *takeover time* as the time that the GA takes to go from 1 super-individual in the initial population to $N-1$ super-individuals at a later generation. The authors calculated analytically the takeover time for a variety of selection schemes including proportionate and order based schemes. The analysis was accompanied by computer simulations that confirmed its correctness. For fitness proportionate selection schemes, the takeover time depends on the fitness function distribution. For order-based selection schemes, such as tournament selection and ranking selection, the takeover time is independent of the fitness distribution and is of order $O(\log N)$ generations.

This work was important because it was the first study that compared selection schemes from a theoretical perspective. From a practical point of view, Goldberg and Deb's work clearly pointed out that proportionate-based selection schemes can suffer from two problems in an actual GA run. The first one is that if early in the search there is an individual that is much better than the others, then that individual quickly takes over the whole population leaving no time for recombination to occur. The second one occurs later in the search when the individuals have nearly the same fitness. Then, there won't be much selection pressure to discriminate among the individuals and the search stagnates.

Mutation

Independently of each other, Mühlenbein (1992) and Bäck (1993) did a theoretical investigation on the effects off the mutation operator in a simple $(1+1)$ evolutionary algorithm. The notation $(1+1)$ is borrowed from the evolution strategies literature. It denotes that the algorithm has one parent and creates one offspring through the application of mutation. Then the better of the two becomes the parent for the next generation. Both Mühlenbein and Bäck, concluded that for a fixed mutation rate throughout the run, the optimal mutation rate in the case of a unimodal problem is $1/\ell$, where ℓ is the problem length. They also concluded that a time-dependent mutation rate decreasing towards $1/\ell$ is beneficial, but only speeds up the search by a little amount.

Mühlenbein also investigated optimal mutation rates for deceptive problems. He concluded that for a maximum order of deception k, the optimal mutation rate is k/ℓ. The analysis was straightforward by recognizing that the most difficult task is to find the very last building block. For example, in the case of a fully deceptive function, the probability of getting the last building block of order k, is equal to the probability that exactly those k bits are mutated, and none of the other bits are. Mühlenbein showed that the simple (1+1) algorithm can solve problems of bounded deception in times of the order $O(\ell^k)$. The result was important in two ways. First, it gave an upper bound on the time complexity for linkage learning GAs. Second, it showed that the optimal mutation rate is problem dependent. Bäck also observed that for the case of multimodal functions, mutation rates other than $1/\ell$ are more adequate. He then suggested that these results may indicate that a self-adaptation mechanism similar to the one used in evolution strategies can be useful in genetic algorithms as well.

Population size

The first attempts to give theoretical estimates of population size in genetic algorithms were given by Holland (1973) and De Jong (1975) who recognized that population sizing in GAs is essentially a statistical decision-making problem. These ideas were further explored by Goldberg (1985, 1989b) and originated the first population sizing equation (Goldberg, Deb, & Clark, 1992) which was later refined by Harik et al. (1997). We have already talked about these studies in the previous chapter, and here we review them with a little bit more detail.

The authors of these studies focused on the population sizing alone, ignoring on purpose issues such as mutation, and poor building block mixing. They viewed the topic of population sizing as being essentially a statistical decision-making problem. Specifically, they recognized that due the finite population size, it is possible that the selection operator makes mistakes when deciding between a building block and one of its competitors. An example is helpful to illustrate this decision-making problem. Consider the four competing schemata:

$$H_1 = * * 11 * * * * * * *$$

$$H_2 = * * 10 * * * * * * *$$

$$H_3 = * * 01 * * * * * * *$$

$$H_4 = * * 00 * * * * * * *$$

These four schemata form a partition of the search space, and compete with each other for population members during the course of a GA run. In the initial generation, the population members will be uniformly distributed among the four schemata. Then, due to the action of selection, the high fit schemata will tend to get more population members, and the low fit schemata will tend to get less population members. Eventually, when the GA converges, one of the four schemata will have won the overall competition.

Let's assume that schema H_1 is a building block. If that's the case, we would like H_1 to be the winner at the end of the GA run. Goldberg et al. reasoned that although the members of H_1 are on average fitter that the members of its competitors, it was possible that in a single competition, a member of H_1 looses with a member of H_2 or H_3 or H_4. This error in decision-making occurs because the don't care symbols ($*$) are a source of noise for the correct evaluation of a schema. Based on this observation, the authors reasoned that the population size should be large enough, so that the GA can correctly decide, in a statistical sense, between a building block and its most tough competitor. Using statistical methods, and given a specified error probability, the authors could compute the population size for problems of known bounded deception. A series of computer experiments showed that the equation was accurate, although a little conservative in its estimates.

The population sizing equation reflects the things that are important for accurate building block decision-making in a GA. However, from a practitioner's perspective, the equation was (and still is) difficult to apply. In order to apply it, the user had to know, or estimate, the selective advantage that building blocks have from their most tough competitors. Goldberg et al. called this selective advantage, the building block's signal. He also has to estimate the building blocks's fitness variance, and the maximum level of deception of the problem that he is interested in solving. And of course, he has to hope that the the building blocks are going to mix well, which may not occur in practice.

Although the theoretical work on population sizing is not easily applicable in practice, it was extremely important in a number of ways. First, it showed for the first time that if there is proper building block mixing, then the time complexity of GAs is sub-quadratic when solving problems of bounded deception. Second, it helped to bootstrap a number of studies including models of building block mixing, linkage learning, and parallelization of GAs. Third, and from a practical perspective, it clearly showed that one shouldn't blindly set the population size to a value in the range of 50-100, and hope that it will work well for most problems.

Control maps

In their study of building block mixing, Goldberg, Deb, and Thierens (1993) were interested in studying the constructive effect of the crossover operator. In order to do that, the authors introduced the concept of a genetic algorithm control map. A control map gives a region of the parameter space where the GA is expected to work well. The effects of mutation were ignored in order to test the limits of a simple GA with selection and crossover alone.

Based on previous studies (Goldberg & Deb, 1991; Goldberg, Deb, & Clark, 1992), the authors built an analytical model of building block exchange. Using a population size that takes into account proper building block decision making, the authors concluded that for a wide range of crossover rates p_c and selection pressures s, the GA is able to work well in the case of a simple order-1 problem. For the case of deceptive problems, it can be shown that the GA is also able to work well for a wide range of combinations of p_c and s, as long as the building blocks are compact. Goldberg, Deb, and Thierens verified that the GA only failed for the extreme cases of very low and very high selection pressures. At very low selection pressures, the GA suffers from the effects of genetic drift. At very high selection pressures, there is an immediate loss of diversity and there's no recombination to be done. Other than these extreme cases, the GA works well for a wide range of values.

Later, Thierens and Goldberg (1993) extended the previous model for problems of bounded deception, but this time without using linkage information (building blocks were not compact). Again, the authors built a mixing model, and the conclusion was that in the absence of proper linkage information, the simple GA requires a population size that grows exponentially with the problem size. That is, even if populations are sized properly by taking into account correct building block decision-making, no combination of selection pressure and crossover rate performs well. The authors pointed out that obeying the schema theorem is a necessary, but not sufficient condition for a successful GA search. Obeying the schema theorem (growing building blocks) is a matter of ensuring that the growth ratio of building blocks is greater than 1, which can be achieved by controlling the selection pressure and the crossover probability. However, even though building blocks are expected to grow, building block exchange may take very long times.

The bottom line of their work, is that a simple GA with selection and crossover alone is limited in what it can do, and that effective linkage-learning mechanisms are needed in order to solve difficult problems with genetic algorithms. Mutation by itself isn't likely to be a very efficient mechanism for solving these difficult problems. Mühlenbein (1992) showed that the time complexity of a mutation-based evolutionary algorithm would be $O(\ell^k)$, where k is the maximum level of deception in the problem.

Regarding the setting of the parameters, the most important lesson from this work, is that if good linkage information is available, and the population is sized properly by taking into account correct building block decision making, then the GA is very robust regarding the settings of selection pressure and crossover rate.

3.4 Parameter adaptation

Parameter adaptation has to do with techniques that change or adapt the parameter values as the search progresses. This topic has been investigated since the early days of the evolutionary computation field. The techniques used include both centralized and decentralized control methods. Centralized methods change the parameter values based on a central learning rule. Decentralized methods have no central control, and have been used more often in evolution strategies. The idea behind decentralized methods is to encode the parameters (typically crossover and mutation rates) in the individual strings themselves. As a consequence, the parameters are also subject to the rules of evolution. Yet another approach for parameter adaptation is the work on meta-GAs. Here, the idea is to run a higher level GA that searches for a good set of parameters for a lower level GA. The higher level GA, or meta-GA, runs on a population whose individuals are encodings of parameter settings, while the lower level GA is a regular GA that uses the settings found by the higher-level GA.

Parameter adaptation techniques have two main advantages. First, the user doesn't have to specify in advance a value for the parameter. Second, a time-dependent value for the parameter may be beneficial for the search. This section is a review of these parameter adaptation methods.

3.4.1 Centralized control methods

Cavicchio 1970

During his dissertation, Cavicchio (1970) developed an adaptive scheme that adjusted the genetic operator rates through time. Cavicchio's parameter modification technique adjusted each parameter probability on the basis of its involvement in creating new population members. By new population member, he meant an individual that performed well enough to take the place of a parent in the current population. Cavicchio reported significant gains by incorporating the above parameter modification scheme. He was interested in these adaptive schemes because he found that it was difficult to have optimal parameter settings for a particular application. Finding such parameters often involved extensive experimentation, and usually turned out not to be very robust. That is, what works well for a particular application, didn't necessarily worked well for another application. He also found that different operators have different performance depending on the stage of the search. Due to these reasons, Cavicchio turned his attention to parameter adaptation techniques with the hope of finding a more flexible approach, something that could give robust settings that work well in a variety of problem domains.

Davis 1989; Julstrom 1995

Davis (1989) introduced a similar adaptation scheme. His idea was to adapt the operator probabilities based on their observed performance. If an operator was responsible for the creation of a good individual, then the operator should be rewarded and be used more frequently. Likewise, if an operator performed poorly, it should be penalized, and be used less frequently. Davis was aware of previous works that attempted to find robust parameter settings for the GA (De Jong, 1975; Grefenstette, 1986), but he recognized that in many applications, the use of other operators and other representations, often yielded superior performance.

To study the aspect of operator probability adaptation, Davis used a steady-state GA. In this kind of GA, the population is not fully replaced in each generation. Instead, after each reproduction event, the children are inserted in the population, and some old member of the population is removed. In addition, Davis didn't apply crossover and mutation in the same reproduction event. Instead, only one operator was applied at each reproduction step. Davis tested his scheme with a GA containing a total of 5 operators (2 crossover operators and 3 mutation operators), each with a probability of being fired. Throughout the run these probabilities changed, rewarding the operators that were performing well, and penalizing the operators that were performing poorly.

Julstrom (1995) also investigated a parameter adaptation scheme in the same spirit of Davis. The main difference between the two was in the adaptation or learning rule itself. While Davis's adaptation scheme required a few external parameters to control it, Julstrom's rule didn't require as many control parameters, and was somewhat simpler. Both Davis and Julstrom found that their parameter adaptation schemes were effective in the problems that they tried. They also observed that a fixed fine tuned parameter setting often yielded superior performance than an adaptive scheme, but of course, finding fine tuned settings is in itself a very time consuming task. The interesting thing about adaptive schemes, however, is that in principle they are more robust than the fixed settings, because they can easily adapt depending on the problem they are operating on.

Lobo and Goldberg 1997

A similar adaptation scheme was also investigated by Lobo and Goldberg (1997). The authors were primarily interested in building a model for investigating how to do efficient hybridization of GAs. In order to do that, they ignored many aspects that need to be considered in a whole hybridization theory, and focused solely on the question of how to combine two different methods. Specifically, they combined a GA with a local search technique by developing a scheme that could autonomously switch between one method or the other. The switching was done based on the observed performance of each method. For the purposes of implementation, the GA was implemented with selection plus crossover, and the local search method was implemented with selection plus mutation. Although the authors were interested in the broader question of how to combine different methods in a hybrid algorithm, the resulting scheme could also be interpreted as a technique for automatically adapting method/operator probabilities.

Adaptive population size, Smith and Smuda 1995

Most of the work on parameter adaptation has focused on adapting mutation and crossover rates, but there has been very little work on adaptive population sizing schemes. An exception is the work of Smith and Smuda (1995) who tried to transfer the population sizing theory of Goldberg, Deb, and Clark to a more practical context. Specifically, Smith and Smuda tried to incorporate Goldberg, Deb, and Clark's population sizing equation in the GA itself. The authors wanted to remove the parameters of the GA, and starting with the population size seemed like a logical first step because experience had shown that the GA was quite robust regarding the other parameters. Moreover, the work of Goldberg, Deb, and Clark had given great insight regarding the role of the population size in a GA.

Based on that theoretical work, Smith and Smuda suggested an algorithm for autonomously adjusting the population size as the search progresses. Smith and Smuda argued that the parameters of the GA are hard to relate to the user's need. That is, the user typically doesn't know how the population size affects the solution quality of the problem. Thus the authors suggested that the parameters should have more meaning from the user's point of view, and proposed an algorithm that only required the user to specify a desired accuracy for the overall search, something equivalent to the building block's signal from Goldberg et al.'s equation. In order to do that, the authors defined *expected selection loss* between two competing schemata as the probability that the GA makes a selection error (chooses the lower fit schema) weighted by their fitness difference. Then, they suggested an algorithm that does online estimates of schema fitness variances, and sizes the population so that the expected selection loss approximates the user's specified target accuracy.

Although they suggested and implemented such a scheme, the resulting algorithm had several limitations. First, it is hard to relate the user's specified selection target loss with the actual accuracy of the GA in solving the problem. Second, the estimation of schema fitness variances is very noisy, because the GA samples many partitions simultaneously. Third, the population sizing theory assumes that the building block mixing is going to be nearly perfect, which may not occur in practice. Nonetheless, the work of Smith and Smuda is a significant one, especially because there has been very few studies that have attempted to automatically adapt the population size. And theory has shown that the population size is a very important parameter in a GA, perhaps the one that is the most difficult to set in the absence of knowledge about the problem.

3.4.2 Decentralized control methods

Bagley 1967

In the parameter adaptation schemes described above, the control rules for changing the parameters were centrally control. Below we describe parameter adaptation schemes that have no central control. Instead the parameter adaptation in done in decentralized fashion. This kind of work has been done more often in a GA-related class of algorithms called Evolution Strategies. But some of the techniques are being transferred to GAs as well. In fact, quiet early in the history of GAs, Bagley (1967) suggested that the control parameters of the GA could be encoded in the individuals themselves and be subject to genetic search as well. Although Bagley didn't implement it, he envisioned that such a scheme could be useful. Next, we review Evolution Strategies (ES). Almost from the beginning of their development, the ES include adaptation of some of its control parameters.

Self-adaptation in Evolution Strategies

Evolution strategies are a class of evolutionary algorithms that were developed in Germany by Rechenberg and Schwefel at about the same time that Holland developed genetic algorithms in the United States. The early evolution strategies were constituted solely by selection and mutation, but later on, started to include crossover as well. The main difference between evolution strategies and genetic algorithms is the emphasis that each class of algorithms gives to their search operators. Evolution strategies rely more on mutation as a search operator, while in genetic algorithms the emphasis is more on crossover. Another important difference is that within genetic algorithms, the parameters are usually set externally and are held constant during the course of the entire search, while in evolution strategies, some control parameters (mutation rates) are encoded in the individuals themselves, and are also subject to adaptation. A paper by Bäck and Schwefel (1995) gives a good overview of evolution strategies, and another paper also by Bäck and Schwefel (1993) gives a good comparison of evolution strategies and genetic algorithms.

The first evolution strategy was the so-called $(1 + 1)$ strategy. The algorithm has a population of size 1. The notation $(1 + 1)$ means that the algorithm uses 1 parent to produce 1 offspring, and the better of the two will be parent for the next generation. Evolution strategies were originally applied to numerical optimization problems with real valued decision variables. The generation of offspring was obtained by adding Gaussian mutations to the variables of a parent individual. Rechenberg did a theoretical investigation on the convergence speed of the $(1 + 1)$ strategy on two simple test functions. Based on that work, he found that the ratio of successful mutations to all mutations is around $1/5$. After that work, he suggested that if this ratio is greater than $1/5$, one should increase the mutation step size; if it is less, one should decrease the mutation step size. In the early evolution strategies, this heuristic was often used to dynamically adjust the mutation step size (standard deviation of a Gaussian random variable).

The $(1+1)$ strategy is essentially a local search strategy. After the algorithm reaches a local optimum, it might take quite a while to jump to another optimum, especially if the rate of successful mutation is low (in a rough landscape, the rate of successful mutations can be low, and due to the 1/5 rule, there will be a decrease of the mutation step size, which may lead to very long waiting times for jumping to another optimum). Following the $(1+1)$ strategy, Rechenberg proposed a multimembered strategy, where μ parents participate in the generation of one offspring. The notation is $(\mu+1)$. With a population of individuals, an operator was devised to mimic sexual reproduction, the equivalent of crossover in genetic algorithms.

More general strategies, the $(\mu+\lambda)$ and the (μ, λ), were introduced later. In these strategies, μ parents produce λ offspring by means of crossover and mutation. The difference between the two strategies is in the selection operator. In the case of the $(\mu+\lambda)$ strategy, the individuals that survive for the next generation are the best μ out of both parents and offspring. In the case of the (μ, λ) strategy, the individuals that survive for the next generation are the best μ out of the offspring only. These modern evolution strategies also have a more robust scheme for adjusting the mutation step sizes. Specifically, the standard deviation for the mutations are coded as part of the individuals, and evolve just like the decision variables do. This process is called self-adaptation of strategy parameters. Each individual can have up to n standard deviations encoded, one for each decision variable. The generation of offspring consists of the following steps.

1. recombine(μ) individuals to generate one offspring

2. mutate the standard deviations of the resulting individual

3. use the resulting standard deviations to mutate the decision variables.

Schwefel also proposed that information about mutation correlations be encoded. In this later scheme, each individual encodes a set of n decision variables, up to n standard deviations of Gaussian random variables, and a covariance matrix. The correlated mutations can be thought of as the equivalent of linkage learning in genetic algorithms, and can be interpreted as a kind of a smart mutation operator.

With these modern evolution strategies, the control of the parameters is done in a decentralized way, as opposed to the centralized 1/5 adaptation rule of the early strategies. Bäck (1992) and Smith and Fogarty (1996) suggested that the techniques of self-adaptation in evolution strategies could be transferred to the context of genetic algorithms as well.

3.4.3 Meta-GAs

Weinberg 1970, Mercer and Sampson 1978, Grefenstette 1986

The relationship among the various GA parameters is a complex one and it is not very clear how they interact with each other. Moreover, the space of all possible parameter configurations is very large. Therefore, finding a good set of parameter values is in itself a problem that seems to need optimization or search. Thus the idea of running a higher level GA, that searches for a good set of parameters for a lower level GA was born. The higher level GA, or meta-GA, runs on a population whose individuals are encodings of parameter settings. Weinberg (1970) was the first to suggest such an adaptive scheme, but he never implemented it. A few years later, Mercer and Sampson (1978) implemented a meta-GA. In their work, a meta-structure is a string of values representing rates (probabilities) of application of parameters. Mercer and Sampson used their adaptive scheme on the same problem that was studied by Cavicchio (1970) some years before. They observed significant improvements over Cavicchio's adaptation scheme.

Grefenstette (1986) also did an investigation with a meta-GA. He identified 6 parameters: population size, crossover rate, mutation rate, generation gap, scaling window, selection strategy. Then he discretized each of these parameters and the resulting parameter space consisted of 2^{18} possible parameter configurations. Each individual in the meta-GA consisted of an 18-bit string which represented a particular GA configuration. To evaluate a particular one, Grefenstette used both online and offline performance measures like De Jong had done previously. The test problems were the 5 test functions from De Jong's study.

Grefenstette's meta-GA more or less confirmed De Jong's results: mutation rates above 0.05 are generally harmful, population sizes in the range 30-100 work best, high crossover rates, low scaling windows (steady selection pressure towards the end of the run), large generation gap (nearly all the population is replaced each generation), and elitist selection strategy. Grefenstette also tested the GA found by the meta-GA in an image registration problem. His idea was to see whether the parameter settings given by the meta-GA could be extended for other problems, even though they only used De Jong's test suite as training examples. Grefenstette compared the standard settings found by De Jong with the best GA found by the meta-GA, and concluded that the GA resulting from the meta-GA was better in online performance and about the same for offline performance. He then speculated that the results obtained may be generally applicable to other optimization problems.

3.5 Summary

This chapter did a survey of the most important research efforts towards understanding the relationship among the various GA parameters, how they affect performance, and attempts to eliminate some of them. The review included empirical studies, facetwise theory, and parameter adaptation techniques. The results from all of them have been important for the understanding of GAs, and have given guidelines and rules of thumb for setting its parameters. Nonetheless, the genetic algorithm still needs a lot of care and tuning.

Most of the previous attempts to automate GA parameters haven't really separated the issue of the parameters themselves (population size, selection rate, operator probabilities) from the operators. Indeed, there has been many research works attempting to adapt parameters and the type of operator at the same time. Within the work of the parameters themselves, most of previous works have focused more on operator probabilities and not so much on population sizing. That is unfortunate because the sizing of the population is one of the most crucial parameters for the success of the GA. Nowadays, it is clear that the standard population settings of $50-100$ individuals often used by GA practitioners can be far from good.

We now proceed to the next chapter where we will start by making a clear separation between parameters and operators. Thereafter, we will investigate how some of the parameters of traditional genetic algorithm, including population sizing, can be eliminated.

Chapter 4

A parameter-less genetic algorithm

4.1 Introduction

From the user's point of view, setting the parameters of a genetic algorithm is far from a trivial task. Moreover, the user is typically not interested in population sizes, crossover probabilities, selection rates, and other GA technicalities. He is just interested in solving a problem, and what he would really like to do, is to handin the problem to a black-box algorithm, and simply press a start button. This chapter explores the development of a GA that fulfills this requirement. It has no parameters whatsoever. The development of the algorithm takes into account several aspects of the theory of GAs, including previous research work on population sizing, the schema theorem, building block mixing, and genetic drift.

This chapter is largely based on the paper by Harik and Lobo (1999). It starts by pointing out some of the pitfalls of current GA practices and categorizes the variety of choices that users are faced with in two categories: (1) coding and operators. (2) parameter settings. Then, it focuses on the second category and describes the development of a parameter-less GA. Thereafter, computer experiments compare its performance with that of a regular GA. We conclude the chapter by saying that the parameter-less GA is economical, and even speculate that future GA practice will not need a GA specialist.

4.2 Genetic algorithms need to be more user-friendly

One thing that stands out from the current literature is that genetic algorithms seem to require quite a bit of expertise in order to make them work well for a particular application. The expertise is needed because users generally don't know how to decide among the various codings and operators, as well as on deciding on a good set of parameter values for the GA.

Many are attracted to GAs because of their broad applicability and simplicity of use. In principle, all that is needed is an encoding of the decision variables of the problem and a fitness function which can measure the efficacy of a given solution. "Here is my problem, please solve it". This should be it as far as the user is concerned, but the reality of current GA practice is quite different from that. The reality is that the user is faced with a vast array of decisions that still need to be made before turning the GA on. He has to choose a specific coding for the decision variables of the problem, he has to choose operators, he has to set parameters for these operators, and he has to set parameters for the GA. Making these decisions properly involves knowledge that have more to do with the way the GA works and not so much with the specific problem that the user is trying to solve.

This situation is unfortunate for the user and it seems that in order to have success with the GA, he needs to have a substantial knowledge of GA techniques; he needs to be a GA expert. But most users are not (and shouldn't be required to be) experts in the field of genetic algorithms. Under this state of affairs, it is no surprise that most users have no other choice other than doing ad-hoc experimentation with a variety of codings, operators, and parameter settings. In the end, instead of being a robust and an easy-to-use method, the genetic algorithm turns out to be a method that needs a lot of tuning and parameter fiddling.

4.2.1 Coding and operators is one thing, parameters is another thing

In general, users make little distinction among the variety of choices that must be made before running the GA. Oftentimes, all these choices are referred to as parameters and are handled in the same way. In the previous chapter we reviewed a variety of research works regarding the topic of parameter setting in GAs, and there too, sometimes the distinction was not made. For example, investigations have been made where not only the GA parameters, such as population size, crossover and mutation probabilities were varied, but also the type of crossover operator was varied. In this chapter we make a clear distinction between these choices or decisions by separating them in two categories. The first, is the choice of an appropriate coding and operators. The second, is the choice of appropriate parameter settings.

- choice of coding and operators
- choice of parameter settings

The distinction is important because they are related to two different things and can be handled separately. This chapter only addresses the second category. Not that the first one is unimportant or irrelevant. It is indeed a very important topic that has been the subject of extensive research. Specifically, one of the motivations of the work in linkage learning GAs, is to relieve the user from having to come up with special purpose codings and operators. Linkage learning GAs are able to detect important similarities (building blocks) in the encoding, and are able to come up with smart operators that are able to efficiently process building blocks. The work contained herein has a similar flavor regarding the second category of choices. The outcome is an algorithm that relieves the user from having to set the parameters of the GA. The resulting parameter-less GA makes life easier to users and goes one step closer in the direction of an algorithm that is robust, efficient, and simple to use.

4.3 Development of a parameter-less GA

In the previous chapter we have categorized the various research works on the topic of parameter setting in GAs into three classes: empirical studies, facetwise theoretical studies, and parameter adaptation techniques. The work that we are about to present falls under the category of facetwise theoretical studies and is built on top of some of the studies mentioned previously. Specifically, the work on genetic algorithm control maps is key for eliminating the selection rate and the crossover probability. The work to be presented is a facetwise study because it ignores mutation for the time being. This doesn't mean that mutation is unimportant. There has been extensive debates in the evolutionary computation community regarding the usefulness of crossover versus mutation and vice-versa. We recognize that mutation can be quite helpful in certain situations and that it shouldn't be discarded lightly, but in this work we ignore it for the sake of simplicity.

There are three parameters that affect the performance of a crossover-based GA: population size, selection rate, and crossover probability. In the remainder of the section we explain how we can eliminate these parameters. The exposition proceeds in two steps. First, we get rid of selection rate and crossover probability. Then, we get rid of population sizing.

4.3.1 Eliminating selection rate and crossover probability

The selection rate and crossover probability can be eliminated in a single shot if one has a proper understanding of the roles of these two parameters. The selection rate s allows the user to control the amount of bias towards better individuals. The crossover probability p_c allows the user to control the amount of recombination or mixing. Together the two parameters work for the same purpose, which is to ensure that building blocks grow from generation to generation.

A closer look at the schema theorem reveals that the survival probability of a schema can be made to increase by either raising the selection rate or lowering the crossover probability. Previous research (Goldberg, Deb, & Thierens, 1993) have shown that the GA is quite robust regarding the settings of these two parameters. Goldberg, Deb, and Thierens observed that when the building blocks are compact, the GA works well for a wide range of combinations of p_c and s. The important thing is to respect the schema theorem and not fall in the extreme cases of very low and very high selection pressures. At very low selection pressures the GA is not able to discriminate between the good and bad individuals. At very high selection pressures the GA only pays attention to the very best individuals, resulting in an immediate loss of diversity and little recombination to be done.

The two extreme cases are easily excluded by setting the selection rate to a value that is neither too low nor too high. Apart from that, the GA just needs to obey the schema theorem by making sure that the growth ratio of building blocks is greater than 1. Let $\phi(H,t)$ be the effect of the selection operator on schema H at generation t, and $\epsilon(H,t)$ be the disruption factor on schema H due to the crossover operator. Then the overall net growth ratio on schema H at generation t is given by:

$$\phi(H,t)[1-\epsilon(H,t)]$$

The above expression is nothing but a simplification of the schema theorem. Under the conservative hypothesis that a schema is destroyed during the crossover operator, and substituting s and p_c in the formula above, we obtain that the growth ratio of a schema is given by the expression:

$$s(1-p_c)$$

This calculation traces its roots to the first messy GA paper (Goldberg, Korb, & Deb, 1989). Setting $s = 4$ and $p_c = 0.5$ gives a net growth factor of 2, and ensures that the necessary building blocks will grow. Whether they will be able to mix in a single individual or not is now a matter of having an adequate population size. If the building blocks are compact, the population sizing requirements will be quite reasonable (Goldberg, Deb, & Clark, 1992), (Harik et al., 1997). If they are not compact, then, unless the GA is able to learn linkage on the fly, the population sizing requirements will be extremely large (Thierens & Goldberg, 1993). In either case, we can get rid of selection rate and crossover probability by setting $s = 4$ and $p_c = 0.5$ for all problems.

4.3.2 Population sizing in theory and practice

In the genetic algorithm, the population can be thought of as a collection of data or observations. The genetic algorithm needs to have enough of these observations so that it can detect the important sub-structures or building blocks of a problem. How big the population should be to detect these building blocks depends on a problem by problem basis and is related to the problem's difficulty; the more difficult the problem is, the larger the population size should be. It is obvious that the larger the population is, the more chance the genetic algorithm has to find a solution of higher quality. However, there is a computational cost associated with having a very large population. Overall, there is a tradeoff between solution quality and cost. It might not be worth to spend the additional cost of having a larger population just to have a very tiny improvement in the solution quality, or in some cases, to have no improvement at all.

We have seen in the previous chapters that theoretical models have been developed to address the topic of population sizing in GAs (Goldberg, Deb, & Clark, 1992), (Harik et al., 1997). The essence of those models is the recognition that the GA can make mistakes when deciding between a building block and one of its competitors. By increasing the size of the population, the GA can get a better sampling of building blocks and reduce the error in decision-making. In simple words, the models dictate that the population size has to be large enough so that the GA can correctly decide, in a statistical sense, between a building block and its competitors.

The result is an equation that says that the population size should be proportional to the problem length, and to building blocks's signal-to-noise ratios. Although accurate, those models are difficult to apply in practice because they rely on several assumptions that may not hold for real-world problems. For example, in order to apply the population sizing equation, the user needs to know or estimate, the maximum level of deception in a problem, and the selective advantage (signal) of a building block over its most tough competitor. This information is usually unknown and is also hard to estimate. Moreover, the equation is only accurate when there is a proper mixing of building blocks. Due to these reasons, it becomes difficult to apply the equation in practice.

The theoretical models on population sizing are extremely important for the design of efficient genetic algorithms. Among other things, an important practical lesson of those models is that setting the population size to a fixed value regardless of the problem's size

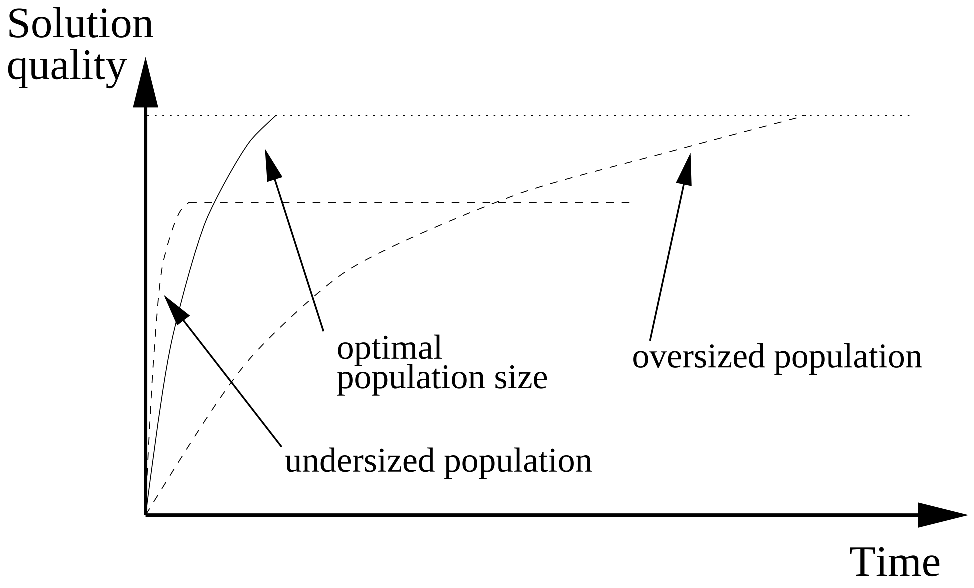


Figure 4.1: Population sizing in genetic algorithms. Too small and the user pays a price in solution quality. Too large and the user pays a price in time.

and difficulty is certainly a mistake. Although the population sizing models are important, it is difficult to use them in practice to size populations for real world genetic algorithm applications due to the reasons mentioned in the previous paragraph. The bottom line is that in practice the user has to do some experimentation and guess the population size. Guessing right is pure luck; most likely the user will guess wrong. There are two types of wrong guesses: (1) a population size that is too small. (2) a population size that is too large.

Type I error: undersized population. If the population size is not big enough, the GA will run out of steam prematurely and converge to sub-optimal solutions; the user pays a price in solution quality.

Type II error: oversized population. If the population size is too large, the GA will waste unnecessary computational resources spending more time than it is necessary; the user pays a price in time.

These two types of errors are depicted in figure 4.1. In the next subsection, we present a technique that eliminates the need of guessing the population size. The basic idea is to continuously increase the population size in an attempt to reach the right sizing. When to stop this growing of the population is left to the user. He will decide when to stop it, and most likely, he will do that based on economic reasons. Simply put, he will stop the GA when he thinks that it is not worth to wait more for an improvement in solution quality. As we will see soon, the parameter-less GA will be a bit slower that a GA that starts out with an optimal population size, but it will do much better than a GA that starts with an undersized or oversized population (the most usual case in current practices).

4.3.3 Eliminating population sizing

Let's step back for a moment and observe what a user typically does to solve a problem with a GA. Most likely, he starts by trying a small population size. Then, he might try a larger one. In the end, perhaps he has tried 5 to 10 different population sizes to have a feeling of how the problem responds to different population settings. This situation is certainly familiar to many GA users around the world, ourselves included. But why should every user need to do this kind of experimentation for every single problem? Why not have the algorithm do the experimentation automatically? That's more or less what we are about to propose here. Next, we present two approaches for getting rid of population sizing. The first is simpler, but has a major drawback. The second comes as a logical consequence of the first one, and constitutes an effective mechanism for eliminating the population sizing parameter once and for all.

First attempt: double the population size at convergence.

This approach consists of iteratively running the GA with different population sizes. Starting with a small population size, say 4, let the GA run until it converges.[1] Following that, the GA fires a new population, twice as large as the previous one, and again let it converge. And the process repeats *ad eternum*. Each time the population converges, the population size is doubled. The GA works forever and only stops if the computer runs out of memory, or the user is happy with the solution quality, and presses a stop button.

The technique, although simplistic, is powerful. It roughly mimics what the user typically does in an ad-hoc way, except this time, the GA is doing it in a systematic way. By doubling the population size each time the population converges, there is a guarantee that the time needed by the parameter-less GA in order to reach a certain solution quality, will be within a factor of 2 of the time needed by a GA that starts with an optimal population size. Moreover, if the user doesn't want to wait that long, and presses the stop button early on the run, the parameter-less GA does probably better than a GA with an optimal population size.

This technique has a drawback that is not immediately obvious. The occurrence of genetic drift may lead to long convergence times. Genetic drift occurs when there is not enough pressure to discriminate between two or more distinct solutions. The simplest way to illustrate it is with a one-bit problem where both individual 0 and individual 1 have the same fitness. Consider a population of N individuals, half of them are like individual 0, and half are like individual 1. When a GA runs on such a population, there's no selection pressure to discriminate between the two different solutions, and eventually, the population will converge to all zeros or all ones. But the whole process looks like a

[1] Convergence means that all the individuals have the same genotype. Notice that this eventually happens because mutation is turned off.

random walk, and the waiting times until convergence can be quite long. This roaming around of the population is called *drift*, and may occur in our proposed parameter-less GA. The consequences are that the waiting times before the population size is doubled can be quite long and unacceptable.

One solution to avoid these long waiting times is to detect if a population is drifting, and in the affirmative case, fire a new population twice as large, even before the old one converges. However, drift detection is a difficult task. One could think of stopping the population from running if the fitness variance of the population members falls within a certain threshold ϵ, but such an approach introduces two problems. First, there's the introduction of a parameter, which goes against the whole philosophy of this work. Second, many real-world problems have noisy fitness functions. On such problems, the threshold ϵ might never be reached. Fortunately, there's another way to get around the drift problem, and without the need of additional parameters.

Second attempt: establish a race among multiple populations.

An alternative approach for tackling the drift problem, consists of running multiple populations simultaneously. The idea is to establish a race among populations of various sizes. The parameter-less GA gives an advantage to the smaller populations by giving them more function evaluations. Consequently, the smaller populations have a chance to converge faster than the larger ones. That is, smaller populations get a head start at the beginning, but if they start to drift too much, they will be caught by a larger population. When that happens, the smaller populations become inactive. Specifically, if at any point in time, a larger population has an average fitness greater than that of a smaller population, then the parameter-less GA gets rid of the smaller population.

The rationale for doing this is as follows. If a larger population has a higher fitness than a smaller population, then there is no point in continuing running the smaller population, since it is very unlikely that the smaller one will produce a fitter individual than the larger one. The drift problem is partially eliminated because if a population starts drifting, it will be caught a larger population. The coordination of this array of populations seems complex, but can be easily implemented with a counter as illustrated by table 4.1.

Table 4.1: Mechanics of the parameter-less GA.

Counter base 4	most significant digit changed	Action
0		
1	1	run 1 generation of population 1
2	1	run 1 generation of population 1
3	1	run 1 generation of population 1
10	2	run 1 generation of population 2
11	1	run 1 generation of population 1
12	1	run 1 generation of population 1
13	1	run 1 generation of population 1
20	2	run 1 generation of population 2
21	1	run 1 generation of population 1
22	1	run 1 generation of population 1
23	1	run 1 generation of population 1
30	2	run 1 generation of population 2
31	1	run 1 generation of population 1
32	1	run 1 generation of population 1
33	1	run 1 generation of population 1
100	3	run 1 generation of population 3
101	1	run 1 generation of population 1
⋮	⋮	⋮

At each time step, a counter of base 4 is incremented, and the position of the most significant digit that changed during the increment operation is noted. That position indicates which population should be ran. In the example, the algorithm initially runs population 1 for 3 generations, then it runs population 2 for 1 generation, then population 1 for 3 more generations, then population 2 for 1 generation, and so on as illustrated in table 4.1.

Overall, population i is allowed to run 4 times more generations than population $i+1$. Just like in the description of our first attempt, each new population that gets fired is twice the size of the previous population. Taking into account both the number of generations and the population sizes, we observe that population i is allowed to spend twice the number of function evaluations of population $i+1$.

Eventually, a population converges (most likely population 1). At that point the algorithm removes it, and resets the counter. Likewise, if the average fitness of a population is less than the average fitness of a larger population, then the smaller population is removed, and the counter is reset.

Table 4.2 gives an example of what the typical state of the parameter-less GA might look like at a particular point in time. In the example, the parameter-less GA is currently running with 3 different population sizes. The smaller one has size 256 and is at generation 30. The next population has size 512 and is only at generation 6. The larger one has size 1024 and is still at generation 1. Tables 4.3 and 4.4 show the population of size 256 being overtaken by the population of size 512. The operation of the parameter-less GA allows the population of size 512 to run its 7^{th} generation, and as a result, the average fitness of that population increased to 13.2 which is larger than 12.6, the average fitness of the population of size 256. Because of this, the parameter-less GA detects that there is no point in continuing running the smaller population and eliminates it.

Table 4.2: Typical state of parameter-less GA.

population index	population size	current generation	average fitness
1	256	30	12.6
2	512	6	11.8
3	1024	1	7.8

Table 4.3: The population of size 512 run its 7^{th} generation and overtakes the population of size 256.

population index	population size	current generation	average fitness
1	256	30	12.6
2	512	7	13.2
3	1024	1	7.8

Table 4.4: The population of size 256 is not active anymore.

population index	population size	current generation	average fitness
1	512	7	13.2
2	1024	1	7.8

The operation of the parameter-less GA uses a counter of base 4 as illustrated in table 4.1. You might wonder why we chose a counter of base 4 and not any other base. Not only that, you might also wonder if there is an optimal base b that should be used. In general, problems that are affected by drift will benefit from a small base, while problems that are not affected by drift will benefit from a large base value. In fact, the first attempt to eliminate the population size that we described can be seen as a particular case of the more general approach using a base $b = \infty$. Notice that the issue of choosing the base b seems to go against the whole purpose of this work. After all, the main motivation of the parameter-less GA is to relieve the user from having to set the parameters of the GA. Because of this, we opted to use a fixed value of $b = 4$, which turns out to be a reasonable compromise between problems that are affected by genetic drift and those that are not.

4.3.4 Worst case analysis

Pelikan and Lobo (1999) did a theoretical analysis for the worst case performance of the parameter-less GA as compared to a GA that starts with an optimal population size. We will not go into the fine details of the analysis here but we would like to give the intuition that is behind it and present a simplified analyses along with the main result of that work.

At a given point in time, the parameter-less GA maintains a number of active populations. A population remains active until it either converges or is overtaken by a larger population. The worst-case scenario for the parameter-less GA occurs when no population ever converges and no population is ever overtaken by a larger population. Let N^* and G^* be the population size and the number of generations required by the GA to reach a certain target solution. Then, the number of fitness function evaluations needed by the GA to reach the target solution is given by $E^* = N^* \, G^*$. In the worst case, the parameter-less GA maintains all the populations up to the size N^* $(2, 4, 8, 16, \ldots, N^*)$. In the parameter-less GA, a population of size N is allowed to spend twice the number of fitness function evaluations than the number allocated to the population of size $2N$. Thus, in the worst case, the number of function evaluations spent by the parameter-less GA, E_{plGA}, is given by:

$$
\begin{aligned}
E_{plGA} &= E^* + 2E^* + 4E^* + 8E^* + \ldots + \frac{N^*}{2}E^* \\
&= E^*(1 + 2 + 4 + 8 + \ldots + \frac{N^*}{2}) \\
&\approx E^* N^*
\end{aligned}
$$

In summary, the worst case analyses says that the performance of the parameter-less GA will be at most N^* times slower than a GA that starts with an optimal population size. Pelikan and Lobo (1999) conducted a more detailed derivation than the one presented here, including the analyses for the case of using a general base b for the counter of the parameter-less GA. Indeed, they found that a base $b = 2$ yields the best result for the worst case scenario, resulting in a time complexity of the order $E^* \log_2 E^*$. This kind of analyses is useful because it gives estimates of the worst-case time complexity of the parameter-less GA. However, in actual practice, the worst-case scenario is very unlikely to occur, and as we will see in the next section, the actual performance of the parameter-less GA is usually within a constant factor of 2 or 3 of a GA that starts with optimal parameter settings.

4.4 Computer experiments

This section presents computer simulations on three test problems, and compares the performance of the parameter-less GA with a regular GA with "optimal" parameter settings for those problems. Both GAs are implemented as simple GAs. By simple GA, we mean a GA that uses a fixed coding and non-adaptive operators. Our simple GA implementation is generational, uses tournament selection without replacement, uniform crossover, and no mutation. For each problem, 20 independent runs were performed in order to get results with statistical significance. In order to compare the parameter-less GA with the regular GA, we run both algorithms until a specified target solution is achieved, and record the number of fitness function evaluations needed to do so.

The three test problems are carefully chosen to illustrate specific aspects of the parameter-less GA. The first problem is the onemax problem, also known as the counting ones problem. This is an easy problem for the GA, and we can use the existing population sizing theory in order to compare the parameter-less GA with a GA with "optimal" population size. The second problem, is the noisy onemax problem. We use this problem to illustrate the need of having multiple populations running simultaneously in order to overcome drift. The third problem is a bounded deceptive problem where the user has no knowledge about the location of the deceptive sub-problems or building blocks. For such a problem, the simple GA is unable to efficiently mix the building blocks, and thus the existing population sizing theory does not hold.

4.4.1 Onemax function

The onemax function simply returns the number of ones of an individual string. For testing purposes, a string length of 100 bits is used. The target solution is the string with all ones. The parameters of the regular GA are: tournament size 2, probability of crossover equal to 1, and population size of 100. These parameters were chosen because they are nearly optimal for this problem under a crossover-based GA. The population size of 100 was chosen based on the population sizing equation (Harik et al., 1997) shown below:

$$N = \frac{-2^k \sqrt{\pi m \sigma_{BB}^2}}{2d} \ln \alpha$$

The population size N depends on the building block size k, the number of building blocks m, the building block's fitness variance σ_{BB}^2, and the building block's fitness signal d. For a 100 bit onemax problem, these values are $k = 1$, $m = 100$, $\sigma_{BB}^2 = 0.25$, and $d = 1$. Plugging these values in the equation, we get:

$$N = -8.86 \ln \alpha$$

where α is the probability that the GA makes a mistake on a building block. Thus, the GA is expected to correctly solve a proportion of $1 - \alpha$ of the building blocks. In our

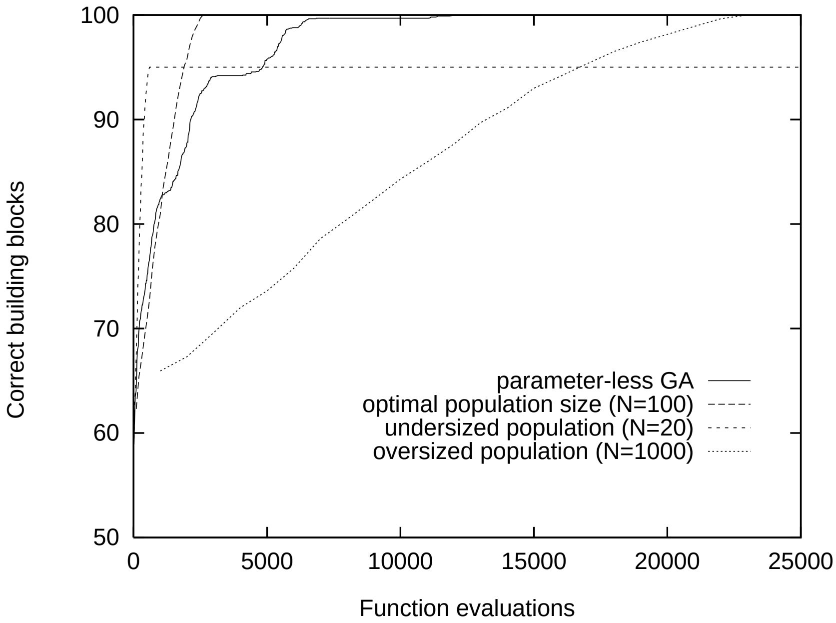


Figure 4.2: Solution quality versus number of function evaluations for the simple GA and its parameter-less version operating on a 100-bit onemax problem. Each line show the best solution quality found up to a given point in time.

case, we would like the GA to get the optimal solution in all the 20 runs. Theoretically, this can only be guaranteed if $\alpha = 0$, which would imply an infinite population size. For practical purposes, we use a population size of 100, which corresponds to a very small error probability ($\alpha \approx 0.00001$).

Figure 4.2 compares the regular GA with the parameter-less GA. The regular GA with optimal parameters takes an average of 2500 function evaluations in order to reach an optimal solution, while the parameter-less GA takes on average 7400 function evaluations. The figure also illustrates what happens in a hypothetical practical situation where the user does not know about building block sizes and noise-to-signal ratios. In such cases, the population sizing equation is not applicable and the user has to guess the population size. Experiments with populations of size 20 and 1000, corresponding to the type I and type II errors that were mentioned earlier in the chapter. With a population of size 20, the GA never reaches the desired target solution. With a population of size 1000, the GA gets to the target solution but it takes too long to do so.

By continuously increasing the population size, the parameter-less GA eliminates the need of guessing the right sizing. It tries a small population size first, and when it runs out of steam, a larger population will come by and take over the search. As expected, the GA with optimal parameter settings can reach the optimal solution faster than the parameter-less GA. But the parameter-less GA does not do badly at all. Without using any knowledge about the problem, the parameter-less GA is able to reach the optimal solution in not more than 3 times the time needed by the optimal GA. The population-sizing scheme is perhaps responsible for a factor of 2. The rest is due to the selection rate $s = 4$ and the crossover probability $p_c = 0.5$ used by the parameter-less GA. On an easy problem such as the onemax, a higher crossover rate and a less aggressive selection scheme, would achieve a higher solution quality faster. But this is a price that must be paid in order to have robust settings. That is, to have parameter settings that work more or less well on both easy and difficult problems. As an aside, notice that we are not doing any clever tricks counting the number of function evaluations. We are simply counting them as the product of population size by number of generations. In a careful implementation, the parameter-less GA could actually spend only half of that number simply by not re-evaluating the population members that don't undergo crossover.

4.4.2 Noisy onemax function

The next example illustrates the need for having multiple populations running simultaneously in the parameter-less GA. To do so, we test it on a noisy onemax problem. In this problem, the fitness f' of an individual is given by:

$$f' = f + noise$$

where f is the fitness of the problem in the absence of noise, and *noise* is a noise component. For practical purposes, the noise component can be simulation by sampling a Gaussian random variable with mean 0 and variance σ^2. We use a 100-bit string length, and a noise variance $\sigma^2 = 1000$. This value corresponds to a large amount of noise. Miller (1997) studied the effects of noise in the population sizing requirements of the GA, and extended Harik et al.'s equation to account for it. The resulting equation is:

$$N = \frac{-2^k \sqrt{\pi} \sqrt{\sigma_f^2 + \sigma_{noise}^2}}{2d} \ln \alpha$$

For the 100-bit onemax problem, the fitness variance without noise is $\sigma_f^2 = 25$. For a noise level of $\sigma_{noise}^2 = 1000$, the population sizing equation becomes:

$$N = -56.7 \ln \alpha$$

For the same level of accuracy that was used before ($\alpha = 0.00001$), we get a population size of 650. Figure 4.3 shows a comparison of the regular GA with optimal parameter settings ($N = 650$, $s = 2$, $p_c = 1$) with the parameter-less GA. The regular GA takes an average of 96000 function evaluations in order to discover an individual with 100 building blocks. To do the same thing, the parameter-less GA takes an average of 197000 function evaluations, a factor of 2. The figure also illustrates what happens in a common scenario of GA practice, a naive user sets the population size to 50 individuals.

Tracing the parameter-less GA, we observed that the smaller populations were not being run until convergence. Instead, they were often overtaken by larger populations. Due to the large amount of noise, there is only a very tiny selection pressure, and the population drifts. Notice that drift is not completely eliminated with the parameter-less

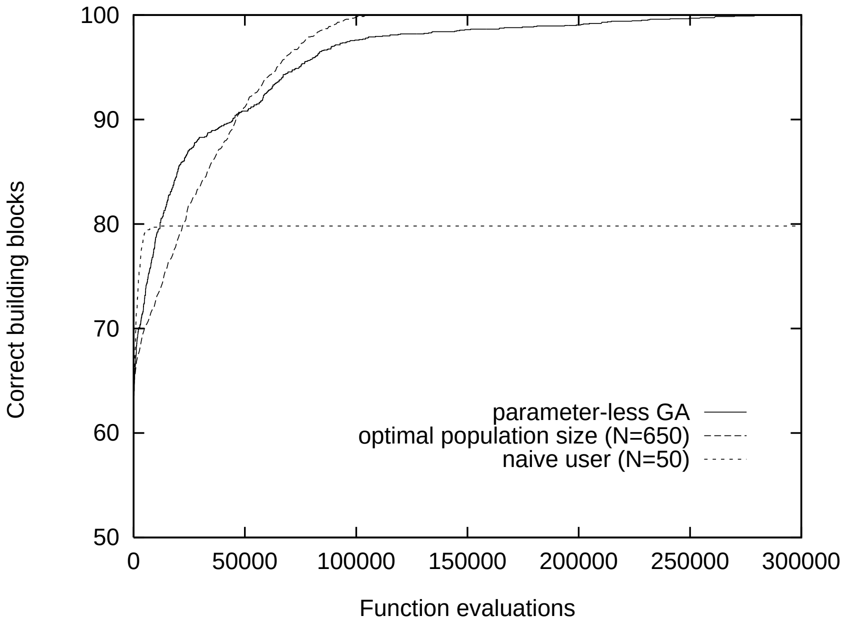


Figure 4.3: Solution quality versus number of function evaluations for the simple GA and its parameter-less version operating on a 100-bit noisy onemax problem. Each line show the best solution quality found up to a given point in time.

scheme. The populations still drift, but they don't do it too much because they are not run until convergence.

4.4.3 Boundedly deceptive function

The last test case illustrates a problem that is known to be difficult for a simple GA. The problem is the concatenation of 10 copies of a 4-bit trap function. A trap function is a function of unitation u, the number of ones in a 4-bit sub-string. It is defined as follows:

$$f(u) = \begin{cases} 3 - u, & \text{if } u < 4; \\ 4, & \text{if } u = 4. \end{cases}$$

The overall fitness function is the sum of the 10 independent sub-functions. Without previous knowledge about the location of each of the 10 sub-functions or building blocks, it is very hard for the simple GA to combine building blocks from different individuals and

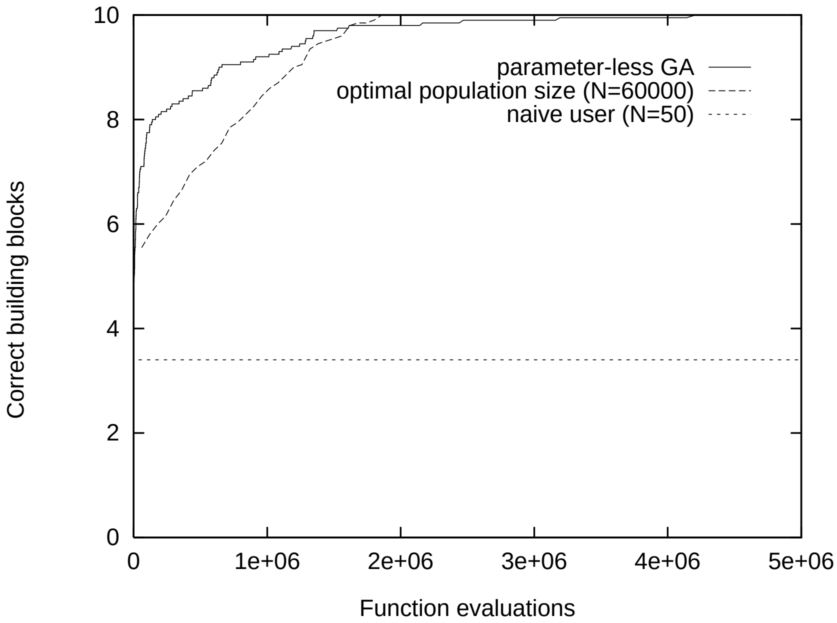


Figure 4.4: Solution quality versus number of function evaluations for the simple GA and its parameter-less version operating on a bounded deceptive function. Each line show the best solution quality found up to a given point in time.

assemble them in a single individual. In order to do that, the simple GA needs to obey the schema theorem, and in addition, it requires a very large population size (Thierens & Goldberg, 1993). For this case, the existing population sizing models are unable to predict accurate population sizes, because the assumption that the building blocks mix well does not hold. The desired target solution is the string with all 10 building blocks solved correctly. In order to obey the schema theorem, the regular GA uses a selection rate of $s = 4$ and a crossover probability $p_c = 0.5$. The minimum population size needed in order to solve all 20 runs to optimality was found empirically to be 60000. Figure 4.4 compares the GAs. The regular GA with optimal population size takes on average 1.5 million function evaluations to reach the target solution, and the parameter-less GA takes about the same number. Once again, the naive user pays a price in solution quality.

This test problem deserves a couple of remarks. The first one is the following. In order to reach the desired target solution, the GA needs millions of fitness function evaluations. In practical real-world problems, evaluating a single solution can take a substantial amount of time and the user might prefer to have an inferior solution quality quickly rather then waiting for a very long time to have a better solution. The parameter-less technique is good for this type of tradeoff situations because it does the best that it can assuming that the user can stop the optimization at any given point in time. The second remark is concerned with GA efficiency. The simple GA is quite inefficient to solve this problem because it is not powerful enough to do what GAs are supposed to do best—building block recombination. The real problem here lies in the crossover operator and has nothing to do with parameters. As mentioned earlier in the chapter, coding and operators is one thing, parameter setting is another thing. In the nest chapter we will see an advanced GA and its parameter-less version solving this exact same problem much faster.

4.5 Summary

This chapter presented a practical approach to eliminate the need for parameters in a crossover-based GA. We illustrated the operation and described how the parameter-less GA can be implemented. The parameter-less GA uses a crossover probability $p_c = 0.5$, and a selection rate $s = 4$ in order to obey the schema theorem, and not fall in the extreme cases of very high and very low selection pressures. At first, fixing the selection rate and the crossover probability for all problems, might look like an approach similar to the one used by those who adopt the so-called standard parameter settings. But in this case, the parameter settings are backed up by sound theoretical work, and constitute robust settings. It is possible to obtain better performance in some problems by tweaking both s and p_c. But with this work, we are not interested in peak performance. What we are really interested is in robustness and simplicity of use. Regarding the population size, the parameter-less GA tackles it by running multiple populations in a cascade-like manner, and getting rid of the ones that converge, and the ones that drift. For laboratory problems, the theoretical models on population sizing can predict the correct sizing of the population, but for an arbitrary real world problem they cannot. The technique presented in this chapter continuously increases the population size in an attempt to reach the right sizing, and it does so in an economical way; without spending too much computation resources.

With the parameter-less GA, the user does not have to guess the parameters of the GA, which oftentimes result in either a poor solution quality or in excessive waiting times. Of course, the parameter-less technique has a cost associated but it is a price that is worth its while. The parameter-less GA is a bit slower than a GA that starts with optimal parameter settings, but the truth is that nobody knows what are the optimal parameter settings for a real-world problem. The technique presented here just requires the user to press a start button. The algorithm runs forever, and does the best that it can until the user is happy with the solution quality, or has waited long enough and decides to press a stop button.

The use of the parameter-less GA is completely independent of the choice of GA implementations. This means that the technique can be used with any kind of crossover-based GA, and thus, we can have a parameter-less simple GA, a parameter-less messy GA, a parameter-less compact GA, and so on. The next chapter illustrates precisely this point by coupling the parameter-less technique with the extended compact genetic algorithm, one of the most powerful GAs that currently exist.

Chapter 5

Parameter-less technique with advanced genetic algorithms

5.1 Introduction

During the past 20-30 years, several variations on the so-called simple genetic algorithm originally proposed by Holland have been investigated and suggested. Among these modifications are a variety of new operators, not only general purpose ones, but also handcrafted operators specifically designed for a particular type of application. The motivation of all these GA variations is to improve the performance of the simple GA. Some of the approaches consist of designing special purpose operators on a problem by problem basis. Other approaches are more general and attempt to automatically learn gene linkage, thus relieving the user from having to come up with special purpose encoding schemes and special purpose operators. These advanced GAs are able to adapt the genetic operators so that they can efficiently process important similarities in the encoding.

This chapter reviews one of the most advanced linkage learning algorithms, the extended compact genetic algorithm (ECGA) invented by Harik (1999), and shows that the parameter-less technique can be used with it just as easily as it can be used with a simple GA. One of the important things about the parameter-less technique is that it is not tied up with a specific GA implementation; it can be used with any kind of GA. This feature is important because it makes the parameter-less approach broadly applicable. Moreover, as improvements are made in GA technology and more powerful algorithms are developed, the technique can be easily integrated to create parameter-less versions of these algorithms.

The chapter starts by reviewing the ECGA. After that, the parameter-less technique is coupled together with it, resulting in an algorithm that not only relieves the user from setting the GA parameters, it also relieves the user from choosing an appropriate coding and operators. Section 5.3 presents computer experiments comparing the performance of the parameter-less ECGA with that of the regular ECGA on a variety of test functions.

5.2 The extended compact genetic algorithm

The ECGA has an alternative view of the operation of the genetic algorithm where instead of generating individuals through pairwise recombination, it generates them by sampling from a probability distribution. The key thing is to learn a good distribution from which future individuals can be generated. Harik (1999) showed that a good probability distribution could be learned from the population itself, and that the whole approach is equivalent to learning linkage. The next section reviews in detail the basic principles of the ECGA.

5.2.1 Principles

The ECGA is based on the recognition that a search algorithm performs induction, that is, it has to guess where to move next based on its past experiences. In the context of GAs, the population represents a collection of points that summarize the past experiences of the algorithm. Based on this set of points, the GA has to guess where to go next. The way the GA typically does this guessing is by choosing another population through the actions of selection, crossover, and mutation. The individuals from the new population will be somehow similar (but different) from the best individuals in the current population. Several researchers observed that it was possible to use a probability distribution to help the GA where to move next. In other words, the probability distribution could be seen as a template to generate populations that are somehow similar (but different) from the best individuals in the current population.

Work on this area started with simple probability models that treated each gene independently. The work of Baluja and Caruana (1995) on population based incremental learning (PBIL), the univariate marginal distribution algorithm (UMDA) by Mühlenbein and Paaß (1996), and the compact GA by Harik, Lobo, and Goldberg (1998) are examples of these simple models, and are often referred to as order-1 models precisely because they treat each gene independently.

The drawback of these simple models is that the resulting distributions cannot represent dependencies among genes, and therefore, cannot effectively process higher-order building blocks. Harik (1999) noticed that it is possible to have distributions that can capture dependencies among genes, and extended the order-1 probabilistic models to more complex ones. Let's illustrate these probability models with an example. Consider a 4-bit problem with a population of 8 individuals as shown next:

1	0	0	0
1	1	0	1
0	1	1	1
1	1	0	0
0	0	1	0
0	1	1	1
1	0	0	0
1	0	0	1

Under an order-1 probabilistic model such as the one used in PBIL or the compact GA, the population can be represented by the following probability vector:

[1]		[2]		[3]		[4]	
1	5/8	1	4/8	1	3/8	1	4/8
0	3/8	0	4/8	0	5/8	0	4/8

These probabilities are the relative frequency counts of the number of 1's and 0's for the different gene positions. This representation treats each gene position independently.

Now, consider another probability model that treats gene 1 and gene 3 together, but gene 2 and gene 4 independently. Under such model, the population can be represented by:

[1,3]		[2]		[4]	
11	0	1	4/8	1	4/8
10	5/8	0	4/8	0	4/8
01	3/8				
00	0				

In this case, the genes are partitioned into $[1,3][2][4]$. Because genes 1 and 3 are treated jointly, the frequencies of the 4 possible outcomes $(11, 10, 01, 00)$ for the 2 bits values need to be computed. In the general case, for a subset of k genes, we need to compute 2^k frequency counts (in practice, only $2^k - 1$ frequency counts need to be computed, because the last one is determined automatically, since the sum of the frequencies is 1). By doing this, Harik showed that a population can be represented by a product of marginal distributions on a gene partition. He called these probability models *marginal product models* (MPMs). The probability model, or MPM, is like a template to generate populations that are somehow similar to the original one. Now comes the big question: "Which probability model is most likely to have generated this population?"

In the example presented, one would intuitively prefer the model that treats genes 1 and 3 jointly, rather than the model which treats each gene independently. Our intuition tends to prefer the first model because it is able to detect correlations between the two genes. These correlations are detected by observing that the frequency counts are far from being uniformly distributed. By inspection, we can see that whenever gene number 1 has value 1, gene number 3 has value 0. And whenever gene number 1 has value 0, gene number 3 has value 1. This is an indication that the two genes might be correlated. In the light of information theory, we say that the distribution of the outcomes $(11, 10, 01, 00)$ has a low *entropy*.

Entropy is a measure that tells the dispersion (or randomness) of a distribution. In information theory, the entropy of a distribution can be interpreted as the average number of bits needed to represent an element drawn at random from that distribution. It is defined as: $\sum -p_i \log_2(p_i)$, where p_i is the probability of observing the outcome i. The entropy is minimum (and equal to zero) when the distribution is maximally skewed, and it is maximum when all outcomes are equally likely to occur. As an example, consider the distribution of genes 1 and 3. The probabilities associated with the four outcomes $(11, 10, 01, 00)$ are $(0,\ 5/8,\ 3/8,\ 0)$, and the entropy of the distribution is:

$$-(5/8)\log_2(5/8) - (3/8)\log_2(3/8) \approx 0.954$$

This means that we would need on average 0.954 bits in order to encode an element from this distribution. Notice that if all four outcomes were equally likely, there would be no correlations (no patterns) to detect, and that would be reflected by a high entropy (2 bits in this case). Summarizing, the entropy measure can be used to detect patterns in data, and in the context of genetic algorithms, it can be used to detect building blocks, which are nothing but patterns in the population that are associated with high fitness.

Let's go back to the question of which probability model is most likely to have generated the population shown earlier in the text. Information theory gives an answer to this question: the most likely probability model is the one that uses the least amount of information to represent the population. The information needed to store a population include two things: the storage for the probability model, and the storage for the compressed population under the given model.

The storage for the probability model is the storage needed for all the frequency counts. In the previous example, model $[1,3][2][4]$ is more expensive than model $[1][2][3][4]$ because it needs to store 5 frequency counts instead of 4. However, under model $[1,3][2][4]$, the population can be better compressed than under model $[1][2][3][4]$ because the joint distribution of genes $[1,3]$ has a lower entropy than the sum of the entropies of the independent distributions of genes [1] and [3]. Overall, the model that is best, is the one that minimizes the sum of the storage needed for both the probability model, and the compressed population under the model. Let's formalize these two measures for the general case.

In a population of size N, a frequency count can only assume one out of $N+1$ possible distinct values $(0, 1, 2, \ldots, N)$, thus, each count can be stored using $\log_2(N+1)$ bits. If S_i is the size of the i^{th} subset of genes, then the storage needed to represent the probability model is given by:

$$Model\ Complexity = \log_2(N+1) \sum_i (2^{S_i} - 1)$$

and the storage for the compressed population under the model is given by:

$$Compressed\ Population\ Complexity = N \sum_i Entropy(M_i)$$

where M_i is the marginal distribution of the i^{th} subset of genes.

The *Combined Complexity* of a model is the sum of *Model Complexity* and *Compressed Population Complexity*, and it can be used to evaluate the efficacy of a particular probability model. The ECGA invented by Harik learns gene linkage by searching for a probability model that minimizes the *Combined Complexity* measure, i.e., by searching for a model that maximally compresses the population. Minimizing the combined complexity measure is nothing but the application of the so-called *minimum description length* (MDL) principle which is widely used in the machine learning literature. The algorithm consists of the following steps:

1. Create a random population of N individuals.

2. Apply selection

3. Model the population using a greedy MPM search.

4. Generate a new population according to the MPM found in step 3.

5. Return to step 2.

The third step is the only one that is different from a simple GA. The greedy MPM search is a steepest descent search based on the combined complexity measure. It starts by assuming that the MPM consisted by the ℓ subsets $[1][2]\ldots[\ell]$ is best. Then, it temporarily merges all possible pairs of subsets, and chooses the one that leads to a lower combined complexity. The search stops, when it is not possible to improve the combined complexity any further. Step 4 of the algorithm uses the resulting MPM to generate a new population of individuals. It is the equivalent of a smart crossover operator.

By incorporating concepts from information theory in the design of the ECGA, Harik showed that is possible for genetic algorithms to learn the structure of a problem, and to use that acquired knowledge to effectively process building blocks.

5.2.2 An example by hand

For completeness, let's illustrate the operation of the ECGA on the 8-member population shown at the beginning of the chapter. First, let's compute the combined complexity measure of the models $[1][2][3][4]$ and $[1,3][2][4]$.

Model: [1][2][3][4]

$$\textit{Model Complexity} = \log_2(9)(1+1+1+1) = 12.7$$

$$\textit{Compressed Population Complexity} = 8(0.954+1+0.954+1) = 31.3$$

$Combined\ Complexity = 12.7 + 31.3 = 44$

Model: [1,3][2][4]

$Model\ Complexity = \log_2(9)(3 + 1 + 1) = 15.8$

$Compressed\ Population\ Complexity = 8(0.954 + 1 + 1) = 23.6$

$Combined\ Complexity = 15.8 + 23.6 = 39.4$

Using model [1][2][3][4], the population can be encoded with 44 bits, while with model [1, 3][2][4] it can be encoded using 39.8 bits. Information theory tells us that we should prefer model [1, 3][2][4] because it is a shorter description of the population. Now, let's illustrate the MPM search for our toy example:

MPM search

The initial MPM is [1][2][3][4] and its combined complexity is 44. The next step is to merge all possible pair of subsets of the current MPM and compute their combined complexity measure as shown below:

MPM	Combined Complexity
[1, 2][3][4]	46.7
[1, 3][2][4]	39.8
[1, 4][2][3]	46.7
[1][2, 3][4]	46.7
[1][2, 4][3]	45.6
[1][2][3, 4]	46.7

The combined complexity of [1][2][3][4] can be improved. [1, 3][2][4] is the MPM that gives the lowest combined complexity, thus the search proceeds from [1, 3][2][4]. Again, all possible pair of subsets are merged:

MPM	Combined Complexity
$[1,3,2][4]$	48.6
$[1,3,4][2]$	48.6
$[1,3][2,4]$	41.4

At this point the MPM search stops because it is not possible to improve the combined complexity any further. Therefore, the model $[1,3][2][4]$ would be used to generate a new population of individuals.

5.3 Experiments with a parameter-less ECGA

This section presents computer experiments comparing the ECGA using optimal parameter settings with its parameter-less version. The test problems consist of additively decomposable subproblems, where each subproblem is a function of unitation that corresponds to a building block. A function of unitation depends only on the number of 1's in the chromosome string. The test problems are divided in the following three categories:

- uniformly scaled problems
- exponentially scaled problems
- noisy problems

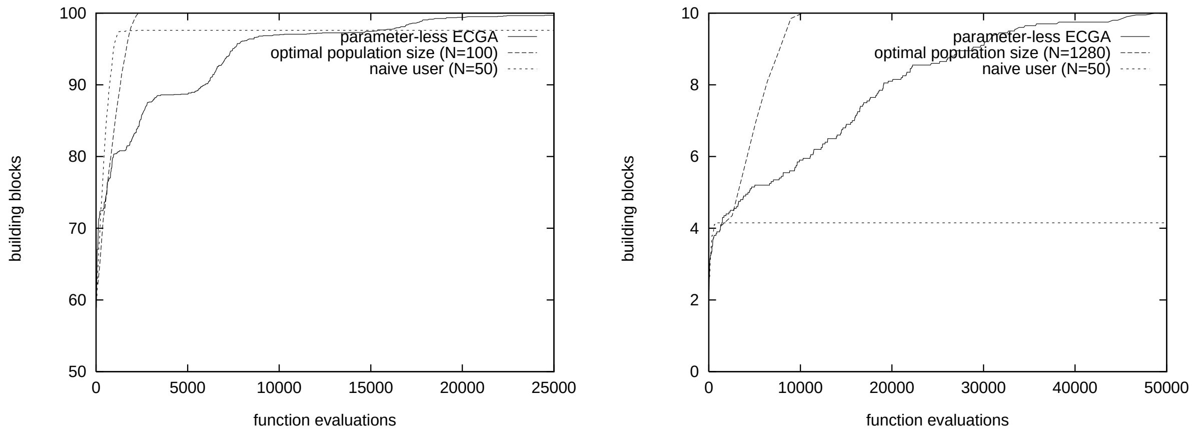
On uniformly scaled problems, the fitness contribution of each building block is the same. On exponentially scaled problems, the fitness contribution of each building block is scaled by some constant power. On noisy problems, the fitness of an individual string has a noise component and is non-deterministic, that is, the same individual can be evaluated to a different fitness value.

For each problem 20 independent runs are performed. In order to compare the parameter-less ECGA with the regular ECGA, we run both algorithms until a specified target solution is achieved and record the number of fitness function evaluations needed to do so.

5.3.1 Uniformly scaled problems

Uniformly scaled building blocks of size 1 (onemax)

The onemax function returns the number of ones of an individual string. We test the onemax function on a problem with 100 bits, and the target solution is the string with all ones. This corresponds to a problem with 100 building blocks of size 1. The ECGA with optimal parameter settings ($s = 2$, $p_c = 1$) needs a population size of 100 in order to reach the target solution on all 20 runs, and takes an average of 2200 fitness function evaluations to do so. To do the same thing, the parameter-less version takes an average of 22380 function evaluations, a factor of 10. The large difference between the regular ECGA and its parameter-less version occurs because the ECGA drifts on the onemax problem when the population size is small. The reason why this happens is not immediately obvious and deserves an explanation. When the population size is small, the GA makes decision-making mistakes in some building blocks. Thus the population might start to be filled with individuals of the form 1101111111. Other individuals might make mistakes in other bit positions, resulting in individuals of the form 1111011111. Now suppose that all the population members are either of the form 1101111111 or 1111011111. What happens is that on bit positions 3 and 5, there will be a lot of 01's and 10's, and the ECGA detects that the two bits are correlated and puts them together in the MPM partition. As a result



(a) 100-bit onemax

(b) 10 times 4-bit trap

Figure 5.1: Solution quality versus number of function evaluations for the ECGA and its parameter-less version operating on uniformly scaled problems. Each line shows the best solution quality found up to a given point in time.

the two bits will not mix, and because the strings have the same fitness, the ECGA drifts. Although the parameter-less technique is able to overcome drift, it still takes some time to do so, especially in this particular case because all smaller populations drift, making it a kind of worst case scenario for the parameter-less scheme.

Uniformly scaled trap functions

We use a 40-bit problem consisting of 10 building blocks, each a 4-bit trap function with signal 0.25. The target solution is the string with all the 10 building blocks solved correctly. The ECGA with optimal parameter settings ($s = 4$, $p_c = 0.5$) needs a population size of 1280 in order to reach the target solution on all 20 runs, and takes an average of 8960 fitness function evaluations to do so. To do the same thing, the parameter-less version takes an average of 28842 function evaluations. Figure 5.1 shows plots of the experiments on uniformly scaled problems.

5.3.2 Exponentially scaled problems

On exponentially scaled problems, the fitness contribution of each building block is scaled by a constant power. In our experiments we use a scale factor of 2. For example, in a 10 building block problem the fitness contribution of the building blocks are 1,2,4,8, ... ,512.

Exponentially scaled building blocks of size 1 (binary integer function)

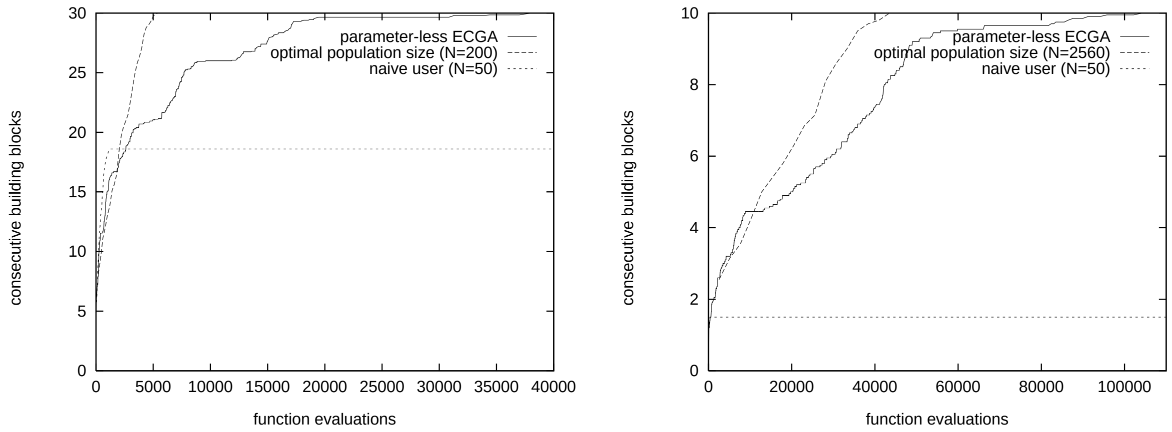
The binary integer problem corresponds to the scaled onemax problem with a scale factor of 2. The fitness of an individual string is its value interpreted as an unsigned binary integer. We use a string length of 30 bits, and the target solution is the string with all 30 bits solved correctly. The ECGA with optimal parameter settings ($s = 2$, $p_c = 1$) needs a population size of 200 in order to reach the target solution on all 20 runs, and takes an average of 4590 fitness function evaluations to do so. To do the same thing, the parameter-less version takes an average of 16994 function evaluations.

Exponentially scaled trap functions

This is the scaled version of the 10 copies of a 4-bit trap function. The subfunctions are also scaled by a factor of 2. The target solution is the string with all 10 subfunctions solved correctly. The ECGA with optimal parameter settings ($s = 4$, $p_c = 0.5$) needs a population size of 2560 in order to reach the target solution on all 20 runs, and takes an average of 36352 fitness function evaluations to do so. To do the same thing, the parameter-less version takes an average of 56456 function evaluations. Figure 5.2 shows plots of the experiments on exponentially scaled problems.

5.3.3 Noisy problems

On noisy problems, the fitness f' of an individual is computed as follows: $f' = f + noise$, where f is the fitness of the problem in the absence of noise, and $noise$ is a noise



(a) 30-bit binary integer problem

(b) 10 times 4-bit scaled trap

Figure 5.2: Solution quality versus number of function evaluations for the ECGA and its parameter-less version operating on exponentially scaled problems. Each line shows the best solution quality found up to a given point in time.

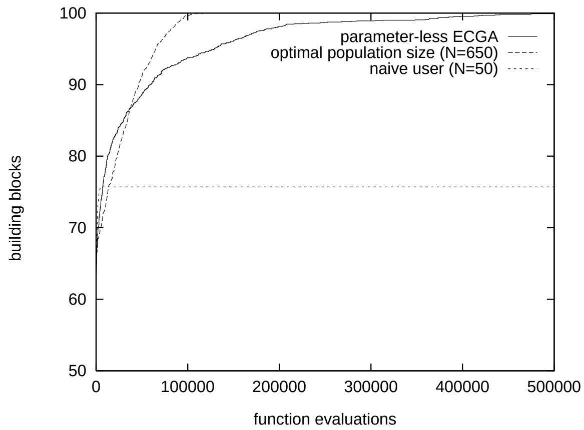
component. For practical purposes, the noise component can be simulated by tossing a Gaussian random variable with mean 0 and variance σ^2.

Noisy onemax

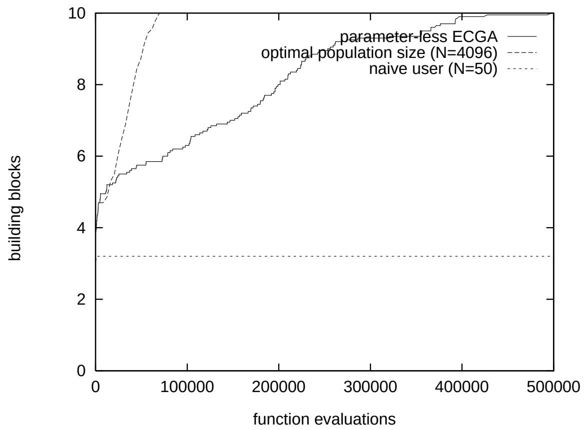
We test a noisy version of the 100-bit onemax problem. The noise component is a Gaussian random variable with mean 0 and variance 1000. The target solution is the string with all 100 bits solved correctly. The ECGA with optimal parameter settings ($s = 2$, $p_c = 1$) needs a population size of 650 in order to reach the target solution on all 20 runs, and takes an average of 95842 fitness function evaluations to do so. To do the same thing, the parameter-less version takes an average of 338538 function evaluations.

Noisy trap functions

We also test a noisy version of the concatenation of the 10 copies of a 4-bit uniformly scaled trap functions. The noise component is a Gaussian random variable with mean 0 and variance 5. The target solution is the string with all 10 subfunctions solved correctly. The ECGA with optimal parameter settings ($s = 4$, $p_c = 0.5$) needs a population size of



(a) 100-bit noisy onemax



(b) 10 times 4-bit noisy trap

Figure 5.3: Solution quality versus number of function evaluations for the ECGA and its parameter-less version operating on noisy problems. Each line shows the best solution quality found up to a given point in time.

4096 in order to reach the target solution on all 20 runs, and takes an average of 60211 fitness function evaluations to do so. To do the same thing, the parameter-less version takes an average of 323419 function evaluations. Figure 5.3 shows plots of the experiments on noisy problems.

5.3.4 Summary of experiments

Table 5.1 summarizes the results of all the experiments. It shows the average number of function evaluations needed by both algorithms in order to reach a specified target solution. The last column of the table indicates how much slower the parameter-less ECGA is compared to the regular ECGA with optimal parameter settings.

As expected, the parameter-less technique is slower than the ECGA with optimal settings, but the slowdown factor is usually small. The worst cases are for the noisy problems and are due to the occurrence of genetic drift. Although the parameter-less technique is able to overcome drift, it still needs some time to do so. As explained in section 5.3.1, the case of the onemax function is exceptional but we argue that similar situations are

Table 5.1: Number of function evaluations needed by the ECGA and its parameter-less version to reach the optimal solution.

Problem	ECGA	parameter-less ECGA	parameter-less ECGA is slower by a factor of
100-bit onemax	2200	22380	10.2
10 x 4-bit trap	8960	28842	3.2
30-bit scaled onemax	4590	16994	3.7
10 x 4-bit scaled trap	36352	56456	1.6
100-bit noisy onemax	95842	338538	3.5
10 x 4-bit noisy trap	60211	323419	5.4

unlikely to happen in real world problems. Functions of unitation are useful to test genetic algorithms because they provide a way of constructing problems with varying degree of difficulty. However, these functions are very symmetrical in the sense that changing 2 bits simultaneously oftentimes yields the same fitness value. The sources of drift in real-world problems are most likely to come from the existence of inherently noisy fitness functions, and not from the fact that fitness functions have somehow special symmetry properties.

5.4 Summary

This chapter presented an advanced genetic algorithm, the ECGA, and used it in conjunction with the parameter-less scheme, showing that the technique developed in the previous chapter is completely independent of the type of genetic algorithm to be used with. Computer experiments were conducted with uniformly scaled, exponentially scaled, and noisy problems. The ECGA alone is a powerful linkage learning algorithm that relieves the user from coming up with special purpose codings and operators. Coupled together with the parameter-less technique, results in an algorithm that in addition to all that, also relieves the user from having to set parameters. The next chapter illustrates how the techniques developed up to now in the dissertation can be transferred and applied to real world problems.

Chapter 6

From theory to practice: a case study using a simplified utility network expansion problem

6.1 Introduction

During the previous chapters we have seen some of the most advanced genetic algorithm techniques that currently exist. They are very recent and up to now have only been tested in artificially constructed problems. The goal of this chapter is to show how those techniques can be transferred to real-world problem, and in some sense, it is a guide for genetic algorithm practitioners. For illustration purposes we will be using a network expansion problem, something that utility companies are often faced with. The example is used as a case study but similar design principles may be applied to other kind of problems as well.

The chapter starts by explaining the differences between artificial and real world problems. It then moves on and describes a network expansion problem as an example of a real world problem, and shows how a genetic algorithm can be applied to it. Along the way, it illustrates how this problem differs from the artificial constructed problems often studied by GA researchers. Following that, the parameter-less technique is applied on 3 problem instances, both with the simple GA and with the ECGA. In either case the advantages of the parameter-less GA stand out. The chapter ends by defining an empirical measure of problem difficulty, something that can be used as a way to compare the difficulty of problems on a problem by problem basis.

6.2 From laboratory problems to real-world problems

Researchers usually test genetic algorithms on artificially problems, and that's precisely what we have done on chapters 4 and 5 when testing the parameter-less GA. Artificial problems are useful for testing GAs due to a number of reasons:

- We can create problems of different sizes.
- We can create problems with varying degrees of difficulty.
- We can study boundary cases of GA performance.

The ability to create problems of different sizes is important because it makes it possible to study how the GA scales up for larger and larger problems. It is also useful to study problems with varying degrees of difficulty. The difficulties can come through a variety of sources including deception, multi-modality, and noise. Another important characteristic of these problems is that it allows researchers to investigate boundary or extreme cases of GA performance. For example, the onemax problem can be seen as an extreme case, it is the easiest of all problems for the GA. But even though it is a very easy problem, it is useful to study how the GA performs on it, because it gives a lower bound for GA performance on other problems. That is, we shouldn't expect the GA to solve a given problem any faster than it does for a onemax problem. On the other end of the problem spectrum we have fully deceptive problems that no algorithm is expected to solve. But by bounding the deception to a certain level, we can create problems that are difficult in some sense, but still solvable by genetic algorithms.

Some researchers have questioned the usefulness of deception for testing GAs with the argument that most real-world problems do not have such properties. While that may be true in some cases, the important thing to note is that deception constitutes another extreme case for testing GA performance. If the GA performs well under these difficulties, we should expect that it will perform well on anything easier than that. Problems of bounded deception have been useful to test GAs because they introduce difficulties. These problems are generally composed of additively sub-functions, each corresponding to a building block. However, the difficulties are usually limited to the building block itself, and in some sense, these problems are still somewhat easy because there are usually no interactions among building blocks (although it is possible to create them).

Practical real-world problems are not as easy as the onemax problem and are not additively decomposable. But certainly they can have some level of deception due to the complex interactions among the decision variables. Before finishing this section we would like to stress out that it is very important to test the GA on these artificial problems. If that was not done, it wouldn't be possible to analyze the GA behavior, it wouldn't be possible to get a better understanding of GAs, and it wouldn't be possible to improve the GA technology. Nonetheless, it is also important to keep an eye on the applications side and see how the GA performs on problems that are not artificially constructed, and if possible, illustrate how the lessons learned from theory can be transferred to a practical context. That's precisely what we try to do in the remainder of the chapter.

6.3 The network expansion problem

This section describes a simplified version of a utility network expansion problem. For the exposition of the problem we focus on the particular case of an electrical network. Specialists in the area of electrical power distribution systems will recognize that the description given here is overly simplified and not very realistic. On the actual real world problem, there would be many more considerations and restrictions to take into account. Nonetheless, there are good reasons for considering a simplified version of the actual real problem because its simplification leads to a straightforward application of a genetic algorithm. Without further considerations, let's describe with an example what the network expansion problem is all about. Figure 6.1 shows a region that doesn't have electrical facilities, an hypothetical instance for the network expansion problem.

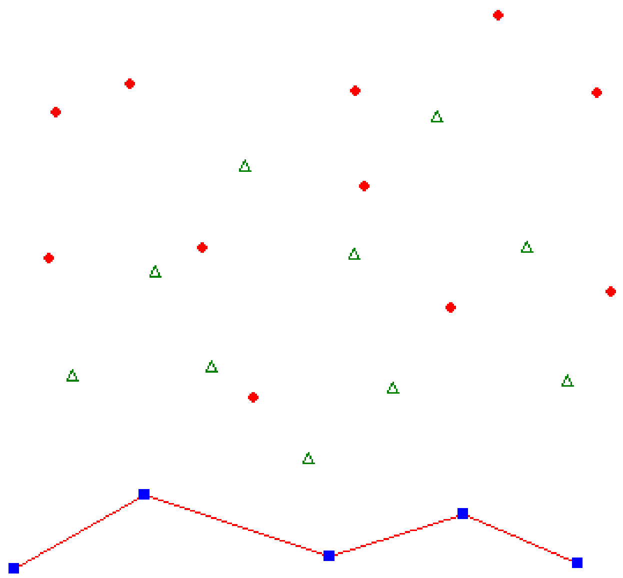


Figure 6.1: The substations, houses, and transformers, are represented by squares, circles, and triangles respectively. In the example there are 5 substations, 11 houses, and 10 possible locations to build transformers.

There are four types of entities depicted in the figure: cables, substations, possible transformer locations, and houses. These entities are represented in the figure by lines, squares, triangles, and circles respectively. In the example there are five substations (squares) connected by cables in a network. The objective of the problem is to expand the network so that the houses (circles) can get electricity. Moreover, the electrical utility company would like to do so by spending the minimum amount of money as possible.

The substations are the only entities that take part of the electrical infrastructure. The transformers (triangles) don't exist yet but there are a number of possible locations where they can be built. These locations are given by the electrical utility company and may take into account a variety of restrictions.

There are many ways in which the houses can be connected to the existing network. For example, a cable can be built to connect a house to one of the existing substation nodes. An alternative solution is to build a transformer and then build a cable from the house to the transformer. The transformer can then be connected to another transformer, and so on, but eventually one of the transformers has to connect to one of the existing substations. The end result is a tree, a graph with no cycles. Figure 6.5 shows a possible solution to the problem. The total cost of expanding the network is the sum of the costs of all the cables and transformers that need to be built. Each transformer that is built has a fixed cost associated and the cost of each cable is proportional to its length, the longer the cable the more expensive it is. The problem that the electrical company is faced with is to decide which cables and transformers should be built in order to give electricity to the houses using the minimum amount of money as possible.

Network problems are usually modeled using graphs. A graph is a mathematical structure that has nodes and edges connecting the nodes. The nodes and edges can have costs associated with them. In our example, the nodes are the houses, transformer locations, and substations. The edges are the electrical cables. There are a number of algorithms that can be performed on graph structures. Throughout the chapter we will sometimes use the graph terminology and use the terms nodes and edges.

The complexity of the problem is to decide which transformers should be built. Once that decision is taken, expanding the network is easy and can be done with a straightforward computation. Figures 6.1 through 6.5 illustrate the construction of the network once the transformers to be built are specified. Figure 6.1 shows the original problem. In figure 6.2, four transformers are selected to be built. They are represented on the figure by the large circles. In figure 6.3, we construct an artificial graph in the following way. For each selected transformer node, we add an edge from that node to all the existing substation nodes and to all the other selected transformers.

In figure 6.4, we find a minimum spanning tree of the graph. A spanning tree of a graph is a tree that contains all the nodes of the graph. A minimum spanning tree is a spanning tree with minimum cost. There are a variety of algorithms to find minimum spanning trees. One such algorithm is Prim's algorithm which can be found on a textbook on algorithms and data structures (see for example Cormen, Leiserson, & Rivest, 1993). Notice that for the minimum spanning tree computation, the edges connecting the substations have zero cost because these edges correspond to existing cables and no additional cost needs to be spent by the utility company. Once the tree is constructed, each house can be connected to the network by adding an edge from the house to the closest node of the tree. The final result is shown in figure 6.5.

In summary, once the decision of which transformers should be built is taken, the computation needed to expand the network is dominated by the construction of the nearly complete graph and its minimum spanning tree. This computation has a time complexity of $O(\ell^2)$, where ℓ is the number of transformer locations.

This problem has similarities with the a well known NP-complete problem from graph theory known as the *minimum steiner tree problem* (Garey & Johnson, 1979). The only difference is that in the minimum steiner tree problem, the possible location for the transformer nodes are not given in advance. There are a number of approximate algorithms and heuristics to solve the minimum steiner tree problem. In this chapter we don't make any comparison between the GA and these other heuristics. In fact, the GA won't take into consideration any special properties of the problem and it will treat it as a pure black-box function. It should be stressed out that the purpose of the chapter is not to show that the GA is superior or even competitive with specialized algorithms specifically tuned for this particular problem. Instead, the purpose of the chapter is to illustrate how GA technology may be applied in an environment where not much is known other than the objective function values of individual solutions.

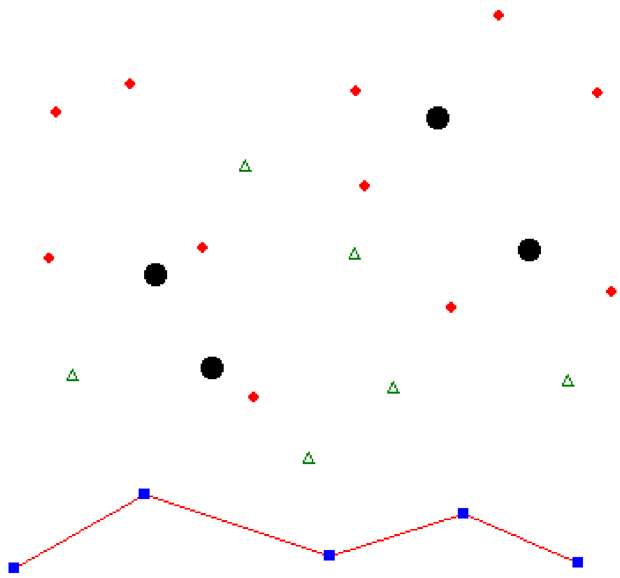


Figure 6.2: Four transformers are turned on. They are represented by the four large circles marked on top of the original transformer locations.

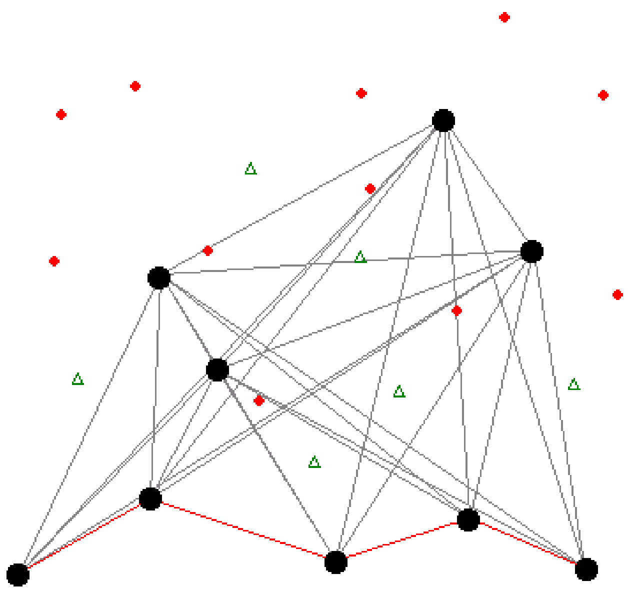


Figure 6.3: An artificial graph is constructed by adding edges from every selected transformer to all the existing substations and to all the other selected transformers.

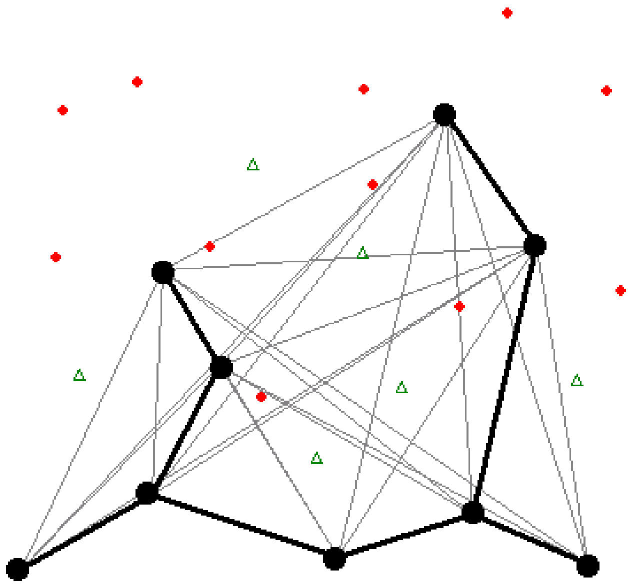


Figure 6.4: The minimum spanning tree of the graph is shown with a thick line.

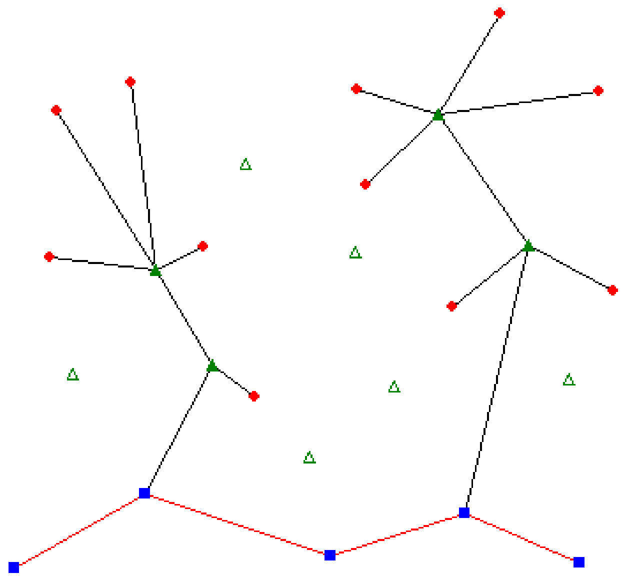


Figure 6.5: Each house connects to the closest node of the minimum spanning tree. Once the 4 transformers are specified this solution becomes the cheapest way to expand the network.

What we have just described is a method to expand the existing network once the transformers to be built are specified. The decision variables of the problem are binary variables, one for each possible transformer location, indicating whether that transformer should be built or not. Thus, if the electrical utility company specifies ℓ possible locations to build transformers, the total number of possible network configurations is 2^ℓ and the application of a genetic algorithm is straightforward. Notice that for a little problem like the one shown in the example there is no need for genetic algorithms at all because we could simply enumerate all the search space and output the best solution.

As mentioned previously, the network expansion problem that we just described is not very realistic in the sense that the actual real world problem would have to incorporate a number of restrictions and a lot of other considerations. From this perspective, we can say that the problem is still a toy problem. Nonetheless, it has several characteristics that contrast with those of the artificial problems that we talked about earlier in the chapter. Some of them are listed below:

- We don't know the global optima.
- We don't know the problem's structure.
- We don't know where the building blocks are.
- We don't know apriori what's a good coding/operator.
- We don't know if the problem is easy or hard.

On the other hand, it is easy to create problem instances of different sizes, and so it is possible to study how the GA scales up when the problem gets larger. In summary, it constitutes a nice example to illustrate the transition from laboratory problems to real-world problems. The next section describes the application of genetic algorithm technology to this problem.

6.4 Genetic algorithm application

This section shows the application of genetic algorithms to the network expansion problem just described. In particular, the parameter-less technique will be applied both with a simple GA and with the extended compact GA. In some sense, the goal of this section is to be a guide to GA application. We will also see how both GAs compare with a mutation-based evolutionary algorithm, and also with GAs using the so-called standard settings. The GA is tested on three different problem instances: a 20-bit problem, a 60-bit problem, and a 100-bit problem. Let's start with the 20-bit problem.

6.4.1 A 20-bit problem

Starting with a small instance is good to have a feeling of how difficult the problem is. For a 20-bit problem, the number of possible solutions is $2^{20} \approx 1$ million. For this particular fitness function, one million evaluations is not so much and can be done reasonably fast. We performed a complete enumeration of the search space and found the optimal solution. Knowing that is useful because we can now test the GA and see how much effort it needs to get to the same solution quality.

We did a single run of the parameter-less SGA and it found the optimal solution when it was running with a population size of 128, spending a total of 4640 function evaluations. We also run the parameter-less ECGA and it needed about the same time that the parameter-less SGA. The result of these experiments seem to indicate that this 20-bit problem instance is not too difficult; even a SGA can find the global optima quite fast.

6.4.2 A 60-bit problem

For a 60-bit problem, a complete enumeration of the search space is out of question. We try both the parameter-less SGA and ECGA, and let them run until the population of size 65536 converged. Table 6.1 shows the best solution quality obtained for each population size tried by the parameter-less scheme.

Table 6.1: The parameter-less GA on a 60-bit problem instance.

population size	solution quality with parameter-less SGA	solution quality with parameter-less ECGA
16	2131.79	2185.12
32	2109.33	2156.45
64	2004.93	2024.10
128	1990.25	2010.44
256	1971.50	1989.38
512	1984.76	1986.49
1024	1986.49	1984.76
2048	1967.21	1967.21
4096	1967.21	1967.21
8192	1967.21	1967.21
16384	1967.21	1967.21
32768	1967.21	1967.21
65536	1967.21	1967.21

Both the SGA and the ECGA found a solution quality of 1967.21 with a population of 2048 individuals. For larger populations, neither the SGA nor the ECGA could find a better solution and always converged to the same solution of 1967.21. We can speculate that the problem remains somewhat easy because the ECGA couldn't do better that the SGA, even with very large population sizes. In conclusion, either the problem is easy or is profoundly deceptive.

Table 6.1 shows the result of a single run. To get statistical significance, we conducted 20 independent runs and recorded the number of fitness function evaluations needed by the parameter-less GA to reach the solution quality of 1967.21. Table 6.2 shows the average and standard deviation of the number of evaluations needed. In both cases, the population size required by the SGA and the ECGA to solve the problem was in the range of 512 to 4096, with the majority of the runs needing a population size of 2048.

Table 6.2: Average and standard deviation of the number of function evaluations needed to reach the target solution of 1967.21.

	parameter-less SGA	parameter-less ECGA
average	161,200	109,400
std. dev.	144,300	78,200

To get some insight in how difficult the problem really is, we run a $(1+1)$ algorithm with a fixed mutation rate of $1/\ell$. If the problem is really easy, a $(1+1)$ algorithm should get the desired target solution faster than the GA. We did 10 independent runs and on 4 of them, the mutation-based algorithm reached the desired target solution quite fast (1K, 27K, 34K, 5K function evaluations). But on the other 6 runs it got stuck in a solution that is 4 bits away from the desired target solution. For a $(1+1)$ algorithm, we can calculate the probability that a local optimal solution jumps to a better solution that is k bits away from it. The probability of such an event is equal to the probability of flipping exactly those k bits and not flipping the remaining $\ell - k$ bits. For a mutation rate m, that probability is:

$$m^k \; (1-m)^{\ell-k}$$

For a mutation rate of $m = 1/\ell$, the success probability becomes:

$$\left(\frac{1}{\ell}\right)^k \left(1 - \frac{1}{\ell}\right)^{\ell-k}$$

The expected number of trials T to have an improvement is the inverse of the success probability:

$$T = \frac{1}{\left(\frac{1}{\ell}\right)^k \left(1 - \frac{1}{\ell}\right)^{\ell - k}} \approx \ell^k \ e$$

This calculation is similar to the one done by Mühlenbein (1992). For a 60-bit problem, the expected number of trials (function evaluations) to jump to a solution that is k bits away is: $60^k \ e$. Table 6.3 shows the expected times for $k = 1, 2, 3, 4$.

Table 6.3: Expected number of function evaluations to jump to a solution that is k-bits away.

k	expected number of trials for an improvement
1	163
2	9,786
3	587,149
4	35,228,932

For the 6 examples that the $(1 + 1)$-algorithm got stuck, we let it run for 5 million function evaluations. After such an amount of time the algorithm still couldn't get to the target solution, a strong indication that to reach the target solution, the $(1 + 1)$ has to do a 4-bit jump, something that will take on average 35 million function evaluations.

This set of experiments with the mutation-based algorithm indicates that the problem is not so easy. There are different paths that lead to the target solution, but many of them lead to solutions that are surrounded by deep valleys that are hard to overcome by a fixed mutation-based algorithm. Although in some cases we can get lucky, in others we get trapped in undesirable solutions. Notice that both the parameter-less SGA and ECGA found the desired solution reliably and with less than a couple thousand function evaluations. Contrary to the GAs, the mutation-based approach shows a very high variance. It can solve the problem very fast sometimes, but other times it takes a very long time.

In another experiment we ran a simple GA with a very large population size but this time with the selection rate $s = 2$ and $p_c = 1$. The SGA was unable to get to the target solution even with a population size of 65536. Again, this is an indication that the problem is not so easy. The parameter-less SGA uses always $s = 4$ and $p_c = 0.5$, and with such parameters, the SGA can solve bounded deceptive problems.

We can speculate that this problem is not too easy but also not too difficult, perhaps it is the equivalent of a problem with a small level of deception. Notice that the SGA with $s = 4$ and $p_c = 0.5$ could also solve bounded deceptive problems because these parameters obey the schema theorem and so the building blocks are expected to grow. The drawback of the SGA is that if the level of deception is not very small, or if the problem is large, then the SGA doesn't scale well. In fact, it requires exponentially larger population sizes as the problem length increases (Thierens & Goldberg, 1993).

6.4.3 A 100-bit problem

The final problem instance is a 100-bit problem. Once again, we try both the parameter-less SGA and ECGA, and let both algorithms run until the population of size 65536 converged. Table 6.4 shows the best solution quality obtained for each population size tried by the parameter-less scheme during a single run.

Both the SGA and the ECGA found a solution quality of 2518.17. The SGA needed a population size of 4096 and the ECGA needed a population size of 1024. For larger populations, neither the SGA nor the ECGA could find a better solution and always converged to the same solution of 2518.17.

Table 6.5 shows the average and standard deviation of the number of evaluations

Table 6.4: The parameter-less GA on a 100-bit problem instance.

population size	solution quality with parameter-less SGA	solution quality with parameter-less ECGA
16	2979.35	2938.68
32	2788.27	2818.21
64	2615.63	2691.68
128	2572.58	2591.16
256	2543.66	2555.07
512	2535.13	2543.70
1024	2530.29	<u>2518.17</u>
2048	2530.29	2518.17
4096	<u>2518.17</u>	2518.17
8192	2518.17	2518.17
16384	2518.17	2518.17
32768	2518.17	2518.17
65536	2518.17	2518.17

needed to reach the solution quality of 2518.17. In both cases, the population size required by the SGA and the ECGA to solve the problem was in the range of 512 to 4096. The majority of times the SGA required a population size of 4096, while the ECGA required 2048 individuals on most of the runs.

Table 6.5: Average and standard deviation of the number of function evaluations needed to reach the target solution of 2518.17.

	parameter-less SGA	parameter-less ECGA
average	226,300	124,300
std. dev.	131,400	67,300

Once more, the results seem to indicate that the problem is not too difficult. Although the ECGA appears to be faster than the SGA, the variance of the results is high and therefore we cannot really say that one is better than the other. Again, we run a $(1 + 1)$ algorithm with a fixed mutation rate of $1/\ell$. We did 10 independent runs and on 3 of them, the mutation-based algorithm reached the desired target solution quite fast (14K, 3K, 67K function evaluations). On the other 7 runs it got stuck in solutions that were far away in hamming distance from the desired target solution. Out of these 7 runs, the algorithm got stuck 3 times on a solution that is 5 bits away from the target, 1 time in a solution that is 6 bits away, 1 time in a solution that is 7 bits away, and 1 time in a solution that is 8 bits away. Notice that a solution that is 8-bits away does not mean that the algorithm would has to make an 8-bit jump because there might be some intermediate solutions in the path to the target. For all the runs we let the algorithm do 5 million function evaluations. A similar calculation that we did for our previous 60-bit problem yields the following expected number of function evaluations to jump to a solution that is k bits away:

Table 6.6: Expected number of function evaluations to jump to a solution that is k-bits away for a 100-bit problem.

k	expected number of trials
1	272
2	27,183
3	2,718,281
4	271,828,100

The conclusion that can be drawn is that the local optimal solutions need at least a 4-bit change to improve, but that requires on average 271 million function evaluations!

In another experiment we run a simple GA with a very large population size but this time with the selection rate $s = 2$ and $p_c = 1$. The SGA was unable to get to the target solution even with a population size of 65536. Again, this is an indication that the problem is not too easy.

Another experiment that was done was to run the SGA with the so-called standard settings: population size of 100, crossover probability of 0.7 and probability of mutation of $1/\ell$. The SGA couldn't reach the target solution not even after a million function evaluations. What happens is that because the population size is not big enough, the GA

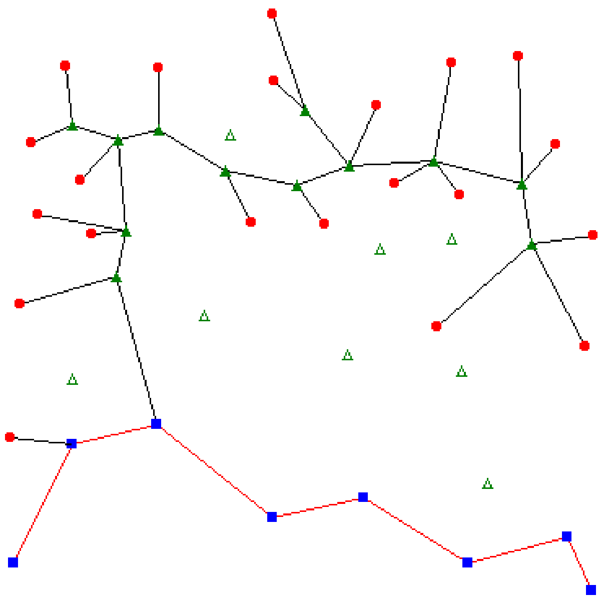


Figure 6.6: The best solution of a 20-bit problem instance.

runs out of diversity quite soon, and after some time, the only source of diversity is the mutation operator. Unfortunately, mutation is too slow when it needs to discover several bits all at once.

Figures 6.6, 6.7, and 6.8, show the best solutions found by the parameter-less GA on the problem instances of 20, 60, and 100 bits.

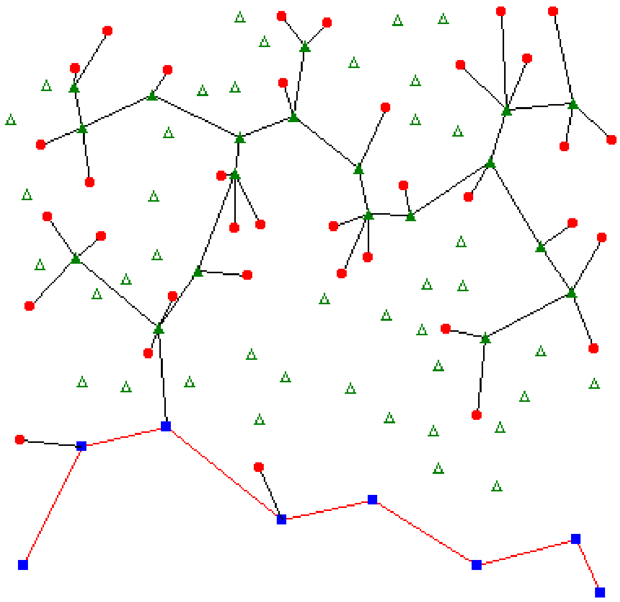


Figure 6.7: The best solution found by the parameter-less GA on a 60-bit problem instance.

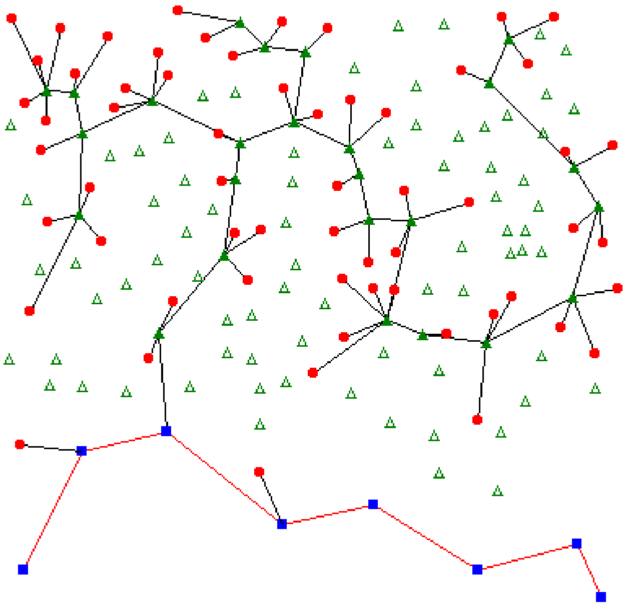


Figure 6.8: The best solution found by the parameter-less GA on a 100-bit problem instance.

6.5 Towards an empirical measure of problem difficulty

The experimental work performed in this chapter showed that the network expansion problem is neither too easy nor too hard, but we don't know how to quantify how easy or hard the problem really is. There are several sources of problem difficulty. The existence of deep building blocks that require linkage learning, bad building block scaling, and noisy fitness functions, are some of the things that can also cause difficulty to the GA.

One thing that can give an overall measure of a problem's difficulty is the amount of effort that is required to solve a given problem. In GAs, this amount of effort can be measured by the minimum population size needed by the parameter-less SGA using uniform crossover to get reliably to some desired target solution. We could go further and define an empirical measure of a problem's difficulty, call it R_{hard}, which can be defined as the minimum population size needed for solving the given problem up to a certain solution quality divided by the minimum population size needed by to the solve a onemax problem of the same size. The onemax problem is chosen as a reference because it is a problem that has been studied theoretically and it is possible to use the population sizing models to predict the minimum population size needed to solve it given a specified error probability. Moreover, the onemax problem is a good reference for comparing the difficulty of problems because in some sense it is the easiest of all problems for the GA. Its building blocks are trivial, are of the same scale, and the function evaluation is not noisy.

For finding the minimum population size needed to solve a problem we could run the parameter-less SGA using uniform crossover. Since the parameter-less GA uses $s = 4$ and $P_c = 0.5$, it should be able to solve all problems given a large enough population size. Let $\hat{n}$ be the minimum population size needed to solve a problem to a certain solution quality, and let n_{onemax} be the minimum population size needed to solve a onemax problem of the same length. The *hardness ratio*, R_{hard}, can be defined as follows:

$$R_{hard} = \frac{\hat{n}}{n_{onemax}}$$

This definition is an empirical measure of a problem's difficulty. A ratio $R_{hard} = 1$ would indicate that the problem is as difficult as the onemax. A ratio $R_{hard} = r$ would indicate that the problem is r times more difficult than its onemax counterpart. Although this empirical measure of problem difficulty can only be estimated after the problem itself is solved, it is a practical measure that can be used to compare the hardness of problems. Thus, when doing GA applications, users can talk about a real-world problem's difficulty and say things like: problem X is $R_{hard} = 10$, problem Y is $R_{hard} = 2000$, and so on.

More specific measures of GA difficulty could be defined as well. For example, we could define a mixing hardness ratio, call it $R_{mixingHard}$, defined as the population size needed by a simple GA divided by the population size needed by a linkage learning GA (such as the ECGA) to reach a certain target solution.

$$R_{mixingHard} = \frac{\hat{n}_{SGA}}{\hat{n}_{ECGA}}$$

Given this measure, it is also be possible to compare the mixing hardness of problems and say things like: problem X is $R_{mixingHard} = 3$, problem Y is $R_{mixingHard} = 1000$, and so on.

6.6 Summary

This chapter showed how the methods presented in chapters 4 and 5 can be applied in a simplified example of a utility network expansion problem. Although it is still a toy problem, it has several characteristics that contrast with the artificial problems that are often used to test and compare genetic algorithm techniques. The network expansion problem is closer to the type of problems that users are likely to encounter when doing real-world GA applications.

For the particular problem instances that we have seen in the chapter, the benefits of the parameter-less technique were strong. For both the 60-bit and the 100-bit problem, the GA needed a population size of around 2000 individuals to get to the desired target solution. In current practices, it is very unlikely that a user guesses 2000 as the right population size. Instead, with the parameter-less GA, the user just presses a button and observes the solutions given by the GA. If he can afford to wait until the parameter-less scheme reaches adequate population sizes, he will get a solution quality that wouldn't be possible to get with the standard parameter settings.

While the benefits of the parameter-less technique were quite strong, the same cannot be said regarding the use of the ECGA. It's true that the ECGA didn't do any worse than the SGA, but it also didn't do any better. They exhibited the same kind of performance, an indication that the problem is not too difficult and that the need of linkage learning is not really essential for the particular problem instances presented herein. Nonetheless, for larger problem instances it is possible that the benefits of linkage learning show up. It is also important to mention that although there is no direct benefits from the ECGA for these particular cases, there is some kind of indirect advantage for using it. An advanced linkage learning GA gives the user a kind of insurance that the solution quality obtained is really good. A user should feel more confident using a linkage learning algorithm because he knows that if the problem has deep building blocks, the algorithm would be able to find it, but the SGA would not. If we only had the SGA we might wonder whether the encoding was good or not. By using the ECGA and observing that there was no real advantages, we can be somewhat confident that even with other encoding or other crossover operators, the SGA would not be able to do any better.

Notice also that if the problem was really difficult, the advantages of the ECGA would show up, precisely like in the cases of the bounded deceptive problems that were illustrated in chapters 4 and 5. There, the parameter-less ECGA was faster than its SGA counterpart by several orders of magnitude.

The experiments with the $(1+1)$ algorithm gave an indication that there are different paths that can lead to a good solution, some are easy and require only a couple of bit changes from one solution to the other, but other paths are not so easy and may require that several bits need to be mutated simultaneously while leaving the remaining bits alone. In order to do that reliably it requires a substantial amount of time.

Below, we present a list of what is likely to occur in practice when solving problems. The categorization includes easy problems, hard problems, and problems that are not too easy but not hard either.

Easy problems. For this category a mutation-based approach is likely to be the fastest scheme and can reliably solve a problem in $O(\ell \log \ell)$ (Mühlenbein, 1992). A crossover-based scheme, either with or without linkage learning should exhibit a similar time complexity, but with a higher constant factor.

Hard problems. A mutation-based approach requires $O(\ell^k)$, where k is the maximum level of deception of the problem (Mühlenbein, 1992). The SGA has an exponential time complexity (Thierens & Goldberg, 1993), but a linkage learning GA reliably solves these problems in no more than $O(\ell^2)$ (Harik et al., 1997), (see also appendix of this dissertation).

Not too easy and not too hard problems. The network problem that we have seen in this chapter is an example of a problem that is not too easy but not too hard either. I believe the majority of real-world problems fall under this category. The time complexities for this case will be something in between the easy and hard cases. However, a fixed mutation-based scheme may not be very reliable, sometimes it can solve the problem fast, but other times not. Coupled with the parameter-less scheme, a crossover-based GA appears to be a more reliable method.

Chapter 7

Summary, extensions, recommendations, and conclusions

This chapter summarizes the work contained in the dissertation, lists the major contributions, outlines a number of topics that deserve additional research, gives practical recommendations for the genetic algorithm user, and finally concludes by stating how the state of knowledge has changed with the work presented herein.

7.1 Summary

This dissertation started by pointing out to some of the pitfalls of current genetic algorithm practice. Specifically, it looked at GAs from an application perspective and realized that users of genetic algorithms usually have to do a lot of tuning with codings, operators, and parameters in order to have success with the GA. We then speculated that GAs should become more user-friendly, and argued that there is a gap between the theory and practice of GA. The gap exists partly because some of the theoretical advances are not easily transferable to a practical context.

We then focused on the topic of parameter setting, and developed an algorithm that took the task of setting critical parameters of the GA away from the user. These parameters were set either automatically (population size) or rationally (selection rate and crossover probability) by the algorithm itself based on solid theoretical foundations. The resulting parameter-less GA was implemented and tested on a number of carefully chosen problems. One of the main advantages of the technique is that it relieved the user from having to guess the parameters of the GA, which oftentimes result in poor performance. Another advantage is that it greatly simplifies genetic algorithm usage; pressing a start and a stop button is all that the user needs to do.

The parameter-less technique was also shown to be a very general one, as it can be applied with different types of GAs. Chapter 5 showed that it could be combined with a linkage learning GA, resulting in an algorithm that not only relieves the user from setting the GA parameters, it also relieves the user from coming up with special purpose codings and operators.

Finally, the techniques developed herein were applied in a simplified version of a utility network expansion problem. Indeed, the application chapter can be seen as a guide to help genetic algorithm practitioners to apply the tools explored in this dissertation in their problem areas.

7.2 Achievements

This dissertation made the following contributions.

- **Developed a selecto-recombinative GA that has no parameters.** With the resulting algorithm, the operation of the GA is simplified because the user does not have to do trial and error experiments to find adequate parameter settings for the GA. Instead, these parameters are set automatically by the algorithm itself.

- **Showed how the technique can be applied in a real world problem.** The application chapter showed the application of the parameter-less GA in a quasi real world problem. The technique was used together with a simple GA and with an advanced linkage learning GA. This was the first time that these techniques were applied in non-laboratory problems. In addition, a comparison was made with previous standard GA techniques. The chapter is useful for practitioners seeking to apply state-of-the-art GA technology.

- **Defined an empirical measure of problem difficulty.** This measure can be useful to compare the difficulty of real world problems on a problem by problem basis.

- **Analyzed the time complexity of GAs on exponentially scaled problems.** A theoretical model was build to explain the dynamics of genetic algorithms on problems with exponentially scaled building blocks (see appendix A). It is important to study the performance of GAs on these type of problems because badly scaled building blocks is one of the things that can cause difficulty to GAs. The model showed that even in the case of exponentially scaled building blocks, the time complexity of genetic algorithms is no worse than quadratic.

7.3 Future research

A number of topics deserve further research and some of them are outlined next.

- **Integration of mutation in the parameter-less GA.** The parameter-less technique that was explored in this dissertation ignored mutation completely. Thus in some sense, it is more accurate to talk about a parameter-less crossover-based GA.

As we have mentioned during chapter 4, there has been extensive debates in the evolutionary computation community regarding the usefulness of crossover versus mutation. Mutation and crossover explore the search space in a different ways. Mutation relies on the principle of locality that says that solutions that are near good solutions are likely to be good as well. Therefore, mutation-based algorithms do local perturbations on good solutions in order to explore new points. Crossover on the other hand, has a less local perspective and its power comes from combining things from past (good) solutions. Due to the different way in which these operators explore the search space, different techniques appear to be more appropriate for one method or the other. Because mutation does local perturbations, it tends to work well with small population sizes. Crossover on the other hand, relies on the population to do effective recombination.

- **Model the population using a mutation-like operator.** The work on probability model-based GAs which started with Baluja and Caruana's Population Based Incremental Learning (PBIL) and continued through the Univariate Marginal Distribution Algorithm (UMDA) (Mühlenbein & Paaß, 1996), the Compact GA (Harik, Lobo, & Goldberg, 1998), the Extended Compact GA (ECGA) (Harik, 1999), and the Bayesian Optimization Algorithm (BOA) (Pelikan, Goldberg, & Cantú-Paz, 1999) have shown that it is possible to mimic the crossover operator by using probability distributions. A similar approach could also be done to mimic the effects of mutation. Such an approach has been suggested (Harik 1999, personal communication). With both a mutation and crossover model in hand, the choice of which operator/model to choose could be decided based on a measure that indicates which one is the more likely to have generated a similar population. Such a measure could be based in information theory concepts such as entropy and the minimum description length principle, just like we have seen in the ECGA. This approach could stop the endless debates of weather crossover is better than mutation and vice-versa. The choice of one or the other could be based on information theory, and the result would be an algorithm that would automatically choose the operators on a problem by problem basis. Preliminary investigations in this topic have already begun and will be reported soon.

- **Extend probabilistic model based GAs to other codes.** Most of the work on probability model based GAs have been done with discrete codes. However, many real-world problems have continuous decision variables. Therefore it is important to develop extensions of these methods to handle real-coded variables. Likewise, the encoding of solutions for many problems consist of permutations. It would also be important to investigate extensions for this case. The result would be the powerful linkage learning algorithms for discrete, real, and permutation codes.

- **Parallelization.** There are guidelines for doing efficient parallelization of traditional genetic algorithms (Cantú-Paz, 1999). However, it is not clear how those guidelines can be transferred to the probability model based GAs. This is very important, because probability model based GAs such as the ECGA and BOA, appear to be the most powerful GAs. It is very important to investigate how the parallelization of these algorithms differs from the case of the traditional simple GAs. Likewise, it is important to investigate how to parallelize the parameter-less technique explored in this dissertation.

- **Do more applications.** Last but not the least, it is important to do more real-world applications. The GA techniques that have been discussed in this dissertation are in their early days. In the laboratory, there is a strong evidence that these techniques are much more powerful than the traditional GAs, but the evidence has to move from laboratory problems to real-world problems. The application spectrum is wide, and in the specific case of environmental engineering, they can be applied in pipe network optimization (Simpson, Dandy, & Murphy, 1994), facility location (Pereira, 1998), pollution source apportionment (Cartwright, 1997), and ecological design of products (Shu, Wallace, & Flowers, 1996), just to name a few.

7.4 Recommendations

Below is a list of recommendations for the genetic algorithm practitioner.

- **Use the parameter-less technique.** There is no reason for future genetic algorithm applications not to use the parameter-less GA. Not only that, current applications of GAs can benefit with the incorporation of the parameter-less scheme. It will not be surprising if applications adopting the parameter-less GA start delivering superior performance when compared to the traditional standard parameter setting techniques.

- **Use the guidelines of chapter 6.** As a practical recommendation for GA applications, we suggest the usage of an advanced linkage learning GA coupled together with the parameter-less technique. Currently, the ECGA (Harik, 1999) and the BOA (Pelikan, Goldberg, & Cantú-Paz, 1999) are the most advanced and reliable linkage learning algorithms, and either one can be used. The only drawback of these linkage learning algorithms is that the operation of the algorithms themselves involves a substantial amount of time. Therefore, when solving very large problems, and if time is really crucial, users may want to sacrifice on solution quality and instead get a rougher solution by using a simpler method such as a parameter-less simple GA or even a fixed mutation-based scheme.

7.5 Conclusions

We finish the chapter by saying how the state of knowledge has changed after this dissertation.

- **GA parameters can be eliminated or adapted.** This was the original thesis or goal statement of the dissertation, and the work presented herein supports, at least partially, the thesis. The only reason why it was not fully achieved is because the mutation operator and its corresponding parameter was ignored. The integration of mutation in the parameter-less GA is a topic that deserves further research as explained already in the future work section. Nonetheless, an important conclusion of this dissertation is that parameters can be eliminated or adapted in a systematic way for GAs of fixed structure.

- **GAs are more economic.** In a recent paper, Goldberg (1999) argued that one of the motives for using GAs in real world applications comes from economics, and in particular, from the economics in method investment. His argument is more or less the following. Instead of mastering a number of specialized optimization techniques which oftentimes have a narrow domain of application, the user might as well master a single technique, the GA, which is a competent method that is applicable to almost any kind of problem. By doing this, engineering costs can be cut down simply by eliminating the need of a specialist in a variety of optimization techniques. His argument makes sense but I would go one step further and eliminate the need of the GA specialist as well. There are not many GA specialist around the world and hiring one of them is certainly expensive. It would be much more economical if the GA could be blindly applied without the need of the GA expert. Only then, we are likely to observe a wide dissemination of the GA in real world applications. The parameter-less technique presented in this dissertation emphasizes the economic nature of GAs.

- **Poor GA performance less likely to occur.** With the parameter-less technique users are less likely to observe poor performance. At least, if the GA does a poor job, it certainly won't be due to poor parameter settings. Up to this date, users had to guess the parameters of the GA but oftentimes they did it wrong. The outcome is either an excessive waste of computation resources or solutions that are not as good as they could be.

Appendix A

Time complexity of genetic algorithms on exponentially scaled problems

A.1 Introduction

This appendix gives a theoretical and empirical analysis of the time complexity of genetic algorithms on problems with exponentially scaled building blocks. Although this research differs slightly from the main flow of the dissertation, it does have important connections with what we have seen so far because it is related to population sizing and run duration.

GA performance is usually measured by the number of fitness function evaluations done during the course of a run. For fixed population sizes, the usual case in GA implementations, the number of fitness function evaluations is given by the product of population size by the number of generations. As far as scaling is concerned, it is useful to investigate how the GA behaves on problems where the building block scaling is extreme. Two extreme cases are when the building blocks are (1) uniformly scaled, and (2) exponentially scaled. An analysis for case 1 has been done both in terms of population sizing (Goldberg, Deb, & Clark, 1992), (Harik et al., 1997) and run duration (Mühlenbein & Schlierkamp-Voosen, 1993). Case 2, exponentially scaled problems, is the topic of this appendix. The study is important because one of the difficulties that GAs are faced with is due to the low scaling or low salience of some building blocks.

The appendix starts by reviewing the GA dynamics on this type of problems, and by taking a look at the model introduced by Thierens, Goldberg, and Pereira (1998). That work presented an investigation of genetic algorithm performance on problems with exponentially scaled genes. Section A.3 extends that work for the case of building blocks rather than single genes. The modeling techniques, however, are the same as the ones used previously by Thierens, Goldberg, and Pereira.

A.2 Background

Users of genetic algorithms oftentimes observe that not all parts of a problem are solved at the same speed, some genes are solved during the first few generations but others take more time to do so. This phenomena occurs because not all parts of a problem are equally important; some of them may be responsible for a high variance in the fitness of an individual while others may only affect an individual's fitness by a little amount. Fitness is what guides the genetic algorithm's search, and the GA focuses its attention on the genes or features that give the highest contribution to the fitness of the individuals. Once these have substantially converged, the GA moves on and tries to solve the remaining parts of the problem. However, the GA may have trouble in solving the low salient features due to the effects of random genetic drift, a topic to be elaborated later on.

A.2.1 Domino convergence

An example of a problem with exponentially scaled building blocks is the binary integer function. In this problem, the fitness of an individual string is the value of the string interpreted as a binary integer number. For example, in a 4-bit problem, string 1101 has fitness $8 + 4 + 0 + 1 = 13$. Each 1-bit corresponds to a building block, and the fitness contribution of each one is a power of 2 increasing from right to left.

When a GA operates on this problem the more significant bits (or genes) converge faster than the less significant ones. Overall, there is a sequential convergence of bits that resemble a row of domino stones falling down one after the other. To see why this is so, consider what happens when the two individuals, A and B, compete.

$$A = 11101101$$

$$B = 11110110$$

Individual B wins because the winner is decided on the first single gene position where the two individuals have a different allele value when scanning both strings from left to right. As can be seen from the example, the selection pressure on the low salient genes is negligible earlier in the search; those genes will only be substantially affected by the selection operator once the more salient genes have almost converged.

Given a large population size, genes converge correctly one after the other from the most salient to the least salient. But if the population size is not large enough, then the GA may have trouble in going all the way down to the least significant gene. The problem occurs because even when there is no selection pressure, allele frequencies fluctuate due to chance variation alone, and may be lost from the population by the time the GA wants to pay attention to them. This effect is known as random genetic drift and is the subject of the next section.

A.2.2 Random genetic drift

Consider a one-bit problem. In such problem there are two type of individuals, call them individual 0 and individual 1. In addition, let us also assume that individual 0 and individual 1 have the same fitness. Since there is no fitness difference between the two type of individuals, we would expect that both remain in the population forever. That does not happen because in each generation there is an element of chance in the selection of individuals that are used to form the next generation.

For instance, consider a population of 100 individuals, 50 of type 0 and 50 of type 1. To obtain the next generation, 100 new individuals are drawn, one at a time, from the population by using random sampling with replacement (individuals are put back into the old population after being selected). In the next generation we won't necessarily get 50 individuals of type 0 and 50 of type 1. Due to chance variation, we may get 52 of type 0 and 48 of type 1, or something like that. The resulting population becomes the starting point for the next generation and it is obvious what is going to happen after some time; eventually the population will be filled with individuals of only one type, either all 0's or all 1's. Once the population reaches such a state, only the mutation operator can reintroduce variation.

This type of process where there is a change in the gene frequency makeup of the population due to chance variation alone is called *random genetic drift* and there are mathematical models that can be used to study it (more about that in section A.3.1).

A.2.3 Model of Thierens, Goldberg, and Pereira (1998)

Domino convergence and genetic drift were recognized by Thierens, Goldberg, and Pereira as the two main things that govern the convergence process of the GA on this type of problems. Based on that observation, the authors built a mathematical model and analyzed the convergence behavior of the GA on the binary integer function. The way they did it was by building two separate models, one to analyze the domino convergence and another to analyze genetic drift. The authors derived two convergence times, $t_{convergence}$ and t_{drift}, to characterize each of the effects. Then they equated the two models and were able to predict at which point in time the drift effect is likely to dominate the convergence process. The next section extends this model for the case of building blocks rather than single genes.

A.3 From genes to building blocks

This section extends the model introduced by Thierens, Goldberg, and Pereira for the case of building blocks. In order to do so, let us start by stating a number of assumptions about the type of problem that we are modeling.

We assume that the fitness function is given by the sum of m non-overlapping subfunctions. Each subfunction is a function of k decision variables (k genes) corresponding to a building block. As in the case of the binary integer function, the building blocks are scaled in such a way that the winner of a pairwise competition is decided on the first building block position where the two individuals have a different value when scanning from left to right. Another assumption is that building block mixing is perfect. This idealized situation occurs when the user has knowledge about the structure of the problem (which is usually not true) or when the GA is able to learn gene linkage automatically.

There is no building block disruption under the assumption of perfect mixing. Thus, a GA operating on a building block of k genes is roughly equivalent to a GA operating on a single gene. This means that a problem with large building blocks can be mapped to a problem with building blocks constituted by single genes, the difference being that the expected proportion of alleles in a randomly initialized population is not half ones and half zeros, but is $1/2^k$ ones and the remaining proportion of zeros. By doing so, the modeling techniques of Thierens, Goldberg, and Pereira can be easily transfered for the case of building blocks. That's precisely what we do in the remaining sections.

A.3.1 Drift model

Consider a gene with two possible allele values, 1 and 0. Consider also a population of size N where a fraction p of the population has allele value 1 and the remaining fraction has allele value 0. The question that we are interested in is the following. How many generations on average does it take for an allele value to be lost from the population assuming that the gene is under the effect of random genetic drift alone. This question has been addressed in the field of population genetics (Kimura & Ohta, 1969) and also in the context of genetic algorithms (Goldberg & Segrest, 1987; Asoh & Mühlenbein, 1994).

The method used by Goldberg and Segrest (1987) and Asoh and Mühlenbein (1994) is a Markov Chain model as follows. Given a population of size N with two alleles, 1 and 0, then the state of the population can be described by the number of 1 alleles in the population. The possible states are 0, 1, 2, ... , N. States 0 and N are the *absorbing states* of the Markov Chain because once the population reaches one of them it cannot get back to another state. For all the other states, it is possible for the population to drift from state i to state j in a single generation. That probability is called the *transition probability*, and can be obtained from the binomial distribution. A population in state i has a frequency of 1 alleles given by $p = i/N$, and a frequency of 0 alleles given by $q = 1 - i/N$. The transition probability of going from i copies of allele 1 to j copies of allele 1 in a single generation is:

$$P_{ij} = \binom{N}{j} \; p^j \; q^{N-j}$$

Given a particular starting condition p, it is possible to solve the Markov Chain to give the expected drift time, but it is difficult to obtain a closed form expression for the general case of an arbitrary starting condition.

An alternative solution is to use a continuous approximation to the discrete Markov Chain model (Kimura, 1964) (see also Hartl and Clark (1997) for a more accessible description). The idea is to analyze genetic drift using a model similar to that of the physical process of diffusion. This is less intuitive than the Markov Chain model, but the basic idea is the following. Consider a distribution of populations, each having an allele frequency in the range from 0 to 1. The number of populations whose frequency is between x and $x + \partial x$ at time t gives a probability density $\phi(x,t)$. Populations may enter this range of allele frequencies by drifting in from a lower frequency, which occurs with a probability flux $J(x,t)$, and populations may leave this range of allele frequencies by drifting out, which occurs with probability flux $J(x + \partial x, t)$. The rate of change is the difference of fluxes, $J(x,t) - J(x + \partial x, t)$. For small ∂x, this difference can be written as $-\frac{\partial}{\partial x} J(x,t)$. Therefore, the rate of change of allele frequency through time is given by:

$$\frac{\partial}{\partial t}\phi(x,t) = -\frac{\partial}{\partial x} J(x,t) \tag{A.1}$$

The probability flux is:

$$J(x,t) = M(x)\phi(x,t) - \frac{1}{2}\frac{\partial}{\partial x}V(x)\phi(x,t) \tag{A.2}$$

where $M(x)$ is the average change in allele frequency in a population whose current allele frequency is x, and $V(x)$ is the variance in change in allele frequency. Substituting equation A.2 into equation A.1 gives:

$$\frac{\partial}{\partial t}\phi(x,t) = -\frac{\partial}{\partial x}\left[M(x)\phi(x,t)\right] + \frac{1}{2}\frac{\partial^2}{\partial x^2}\left[V(x)\phi(x,t)\right] \tag{A.3}$$

Equation A.3 is used widely in physics to model heat diffusion and is known as the Fokker-Planck equation. For the case of random genetic drift, $M(x) = 0$ and $V(x)$ is the binomial sampling variance which is equal to $x(1-x)/N$. Thus, the diffusion equation for random genetic drift is:

$$\frac{\partial}{\partial t}\phi(x,t) = \frac{1}{2N}\frac{\partial^2}{\partial x^2}\left[x(1-x)\phi(x,t)\right] \tag{A.4}$$

The diffusion approximation for the random genetic drift model is a second-order partial differential equation that gives a distribution $\phi(x,t)$, giving the number of populations with allele frequency x at time t. The solution of this equation requires advanced mathematics that won't be mentioned here (the interested reader should see (Kimura, 1964; Crow & Kimura, 1970) for the mathematical derivation). One important application of the diffusion approximation is the determination of the expected extinction time for an allele. Kimura and Ohta (1969) showed that in a population of size N, the average number of generations to lose an allele that starts with a proportion p is:

$$t_{drift}\ (p) \ = \ \frac{-2N}{1-p}\, p \ln p \tag{A.5}$$

The diffusion model for genetic drift was originally derived for diploid individuals. There, a population of N individuals has $2N$ alleles, and the binomial variance in that case is $V(x) = x(1-x)/2N$. Equations A.4 and A.5 are written assuming that $V(x) = x(1-x)/N$, which is the correct value for haploid individuals, the usual case in genetic algorithm implementations.

Equation A.5 gives the one-sided drift time of allele extinction, that is, it doesn't consider the cases when the allele gets fixed to the correct value. This one-sided drift time is the one that matters for the GA analysis because we don't want to consider the lucky occasions when a building block drifts to the correct value. Substituting p by $1/2^k$ in equation A.5 allow us to use the genetic algorithm terminology and say that on a population of size N, the average number of generations to lose a building block of size k genes under the effect of random genetic drift alone is:

$$t_{drift}\ (k) \ = \ \frac{2\ k\ \ln 2}{2^k - 1}\ N \tag{A.6}$$

Equation A.6 says that the extinction time for building blocks under random genetic drift alone grows linearly with the population size, and the constant factor decreases with the building block size. Table A.1 shows the expected extinction time for building block sizes

1 through 5 using equation A.6.

Table A.1: Expected extinction time for various building block sizes under the effect of random genetic drift.

Building block size	Drift time
1	1.386 N
2	0.924 N
3	0.594 N
4	0.370 N
5	0.224 N

Computer simulations of random genetic drift match the results obtained from the diffusion model quite well (see figure A.1).

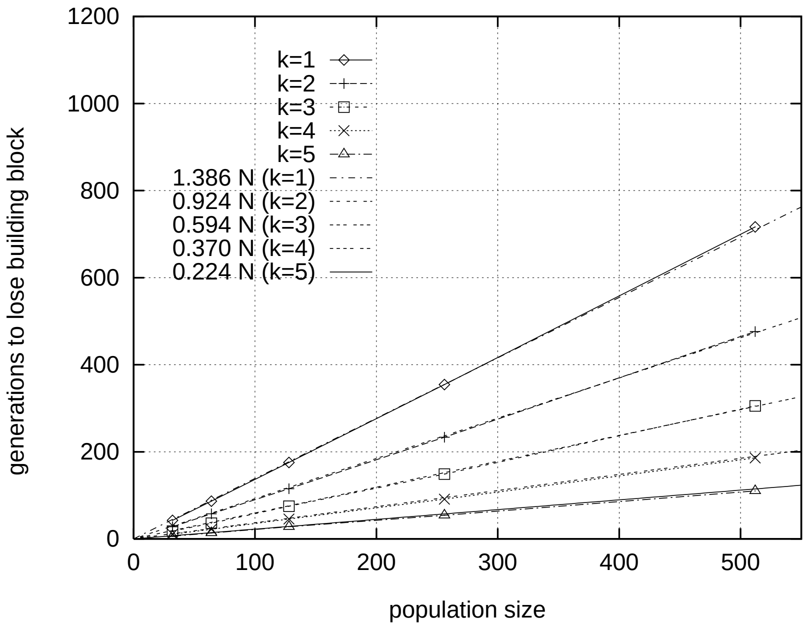


Figure A.1: Average time to lose a building block. The solid lines are from the diffusion model of Kimura and Ohta (1969). The isolated points were obtained by computer simulation.

A.3.2 Domino model

This section applies the model developed by Thierens, Goldberg, and Pereira (1998) for building blocks. The terminology is the same as the one adopted by the previous authors. The model assumes that when the most salient λ building blocks have converged, the remaining $m - \lambda$ building blocks are still in their initial random state. The convergence model is based on the selection differential $S(t)$, the difference between the mean fitness of the population at generation $t+1$ and the population mean fitness at generation t. The selection intensity $I(t)$ is the selection differential $S(t)$ scaled by the standard deviation $\sigma(t)$ of the population fitness.

$$I(t) = \frac{S(t)}{\sigma(t)} = \frac{\mu(t+1) - \mu(t)}{\sigma(t)}$$

The increase in the population mean fitness from one generation to the next can also be written as a function of λ.

$$I(t) = \frac{\mu_{t+1}(\lambda) - \mu_t(\lambda)}{\sigma_t(\lambda)}$$

Next, we compute the mean and variance of the population fitness when λ building blocks have converged, but before doing that let's compute the mean and variance of a single building block. In order to make the analysis simple, we consider that a k-bit building block correspond to a k-bit needle in a haystack (NIAH) function. In such a function, one solution has fitness f_{max} and the remaining $2^k - 1$ solutions have fitness f_{min}. Let $p = 1/2^k$, then the mean μ and variance σ^2 of a k-bit NIAH function is:

$$\mu = p\ f_{max} + (1-p)\ f_{min}$$

$$\sigma^2 = p\ (f_{max} - \mu)^2 + (1-p)\ (f_{min} - \mu)^2$$

In the case of our exponentially scaled problem, $f_{min} = 0$ and f_{max} is a power of 2 whose value depends on the building block's salience (or importance). The mean and variance of the i^{th} most salient building block is:

$$\mu_{bb(i)} = p\ f_{max}\ =\ p\ 2^{m-i}$$

$$\sigma^2_{bb(i)} = p\ (1-p)\ (f_{max})^2\ =\ p\ (1-p)\ (2^{m-i})^2$$

We are now ready to calculate the mean and variance when λ building blocks have converged. Throughout the text, $p = 1/2^k$ denotes the expected proportion of building blocks on a randomly initialized population. The mean fitness $\mu(\lambda)$ is calculated by assuming that the first λ building blocks have already converged to the correct value, and the remaining $m - \lambda$ building blocks are still in their initial random proportion.

$$\begin{aligned} \mu(\lambda) &= \sum_{i=1}^{\lambda} 2^{m-i} + \sum_{i=\lambda+1}^{m} p\ 2^{m-i} \\ &= \sum_{j=0}^{m-1} 2^j - \sum_{j=0}^{m-\lambda-1} 2^j + p \sum_{j=0}^{m-\lambda-1} 2^j \\ &= 2^m - 1 + (p-1)(2^{m-\lambda} - 1) \end{aligned} \tag{A.7}$$

For the variance calculation, it is only necessary to consider the non-converged region because the region that has already converged contributes nothing to the variance.

$$\begin{aligned} \sigma^2(\lambda) &= \sum_{j=0}^{m-\lambda-1} p\ (1-p)\ (2^j)^2 \\ &= p\ (1-p) \sum_{j=0}^{m-\lambda-1} 4^j \\ &= p\ (1-p)\ \frac{4^{m-\lambda} - 1}{4 - 1} \\ &\approx p\ (1-p)\ \frac{4^{m-\lambda}}{3} \end{aligned} \tag{A.8}$$

Thierens, Goldberg, and Pereira derived convergence times for both constant and variable selection intensities. Here, we limit ourselves to the case of constant selection intensities because they are the ones that yield faster convergence times. Examples of such selection schemes are tournament selection, truncation selection, and rank-based selection. In these schemes the average fitness increase from one generation to the next is equal to the product of the selection intensity I by the standard deviation of the population fitness:

$$\mu_{t+1}(\lambda) - \mu_t(\lambda) = \sigma_t(\lambda)\ I$$

Substituting the values for the mean $\mu(\lambda)$ and standard deviation $\sigma(\lambda)$ obtained in equations A.7 and A.8 gives:

$$(p-1)\left(2^{m-\lambda_{t+1}} - 1\right) - (p-1)\left(2^{m-\lambda_t} - 1\right) = \sqrt{\frac{p\ (1-p)}{3}}\ 2^m\ 2^{-\lambda_t}\ I$$

Simplifying gives:

$$2^{-\lambda_{t+1}} = 2^{-\lambda_t}\ \left(1 - I\sqrt{\frac{p}{3\ (1-p)}}\right)$$

When $t = 0$, $\lambda_0 = 0$. Therefore:

$$2^{-\lambda_t} = \left(1 - I\ \sqrt{\frac{p}{3\ (1-p)}}\right)^t$$

which is equivalent to:

$$t = \frac{-\ln 2}{\ln\left(1 - I\ \sqrt{\frac{p}{3\ (1-p)}}\right)}\ \lambda_t$$

Substituting p by $1/2^k$ gives convergence time as a function of the building block size k:

$$t = \frac{-\ln 2}{\ln\left(1 - I\,\frac{1}{\sqrt{3}\,\sqrt{2^k - 1}}\right)}\,\lambda_t \tag{A.9}$$

Equation A.9 says that the number of generations t until convergence is a linear function of the building block's salience in the string. For example, for binary tournament selection the selection intensity is $I = 1/\sqrt{\pi}$, and the expected number of generations until the entire string converges ($\lambda_t = m$) for the case of building blocks of size $k = 1$ is precisely the same equation that Thierens, Goldberg, and Pereira obtained:

$$t = \frac{-\ln 2}{\ln\left(1 - \frac{1}{\sqrt{3\pi}}\right)}\,m = 1.76\ m$$

As another example, for building blocks of size $k = 3$, the equation becomes:

$$t = \frac{-\ln 2}{\ln\left(1 - \frac{1}{\sqrt{21\pi}}\right)}\,m = 5.28\ m$$

Just like in the original paper of Thierens, Goldberg, and Pereira, the constant factor obtained with the domino convergence model shouldn't be taken too literally because this kind of modeling is not completely exact. Nevertheless, the dimensional relations or functional form is correct (it is confirmed by experiments in section A.3.4) and says that the average number of generations until convergence grows linearly with respect to the number of building blocks.

A.3.3 Domino and drift model together

By joining the drift model with the domino convergence model, it is possible to predict where in the string is the drift stall likely to occur. That's precisely what we do next by equating the times for the two models, $t_{drift} \approx t_{convergence}$:

$$\frac{2\ k\ \ln 2}{2^k - 1}\ N \approx \frac{-\ln 2}{\ln\left(1 - I\ \frac{1}{\sqrt{3}\ \sqrt{2^k-1}}\right)}\ \lambda^*$$

Rearranging:

$$\lambda^* \approx \frac{-2\ k\ \ln\left(1 - I\ \frac{1}{\sqrt{3}\ \sqrt{2^k-1}}\right)}{2^k - 1}\ N \tag{A.10}$$

Equation A.10 says that given a population size N, it is possible to predict the number of building blocks λ^* that will be solved correctly by the GA. Likewise, given a number of building blocks λ^* that we wish to solve correctly, it is possible to predict the necessary population size needed by the GA to do so. Looking only at the functional form, equation A.10 can be written as $N = c\ \lambda^*$, where c is a constant that depends on the building block size k and the selection intensity I.

The population size grows linearly with respect to the number of building blocks λ^*. In section A.3.2 we also observed that the average number of generations until convergence under the domino model is also a linear function of the number of building blocks. The number of fitness function evaluations taken by the GA is the product of the population size by the number of generations, and both are linear functions of the number of building blocks. Therefore, the overall time complexity of genetic algorithms on exponentially scaled problems, under the assumption of perfect mixing, is quadratic. The next section presents computer experiments that confirm a quadratic time complexity.

A.3.4 Computer experiments

Computer experiments were performed on problems with 200 exponentially scaled building blocks. The experiments were done using binary tournament selection with replacement, and the building blocks mixed perfectly using a population-wise crossover operator similar to the one used in the compact GA (Harik, Lobo, & Goldberg, 1998) or UMDA (Mühlenbein & Paaß, 1996). The experiments were done for building blocks of size 1 and 3. For size 1 building blocks, the population was initialized with exactly 50% ones and 50% zeros in each bit position. 100 independent runs were performed for population sizes 32, 64, 128, 256, 512, 1024, and 2048.

Figure A.2 plots the average number of generations needed to either solve or lose the building block. Building block number 0 is the most salient and building block number 199 is the least salient. The plot is identical to the one obtained by Thierens, Goldberg, and Pereira, except that in this case the results are shown for various population sizes. For very small population sizes the convergence process is largely dominated by the drift model, while for large population sizes the convergence is dictated by the domino model. For population sizes in the middle range, the most salient building blocks follow the domino model, and after some point, the drift effect starts to dominate.

Figure A.3 is a similar plot for building blocks of size $k = 3$. Each building block is a NIAH function. The experiments were conducted in a similar way as the ones for $k = 1$, with the difference that in this case, the initial population had a proportion of 1/8 ($1/2^k$, for $k = 3$) ones and the remaining proportion zeros.

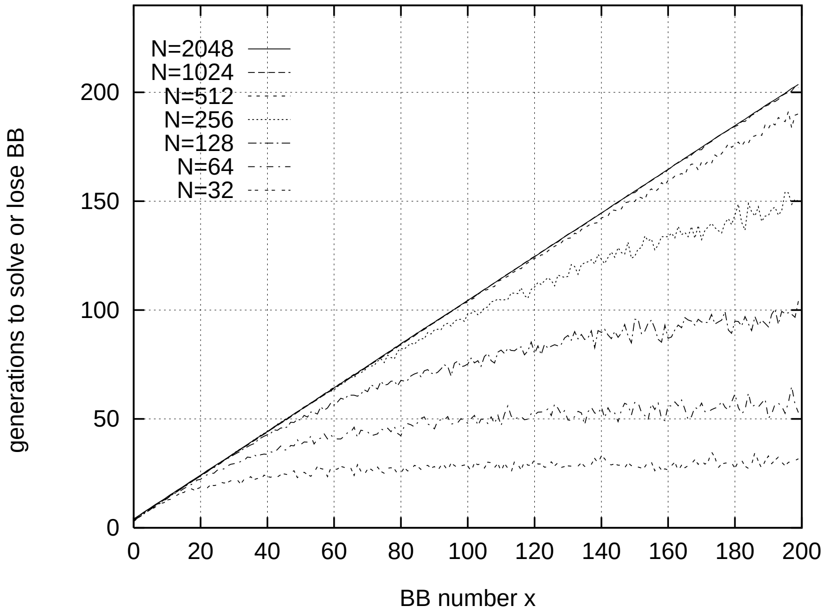


Figure A.2: Convergence behavior of an exponentially scaled problem with 200 building blocks of size $k = 1$ for various population sizes.

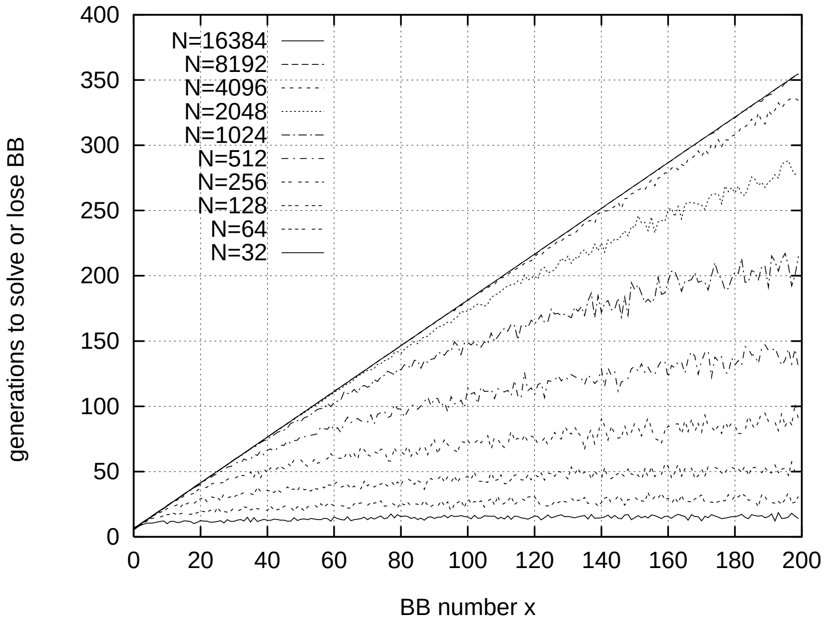


Figure A.3: Convergence behavior of an exponentially scaled problem with 200 building blocks of size $k = 3$ for various population sizes.

The experiments were done is this way to reflect an idealized building block mixing behavior. On an actual GA run, we should expect to observe slight differences from the results obtained in figures A.2 and A.3. Notice the straight lines obtained for the experiments with large population sizes in both figures A.2 and A.3. These results confirm the linear time complexity for run duration. The constant factor obtained from the theoretical model, however, is a conservative estimate. The difference between model and experiment can be explained due to the assumption that when the most salient λ building blocks have converged, the others are still in their initial random state. This is only an approximation of what actually happens in a GA run. There, when the λ^{th} building block has converged, the $(\lambda+1)^{th}$ building block has already converged substantially, and is far from its initial random state. Therefore, on an actual GA run the convergence time is faster than the one predicted by the domino model.

The experiments show a linear time complexity for run duration. In terms of population sizing, a look at the experimental data from our GA runs also reveals a linear growth. For example, in order to correctly solve the first 25 building blocks (in the case of $k = 1$) in at least 99% of the runs, the GA needed a population size of 256. To do the same thing for the first 50 building blocks a population size of 256 was not enough. Table A.2 shows the population size needed to correctly solve the first 25, 50, 100, and 200 building blocks in at least 99% of the runs, for the case of size-1 building blocks. Table A.3 shows the same thing for the case of size-3 building blocks.

In both cases, when the number of building blocks doubles, the population sizing requirements also doubles. In summary, both population size and run duration grow linearly with the number of building blocks, giving an overall quadratic time complexity in terms of fitness function evaluations.

Table A.2: Population size needed to correctly solve the first 25, 50, 100, and 200 size-1 building blocks in at least 99 out of 100 runs.

Building blocks	Population size
25	256
50	512
100	1024
200	2048

Table A.3: Population size needed to correctly solve the first 25, 50, 100, and 200 size-3 building blocks in at least 99 out of 100 runs.

Building blocks	Population size
25	2048
50	4096
100	8192
200	16384

A.4 Summary

This study extended the model of Thierens, Goldberg, and Pereira (1998) to analyze the time complexity of genetic algorithms on exponentially scaled problems. An extended drift and domino convergence model was presented along with computer simulations. The main result of this work is that under the assumption of perfect building block mixing, the overall time complexity of genetic algorithms on problems with exponentially scaled building blocks is a quadratic function of the number of building blocks.

A.5 Conclusions

For the two extreme cases of building block scaling, uniform and exponential, genetic algorithms with perfect mixing have time complexities of $O(m)$ and $O(m^2)$ respectively. The linear time complexity for uniformly scaled problems occurs because the population sizing grows with the square root of m (Harik et al., 1997) and the time to convergence also grows with the square root of m (Mühlenbein & Schlierkamp-Voosen, 1993).

A quadratic time complexity for exponentially scaled problems does not seem that bad, and we may speculate that it diminishes the relevance of a fixed mutation operator as a means of introducing diversity in the population. Notice that a fixed mutation operator needs $O(\ell^k)$ trials in order to discover a k-bit building block without messing up with the remaining parts of the problem (Mühlenbein, 1992).

The relevance of other building block diversity preservation techniques such as the one used in the Linkage Learning Genetic Algorithm (LLGA) (Harik, 1997) may also be not so important after all. The LLGA was able to solve exponentially scaled problems in almost linear time (Harik, 1997), (Lobo et al., 1998) due to a built-in probabilistic expression mechanism that preserved building block diversity and partly eliminated the problem of random genetic drift. Nevertheless, while the LLGA excelled in exponentially scaled problems, it did not do as well when solving other type of problems.

The time complexity estimates presented herein are for idealized situations because they assume perfect building block mixing. With traditional simple GAs, this assumption only occurs if the building blocks are trivial (size-1 building blocks), or when the user has knowledge about the structure of the problem (which is usually not true). However, with more advanced GAs that are able to learn gene linkage automatically, the assumption of perfect mixing is well approximated, and we should only expect to observe a slightly worse performance than in the idealized perfect mixing situation. The reason is that there may be a little overhead in population sizing and run duration needed by these advanced GAs in order to learn a problem's structure, without which efficient building block mixing is not possible. Examples of such GAs are the ECGA (Harik, 1999) and the BOA (Pelikan, Goldberg, & Cantú-Paz, 1999).

Bibliography

Asoh, H., & Mühlenbein, H. (1994). On the mean convergence time of evolutionary algorithms without selection and mutation. In Davidor, Y., Schwefel, H.-P., & Männer, R. (Eds.), *Parallel Problem Solving from Nature, PPSN III* (pp. 88–97). Berlin: Springer-Verlag.

Bäck, T. (1992). Self-adaptation in genetic algorithms. In Varela, F. J., & Bourgine, P. (Eds.), *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life* (pp. 263–271). Cambridge, MA: The MIT Press.

Bäck, T. (1993). Optimal mutation rates in genetic search. In Forrest, S. (Ed.), *Proceedings of the Fifth International Conference on Genetic Algorithms* (pp. 2–8). San Mateo, CA: Morgan Kaufmann.

Bäck, T. (Ed.) (1997). *Proceedings of the Seventh International Conference on Genetic Algorithms*. San Francisco, CA: Morgan Kaufmann.

Bäck, T., & Schwefel, H.-P. (1993). An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, *1*(1), 1–23.

Bäck, T., & Schwefel, H.-P. (1995). Evolution strategies I: Variants and their computational implementation. In Winter, G., Périaux, J., Galán, M., & Cuesta, P. (Eds.), *Genetic Algorithms in Engineering and Computer Science* (Chapter 6, pp. 111–126). Chichester: John Wiley and Sons.

Bagley, J. D. (1967). *The behavior of adaptive systems which employ genetic and correlation algorithms*. Doctoral dissertation, University of Michigan. (University Microfilms No. 68-7556).

Baluja, S., & Caruana, R. (1995). *Removing the genetics from the standard genetic algorithm* (Tech Rep. No. CMU-CS-95-141). Pittsburgh, Pennsylvania: Carnegie Mellon University.

Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., & Smith, R. E. (Eds.) (1999). *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*. San Francisco, CA: Morgan Kaufmann.

Cantú-Paz, E. (1999). *Designing efficient and accurate parallel genetic algorithms*. Doctoral dissertation, University of Illinois at Urbana-Champaign, Urbana. Also IlliGAL Report No. 99017.

Cartwright, H. M. (1997). Genetic algorithms for the analysis of the movement of airborne pollution. In Bäck, T., Fogel, D. B., & Michalewicz, Z. (Eds.), *Handbook of Evolutionary Computation* (pp. G5.1:1–8). Bristol and New York: Institute of Physics Publishing and Oxford University Press.

Cavicchio, Jr., D. J. (1970). *Adaptive search using simulated evolution*. Unpublished doctoral dissertation, University of Michigan, Ann Arbor, MI. (University Microfilms No. 25-0199).

Cormen, T., Leiserson, C., & Rivest, R. (1993). *Introduction to algorithms*. MIT Press, McGraw-Hill.

Crow, J. F., & Kimura, M. (1970). *An introduction to population genetics theory.* New York: Harper and Row.

Davis, L. (1989). Adapting operator probabilities in genetic algorithms. In Schaffer, J. D. (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 61–69). San Mateo, CA: Morgan Kaufmann.

De Jong, K. A. (1975). *An analysis of the behavior of a class of genetic adaptive systems.* Doctoral dissertation, University of Michigan, Ann Arbor. (University Microfilms No. 76-9381).

Deb, K. (1991). *Binary and floating-point function optimization using messy genetic algorithms* (IlliGAL Report No. 91004 and doctoral dissertation, Unversity of Alabama, Tuscaloosa). Urbana: University of Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory.

Eiben, A. E., Bäck, T., Schoenauer, M., & Schwefel, H.-P. (Eds.) (1998). *Parallel Problem Solving from Nature, PPSN V.* Berlin: Springer-Verlag.

Fogel, D. B., Schwefel, H.-P., Bäck, T., & Yao, X. (Eds.) (1998). *Proceedings of 1998 IEEE International Conference on Evolutionary Computation.* Piscataway, NJ: IEEE Service Center.

Frantz, D. R. (1972). *Non-linearities in genetic adaptive search.* Doctoral dissertation, University of Michigan. (University Microfilms No. 73-11,116).

Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability: a guide to the theory of NP-completeness.* Freeman.

Goldberg, D. E. (1985). *Optimal initial population size for binary-coded genetic algorithms* (TCGA Report No. 85001). Tuscaloosa, AL: University of Alabama, The Clearinghouse for Genetic Algorithms.

Goldberg, D. E. (1987). Simple genetic algorithms and the minimal, deceptive problem. In Davis, L. (Ed.), *Genetic Algorithms and Simulated Annealing* (Chapter 6, pp. 74–88). Los Altos, CA: Morgan Kaufmann.

Goldberg, D. E. (1989a). *Genetic algorithms in search, optimization, and machine learning.* Reading, MA: Addison-Wesley.

Goldberg, D. E. (1989b). Sizing populations for serial and parallel genetic algorithms. In Schaffer, J. D. (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 70–79). San Mateo, CA: Morgan Kaufmann. (Also TCGA Report 88004).

Goldberg, D. E. (1999). *Genetic and evolutionary algorithms in the real world* (IlliGAL Report No. 99013). Urbana: University of Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory.

Goldberg, D. E., & Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. *Foundations of Genetic Algorithms*, *1*, 69–93. (Also TCGA Report 90007).

Goldberg, D. E., Deb, K., & Clark, J. H. (1992). Genetic algorithms, noise, and the sizing of populations. *Complex Systems*, *6*, 333–362.

Goldberg, D. E., Deb, K., Kargupta, H., & Harik, G. (1993). Rapid, accurate optimization of difficult problems using fast messy genetic algorithms. In Forrest, S.

(Ed.), *Proceedings of the Fifth International Conference on Genetic Algorithms* (pp. 56–64). San Mateo, CA: Morgan Kaufmann.

Goldberg, D. E., Deb, K., & Thierens, D. (1993). Toward a better understanding of mixing in genetic algorithms. *Journal of the Society of Instrument and Control Engineers*, *32*(1), 10–16.

Goldberg, D. E., Korb, B., & Deb, K. (1989). Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, *3*(5), 493–530.

Goldberg, D. E., & Segrest, P. (1987). Finite Markov chain analysis of genetic algorithms. In Grefenstette, J. J. (Ed.), *Proceedings of the Second International Conference on Genetic Algorithms* (pp. 1–8). Hillsdale, NJ: Lawrence Erlbaum Associates.

Grefenstette, J. J. (1986). Optimization of control parameters for genetic algorithms. In Sage, A. P. (Ed.), *IEEE Transactions on Systems, Man, and Cybernetics*, Volume SMC–16(1) (pp. 122–128). New York: IEEE.

Harik, G. R. (1997). *Learning gene linkage to efficiently solve problems of bounded difficulty using genetic algorithms*. Doctoral dissertation, University of Michigan, Ann Arbor. Also IlliGAL Report No. 97005.

Harik, G. R. (1999). *Linkage Learning via Probabilistic Modeling in the ECGA.* (IlliGAL Report No. 99010). Urbana: University of Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory.

Harik, G. R., Cantú-Paz, E., Goldberg, D. E., & Miller, B. L. (1997). The gambler's ruin problem, genetic algorithms, and the sizing of populations. In Bäck, T. (Ed.), *Proceedings of 1997 IEEE International Conference on Evolutionary Computation* (pp. 7–12). New York: IEEE Press.

Harik, G. R., & Lobo, F. G. (1999). A parameter-less genetic algorithm. See Banzhaf, Daida, Eiben, Garzon, Honavar, Jakiela, and Smith (1999), pp. 258–267.

Harik, G. R., Lobo, F. G., & Goldberg, D. E. (1998). The compact genetic algorithm. In *Proceedings of 1998 IEEE International Conference on Evolutionary Computation* (pp. 523–528). Piscataway, NJ: IEEE.

Hartl, D. L., & Clark, A. G. (1997). *Principles of population genetics (3 ed.)*. Sunderland, Massachusetts: Sinauer Associates.

Holland, J. H. (1973). Genetic algorithms and the optimal allocation of trials. *SIAM Journal on Computing*, *2*(2), 88–105.

Holland, J. H. (1975). *Adaptation in natural and artificial systems*. Ann Arbor, MI: University of Michigan Press.

Johnson, R. E. (1997). Frameworks = (components + patterns). *Communications of the ACM*, *40*(10), 39–42.

Julstrom, B. A. (1995). What have you done for me lately? Adapting operator probabilities in a steady-state genetic algorithm. In Eschelman, L. (Ed.), *Proceedings of the Sixth International Conference on Genetic Algorithms* (pp. 81–87). San Francisco, CA: Morgan Kaufmann.

Kargupta, H. (1995). *SEARCH, polynomial complexity, and the fast messy genetic algorithm*. udd, University of Illinois at Urbana-Champaign, Urbana, IL. (Also IlliGAL Report No. 95008).

Kargupta, H. (1996). The gene expression messy genetic algorithm. In *Proceedings of 1996 IEEE International Conference on Evolutionary Computation* (pp. 814–819). New York: IEEE Press.

Kimura, M. (1964). Diffusion models in population genetics. *J. Appl. Prob.*, *1*, 177–232.

Kimura, M., & Ohta, T. (1969). The average number of generations until fixation of a mutant gene in a finite population. *Genetics*, *61*, 763–771.

Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H., & Riolo, R. L. (Eds.) (1998). *Genetic Programming 98*. San Francisco: Morgan Kaufmann Publishers.

Lobo, F. G., Deb, K., Goldberg, D. E., Harik, G. R., & Wang, L. (1998). Compressed introns in a linkage learning genetic algorithm. See Koza, Banzhaf, Chellapilla, Deb, Dorigo, Fogel, Garzon, Goldberg, Iba, and Riolo (1998), pp. 551–558.

Lobo, F. G., & Goldberg, D. E. (1997). Decision making in a hybrid genetic algorithm. In *Proceedings of 1997 IEEE International Conference on Evolutionary Computation* (pp. 122–125). Piscataway, NJ: IEEE.

Mercer, R. E., & Sampson, J. R. (1978). Adaptive search using a reproductive metaplan. *Kybernetes*, *7*, 215–228.

Miller, B. L. (1997). *Noise, sampling, and efficient genetic algorithms*. doctoral dissertation, University of Illinois at Urbana-Champaign, Urbana. Also IlliGAL Report No. 97001.

Mühlenbein, H. (1992). How genetic algorithms really work: I.Mutation and Hillclimbing. In Männer, R., & Manderick, B. (Eds.), *Parallel Problem Solving from Nature, 2* (pp. 15–25). Amsterdam, The Netherlands: Elsevier Science.

Mühlenbein, H., & Paaß, G. (1996). From recombination of genes to the estimation of distributions I. binary parameters. In Voigt, H.-M., Ebeling, W., Rechenberg, I., & Schwefel, H.-P. (Eds.), *Parallel Problem Solving from Nature, PPSN IV* (pp. 178–187). Berlin: Springer-Verlag.

Mühlenbein, H., & Schlierkamp-Voosen, D. (1993). Predictive models for the breeder genetic algorithm: I. Continuous parameter optimization. *Evolutionary Computation*, *1*(1), 25–49.

Pelikan, M., Goldberg, D. E., & Cantú-Paz, E. (1999). BOA: the bayesian optimization algorithm. See Banzhaf, Daida, Eiben, Garzon, Honavar, Jakiela, and Smith (1999), pp. 525–532.

Pelikan, M., & Lobo, F. G. (1999). *Parameter-less genetic algorithm: a worst-case time and space complexity analysis* (IlliGAL Report No. 99014). Urbana: University of Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory.

Pereira, A. G. (1998). *Extending environmental impact assessment processes: Generation of alternatives for siting and routing infrastructural facilities by multi-criteria evaluation and genetic algorithms*. Doctoral dissertation in environmental engineering, Universidade Nova de Lisboa, Lisboa.

Schaffer, J. D., Caruana, R. A., Eshelman, L. J., & Das, R. (1989). A study of control parameters affecting online performance of genetic algorithms for function optimization. In Schaffer, J. D. (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 51–60). San Mateo, CA: Morgan Kaufmann.

Shu, L. S., Wallace, D. R., & Flowers, W. C. (1996). Probabilistic methods in life-cycle design. In *Proceedings of the 1996 IEEE International Symposium on Electronics and the Environment* (pp. 7–12). Piscataway, NJ: IEEE.

Simpson, A. R., Dandy, G. C., & Murphy, L. J. (1994). Genetic algorithms compared to other techniques for pipe optimization. *Journal of Water Resources Planning and Management*, *120*(4), 423–443.

Smith, J., & Fogarty, T. C. (1996). Self adaptation of mutation rates in a steady state genetic algorithm. In of Electrical, I., & Engineers, E. (Eds.), *Proceedings of 1996 IEEE International Conference on Evolutionary Computation* (pp. 318–323). Piscataway, NJ: IEEE Service Center.

Smith, R. E., & Smuda, E. (1995). Adaptively resizing populations: Algorithm, analysis, and first results. *Complex Systems*, *9*, 47–72.

Thierens, D. (1995). *Analysis and design of genetic algorithms.* Leuven, Belgium: Katholieke Universiteit Leuven.

Thierens, D., & Goldberg, D. E. (1993). Mixing in genetic algorithms. In Forrest, S. (Ed.), *Proceedings of the Fifth International Conference on Genetic Algorithms* (pp. 38–45). San Mateo, CA: Morgan Kaufmann.

Thierens, D., Goldberg, D. E., & Pereira, A. (1998). Domino convergence, drift, and the temporal-salience structure of problems. In *Proceedings of 1998 IEEE International Conference on Evolutionary Computation* (pp. 535–540). Piscataway, NJ: IEEE.

Weinberg, R. (1970). Computer simulation of a living cell. *Dissertations Abstracts International*, *31*(9), 5312B. (University Microfilms No. 71-4766).