Contract number° 507953

Workpackage 8
Dynamics, selection, aggregation

Deliverable 8.1
Report on Evolution of High-Level Software Components

Project funded by the European Community under the “Information Society Technology” Programme

Contract number: 507953
Project acronym: DBE
Title: Digital Business Ecosystem

Deliverable Nº: D8.1
Due date: 30/04/2005
Delivery date: 30/04/2005 (prospective)

Short description:
This report brings together material of a theoretical nature concerning the process of evolution as it might be applied to the problem of the automatic creation of high-level software, in response to requests by a user. In addition, some related technical problems in the theory of evolutionary computation are addressed.

Partners owning: UBHAM (Jonathan E. Rowe, Boris Mitavskiy)
Partners contributed: STU, LSE, ICL
Made available to: DBE Consortium and European Commission

Versioning		
Version	**Date**	**Author, Organisation**
1.0	22/04/2005	UBHAM
0.9	31/03/2005	UBHAM

Quality check
1st internal reviewer: Giulio Marcon (STU)
2nd internal reviewer: Gerard Briscoe (ICL)

Contents

Executive Summary

This report brings together material of a theoretical nature concerning the process of evolution as it might be applied to the problem of the automatic creation of high-level software, in response to requests by a user. In addition, some related technical problems in the theory of evolutionary computation are addressed.

The report begins with an overview, describing each of the following chapters in turn. The first three of these chapters have already been circulated to the Science partners as discussion documents, and include: an initial view of the evolution of high-level software; an empirical study of genetic algorithms in comparison with other techniques for solving an abstracted version of the "DBE problem"; and a tutorial-style summary of the relevant mathematical background. Two technical chapters follow: one on co-evolution (which is on-going work) and one on variable-length structures (which has been presented at a recent conferences and will be published). The latter is rather mathematical and the details are unlikely to appeal to the non-specialist. The main points are summarised in the overview.

It will be seen that a sound mathematical basis for the Evolutionary Environment is a difficult technical challenge. Much progress has been made so far, and we will continue to develop this work in the second phase of the project. The main areas of development will be:

- Fitness landscape analysis, and the design of genetic operators.
- Continuing work on variable-sized structures. In particular we aim to adapt our results to the structures proposed by STU for representing services and workflows.
- Co-evolution and dynamic environments. This work will be important for studying the long-term evolution of the DBE, as users' requirements change in time, and new services become available.

The direct "customers" for this work are the other Science partners. In particular:

- STU — understanding the effects of genetic operators (crossover, mutation) and their interaction with the fitness landscape.
- ICL — mathematical modelling of evolving populations, providing the essential background to their work on entropy and efficiency in evolution.
- UniS — evolution as optimisation, especially the comparative experimental work.

- LSE — development of a formal, abstract view of evolution within the DBE.

It should be noted that the deliverable was originally intended to be a joint work with UniS, who were to contribute a first release of an optimisation algorithm. However, due to management difficulties, their contribution will not be appearing here.

Chapter 1

Overview

1.1 Introduction

This report presents a collection of pieces of work by the University of Birmingham relating to theoretical aspects of evolutionary computation and its application to evolving high-level software components within the framework of the Digital Business Ecosystem project. This work has been done using both empirical work and mathematical analysis. While some of it is highly technical, and somewhat abstract, we have made efforts to communicate basic results in an understandable way to the other partners of the project, through reports and through presentations and discussion. This has taken place throughout the project thus far, and some of these documents are reproduced here as a record of the engagement of this more theoretical work with the rest of the project.

The bulk of this report comprises a sequence of reports. Some of these have been distributed at an early stage to the project partners. Some have been, or are in the process of being, published as academic research papers. Much of the work is ongoing fundamental research into the nature of evolutionary computation. This chapter presents an overview of the rest of the report, providing a non-technical summary of the chief elements of the other chapters. The remaining chapters are as follows:

- An initial view.
- Alternative optimisation algorithms.
- Mathematical background.
- Co-evolution and extinction.
- Variable-sized structures.

The first of three reports should be fairly accessible to the non-specialist. They present some preliminary suggestions regarding evolution and the Digital Business Ecosystem; some empirical evidence that evolutionary techniques are a suitable technology for solving these kinds of problems; and an outline of the relevant mathematical background. There then follow two reports

of a more specialised, technical nature. The first combines an analytic and empirical approach to studying the dynamics of co-evolutionary systems, and the latter is a technical mathematical work relating to evolution of variable-sized structures. The details presented are likely to be of interest only to a specialist, but since this work represents a substantial research contribution to the field, it is included for completeness. A non-technical presentation of the chief results, and their relationship to the DBE project is given in the remainder of this chapter.

1.2 An initial view

This document was circulated early on in the project (April 2004). Its aim was to provoke discussion and offer some insights from existing understanding of evolutionary computation for how these techniques might be adapted to the DBE. It starts with the problem of verification, which is a key problem in Genetic Programming research: how can one be certain that the evolved program is guaranteed to perform the task for which it has been evolved? This leads directly to the idea that there could be a separation of the underlying implementation of a service (be it in code, or as a real-world service), and its description in some formal language (a specification). This separation has now become fundamental to the description of services in the DBE. Moreover, it potentially solves the verification problem, since the behaviour of an evolved composite service may be deduced from the specifications associated with each of its components.

Secondly, there is a discussion of the types of structure that might be used to combine services into composites. Fixed-length strings have, of course, a long history in genetic algorithms, and may be appropriate for the early stage implementation of the DBE: the user specifies a fixed list of the *types* of component services required, and the evolution fills these slots with (near) optimal components from the current pool (habitat). This approach can be found in the current implementations developed by Trinity College Dublin, and modelled by STU. More realistically, strings that can vary in length might be more suitable for composite services of arbitrary complexity. This was a view being developed in parallel at ICL by Gerard Briscoe. Indeed, one of the consequences of this discussion document was a joint position statement authored by UB-HAM, ICL and LSE which formally proposed the habitat architecture for the DBE. This paper is not included in the current deliverable. Other potential structures for combining services include trees and networks. There has been ongoing discussion between the Science partners and TCD as to the most appropriate structure, and this issue remains a matter of on-going research.

A third issue raised is that of fitness, and how it is to be measured. One key problem here is how to cope with the fact that the fitness of services has many aspects, deriving from a number of sources. Ideas from *multi-objective* optimisation are introduced as a potential way forward. These suggestions have fed directly into the development of the "fitness landscape" by STU.

The document finishes with some speculations about modularity and the potential for *virtual* services which exist only at the specification level, with no underlying concrete implementation. The idea would be for such services to be implemented "on-demand" — an idea which has taken shape through the course of the project so far, and relates to issues in the automatic generation of code.

1.3 Alternative optimisation algorithms

At the time of the DBE first annual review, a concern was expressed that the underlying assumption that evolutionary methods were a suitable technology had not been properly tested. Consequently, a set of empirical experiments were developed to investigate this matter. Since the exact nature of the DBE as an optimisation problem is still under development, an abstraction of the problem was put forward, based on the *set cover* problem. In this formulation, a user puts forward a set of requirements, taken from some larger set. Each component service can satisfy a number of these requirements, and the goal is to find some minimal collection of services so that all the user's requirements are met. To make the scenario more realistic, we also assume that each service has a cost associated with it, so the goal becomes finding a minimal cost collection of services that satisfy the requirements. A fitness function was derived to reflect this.

A restriction was made that the algorithms to be considered would be *black box* algorithms. That is, they would have no access to the structure of the problem under consideration — the only information they can gain comes from evaluating solutions with the fitness function. This rules out the possibility of using special-purpose heuristics and approximation algorithms. This restriction reflects the reality that the algorithms employed in the DBE will not necessarily have access to the *reasons* for the assignment of fitness values (since these may come, for example, from user feedback). The optimisation algorithms used were:

- Random search.
- Steepest descent (with restarts).
- Simulated annealing.
- Tabu search.
- Genetic algorithm.

In order to compare a collection of optimisation algorithms, a methodology had to be developed. A random problem-generator was developed, and the candidate algorithms were tried on a large sample of problems. The performance of each algorithm on each problem were then analysed statistically. They were compared pair-wise on two different measures: a paired t-test on the best fitness found by each algorithm on each problem, and a count of how many times one algorithm beat the other (regardless of by how much). The results on these measures were consistent on two different problem sets. The algorithms were ranked in the following order (best first): Genetic algorithm, simulated annealing, steepest descent, tabu search, random search. We take this to be at least a *prima facie* case for using evolutionary methods within the DBE.

These conclusions are discussed briefly in relation to the structure of the search space and, in particular, the distribution of local optima. Briefly, it is believed that tabu search, for example, performs best when the local optima form a single dense cluster, whereas steepest descent prefers widely distributed optima. It is unknown how (if at all) the distribution of optima affects simulated annealing and genetic algorithms, and indeed this is a subject of ongoing research in the field.

1.4 Mathematical background

The material in this chapter was developed during the first year of the project and is a semi-technical tutorial on theoretical aspects of genetic algorithms aimed at the non-specialist. It has been circulated amongst the Science partners of the project to provide them with an awareness of the main elements of the theory, and to show how some of these elements impact on issues of concern to the DBE project. It is to appear as a chapter in the book *Foundations of Learning Classifier Systems* edited by Larry Bull and Tim Kovacs (to be published by Springer).

The chapter begins by giving an overview of the basic theory of Simple Genetic Algorithms, as developed by Micheal Vose and co-workers (including J. Rowe). Fundamentally, genetic algorithms are Markov processes. That is, they are stochastic processes in which the state at one generation depends only on the state at the previous generation. There are two important types of Markov process, which have very different long-term behaviours. Ergodic Markov chains can access any state from any other state (at least, eventually), and in the long-term converge to a *stationary distribution* which determines the probability that each state will be visited. In contrast, other Markov chains have *absorbing states*. These are states which cannot be escaped from, once they have been reached. In the long-term, the system inevitably ends up in one of these states and stays there forever. The question of interest is: how long does it take before this happens? Genetic algorithms can exhibit either type of behaviour, depending on which forms of selection, crossover and mutation are used. This is an important issue for the design of the Evolutionary Environment. Do we want the population to converge to a particular solution, or would we rather it continue to explore the entire search space? This trade-off between exploration and exploitation is a key issue in the design of search algorithms of all kinds, and evolutionary ones in particular.

The transition matrix determining the probability of moving from one state to another is derived from the underlying replicator equation for the system. In a generational genetic algorithm, this is a discrete time equation, referred to as the Hardy-Weinberg model in population genetics (and the Nix-Vose model in genetic algorithm theory). This equation describes the expected next generation, given the current one. There is an important theorem which establishes that the *variance* of the next generation decreases in inverse proportion to the population size. Consequently, a large population will, with high probability, stay close to the expected trajectory. In the infinite population limit, the system becomes deterministic and is described directly by the replicator equation, which is therefore sometimes referred to as the "infinite population" model.

Of prime importance is the relationship between finite (stochastic) population behaviour and the trajectory of the infinite population. It has been established that finite populations tend to spend time in the vicinity of fixed-points of the deterministic system. It is also sometimes possible to find large regions of the state space in which a finite population will wander for long periods, before finally escaping to another such state. These regions are referred to as *metastable* states and the tendency of a finite population to inhabit them for a while before moving suddenly to another, gives rise to the phenomenon known as *punctuated equilibria* in the biology literature. This notion of metastability has been taken up by LSE in their study of systems "far from equilibrium" as a way of understanding how a system such as the DBE (or some other form of ecosystem) might maintain itself for a period of time in the face of increasing entropy.

The exact form of the replicator equation depends, of course, on the choice of operators that are to be used. The study of different operators and their properties is a major part of genetic algorithm theory. Selection is the process of emphasising high quality individuals in the population. In physics terms, it decreases the entropy of the population (as described in the work of ICL). There are several popular choices for this operator, including *proportional* selection (in which the probability of being selected is proportional to fitness) and *tournament* selection (in which a subset of individuals are selected randomly and the best one chosen). Proportional selection gives rise to linear equations, and is easier to deal with analytically. Tournament selection (based on sets of size two) is quadratic, which adds complexity to the mathematics. In any case, the effects of just applying selection repeatedly is to converge on the best solution in the original population. It is pure exploitation with no exploration.

Mutation, by contrast, is pure exploration. It tends to randomise the population, that is it increases entropy. If left to run by itself, mutation will always create a completely (uniform) random population. The choice of how to perform mutation depends largely on the representation employed for elements of the search space, and the underlying symmetries inherent in that representation. For example, for fixed-length binary strings, the usual mutation method is to flip each bit with a certain probability, referred to as the *mutation rate*. It is of interest that this mutation satisfies the following property:

$$\text{Prob}\left[x \text{ mutates to } y\right] = \text{Prob}\left[u \oplus x \text{ mutates to } u \oplus y\right]$$

for all strings u, x, y, where $\oplus$ indicates bit-wise exclusive-or. We say that mutation *commutes* with this operator. It turns out that this phenomenon occurs in a large number of cases, and it is examined further in the chapter on structural search spaces (see below).

Crossover is not so simple to describe in terms of entropy. Its effect on structural search spaces (such as fixed-length strings) is to *de-correlate* the components of the solution. If run by itself, it produces a population in *linkage equilibrium* in which the probability of finding a component in a particular solution is independent of the other components in that solution. This convergence result, for fixed-length binary strings, is a basic theorem of population genetics known as Geiringer's Theorem. It has recently been extended by Rowe and others to variable-length structures and trees (see below on variable-sized structures).

The replicator term for crossover is a quadratic operator, and complicates things considerably. However, it shares a similar property to mutation in that it typically commutes with an underlying symmetry of the search space. For example, in the case of fixed-length binary strings:

$$\text{Prob}\left[x, y \text{ cross to form } z\right] = \text{Prob}\left[u \oplus x, u \oplus y \text{ cross to form } u \oplus z\right]$$

for all u, x, y, z. This inherent symmetry enables the replicator equation to be simplified considerably, and progress made on determining its dynamics. This issue will also be considered further below.

After introducing these operators, we then consider them in various combinations. Firstly, the combination of proportional selection and mutation is analysed. Mathematically, the situation is relatively straightforward — there is a change of basis under which the dynamics becomes equivalent to proportional selection alone. There is a single globally attracting fixed-point, which

is a perturbation of the optimal population: the fixed-point is pushed away by mutation. We see here a balance between the two operators. Selection is trying to reduce entropy, driving the population to a uniform one, wheras mutation is increasing entropy and moving the population away. The end result is a balance between these two opposing forces.

Selection plus crossover is a little more complicated. All uniform populations are fixed-points of the system (and absorbing states of the Markov process), The asymptotically stable fixed-points of this type are characterised as those containing local optima in the search space. However, it is an open question as to whether or not there are any asymptotically stable *mixed* populations. Selection plus crossover plus mutation is, of course, the most complex of all, and there is much that is not known about this situation in general. progress can be made by considering particular classes of fitness function and using statistical mechanics techniques to simplify the equations.

The chapter then moves on to address several variations and extensions to the Simple Genetic Algorithm model. The first of these is the *steady state* model, in which only one individual in the population is processed at one time. The new individual is inserted back into the population in one of a number of possible ways. The two simplest are: replace a random element; and replace the worst element. It can be shown that if a random element is replaced, then the fixed-points of the system are identical to a generational model. Replacing the worst element is, however, the most popular choice, as it strengthens selection pressure and is, moreover, *elitist*, meaning that the best individual is guaranteed to survive. Elitist algorithms have the property that they converge (almost surely) to the optimum. Of course, the amount of time this takes is not necessarily efficient.

The extension to variable-sized strings is considered next, a subject of great relevance to the DBE project. In this case, the usual definitions of crossover and mutation have to be adapted. There are a number of commonly used operators:

Crossover Two strings are truncated at random places along their length. The left part of one of the strings is appended to the right part of the other string.

Grow mutation A string is truncated at some random point along its length. New symbols are then added according to a sequence of Bernoulli trials. That is, there is some probability q that a new symbol (chosen arbitrarily from the available alphabet) will be appended, and a probability of $1 - q$ that the growth process will stop. Growth continues until this latter event happens.

Full mutation A string is truncated at some random point along its length. A new random string of length D (which is a parameter to be set by the user) is appended.

There are a number of results proved by Rowe and others concerning the limiting size distribution for these operators (assuming they are run forever). Some of these results are presented in the chapter, and indicate the inherent biases of the different operators with respect to the size of structure they prefer. This could have some considerable influence on the choice of operator for an application.

When these operators are combined with selection, a commonly observed phenomenon is *bloat*: the tendency of the population to evolve bigger and bigger structures, without appreciable

gain in fitness. There have been a number of investigations into the causes of bloat, though it is still not fully understood. One contribution made in this paper was that the average size of individuals in the population can diverge even though the evolution of the population itself is converging. The issue of bloat is important for Genetic Programming in general and for the DBE in particular. There is a real danger that unnecessary (and potentially harmful) services will "hitch-hike" on other services, gaining fitness simply by association, leading to the equivalent of parasites and viruses within the system.

The chapter concludes by examining two situations in which the fitness of individuals can change with time. The first case is where the external environment changes. This could well happen in the DBE, for example, when a user changes requirements. A simplifying assumption of periodicity[1] enables the techniques developed previously to be applied in a straightforward manner. Non-periodic functions are potentially much harder to analyse. The other case of changing fitness, is where the fitness of an individual depends on the other elements of the population. This forms a co-evolutionary system. This is the case, for example, in a learning classifier system in which the fitness of one classifier depends strongly on what other classifiers are around which it can work with. A similar situation arises in the DBE when we consider the long-term evolution of a habitat: the long-term fitness of a service will depend on what others it has had the chance to combine with. These issues are further considered in the chapter on co-evolution and extinction (see below).

1.5 Co-evolution and extinction

Here we present some results of on-going research into co-evolutionary dynamics, in which the fitness of an individual at any time is a function of the current population. The basic model is a discrete-time version of the standard replicator equation of *evolutionary game theory*. Each individual in the population is assumed to encounter all the other members of the population in each generation. As a result of these interactions, there is an increase or decrease in fitness. The expected fitness of an individual is then a linear combination of these effects. That is, there is a matrix C, called the *game matrix* such that $C_{i,j}$ gives the effect on i of encountering j. If p_j is the probability of encountering j (that is, it is the proportion of j in the population) then the fitness of i is given by $\sum_j C_{i,j} p_j$. In other words, the fitness function is given by a vector $\boldsymbol{f} = C\boldsymbol{p}$, where the fitness of i is f_i.

The first part of the chapter extends this model to non-linear interactions, in which C now becomes a (potentially non-linear) map, and $\boldsymbol{f} = C(\boldsymbol{p})$. The fixed-points of evolution in this system can be calculated (as long as C is invertible) by

$$\boldsymbol{p} = C^{-1}(\lambda \mathbf{1})$$

where λ is a normalising factor and $\mathbf{1}$ is the vector containing all ones. An immediate corollary is that at any fixed-point, all individuals have identical fitness. This result is well-known in the case of linear games, but is here generalised to non-linear ones.

[1] This assumption is clearly unrealistic, but might open the way to applying a Fourier analysis to more complicated time dependences.

Some examples of linear and non-linear games are given, along with their fixed-point analysis. One consequence of the fixed-point theorem is that there are potentially many fixed-points. If the set of possible species is S, then for every subset $A \subseteq S$ there is (potentially) a distinct fixed-point, corresponding to the situation in which all species in A have become extinct. This observation is important in understanding the stochastic behaviour of finite populations. Since, as we have seen above, these tend to fluctuate around the infinite population trajectory, it is possible for a species to go extinct by "accident", when the predicted quantity of individuals of that species is small. Of course, once a species is extinct, it cannot come back. We therefore often observe a cascade of extinctions. One species is eliminated by some random fluctuation, removing it from the game. A new function C then comes into operation (a projection of the original one) with its own fixed-point which then drives the evolution. New fluctuations during this passage of evolution may cause further extinctions. One can often observe a *domino effect* in which one initial extinction can trigger a landslide of others.

The second part of this chapter is an empirical study of a multi-population co-evolutionary system. In particular we look at two populations in which the fitness function to be applied to one is a function of the state of the other population (and *vice versa*). A biological interpretation of this situation is the evolution of a so-called *arms race*. For example, the fitness of lions is a function of how quickly its prey can escape. Similarly, the fitness of antelope is a function of the lions ability to catch them. This system leads to the evolution of faster antelope, which in turn increases the selective pressure for faster lions. A positive feedback occurs, leading to ever increasing speeds in both species.

The arms-race scenario has been used in evolutionary computation to solve optimisation problems. The idea is to have one population which is a set of problems, and a second which is a set of solutions. As the solutions evolve to be more efficient, so the problems evolve to become harder. The fitness of a solution is thus a measure of its ability to solve the current set of problems. The fitness of a problem is the extent to which it evades solution. The idea is that the ensuing arms-race leads to harder problems, and more sophisticated solutions.

We consider a particular system which can be viewed as an abstraction of this process, and is a model of the long-term evolution of the DBE. We again use the set-cover problem as the basic structure. Suppose there is some global set of of requirements. A solution is characterised by the subset of requirements which it can meet. A problem, on the other hand, is characterised by the subset of requirements which it needs to be satisfied. At each time step, a random problem and solution are selected from their respective populations. If the solution can solve the problem then its fitness increases, and the corresponding problem's fitness decreases. Conversely, if the solution fails to solve the problem, it is the problem that gets an increase in fitness. The idea is that problems (or sets of requirements) put forward by users are either "solved" by particular services in the current habitat, in which case they can be removed, or they remain "unsolved", and pressure is exerted to drive the evolutionary process to form better solutions.

The fixed-point analysis developed earlier can be applied to this situation (extended to multiple populations). However, it turns out that in this case the fixed-points are not stable. In fact the ensuing evolutionary dynamics turn out to be highly chaotic (at least empirically), even for very low-dimensional situations. This observation has implications for finite population behaviour, since chaotic systems are known to be highly sensitive to small fluctuations. Consequently, even

large populations cannot track the infinite population trajectory for long, and a series of extinctions quickly follow.

An immediate consequence for the DBE project is that the evolutionary pressures exerted by changing user requirements have to be understood in detail. We have assumed, in this model, that they follow rather simple evolutionary dynamics: requirements that can be easily met will disappear quickly (as they are met quickly), whereas difficult sets of requirements will tend to dominate (as they remain unsatisfied requests from users). This is very simplistic, but already leads to potentially chaotic dynamics with the potential for mass extinctions caused even by very small fluctuations in the system.

1.6 Variable-sized structures

Geiringer's Theorem (1944) is a classic result in population genetics, and concerns the long-term limit of repeatedly applying crossover to a population of fixed-length strings. The result applies to strings over any finite alphabet, and a variety of crossovers — the crossovers defined by binary masks. The only restriction is that, given any two positions in the string, there must be a non-zero probability of selecting a mask which will split those positions. The standard one-point, two-point and uniform crossovers are compatible with this restriction.

The Theorem states that, whatever the initial population, repeated application of crossover will eventually lead to a limiting distribution over the strings. In this limit, the probability of finding a particular string $x_1 x_2 \ldots x_n$ depends only on the distribution of the alleles $x_1, x_2, \ldots$ in the initial population. Formally, if $p(x_1 x_2 \ldots x_n, t)$ is the probability of finding this string in the population at time t, and $p(x_k, t)$ is the proportion of strings carrying allele x_k at time t then

$$\lim_{t\to\infty} p(x_1 x_2 \ldots x_n, t) = \prod_{k=1}^{n} p(x_k, 0)$$

We say that crossover *de-correlates* the population, as the probability of finding an allele in a string is independent (in the limit) of what else is in the string. Selection, in contrast, tends to correlate the population as it is biased towards allele combinations which contribute to high fitness values.

This result has recently been generalised to the case of variable-length strings, under *homologous* crossover. This form of crossover is like one-point crossover, with the cut point being selected at a position less than the length of the shorter of the two strings. For example, given 00010 and 111110001, we select a cut point $0 \leq c < 5$. Choosing, say, $c = 3$ gives the result: 000110001. The extension of Geiringer's Theorem to this case gives the limiting distribution as follows:

$$\lim_{t\to\infty} p(x_1 x_2 \ldots x_n, t) = p(*^{n-1} x_n, 0) \prod_{k=1}^{n-1} \frac{p(*^{k-1} x_k \#, 0)}{p(*^k \#, 0)}$$

where $*$ is a "don't care" symbol that matches a single allele, and $\#$ stands for a string of arbitrary length. The interpretation of this result is that, if we chose a random string from the limiting

distribution, the probability of finding allele x_k in that string depends only on the number of copies of x_k out of all the strings of length at least k in the initial population.

Proving that this distribution is a fixed-point for homologous crossover is relatively straightforward. Proving that it is the limit is harder. Details are in the paper:

> Poli, R., Stephens, C.R., Wright, A.H. and Rowe, J.E. (2003) A schema theory based extension of Geiringer's theorem for linear GP and variable length GAs under homologous crossover. In Foundations of Genetic Algorithms. Vol. 7. De Jong, Poli and Rowe (eds). Morgan Kaufmann Publishers. Pages 45-62.

This is all by way of background to chapter 6, which extends them in two important directions. Notice first that the results quoted so far apply to the *infinite population model*. The work presented here extends these results to finite populations. Secondly, the results apply not just to variable-length strings, but to *trees* under homologous crossover (where the cut point exists in the *common region* of the two parent trees).

The details of the extension to finite populations are presented in:

> Mitavskiy B. (2004). A Generalization of Geiringer Theorem for a wide class of evolutionary search algorithms. Submitted to Evolutionary Computation.

In essence, the idea is to consider the following model of the Markov chain. Suppose we have a set of (deterministic) maps, from populations to populations (of a given size). Suppose further, that there is a probability distribution defined over this set of maps. Our Markov process then proceeds from an initial population by repeatedly selecting a map (according to the distribution) and applying it to the current population, to give the next generation. We then investigate properties of the stationary distribution of this process. For example, if the set of maps generates a group, then one can use the following theorem: the stationary distribution of a random walk on a group is uniform. That is, each state is equally likely to appear in the long-run. For if G is a group and μ a probability distribution over G, then the probability to move from state j to state i is $\sum_{g\in G}\mu(g)[g(j)=i]$. The square brackets indicate a 1 if the enclosed statement is true and 0 otherwise. Then if we start with the uniform distribution $\pi = (1/N, 1/N, \ldots, 1/N)$ (where N is the number of states), the probability of seeing state i at the next time-step is

$$\begin{aligned}
\sum_j \sum_{g\in G} \mu(g)[g(j)=i]\pi_j &= \frac{1}{N}\sum_j \sum_{g\in G}\mu(g)[g(j)=i] \\
&= \frac{1}{N}\sum_{g\in G}\mu(g)\sum_j [g(j)=i] \\
&= \frac{1}{N}\sum_{g\in G}\mu(g)\sum_j [j=g^{-1}(i)] \\
&= \frac{1}{N}\sum_{g\in G}\mu(g) \\
&= \frac{1}{N}
\end{aligned}$$

This shows that the uniform distribution is a fixed-point. If the group action is also transitive, and $\mu(g)$ is non-zero (at least for some subset that generates G), then is it also the unique stationary distribution.

Chapter 6 applies this simple yet powerful observation to the case of populations of trees under homologous crossover. Much of the technical detail is in defining trees and tree-schemata, and establishing that homologous crossovers fit within the framework just described. Once this is done, however, the Geiringer Theorem for trees can be deduced as a direct corollary.

This work speaks directly to the effects of applying crossover repeatedly to a population. Of course, in practice, we also have a selection pressure. However, even a single application of crossover takes the population very close to the Geiringer limit. Understanding this limit tells us something about the bias imposed by crossover on the search process. The net result will be a balance between this force, and the pressure of selection towards correlated populations. When we design specific crossover operators for the structures in the DBE habitats (representing service-chains, for example), these results will help us understand the effects that they have on the overall search, and the kinds of populations which they favour.

1.7 Further work

There is much further work to be done in developing the theoretical ideas presented here. In this regard we welcome the encouragement of the project reviewers to conduct scientific research which contributes directly to the DBE project, and which progresses "the state of the art in evolutionary systems regardless of their use in the DBE context." The work planned for the second phase of the project will continue to be of both kinds.

There are three main areas in which this work will be extended. The first is in *landscape analysis*. It is clear that there is something about the fitness landscape of the weighted set-cover problem (and, by extension, about the "DBE problem") which makes it amenable to solution by an evolutionary system (rather than, say, tabu search). We have hinted that this might have something to do with the statistical distribution of local optima in the search space. This hypothesis needs to be further formalised and tested. This is an area of general interest in the research community, but also of particular interest to the DBE project, with strong overlaps with the work of the other Science partners.

The study of landscape structure also has potential impact in the development of the theory of operators in terms of their relationship with group symmetries. One might wonder where such symmetries come from, and a natural response would be to look at the symmetries and structure of the underlying landscape. A landscape can be defined on a search space by setting up a *neighbourhood* structure. For each point, we assign a set of neighbours. This can usually be done in a natural way, and provides the basic mechanism for implementing local search algorithms. For our purposes, it also defines a (possibly directed) graph, with the elements of the search space as vertices, and edges between neighbours. One could then analyse the structure of the graph by looking at its *automorphisms*, that is, permutations of the vertex set which preserve the neighbourhood structure. By doing this, one is acknowledging that the representation used for the points in the search space is rather arbitrary: what is of significance is the relationship

between the points. The set of automorphisms of a graph forms a natural group action on the search space (since the composition of two automorphisms is also an automorphism), and is therefore a candidate for the foundations of our theory.

The second area of research will then be to continue extending the existing theory of evolutionary algorithms to variable-sized structures, including incorporating the effects of selection pressure into our new Geiringer Theorems. We have already made some progress in this direction, but much more remains to be done. There are a number of technical problems that have to be addressed, including finding the best mathematical representation of populations over a potentially infinite search space. We envisage that this work, in addition to introducing new fundamental results, will be highly relevant to the DBE project in its second phase as it is thought that the representation of arbitrary composite services will require (at least) variable-length strings, and potentially trees (for hierarchies of services) and networks (for complex workflows).

The work on co-evolution and dynamic environments is the third subject of our future research. It is clear from the empirical results that have been presented that a simple fixed-point analysis is insufficient to characterise the dynamics of even simple co-evolving systems. We need to study their stability, and the conditions under which chaotic dynamics arise. As well as being an important topic in its own right, it may impact the DBE project in the long-term, as we try to take into account the effects of user changing their requirements, and solutions adapting in response to these changes. In particular, it may be necessary to come up with practical ways to control the on-set of chaos and to prevent sudden extinctions due to small fluctuations.

The DBE project has, to date, provided an excellent test-bed for the development of evolutionary computation theory and, we hope, the other partners in the project will benefit from the insights that are being generated. We plan to continue this work, in collaboration with the other Science partners, into the next phase of the project and beyond.

Chapter 2

An initial view

> This chapter was circulated as a project discussion document in April 2004, under the title: “The evolution of high-level software”.

This paper is a discussion document on what it might mean for (relatively) high-level software components to evolve autonomously. Current evolutionary algorithms for evolving programs (so-called Genetic Programming) suffer from a number of weaknesses. Two critical ones are:

- While being moderately successful at evolving simple programs, it seems very difficult to get them to scale up and evolve programs with any kind of complexity.
- The fitness of a program is normally given by a measure of how accurately it computes a given function, as represented by a set of input/output pairs. There is no guarantee therefore that the evolved program actually does the intended computation.

These issues are particularly important if we want to evolve high-level, complex structured software.

An abstract model

Let’s first start with an abstract model of a system of evolving software. We imagine that there is some underlying set of components. Each component has two properties: a piece of executable code (perhaps stored on some remote machine via a proxy) and a guarantee that specifies something about what the code does.

Examples:

- each component is a line of code in some programming language
- each component executes some function within a spreadsheet
- each component provides some kind of web service
- each component provides the front-end to a “real-world” service

Now the problem is that we want to find a collection of components that satisfies some given specification. It may be possible to do this using some optimisation algorithm. But it may also arise as the product of an evolutionary process. The DNA in this case would correspond to a structure which specifies which components are to be used and the relationship between them. Mutations may occur by switching some components into and out of the structure, or by changing the relationships. Crossover (recombination) may occur by combining elements of two or more structures into a new structure.

In this way, a piece of DNA has an interpretation (or phenotype) as a large complex-structured piece of software which can execute some task. However, there is more. Because each component also carries a guarantee, or specification, of what it does, it ought to be possible to use some software validation technique to verify whether or not the overall collection satisfies the required specification. Indeed, this can be the heart of the fitness function which drives the evolutionary process.

Structure

Collections of software components may be collected together in different structures. If they are providing real-world services, then maybe all we require is a set of such services. If they are lines of code from a programming language, then the components will be ordered as a sequence. A collection of web-services may have a tree structure. And functions in a spreadsheet may be related by a directed network of inputs and outputs.

The kind of structure that is being used makes a big difference to a number of aspects of evolution. A set may be represented easily by a binary string, in which the bits indicate whether or not a component is in the set. Mutation is done bitwise and crossover by binary masks. The result will look like a fairly standard genetic algorithm. Sequences of code and services may be represented by variable length strings over some alphabet. The dynamics of evolutionary processes over such strings has been studied a little in the last few years. There are a number of ways of defining mutation and crossover, each introducing their own biases into the kinds of program (e.g. in terms of size or "scale") that will be sampled most often. Standard genetic programming usually works on tree structures, although there has been recent work done on networks (e.g. "Cartesian Genetic Programming"). Theory here is more sparse but some progress has been made. Of particular interest is the phenomenon in which trees tend to grow bigger and bigger as the evolution goes on, without necessarily any corresponding increase in fitness — a phenomenon called "bloat". There are some recent results which suggest methods for controlling this.

Fitness

We have already suggested that the prime driver of the evolutionary process should be the extent to which a candidate collection can verifiably satisfy the specified requirements. It may be that this can be measured probabilistically, or perhaps some kind of theorem-proving can be used to validate the system. However, there will also be other pressures on fitness. For example, one may seek the most parsimonious solution to a problem (one that provides exactly the specified

features and no more), or the cheapest solution, or one with a good "reputation". Some aspects of fitness will be implicit in the evolutionary process (e.g. collections which are often used may gain more fitness) while others will require explicit measures (e.g. price, or user satisfaction).

One way to handle this multiplicity of fitness values (some qualitative!) is to explicitly recognise the multi-objective nature of the optimisation problem. In this way, we are seeking not the single best solution, but a range of possible trade-offs that can be made most optimally. The set of solutions for which there exist no better trade-offs, is called the Pareto-set. Evolutionary techniques have been adapted to solve such problems with considerable success. The main point is that selection has to be driven not by an absolute value of fitness, but rather by a notion of what it means for one solution to be better that another. We say one solution *dominates* another if it is better in at least one respect, and no worse in any of the others.

Another way in which multiple objectives will arise is from the fact that there will be, at any time, many different specifications needing to be satisfied. It would be a mistake to try to find one solution which attempts to meet all the requirements. Instead, each specification corresponds to a different niche in the evolving ecosystem.

Modules and hierarchical dynamics

In order to construct ever more complex software solutions it seems clear that some form of modularity is needed. This has been attempted by a number of researchers with mixed results. The basic idea would be to make successful solutions at one time step become individual components in the future. That is, it becomes possible to plug-in an entire package into an evolving collection. All components must provide a guarantee as to what it is they compute (or what service they provide). For a module (composite system) this would derive from the specification it was previously evolved to satisfy. Therefore it will be able to take part in further software validation.

In this way, the system can become open-ended (or "evolvable"). As the system receives more and more sophisticated requests, so more and more complex software collections become available in the population to be used as modular components.

Evolution of specifications

One final speculation:

It has been suggested that the executable code (or service or whatever) may exist at some remote site. The evolutionary process only requires the existence of the guarantees to work. The actual underlying code or service only comes into play once the whole package has been assembled. There is therefore no strict requirement for the underlying code to actually exist until this stage. That is, evolution could take place entirely at the level of the specifications. Software component providers would only need to actually supply the code once there is a demand for it. That is, when one of their specifications (or guarantees) has been used in the construction of a software solution which meets a user's requirements. The whole system then becomes more like an evolving futures market.

Chapter 3

Alternative optimisation algorithms

This chapter has been circulated to the Science partners under the title: "A comparison of heuristic search techniques on the weighted set cover problem".

3.1 Digital Business Ecosystems and the Set Cover Problem

A *digital business ecosystem* (DBE) is an environment in which a user's requirements are to be met by combining a number of services together into a *service chain*. It is *digital* because this all happens in software (making use of proxies to stand in for real-world services); it helps address the *business* needs of, especially, small to medium size enterprises; and it forms an *ecosystem* in the sense that the environment will change, both as a function of the changing user requirements, and as a function of the available services.

There is also an implication that the automatic assembly of the required service chains is an *evolutionary* process, driven by the needs of the user. Consequently, genetic algorithms are being studied as appropriate optimisation tools for solving this problem. However, it is prudent to step back and consider whether this is necessarily the best choice of optimisation algorithm. In this paper, we present some empirical evidence that compares a genetic algorithm with four other heuristic techniques on a collection of problems, based on an abstraction of the DBE idea. That abstraction is the *weighted set cover problem*.

We imagine that the user has presented to the system a set of requirements, S. We also imagine that there is a current set of services available C. Each service in C can satisfy a subset of the requirements in S. Each service also has a cost associated with it. The problem is to find a collection of services from C that jointly satisfy all the requirements of S, at minimum cost.

3.2 Methodology

We seek to compare five different heuristic techniques on this class of problem. They are:

- genetic algorithms
- simulated annealing

- steepest descent (with restarts)
- tabu search
- random search

A common way to proceed in this investigation would be to generate a small set of problems, and to run each algorithm many times on each. A t-test conducted on each pair of algorithms would determine if one algorithm had performed significantly better (on average) on each problem. However, there are difficulties with this approach. Firstly, due to time constraints, only a very small number of problems can usually be used. It is therefore unclear how representative these problems are. Secondly, we compare the performance of the algorithms averaged over many runs. In practice, you are allowed one attempt at solving the problem at hand, and the average performance on a single problem may not be indicative of typical behaviour. Thirdly, with a small set of problems, it is possible to fine-tune the parameters of an algorithm to make it perform well on that small set. It is never clear whether or not one algorithm has been better tuned than another, and also whether the tuned parameters would give acceptable performance on problems other than those in the given set.

Consequently, we prefer an alternative methodology. We use an automatic problem generator to generate many random instances of the problem class. For each instance, each of the search algorithms is run *once* and the best result found during that run is recorded. The algorithms are then compared pair-wise using a *paired* t-test, to indicate whether or not differences in performance are significant. A paired t-test is required, since some of the generated problems may be much harder than others, and we need to compare the algorithms on each problem, like for like. In effect, the question we are asking is:

> Given a random instance of the problem class, if you can choose between two algorithms to run once only on it, which should you chose?

3.3 The Problem-Generation Algorithm

Instances of the weighted set cover problem are generated as follows. First, costs are assigned randomly to each element in S. These represent the cost of achieving each of the user's requirements. In our experiments, these are integers in the range $1 \ldots 100$. Second, the elements of C (the set of available services) are generated randomly as follows. A probability p is chosen uniformly at random in the range $0 \leq p < 0.5$. The service is generated randomly using p as the probability that any given requirement in S is satisfied by the service. A new value of p is generated for each service, and therefore represents the proportion of requirements from S that the service will satisfy. The cost of a service (that is, an element of C) is then the sum of the costs associated with each requirement that it can meet.[1] The generation of a problem instance is therefore parameterised by the size of the sets S and C.

[1] This direct relationship is rather a simplifying assumption.

We represent attempts at solving the problem (that is, service chains) as subsets of C (that is, a subset of services). We define an objective function in two stages. If the service chain satisfies all of the requirements in S, the value of the objective function is simply the total cost of the all the services in the chain. If, however, it does not meet all the requirements, then in addition to the cost of all the services, we also add penalty, comprising two terms: a constant, reflecting the maximum possible cost of a service chain (guaranteeing that its fitness is worse than any service chain which does cover all the requirements), and a term proportional to the number of missing requirements. Thus, if $X \subseteq C$ is a service chain:

$$f(X) = \begin{cases} \sum_{i \in X} c(i) & \text{if } X \text{ covers } S \\ \sum_{i \in X} c(i) + 100m + \sum_{z \in C} c(z) & \text{otherwise} \end{cases}$$

where $c(i)$ is the cost of service i and m is the number of missing requirements. It can be seen that X meets all the requirements if and only if $f(X) < \sum_{z \in C} c(z)$.

3.4 The Heuristic Search Algorithms

The five heuristic algorithms tested in these experiments are now described in detail. An important point is that all the algorithms are completely standard, and, with the exception of simulated annealing (see below), no attempt has been made to fine-tune the parameters to suit the problem. Subsets of C (service chains) are represented using binary strings of length $|C|$, indicating which services are in the chain.

Genetic algorithm

We use a standard *steady-state* genetic algorithm, with population size equal to the size of the problem $|C|$. The mutation rate is $1/|C|$ and uniform crossover is employed with probability 1. Selection is by using the binary tournament method. The overall algorithm is as follows:

1. Initialise the population randomly.
2. Select two elements of the population randomly — keep the best one.
3. Select two more elements of the population randomly — keep the best one.
4. Crossover these chosen two elements using uniform crossover
5. Mutate the result
6. Insert the result into the population, replacing the worst element.
7. Go to 2.

Simulated annealing

We follow a standard simulated annealing algorithm, that makes use of parameters T (the temperature) and α (the cooling rate).

1. Initialise the temperature T.
2. Generate a random solution x.
3. Randomly change one bit of the current solution to give a proposed solution y.
4. If the result is an improvement, $f(y) < f(x)$, accept it as the new current solution. That is, let $x := y$.
5. Otherwise, let $x := y$ with probability $\exp(-\Delta/T)$, where $\Delta = f(y) - f(x)$ is the proposed increase (or *jump*) in the objective value.
6. Let $T := \alpha T$.
7. Go to 3.

The difficulty with setting the initial temperature and the cooling rate, is that they are problem dependent. That is, you have to tune it to suit the problem under consideration. In the first set of experiments we conducted, the values $|S| = 500, |C| = 50$ were used, and we tuned the parameters to these values as follows. Firstly, experiments indicated that poor solutions had objective values of around 300,000, whereas good solutions were about 80000. In the early stages of the algorithm, one want to accept big jumps with relatively high probability. Given the data, we decided that initially a jump of size 10,000 should be accepted with probability 0.5. This leads to an initial temperature of 14,427. At the end of the run, however, we want even small jumps to be accepted with only small probability. We therefore required that jumps of size 1000 should be accepted with probability 0.01. This leads to a final temperature of 217. Since each algorithm was to be allowed 10,000 function evaluations, this corresponds to a cooling rate of $\alpha = 0.99958$.

The fact that these parameters have been set to correspond to knowledge about the problem and the value of good and bad solutions, should give simulated annealing an advantage over the other algorithms, which have no such access to information about the problem. To test for this, we ran a second set of experiments with problems generated using the values $|S| = 700, |C| = 70$. However, we left the values of α and the initial temperature *the same* as in the first experiment. These values are no longer tuned to the problem, and some degradation in performance is to be expected, as indeed was found to be the case (see below).

Steepest descent (with restarts)

This simple algorithm involves checking all the Hamming-one neighbours of the current solution and taking the best one, if it is an improvement on the current solution. If there are no improving neighbours, the search starts again with a new random solution.

1. Generate a random solution.
2. Check all neighbours of current solution.
3. If the best neighbour is better than the current solution, accept it as the new current solution and go to 2.
4. Otherwise, go to 1.

This algorithm is expected to perform well when local optima are spread out in the search space. This is in contrast to tabu search (below).

Tabu search

Tabu search generates all the neighbours of the current point and accepts the best one, regardless of whether it is better or worse than the current point. This enables the search to walk away from a local optimum. However, whenever a move is made, it is stored in a memory for a certain amount of time (the *tabu tenure* — we used a value of 10). The same move is disallowed while it remains in the memory. This is to stop the search process walking back into the most recently visited optimum. An exception to this rule is if the move would generate a solution better than any previously seen, in which case it is allowed.

1. Initialise memory to be empty.
2. Generate a random solution
3. Check all the neighbours of the current solution.
4. If the best neighbour is better than any previously visited solution, accept it as the current solution.
5. Otherwise, accept the best neighbour that is not forbidden (that is, does not correspond to a move that is currently in memory).
6. Add the move (that is, the bit position that was flipped) to the memory.
7. Remove anything from memory that has been there longer than the tenure.
8. Go to 3.

Unlike steepest descent (with restarts), tabu search performs best when the local optima are clustered close together, so it can easily walk from one to the other.

Random search

This is the simplest algorithm of all. It is included to provide a benchmark.

1. Generate a random solution.
2. Go to 1.

3.5 Results

As explained above, we conducted two sets of experiments. The first used the values $|S| = 500, |C| = 50$ and the second set had $|S| = 700, |C| = 70$. In both cases, a set of 100 problems was generated, and each algorithm run once on each problem. The algorithms were each allowed a maximum of 10,000 objective function evaluations, and reported the best result they found. The algorithms were then compared pairwise, using a paired t-test. This enabled a ranking of algorithms to be established, using the 95% significance level.

For the first set of experiments, the genetic algorithm and simulated annealing were not significantly different. However, they both performed significantly better than the other three algorithms. Steepest descent was significantly better than tabu or random search. These last two algorithms were not significantly different. The average fitness for each algorithm, over the set of problems, was as follows:

Genetic algorithm	76942.56
Simulated annealing	76627.43
Steepest descent	77923.86
Tabu search	100747.04
Random search	98097.7

The results of the pair-wise t-test are as follows (showing the probability that the differences are *not* significant):

	Genetic algorithm	Simulated annealing	Steepest descent	Tabu search
Simulated annealing	0.552			
Steepest descent	0.061	0.0075		
Tabu search	0.0004	0.0004	0.0007	
Random search	10^{-56}	10^{-60}	10^{-62}	0.690

To help separate out the algorithms further, we also looked at the number of times one algorithm beat another. For example, tabu search beat random search 91 times out of 100. Simulated annealing beat the genetic algorithm 55 times out of 100. This latter result is not significant at the 95% level, however. The results are shown below (indicating the number of times the algorithm in the column beat the algorithm in the row):

	Genetic algorithm	Simulated annealing	Steepest descent	Tabu search
Simulated annealing	45			
Steepest descent	59	64		
Tabu search	80	83	78	
Random search	100	100	100	91

In the second set of experiments (when simulated annealing is no longer so well tuned for the problem), the algorithms could be clearly ranked in order, by both the above methods. The

order (from best to worst, by average fitness) is:

Genetic algorithm	113953.17
Simulated annealing	117224.95
Steepest descent	119183.46
Tabu search	122495.21
Random search	169666.23

The results of the pair-wise t-test show that all the differences are significant:

	Genetic algorithm	Simulated annealing	Steepest descent	Tabu search
Simulated annealing	10^{-5}			
Steepest descent	10^{-12}	0.0049		
Tabu search	10^{-15}	10^{-8}	10^{-5}	
Random search	10^{-73}	10^{-70}	10^{-68}	10^{-59}

The matrix of how many times one algorithm beat another is:

	Genetic algorithm	Simulated annealing	Steepest descent	Tabu search
Simulated annealing	67			
Steepest descent	77	61		
Tabu search	81	72	60	
Random search	100	100	100	100

All results are significant (at the 95% level or better).

3.6 Discussion

It is of interest that the genetic algorithm came out best (or joint best) in both experiments. It is significant that simulated annealing, when moved to a set of problems for which it is not fine-tuned, degrades in performance. One should only contemplate using it if one has accurate estimates of the range of solution values in the set of problems to be encountered. The fact that steepest descent beat tabu search indicates that the local optima in this class of problem are likely to be spread out. This is also an indication of why a population-based method is effective. Tabu search performs very badly on the first set of experiments. This is because it occasionally gets stuck in a cluster of optima that do not form covers of S, and therefore the value it returns includes the high penalty factor. It is, however, more likely to beat random search. This is reinforced in the second set of experiments, where tabu search always locates a covering service chain and, in this case, performs significantly better than random search.

3.7 Conclusions

Of course, there are many other experiments of this kind that could be tried. One could generate problems using different parameter settings for $|S|$ and $|C|$. One could try to generate more

realistic problems. The parameters of the algorithms could be experimented with and fine-tuned. However, given that the weighted set cover problem is an abstraction of an actual DBE, and that we currently do not know the exact details of how a DBE problem will be specified, it is unlikely that we would learn much more than has been shown above. We therefore present these results as a *prima facie* case for the use of evolutionary techniques in addressing the DBE problem, with the added observation that simulated annealing might also be made to work, provided the relevant parameters can be estimated sufficiently accurately.

An important extension to this work will be to repeat these experiments using a more realistic representation of services and service-chains. Once BML is sufficiently well-defined, we plan to repeat this work to continue to investigate the efficiency of evolutionary search methods for this class of problem.

Chapter 4

Mathematical background

This chapter is to be published in the book *Foundations of learning classifier systems* (Larry Bull and Tim Kovacs (eds.)), Springer 2005, under the title: "Population dynamics of genetic algorithms".

4.1 Introduction

The theory of evolutionary algorithms has developed significantly in the last few years. A variety of techniques and perspectives have been brought to bear on the analysis and understanding of these algorithms. However, it is fair to say that we are still some way away from a coherent theory that explains and predicts behaviour, and can give guidance to applied practitioners. Theory has so far developed in a fragmented, piecemeal fashion, with different researchers applying their own perspectives, and using tools with which they are familiar. This is beginning to change, as the research community develops and individual insights become shared. Consequently, the work presented in this chapter is a somewhat biased selection of results. However, I hope that other researchers will appreciate this material, even if they would themselves have concentrated on a different approach. Readers who are interested in a survey of current theory are referred to the books *Genetic algorithms: principles and perspectives* [15] and *Theoretical aspects of evolutionary computation* [9].

We begin, then, by considering the basic framework for studying genetic algorithms, laid out by Michael Vose in his book *The Simple Genetic Algorithm* [25]. We will introduce only the basic concepts: other theoretical approaches map easily onto this framework at this level. Genetic algorithms are Markov processes, and we describe, in quite general terms, how they may be described as such, as well as their relationship to the so-called infinite population model. We will then concentrate on a particular example: the "simple" genetic algorithm, comprising proportional selection, mutation and crossover (by masks) acting on fixed-length strings (over some alphabet). Some results relating to the fixed-points of these operators (in various combinations) will be described, as well as the variant known as *genepool* crossover, which has recently been investigated.

The second half of the chapter then looks at some extensions to the basic model. Firstly,

we will look at the possibility of having variable-sized structures in the search space. This kind of thing arises, for example, when considering rules (in which the action part might be of arbitrary length), grammars (to represent developmental encodings, for example), and programs (as in Genetic Programming). Much of the theory here has been developed by Riccardo Poli and colleagues [10]. We will relate a few results which fit nicely into the Vose framework.

Secondly, we will consider what happens when the fitness function (or the *environment*) changes with time [27, 3]. A simple case is when the fitness function is periodic, in which case the fixed-point analysis for the stationary case generalises in a straightforward way. Another situation in which the fitness changes with time, is when it is, in fact, a function of the population itself. This leads us to our third extension: co-evolutionary systems. Such algorithms have been used, for example, in cooperative problem solving with parallel populations, as well as with learning classifier systems, in which the fitness of a rule depends on the context in which it is used. Such systems have been studied extensively in theoretical biology using evolutionary game theory [6]. We present a simple discrete-time version, which maps directly into our mathematical framework.

Clearly, there is a lot of theoretical work that will not be covered in this chapter. I would like to mention some, and give pointers to the interested reader. Staying within the framework developed by Michael Vose, there is a considerable amount of more advanced material. Some of this is covered in Vose's book. More recent work by Vose, and collaborators, generalises this approach to arbitrary finite search spaces, with particular emphasis on algebraic properties of the genetic operators [20, 21]. Related work has been done by Christopher Stephens, and co-workers [24].

In parallel to this approach, are a number of models of specific systems which make use of techniques from statistical physics [23, 1]. These techniques are useful in helping to understand the effects of finite population sizes on the underlying dynamics. A further parallel development is the application of techniques from algorithmic analysis to evolutionary algorithms applied to different optimisation problem classes [8]. These consider evolutionary algorithms as being "black-box" function optimisers, and ask what the expected running time is to finding the optimal solution. Finally, in the case when the search space comprises real Euclidean space (that is, the individuals are vectors of real numbers), a considerable amount of impressive theoretical work has been done [2]. Readers who are keen to understand the current state of the art in evolutionary computation theory are strongly advised to study all these areas.

4.2 Genetic algorithms as Markov processes

We will be considering a generational genetic algorithm, in which, at each generation, the entire population is updated.[1] If we are working, as is usual, with a fixed population size, then the *state* of the algorithm at a given time step is simply the current population. The behaviour of a genetic algorithm can therefore be traced through time as a (random) sequence of populations. Of course, the population that we see at any one generation depends rather heavily on the population

[1] Some of the theory has been extended to *steady-state* algorithms — see [28].

of the previous generation, and also on the particular genetic operators and fitness function that were applied to that population. In fact, given the knowledge of these things, the probability of obtaining any particular population in the next generation is completely determined. A genetic algorithm is therefore an instance of a *Markov process*, since the state at any time step depends only on the previous time step. If the search space is finite (or countable), then it is a Markov chain.

Markov chains can be characterised by their corresponding *transition matrix*. This matrix contains the probabilities that the chain will move from one state j to another i as follows:

$$Q_{i,j} = \text{Prob}\,[i|j]$$

(that is, the probability of state i given state j).

Markov chains come in various types (see [7] for an introduction to the theory). The first important type (for our purposes) is when the chain is *irreducible*. This means that given any two states (that is, populations), there is always a non-zero probability of going from one to the other in a finite number of generations. Typically, this happens in a genetic algorithm if there is mutation present: there is always a chance (however small) of all the strings of one population mutating to those of another. One of the key properties of an irreducible Markov chain is that it visits every possible state infinitely often. Some states may be much more likely to occur than others, however, and we would like to be able to characterise these states. One characterisation comes from the transition matrix: the vector $\boldsymbol{v}$ that satisfies the equation

$$Q\boldsymbol{v} = \boldsymbol{v}$$

contains the probabilities of seeing each state over an infinitely long run. That is, state k will occur with frequency v_k if the algorithm is run for long enough. $\boldsymbol{v}$ is referred to as the *stationary distribution*.

If a Markov chain is not irreducible, then it might have *absorbing states*. These are states which, should the process ever arrive in one of them, it will remain stuck there forever. This is typically the case for genetic algorithms without mutation. A subset of states may also be absorbing in the sense that, having arrived at one state in the subset, the system remains forever within that subset (even though it might move around within it). An example here would be a genetic algorithm with mutation, but also with elitism (so that the best individual of a population is preserved to the next generation untouched). It is fairly straightforward to see that the fitness of the best of the population cannot decrease, due to the elitism, and in fact that this fitness will converge to the optimum (see [22] for a formal proof). This means that the algorithm is not irreducible, since it can never move to a population with a worse best fitness. Once it finds the optimum, it can never lose it. However, the rest of the population is free to be mutated into any other individual. Therefore, the set of all populations containing the optimum comprises an absorbing subset.

When the system has absorbing states, we know immediately, of course, where the system will end up (if run for long enough). The question of interest is: how long will this take. Theoretically, this question can also be answered by examining the transition matrix. However, as

with finding the stationary distribution, this is intractable in practice, as the number of populations associated with a genetic algorithm is very large. Approximate methods can sometimes be employed here (as in the work on analysing the time complexity of evolutionary algorithms [8]).

In the following section, we will show how the transition probabilities of the Markov process are related to the underlying operators of selection, mutation and crossover. First, we will describe the general setting: the *random heuristic search* framework of Michael Vose.

To start with, we need a way to describe populations (which are the states of the Markov chain) mathematically. We will represent a population with a corresponding *population vector*

$$\boldsymbol{p} = (p_0, p_1, \ldots, p_{n-1})$$

where p_k represents the proportion of item k in the population, and n is the size of the search space. Notice that we associate the search space with the set $\Omega = \{0, 1, \ldots, n-1\}$ by applying some arbitrary ordering. If the population size is N, then we can find the number of copies of item k simply by multiplying: Np_k. Population vectors are a subset of the following set, the *simplex*:

$$\Lambda = \left\{ \boldsymbol{x} \in \mathbb{R}^n \,|\, \sum_k x_k = 1 \text{ and } x_k \geq 0 \text{ for all } k \right\}$$

We can therefore think of the genetic algorithm as mapping out a random sequence of points in the simplex. This sequence will arise in the following way. Any element of the search space has a certain probability of being in the next generation, given the current population. This probability depends on the correct combination of selection, mutation and crossover happening, in just the right way, so as to produce the element under consideration. Let us write $\mathcal{G}(\boldsymbol{p})_k$ to be the probability that item k is produced by the genetic operators, starting with population vector $\boldsymbol{p}$. If we can write down these probabilities for all elements $k \in \Omega$ then we define a map:

$$\mathcal{G} : \Lambda \to \Lambda$$

Notice that Λ is here serving a dual purpose of representing probability distributions over the search space Ω.

The random process of the genetic algorithm is now exactly equivalent to the following:

1. Start with an initial population vector $\boldsymbol{p}$.
2. Calculate the probability distribution $\mathcal{G}(\boldsymbol{p})$.
3. Sample this distribution N times (with replacement) to form the next population $\boldsymbol{q}$
4. Set $\boldsymbol{p} := \boldsymbol{q}$ and go to 2.

The function $\mathcal{G}$ is referred to as the *heuristic* of the search process, and has to be appropriately defined to take into account the effects of the genetic operators (see the following section).

Even at this level of generality (that is, without going into the details of $\mathcal{G}$) it is possible to say something about the behaviour of the system. In the first place, we know that, given our current

population, the next will be chosen according to some probability distribution over all possible populations (of size N). These probabilities form the contents of the Markov transition matrix, which we can write down as follows (theorem 3.4 of [25]).

$$\text{Prob}\,[\boldsymbol{q}|\boldsymbol{p}] = N! \prod_{k\in\Omega} \frac{(\mathcal{G}(\boldsymbol{p})_k)^{Nq_k}}{(Nq_k)!}$$

We can ask what the *expected* next population is (that is, the mean of this distribution). Theorem 3.3 of [25] tells us that if the current population vector is $\boldsymbol{p}$ then the expected next population is in fact simply $\mathcal{G}(\boldsymbol{p})$. In other words, the map $\mathcal{G}$ not only tells us the probability distribution from which the next generation is sampled, it also tells us the average result, over all possible populations.

Further, we can also ask what the *variance* of the distribution is. Theorem 3.5 of [25] tell us that it is:

$$\frac{1 - \mathcal{G}(\boldsymbol{p})^T \mathcal{G}(\boldsymbol{p})}{N}$$

(where $\boldsymbol{x}^T$ indicates the transpose of the column vector $\boldsymbol{x}$). What is of interest here is that the variance decreases as the population size N increases. In other words, as the population size gets bigger, so it becomes more and more likely that the actual next population is very close to the expected next population. This has led to the function $\mathcal{G}$ being referred to as the *infinite population model*, since in the limit as $N \to \infty$ the next generation actually is $\mathcal{G}$. Moreover, this is true for any *finite* number of time steps. It is, of course, not true that the random process follows the deterministic sequence $\boldsymbol{p}, \mathcal{G}(\boldsymbol{p}), \mathcal{G}^2(\boldsymbol{p}), \ldots$ forever, since if the Markov chain is irreducible, then all populations will be visited infinitely often, whereas the deterministic sequence may converge to some fixed-point of $\mathcal{G}$.

Actually, it turns out that the fixed-points of $\mathcal{G}$ do have something to say about the long-term behaviour of the Markov process, as long as $\mathcal{G}$ satisfies certain technical conditions (which are usually satisfied in the case of the simple genetic algorithm). Put simply, the genetic algorithm likes to spend its time in populations which (when considered as vectors) are near fixed-points of $\mathcal{G}$. The long-term behaviour of the system is characterised by relatively stable periods near such points (sometimes referred to as *metastable states*) followed by rapid transitions to other such states. This gives rise to a picture of *punctuated equilibria*, which is described in more technical detail in chapters 13 and 14 of Vose's book. In terms of the Markov chain, we would say that states near fixed-points tend to have higher probability in the stationary distribution.

These results apply, technically speaking, in the case when the population size is sufficiently large. However, there is experimental evidence which suggests that populations stay near fixed-points even for small population sizes. To see why this might be the case, consider the following. Call the distance $|\mathcal{G}(\boldsymbol{p}) - \boldsymbol{p}|$ the *force* of $\mathcal{G}$ at the point $\boldsymbol{p}$. We would expect that if the force were large, then the GA is more likely to jump to a population which is very different to the current one. Obviously, at a fixed-point the force is zero. And since $\mathcal{G}$ is continuous (for genetic algorithms, anyway), then the force near a fixed-point is small. Moreover, when the population size is small, the population vectors representing the possible populations are rather spread out

in the simplex. That is, they are a long distance apart. This seems to imply that, if a population is near a fixed-point, the probability of making the jump to the next nearest population should be rather small. The genetic algorithm will therefore tend to "stall" in such states. This would explain the punctuated equilibria effect, even for small populations. So far, there has been no formal proof of this idea, but it seems that finding the fixed-points of $\mathcal{G}$ is an important tool for describing the behaviour of the genetic algorithm. We will see some examples of this later.

4.3 The simple genetic algorithm

We define the *simple genetic algorithm* to be a generational GA, comprising selection (usually fitness-proportional selection), mutation and crossover. The search space is usually the set of binary strings of length ℓ. We identify this search space with the set $\Omega = \{0, 1, \ldots, n-1\}$ by setting $n = 2^\ell$ and interpreting each binary string as being an integer written in base 2 notation. Given a current population of size N, we create the new one as follows.

1. Select first parent from population (with probability proportional to fitness).
2. Select second parent from population (with probability proportional to fitness).
3. Cross the two parents to form an offspring.
4. Apply mutation to the offspring.
5. Add the result to the next generation.
6. Repeat until next generation contains N individuals.

As we described above, this process is equivalent to the Random Heuristic Search, as long as we define the operator $\mathcal{G}$ in such a way as to accurately reflect the effects of the three operators. In this section, we will describe how this can be done, and give some examples of a fixed-point analysis.

We start by describing fitness-proportional selection, although it is quite possible to describe other kinds of selection within the same mathematical framework (for example, tournament selection, rank-based selection). We denote the heuristic operator corresponding to proportional selection by $\mathcal{F} : \Lambda \to \Lambda$. Recall that $\mathcal{F}(\boldsymbol{p})_k$ should be the probability of selecting individual $k \in \Omega$ from a population described by the vector $\boldsymbol{p}$. It is simple to check that the following definition achieves this:

$$\mathcal{F}(\boldsymbol{p})_k = \frac{f_k p_k}{\widehat{f}(\boldsymbol{p})}$$

where f_k is the fitness of item k and $\widehat{f}(\boldsymbol{p})$ represents the average fitness of the population $\boldsymbol{p}$. We get a slightly simpler formulation by placing the elements of the vector $\boldsymbol{f} = (f_0, f_1, \ldots, f_{n-1})$

along the diagonal of a diagonal matrix, which we will denote $\mathrm{diag}(\boldsymbol{f})$. We also note that the average fitness may be found by calculating the inner product $\boldsymbol{f}^T\boldsymbol{p}$, giving

$$\mathcal{F}(\boldsymbol{p}) = \frac{\mathrm{diag}(\boldsymbol{f})\boldsymbol{p}}{\boldsymbol{f}^T\boldsymbol{p}}$$

It is worth remarking that, if you run a genetic algorithm with just selection, then the system is a Markov chain with absorbing states. The absorbing states are the *uniform* populations: that is, populations containing only copies of a single individual. The corresponding population vectors are the standard basis vectors, containing a one in the position of the individual that is in the population and zeros elsewhere. These vectors are the corners of the simplex. We denote them $\mathbf{e}_j$ where j is the individual in the population. That is:

$$(\mathbf{e}_j)_k = [j = k]$$

The square bracket notation $[expr]$ denotes 1 if the expression $expr$ is true, and 0 if it is false.

It is easy to check that these vectors are fixed-points of $\mathcal{F}$ as follows:

$$\begin{aligned}\mathcal{F}(\mathbf{e}_j)_k &= \frac{f_k(\mathbf{e}_j)_k}{\boldsymbol{f}^T\mathbf{e}_j} \\ &= \frac{f_k[j=k]}{f_j} \\ &= [j=k]\end{aligned}$$

Starting with a population $\boldsymbol{p}(0)$, write $\boldsymbol{p}(1) = \mathcal{F}(\boldsymbol{p}(0))$, $\boldsymbol{p}(2) = \mathcal{F}(\boldsymbol{p}(1))$ and so on. The dynamics can be determined as:

$$\boldsymbol{p}(t)_k = \frac{(f_k)^t\boldsymbol{p}(0)_k}{\sum_j (f_j)^t\boldsymbol{p}(0)_j}$$

(see [15] for a proof).

The effects of mutation will be given by an operator $\mathcal{U} : \Lambda \to \Lambda$. The quantity $\mathcal{U}(\boldsymbol{p})_i$ should be the probability of creating item $i \in \Omega$ from population $\boldsymbol{p}$ using mutation. If we denote the probability that item j mutates to item i by $U_{i,j}$ then

$$\mathcal{U}(\boldsymbol{p})_i = \sum_j U_{i,j}p_j$$

and so the effects of mutation are given by a matrix U with entries $U_{i,j}$ so that

$$\mathcal{U}(\boldsymbol{p}) = U\boldsymbol{p}$$

In the usual case in which mutation is implemented by flipping each bit of a string independently, with probability u, we find an explicit formula for U

$$U_{i,j} = u^{d(i,j)}(1-u)^{\ell - d(i,j)}$$

where $d(i,j)$ is the Hamming distance between the strings i and j.

The set of binary strings of length ℓ can be given a natural algebraic structure. We define the *sum* of two strings i and j, which we will denote $i \oplus j$, by combining them together using bitwise exclusive-or (or, equivalently, bitwise addition modulo-2). In fact, Ω forms a *group* under this definition, with identity element 0. Every element of Ω is its own inverse: $i \oplus i = 0$ for all $i \in \Omega$. Mutation, as defined above, has the interesting property that it *commutes* with the group. That is:

$$U_{k\oplus i,k\oplus j} = U_{i,j}$$

for all $i,j,k \in \Omega$. Many other types of mutation also have this property. In general, define mutation *by mask* by assigning a probability distribution $\mu \in \Lambda$ to the set Ω. That is, the probability of picking $k \in \Omega$ is μ_k. Then we mutate an element j in the population by picking k according to this distribution and forming $j \oplus k$. Bitwise mutation by a rate is just a special case of this with

$$\mu_k = u^{d(k,0)}(1-u)^{\ell - d(k,0)}$$

We can use the fact that mutation commutes with the group to prove that the centre of the simplex, $\boldsymbol{v} = (1/n, 1/n, \ldots, 1/n)$ is a fixed-point of such a mutation:

$$(U\boldsymbol{v})_i = \sum_j U_{i,j} v_j = \frac{1}{n}\sum_j U_{i,j} = \frac{1}{n}\sum_j U_{j\oplus i,0} = \frac{1}{n}\sum_k U_{k,0} = \frac{1}{n} = v_i$$

The Perron-Frobenius theorem tells us that, if $U_{i,j} > 0$ for all i,j (for example, if mutation is bitwise by a rate $u > 0$), then this is in fact the only fixed-point in the simplex for $\mathcal{U}$.

Finally, we will deal with crossover. This will be represented as an operator $\mathcal{C} : \Lambda \to \Lambda$. Naturally, we want $\mathcal{C}(\boldsymbol{p})_k$ to be the probability of producing item $k \in \Omega$ from the population p. Let us denote the probability that parents i and j combine to form k by $r(i,j,k)$. Then we want

$$\mathcal{C}(\boldsymbol{p})_k = \sum_{i,j} p_i p_j r(i,j,k)$$

This suggests that, for each item k, we put the probabilities $r(i,j,k)$ into a matrix M_k, so that

$$\mathcal{C}(\boldsymbol{p})_k = \boldsymbol{p}^T M_k \boldsymbol{p}$$

which is a *quadratic form*. $\mathcal{C}$ is called a *quadratic operator* [18]. The entries of M_k are

$$(M_k)_{i,j} = \frac{r(i,j,k) + r(j,i,k)}{2}$$

and so M_k is symmetric.[2]

To go further, we need the details of $r(i,j,k)$. For fixed-length binary strings, the most common crossovers are done by *masks*. Let b be a binary string of length ℓ. Then define $i \otimes b$ to

[2]We get exactly the same quadratic form if we set $M_k = r(i,j,k)$, so one may as well assume that the matrices M_k are symmetric.

be the bitwise AND operator. We define a probability distribution χ over the set of masks. That is, χ_b is the probability of picking b. Then

$$r(i,j,k) = [(i \otimes b) \oplus (j \otimes \bar{b})]\chi_b$$

(where $\bar{b}$ is the complement of b). For example, if $i = 10110011$ and $j = 01100110$, then mask $b = 11110000$ produces offspring

$$\begin{aligned}(10110011 \otimes 11110000) \oplus (01100110 \otimes 00001111) &= (10110000 \oplus 00000110) \\ &= 10110110\end{aligned}$$

which corresponds to a one-point crossover, with the cut point in the middle of the string. We get different forms of crossover by choosing different distributions χ. For example, uniform crossover is given by assigning all masks equal probabilities. If crossover is performed with probability less than 1, then this corresponds to assigning higher probability to the mask $b = 0$, which has the effect of cloning the first parent.

Crossover by masks shares with mutation by masks the nice property of commuting with the group. That is

$$r(i,j,k) = r(a \oplus i, a \oplus j, a \oplus k)$$

for all $i, j, k, a \in \Omega$. This property means that we can simplify the definition of the crossover operator as follows. For each $k \in \Omega$ define an $n \times n$ matrix σ_k by

$$(\sigma_k)_{i,j} = [i = k \oplus j]$$

Then it is easy to check that, for all $k \in \Omega$

$$M_k = \sigma_k M_0 \sigma_k^T$$

In other words, the probabilities found in the matrix M_k are identical to those found in M_0, but have been moved around (by the permutation matrix σ_k). This means that, once the group is defined, all the information necessary to describe crossover is in the matrix M_0. This matrix is referred to as the *mixing matrix* of crossover. In fact, this happens whenever the search space has a group structure, or indeed, has a group acting transitively upon it — see [20, 21] for details.

If we just run crossover over and over again, starting from some initial population, we again get to a fixed-point. The fixed-points of crossover are described by Geiringer's Theorem [5], a well-known theorem from population genetics. Notice that applying crossover to a population cannot change the number of ones and zeros in any bit position. All that happens is that these bits get shuffled around. This means that the fixed-point we get to depends on the initial population. It is characterised by the fact that, if we draw a string at random from the fixed-point population, the probability that we see a one or zero in any position is *independent* of the bit values at the other positions. Moreover, that probability is given simply by the frequency of ones and zeros in that position in the starting population. In other words, crossover tends to *de-correlate* the bit values, whilst not introducing any new genetic material. By contrast, selection tends to drive

evolution towards uniform (high fitness) populations, losing genetic material. Mutation tends to randomise the population (move it towards the centre of the simplex), creating new individuals. Of course, the behaviour that we see in an actual genetic algorithm will be a balance of these three effects working in combination.

Let's consider what happens when we combine selection and mutation. In other words, we set the crossover probability to zero. This is equivalent to setting the probability of mask $b = 0$ to 1. This has the effect of always cloning the first parent selected, and ignoring the second parent. A single generation of the genetic algorithm therefore reduces to:

1. Select a parent (with probability proportional to fitness).
2. Apply mutation to the parent to create an offspring.
3. Add offspring to the next generation.
4. Repeat until next generation contains N individuals.

The heuristic function for selection plus mutation is found by composing the heuristic for the two operators:

$$\mathcal{G} = \mathcal{U} \circ \mathcal{F}$$

Putting in the definitions of these operators we get

$$\mathcal{G}(\boldsymbol{p}) = \frac{U\text{diag}(\boldsymbol{f})p}{\boldsymbol{f}^T\boldsymbol{p}}$$

The fixed-points of $\mathcal{G}$ satisfy the equation

$$U\text{diag}(\boldsymbol{f})\boldsymbol{p} = (\boldsymbol{f}^T\boldsymbol{p})\boldsymbol{p}$$

In other words, the fixed-point population is an eigenvector of the matrix $U\text{diag}(\boldsymbol{f})$. The corresponding eigenvalue is the average fitness of this population. Again, the Perron-Frobenius Theorem tells us that there is exactly one fixed-point in the simplex, and that this corresponds to the eigenvalue of largest magnitude. It is also interesting to note that every point in the interior of the simplex corresponds to a fixed-point for some fitness function [17]. Choosing

$$\boldsymbol{f} = (U\text{diag}(\boldsymbol{p}))^{-1}\boldsymbol{p}$$

(or a scalar multiple) does the trick, assuming that U is invertible (which it usually is).

Finding the eigenvectors of $U\text{diag}(\boldsymbol{f})$ becomes computationally intractable for large search spaces. However, in the case of *functions of unitation* (where the fitness of a string depends only on the number of ones it contains), the search space can be collapsed onto one of size $\ell + 1$. Each element of the new search space simply counts the number of ones in the strings of the old search space. One defines the mutation matrix U by

$$U_{i,j} = \sum_{k=0}^{\ell-j}\sum_{l=0}^{j}[i = j + k - l]\binom{\ell - j}{k}\binom{j}{l}u^{k+l}(1-u)^{\ell-k-l}$$

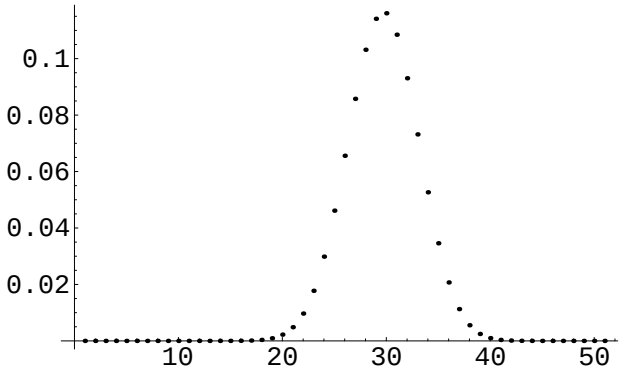


Figure 4.1: The fixed-point population, shown as a distribution over unitation classes, for the OneMax function with $\ell = 50$, $u = 0.05$ and no crossover. The corresponding average fitness is 28.65.

(see [14] for details). Note that i and j are now unitation classes, taking values from $\{0, 1, \ldots, \ell\}$.

As an example, consider the OneMax function $f_k = d(k, 0)$. This is obviously a function of unitation. Working with the reduced search space $\Omega = \{0, 1, \ldots, \ell\}$, we can calculate the fixed-points of $\mathcal{G}$ for quite large problems. For example, for $\ell = 50$ and $u = 0.05$, the fixed-point distribution is illustrated in figure 4.1, with a corresponding average fitness of 28.65. As hinted at earlier, the effects of other fixed-points, even those outside the simplex, may be important in explaining where the genetic algorithm will spend most of its time. A number of examples are worked out in [17].

Now let us consider the case when there is crossover, but no mutation (that is, set $u = 0$ and $\chi_0 < 1$). A moment's thought tells us that, as with selection-only, the uniform populations are again absorbing states. This is because neither selection nor crossover can introduce new genetic material. It is also easy to check that such populations are also fixed-points of $\mathcal{G} = \mathcal{C} \circ \mathcal{F}$. However, it turns out that some of these points are *asymptotically stable*, while others are not. An asymptotically stable fixed-point is one for which, if the initial population is chosen close enough, then the evolutionary trajectory is guaranteed to converge to the fixed-point. A theorem by Michael Vose and Alden Wright proves the following interesting property. Each vertex of the simplex corresponds to a uniform population, and therefore also corresponds to a single point of the search space (with N copies). If the vertex is an asymptotically stable fixed-point of $\mathcal{G}$ then the corresponding element of the search space must be a *local optimum* with regards the usual Hamming neighbourhood. That is, it must have a fitness that is greater than all strings which differ from it in exactly one bit position [26]. It may happen that there are also some fixed-points that are not vertices. However, it is believed that any such points would not be asymptotically stable. This has not be proved, however, and remains an open question.

Slightly more is known about a different form of crossover — one that is not a quadratic operator. The so-called *genepool* crossover works on the whole population simultaneously. We generate each new string of the next generation one bit at a time, by choosing its value according to the frequency of ones and zeros in the corresponding position in the current population. The

effect of this is to immediately de-correlate the bit values. In effect, the algorithm goes in one step to the Geiringer limit. If we use genepool crossover with proportional selection, the algorithm is known as UMDA (Univariate Marginal Distribution Algorithm), and is related to the class of *estimation of distribution* algorithms. This algorithm has been analysed thoroughly by Heinz Mühlenbein [13] who shows, amongst other things:

- The average fitness of the population always increases, except at vertices (that is, uniform populations).
- The vertices are fixed-points.
- The asymptotically stable vertex fixed-points are precisely those vertices which correspond to local optima (in the sense of Hamming neighbourhoods).
- There are no asymptotically stable fixed points inside the simplex.

The situation when mutation is added to genepool crossover and proportional selection has been analysed for a few simple case (see [29]).

Some of the techniques described in this section can be adapted to the study of *steady-state* genetic algorithms, in which a single offspring is created in a given time-step, and it is inserted back into the population. The dynamics of this approach can be approximated by considering time to be continuous [28]. One of the crucial implementation decisions that has to be made is how to decide which element of the population is to be replaced by the newly created offspring. Two popular choices are:

1. Replace the worst item in the population.
2. Replace a random item.

This choice has a strong influence on the fixed-points of the infinite population model. In the first case (replace worst), it can be shown that the fixed-points are uniform populations provided some technical conditions are met (which they are, for example, if there is a positive mutation rate). In the second case, however, (replace random) the fixed-points turn out to be identical to the fixed-points of the generational GA, using the same selection and mixing heuristic.

4.4 Search spaces with variable-sized objects

The simple genetic algorithm has, as its canonical search space, the finite set of binary strings of some fixed length. Aspects of the theory have been generalised to other finite search spaces, for example, for combinatorial problems such as the Travelling Salesman Problem [20]. However, there are a number of application areas in which the search space is not necessarily finite, due to the fact that the objects in the search space have variable size. One example of this occurs when the elements of the search space are rules, made up of IF and THEN parts, each of which may contain a variable number of clauses (conditions which have to be true, in the case of the IF-part, and things that become true in the case of THEN-parts). A related example is the problem of

trying to evolve rules for a grammar, for example, if the representation is a developmental one, like an L-system. In the case of Learning Classifier Systems, variable-length structures have been used in conjunction with XCS [11]. They have also been used to grow hidden layers in neural classifiers [4].

Perhaps the most common example of variable-length structures comes from the field of Genetic Programming, in which the individuals are (representations of) computer programs. Programs can be represented as sequences of instructions (for example, an assembly language program) or as a parse-tree of a lisp-like language. While a lot of work has gone into developing these representations and the operators that act upon them, as well as empirical work justifying their use in practice, there is nothing like a comparable amount of theoretical analysis or understanding. Most of what is known has been developed by Riccardo Poli, William Langdon and colleagues [10]. Much of this work concerns the generalisation of ideas from genetic algorithm theory to the case of trees. We will look briefly at the rather simpler case of variable-length linear structures (that is, strings), where there is a more obvious extension of the situation for genetic algorithms.

There are many ways in which mutations and crossovers can be defined for variable-length strings. We will give as examples three of the most common (which are, in fact, counterparts of operators defined for trees).

Crossover Two strings are truncated at random places along their length. The left part of one of the strings is appended to the right part of the other string.

Grow mutation A string is truncated at some random point along its length. New symbols are then added according to a sequence of Bernoulli trials. That is, there is some probability q that a new symbol (chosen arbitrarily from the available alphabet) will be appended, and a probability of $1 - q$ that the growth process will stop. Growth continues until this latter event happens.

Full mutation A string is truncated at some random point along its length. A new random string of length D (which is a parameter to be set by the user) is appended.

Some of the effects of applying these operators have been investigated [19], if we just consider what is happening to the *average length* of strings in the population. In the absence of selection, this is the only significant factor.

For crossover, it should first be noted that the average length of items in the population does not change. The fixed-point therefore depends on the average length of the initial population. It can be shown that populations of the form

$$p_k = (1 - a)^2 k a^{k-1}$$

are fixed-points of this kind of crossover, where a is a parameter related to the average length, m, of the initial (and subsequent) populations by

$$a = \frac{m - 1}{m + 1}$$

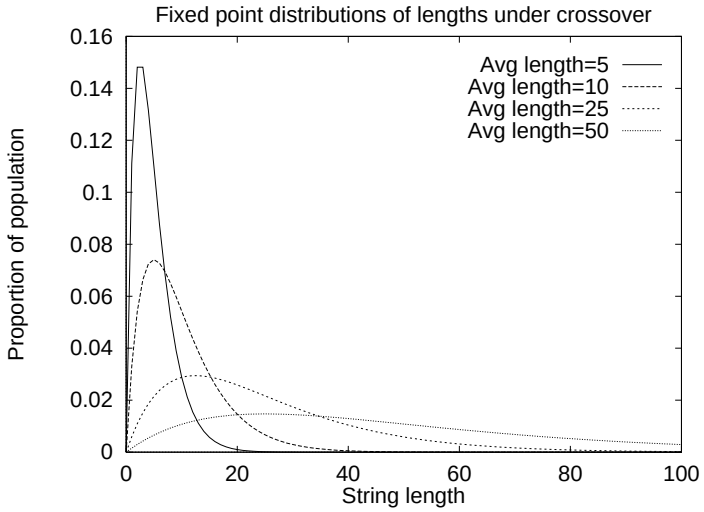


Figure 4.2: Fixed-point populations for crossover, acting on variable length strings, for different average lengths. (Graph taken from [19].)

This was originally proved in [12]. Some examples of this distribution are illustrated in figure 4.2.

Remarkably, for grow mutation, the fixed-points have exactly the same form as for crossover, with the parameter a set to the value q. That is, the fixed-point of grow mutation is:

$$p_k = (1-q)^2 k q^{k-1}$$

(see [19]).

For full mutation, the result obviously depends on the value of the parameter D. For $D > 2$ there is no known closed form for the fixed-point. However, for $D = 2$ we have:

$$p_k = \frac{[k > 1]}{e(k-2)!}$$

[19] which is illustrated in figure 4.3. Notice that the height of the "plateau" is exactly $1/e$.

There has been very little work done on studying the combination of selection and crossover or mutation for variable-length structures. This is partly because the subject is rather difficult, but mostly because there are very few researchers working on these problems.[3] We give a simple example to illustrate some of the technicalities.[4]

Let us, then, consider the simplest possible case of an infinite search space, namely taking Ω to be the natural numbers $\{1, 2, 3, 4, \dots\}$. You can think of each integer k as representing the *size* of a program or string. Populations are then associated with probability distributions over this set:

$$\boldsymbol{p} = (p_1, p_2, p_3, \dots)$$

[3] I counted just six researchers at the time of writing!

[4] These ideas were originally presented at a Dagstuhl Seminar in January 2002 by the author.

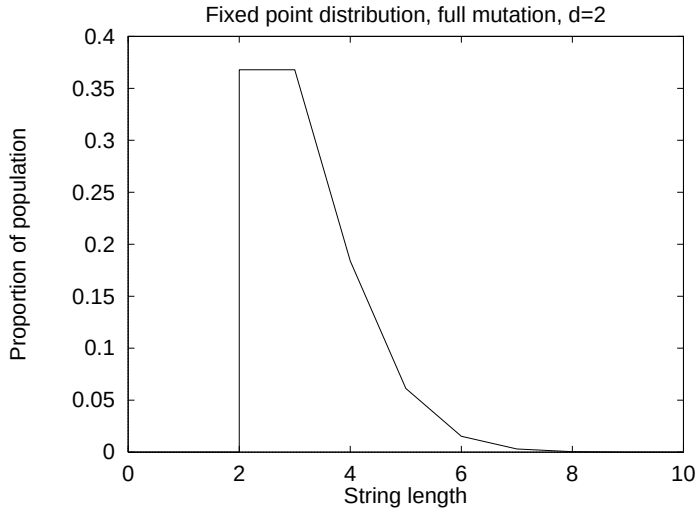


Figure 4.3: The fixed-point population for full mutation with $D = 2$. (Graph taken from [19].)

where as usual p_k indicates the proportion of the population which is made up of copies of item k. The set of all such probability distributions is:

$$\Lambda = \left\{ (p_1, p_2, p_3, \dots) | \sum_k p_k = 1 \text{ and } p_k \geq 0 \text{ for all } k \right\}$$

We will use fitness proportional selection (which is well-defined as long as the fitness function is bounded) and the following mutation operator (for some rate $0 < u < 1$):

$$\mathcal{U}(\boldsymbol{p})_k = \begin{cases} (1-u)p_1 & \text{if } k = 1 \\ (1-u)p_k + up_{k-1} & \text{if } k > 1 \end{cases}$$

The effect of this mutation is that item k is mutated into item $k+1$ with a probability u. This mutation was christened the *super-bloater* by Riccardo Poli, for reasons which we will see.

First let's see what would happen if the fitness function is a constant $f(k) = c$ for all k. This makes the proportional selection operator equivalent to the identity. In other words, we are just iterating $\mathcal{U}$. We define the average size of the population to be

$$s(\boldsymbol{p}) = \sum_k kp_k$$

It is simple to show that the average size increases without bound if we just iterate $\mathcal{U}$. Such a situation is called *bloat* in the GP literature. In fact

$$s(p(t+1)) = s(p(t)) + u$$

The reason for this is that the sequence of populations is itself not converging to anything: there is no limit! The populations keep sampling bigger and bigger elements of the search space. Notice

that this is very different from the case when the search space is finite. In that case, Brouwer's Fixed-Point theorem tells us that $\mathcal{G}$ must have at least one fixed-point, and it typically happens (as we saw, for example, with selection plus mutation) that all populations eventually converge to that fixed-point.

We could try to change this by having a non-constant fitness function, that favours smaller individuals. The resulting behaviour should be a balance of these two forces. Let's suppose we have a fitness function such that there is a fixed-point which is the population $\boldsymbol{v}$ where

$$v_k = \frac{6}{(\pi k)^2}$$

This is a valid population since

$$\sum_k v_k = \sum_k \frac{6}{(\pi k)^2} = \frac{6}{\pi^2} \sum_k \frac{1}{k^2} = 1$$

But the average size of this population is:

$$s(\boldsymbol{v}) = \sum_k k v_k = \sum \frac{6}{k\pi^2} = \frac{6}{\pi^2} \sum_k \frac{1}{k} = \infty$$

So it's possible that a sequence of populations might converge to this fixed-point, but nevertheless, the average size would still grow to infinity! The only question is: does there exist a fitness function such that $\boldsymbol{v}$ is the fixed-point? It turns out that the argument given above for finite search spaces, showing that any point in the simplex is a fixed-point for some fitness function, can be adapted to the infinite case, and the answer is, yes there is such a fitness function. The details are rather technical and are omitted here, but if you run an evolutionary algorithm with that fitness and the given mutation operator, you indeed see two things happening simultaneously: the population converges to the fixed-point; and the average length increases without bound (that is, the system bloats, even though it is converging to a limit).

Clearly, much more theoretical investigation is required if we are going to understand how evolution can take place in these infinite search spaces. Our example is highly simplistic, yet it already highlights some difficulties, as well as interesting phenomena that do not occur in the finite case.

4.5 Dynamic fitness functions

We now return to finite search spaces, and the simple genetic algorithm, but consider the situation when the fitness function changes over time. There are a number of situations when this might happen, for example:

- We are using the genetic algorithm to try to find an optimal control strategy for a manufacturing process. The correct control settings will vary depending on parameters governing that process.

- We are trying to evolve solutions to a problem set by a user, whose requirements change with time.
- We are trying to model an ecosystem, in which the external environment is changing.

In this section we will consider the simpler case when the fitness varies in a known way with time. When the fitness also depends on the state of the population, we have a *co-evolutionary* algorithm, which we address in the following section. One option, if the fitness function changes, is simply to run the genetic algorithm over again from scratch. However, we will assume that the changes are rapid enough that this method would not be able to keep track of them, and we will assume that the changes are smooth enough that continuing the evolution from the current population will give viable results.

Let us suppose we have a genetic algorithm with proportional selection and mutation, but no crossover. We consider the case when the fitness function is varying in a known periodic manner, with period τ. That is, there is a sequence of fitness functions, $\boldsymbol{f}(0), \boldsymbol{f}(1), \ldots, \boldsymbol{f}(\tau-1)$, and the function that applies in generation t is $\boldsymbol{f}(t \bmod \tau)$. We define a corresponding set of selection heuristics:

$$\mathcal{F}_i(\boldsymbol{p}) = \frac{\text{diag}(\boldsymbol{f}(i))\boldsymbol{p}}{\boldsymbol{f}(i)^T\boldsymbol{p}}$$

Of course, we no longer expect the evolutionary dynamics to have a fixed-point, but rather a periodic cycle, of order τ. For each $i \in \{0, 1, \ldots, \tau-1\}$, define an operator $\mathcal{G}_i = \mathcal{U} \circ \mathcal{F}_i$, and then define operators $\mathcal{H}_i$ as follows:

$$\begin{aligned}
\mathcal{H}_0 &= \mathcal{G}_{\tau-1} \circ \mathcal{G}_{\tau-2} \circ \cdots \circ \mathcal{G}_0 \\
\mathcal{H}_1 &= \mathcal{G}_{\tau-2} \circ \mathcal{G}_{\tau-3} \circ \cdots \circ \mathcal{G}_0 \circ \mathcal{G}_{\tau-1} \\
\mathcal{H}_2 &= \mathcal{G}_{\tau-3} \circ \mathcal{G}_{\tau-4} \circ \cdots \circ \mathcal{G}_0 \circ \mathcal{G}_{\tau-1} \circ \mathcal{G}_{\tau-2} \\
&\vdots \\
\mathcal{H}_{\tau-1} &= \mathcal{G}_0 \circ \mathcal{G}_{\tau-1} \circ \cdots \circ \mathcal{G}_2 \circ \mathcal{G}_1
\end{aligned}$$

Then the periodic attractor for the system is $\boldsymbol{v}(0), \boldsymbol{v}(1), \ldots, \boldsymbol{v}(\tau-1)$ where

$$\mathcal{H}_i(\boldsymbol{v}(i)) = \boldsymbol{v}(i)$$

for each $i \in \{0, 1, \ldots, \tau-1\}$. We can solve for these vectors using a method similar to the case of a static fitness function. For example, the vector $\boldsymbol{v}(0)$ is an eigenvector of the matrix

$$U\text{diag}(\boldsymbol{f}(\tau-1))U\text{diag}(\boldsymbol{f}(\tau-2))\ldots U\text{diag}(\boldsymbol{f}(0))$$

The corresponding eigenvalue is not the average fitness, but rather

$$\prod_{i=0}^{\tau-1} \boldsymbol{f}(i)^T\boldsymbol{v}(i)$$

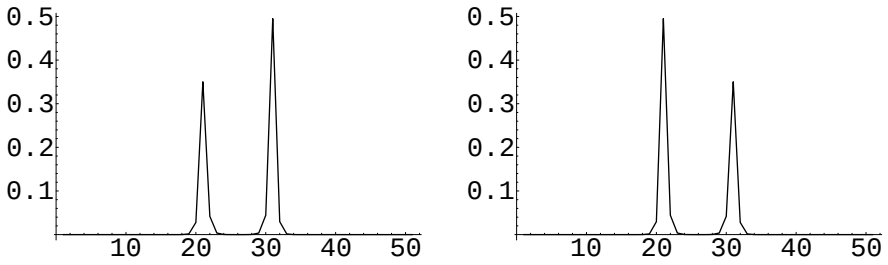


Figure 4.4: The periodic attractor corresponding to f_0 and f_1 described in the text.)

As a simple example, let's consider the following pair of functions of unitation:

$$f_0(x) = [d(x,0) = \ell/2 - 5] + 1,\ f_1(x) = [d(x,0) = \ell/2 + 5] + 1$$

That is, f_0 scores 2 for strings that contain exactly $\ell/2 - 5$ ones, and 1 otherwise and f_2 scores 2 for strings with $\ell/2 + 5$ ones, and 1 otherwise. Using a string length $\ell = 50$ and a mutation rate $u = 0.001$, we calculate the fixed-points of $\mathcal{H}_0$ and $\mathcal{H}_1$ as described above to find the periodic attractor, shown in figure 4.4. We can see that at each generation, the majority of the population is clustered around the optimal point. However, a large minority of individuals are placed at the optimal point for the other fitness function. In this way, the population manages to keep track of the changing optimum over time.[5]

4.6 Co-evolutionary dynamics

In this final section, we will continue to consider dynamic fitness functions, but now we assume that the fitness changes in a way that depends on the current population. That is, the fitness of an individual will depend on which other individuals are sharing the same population. This situation is sometimes referred to as *co-evolution*. One of the most common examples of this phenomenon in evolutionary algorithms, is in *classifier systems*. Here, the aim is to evolve a population of rules which will control the activities of an agent in some environment. The success of a rule depends not only on the environment, but also on what other rules are in the population, which it can work with to form chains of activations.

In theoretical biology, the situation is described using evolutionary game theory [6]. It is assumed that each member of the population interacts with all the other members (or perhaps randomly encounters a sample) during one time period. As a result of each interaction, the individual either gains or loses some fitness. Its final fitness is a measure of its ability to reproduce. In order to represent what is happening, we need to store all of the possible interactions. We do this in a matrix, called the *payoff matrix*, which we will denote P. The entry $P_{i,j}$ gives us the increment in fitness to individual i as a result of an encounter with individual j. Negative entries correspond to a decrease in fitness. We assume that each individual i has some baseline fitness

[5] For an alternative, but equivalent, method, see [16]

c_i, which is how fit it would be if there were no encounters. If the current population is $\boldsymbol{p} \in \Lambda$, then we can work out the fitness of individual i in this population as

$$f_i = c_i + \sum_j P_{i,j} p_j$$

In other words, the fitness function (considered as a vector) can be written as a function of the population as follows:

$$\boldsymbol{f}(\boldsymbol{p}) = P\boldsymbol{p} + \boldsymbol{c}$$

We will assume that the time period under consideration is one generation, and also that individuals reproduce in proportion to their fitness. In this case, we get a discrete-time *replicator equation*:

$$\boldsymbol{p}(t+1) = \frac{\text{diag}(\boldsymbol{f}(\boldsymbol{p}))\boldsymbol{p}(t)}{\boldsymbol{f}(\boldsymbol{p})^T \boldsymbol{p}(t)}$$

which is the same as we had for proportional selection for the simple genetic algorithm, except now the fitness varies as a function of $\boldsymbol{p}$.

The assumption that each individual interacts with all the others in the population (with equal probability), and that the results of these interactions are independent in their effects, results in the fitness being a *linear* function of the population vector. It is possible, in what follows, to relax this assumption, and consider fitness to depend in any (possibly non-linear) way on $\boldsymbol{p}$. However, to keep things simple, we will stick with the assumption of linearity.

We can solve for the fixed-points of the system as follows. First, let's suppose that there is a fixed-point $\boldsymbol{v}$ inside the simplex (so all its entries are non-zero). Let the average fitness of this population be λ. Then $\boldsymbol{v}$ must satisfy

$$\boldsymbol{v} = \frac{\text{diag}(\boldsymbol{f})\boldsymbol{v}}{\lambda} = \frac{\text{diag}(\boldsymbol{v})\boldsymbol{f}}{\lambda} = \frac{\text{diag}(\boldsymbol{v})(P\boldsymbol{v} + \boldsymbol{c})}{\lambda}$$

and so, rearranging,

$$\boldsymbol{v} = \lambda P^{-1}\mathbf{1} - P^{-1}\boldsymbol{c}$$

where $\mathbf{1}$ is the vector containing all ones, and we have assumed that P is invertible. The value of λ can be found from the fact that $\sum_k v_k = 1$, giving

$$\lambda = \frac{1 + \sum_k (P^{-1}\boldsymbol{c})_k}{\sum_k (P^{-1}\mathbf{1})_k}$$

The assumption that all the entries of $\boldsymbol{v}$ were non-zero was important, because during this calculation, we need the inverse of the matrix $\text{diag}(\boldsymbol{v})$ to exist. So what happens if this is not the case, and $v_k = 0$ for some $k \in \Omega$? If this happens, then the type k is extinct — we only have selection, so it can never come back. Since it is no longer a player in the game, we simply remove the corresponding row and column from P to get a reduced payoff matrix. We also delete c_k from $\boldsymbol{c}$. Then we can proceed as before (assuming the new P is invertible) to find a new fixed-point. There are, therefore, potentially 2^n possible fixed-points of the system. An algorithm for finding them is:

1. For each subset $A \subset \Omega$ repeat the following:
2. Remove rows and columns corresponding to elements of A from the payoff matrix.
3. Remove the corresponding elements from $\boldsymbol{c}$.
4. If the resulting payoff matrix is invertible, find the fixed-point (as above).

There is no point taking $A = \Omega$ as this corresponds to the case where everything is extinct. Also, the cases where there is just one species in the game give rise to the trivial fixed-points $\mathbf{e}_j$ at the corners of the simplex.

Let's work through an example with three strategies. We expect to get a maximum of $2^3 - 1 = 7$ fixed-points (ignoring 0). We assume that $\boldsymbol{c} = 0$ and consider the payoff matrix:

$$P = \begin{pmatrix} 4 & 2 & 1 \\ 3 & 3 & 1 \\ 2 & 1 & 2 \end{pmatrix}$$

This matrix is invertible and

$$P^{-1} = \begin{pmatrix} 0.5556 & -0.3333 & -0.1111 \\ -0.4444 & 0.6667 & -0.1111 \\ -0.3333 & 0 & 0.6667 \end{pmatrix}$$

Multiplying P^{-1} by $(1, 1, 1)$ and normalising gives the fixed-point $(0.2, 0.2, 0.6)$ corresponding to $\lambda = 1.8$, which is the average fitness value at that fixed-point.

Now we must consider what happens when $p_0 = 0$. The projection of P is

$$\begin{pmatrix} 3 & 1 \\ 1 & 2 \end{pmatrix}$$

which is also invertible. Multiplying the inverse by $(1, 1)$ and normalising gives us the fixed-point $(0, 1/3, 2/3)$ corresponding to fitness value $\lambda = 5/3$. Note that this is the fitness value for strategies 1 and 2. The fitness for the extinct strategy 0 is not defined (and, of course, irrelevant).

When $p_1 = 0$ a similar process gives us the fixed-point $(1/3, 0, 2/3)$ corresponding to $\lambda = 2$. And when $p_2 = 0$ we get the fixed-point $(0.5, 0.5, 0)$ with $\lambda = 3$.

You can also check that $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$ are fixed-points with corresponding average fitness 4, 3 and 2 respectively. We therefore have seven fixed-points for this system, all of which could, in fact, correspond to actual finite populations. Only one of the fixed-points, however, contains copies of all three strategies.

The fact that there are potentially so many fixed-points can have a large influence on the dynamics of the system, when run with a finite population. Suppose the initial population contains members of all elements of Ω, and there is a fixed-point in the simplex. As the evolution unfolds, sampling effects due to the finite size of the population will create fluctuations. This might lead to an accidental extinction, if there aren't enough representatives of a particular individual in the

population. Once this happens, the game changes, and there is a new fixed-point. As the evolutionary trajectory heads in that direction, it is possible that further extinctions might take place. Such events are often observed in runs of co-evolutionary genetic algorithms, with the end result being a population with many elements of the search space missing. One extinction event can trigger others in a cascade. For an investigation into the dynamics of extinctions, see [27].

Some co-evolutionary algorithms are implemented with a number of isolated populations. The system is co-evolutionary in the sense that the fitness function that is applied to one population depends on the contents of the other populations. An example of this is *co-operative* co-evolution, in which different parts of a problem are assigned to different populations. The fitness of an element from one population depends on which members of the other populations it is put together with in order to be evaluated. We can extend our analysis to this situation in a straightforward manner. Let us suppose, for simplicity, that there are two populations, $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$. The fitness function to be applied to the first population is a (linear) function of the second, and vice versa:

$$\boldsymbol{f}_1 = P_1\boldsymbol{p}_2, \ \boldsymbol{f}_2 = P_2\boldsymbol{p}_1$$

In this example, we take the baseline fitness to be zero. Again, by assuming that there are fixed-points for each population $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ in the interior of the simplex, we find:

$$\boldsymbol{v}_1 = \lambda_1(P_2)^{-1}\mathbf{1}$$

and

$$\boldsymbol{v}_2 = \lambda_2(P_1)^{-1}\mathbf{1}$$

where the constants λ_1 and λ_2 can be found by normalising. The same considerations as before apply if certain elements become extinct.

4.7 Conclusions

Learning classifier systems adapt their behaviour on two different time-scales. In the short-term, the existing population of classifiers responds to the changing environment. In the long-term, the underlying genetic algorithm seeks to evolve better populations. As such, the theory of genetic algorithms provides a starting point for building a framework of LCS population dynamics. However, the description of the Simple Genetic Algorithm must be extended in important ways, both structurally and dynamically. Structurally, because it is possible for classifiers to take on a number of syntactic forms, rather than simply being fixed-length strings (for example, S-expressions and neural classifiers have recently been proposed [11, 4]). Dynamically, because the fitness of an individual classifier depends critically on the context of the population in which it resides. This makes the long-term process a *co-evolutionary* one, based on performance data from the short-term success of the system. Moreover, the environment faced by the system may itself change in time. Consequently, these extensions to standard GA theory, which have only recently begun, are essential to the development of a fuller understanding of learning classifier systems.

Bibliography

[1] A.Prügel-Bennett and A.Rogers. Modelling genetic algorithm dynamics. In L. Kallel, B. Naudts, and A. Rogers, editors, *Theoretical aspects of evolutionary computation*, pages 59–86. Springer, 2001.

[2] H.-G. Beyer. *Theory of evolution strategies*. Springer, 2001.

[3] J. Branke. *Evolutionary optimization in dynamic environments*. Kluwer Academic Publishers, 2001.

[4] L. Bull. On using constructivism in neural classifier systems. In J. Merelo, P. Adamidis, H.-G. Beyer, J.-L. Fernandez-Villacanas, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature — PPSN VII.*, pages 558–567. Springer Verlag, 2002.

[5] H. Geiringer. On the probability theory of linkage in mendelian heredity. *Annals of Mathematical Statstics*, 15(1):25–57, 1944.

[6] J. Hofbauer and K. Sigmund. *Evolutionary games and population dynamics*. Cambridge University Press, 1998.

[7] D. L. Isaacson and R. W. Madsen. *Markov chains: theory and applications*. John Wiley & Sons, 1976.

[8] T. Jansen and I. Wegener. Real royal road functions — where crossover provably is essential. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, pages 1034–1041. Morgan Kaufmann, 2001.

[9] L. Kallel, B. Naudts, and A. Rogers, editors. *Theoretical aspects of evolutionary computation*. Springer, 2001.

[10] W. B. Langdon and R. Poli. *Foundations of genetic programming*. Springer, 2002.

[11] P.-L. Lanzi. Extending the representation of classifier conditions part II: from messy coding to S-expressions. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honovar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999)*, pages 11–18. Morgan Kaufmann, 1999.

[12] N. F. McPhee, R. Poli, and J. E. Rowe. A schema theory analysis of mutation size biases in genetic programming with linear representations. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC 2001*, pages 1078–1085, Seoul, Korea, May 2001.

[13] H. Mülenbein and T. Mahnig. Convergence theory and applications of the factorized distribution algorithm. *Journal of Computing and Information Technology*, 7:19–32, 1999.

[14] E. Van Nimwegen, J. P. Crutchfield, and M. Mitchell. Finite populations induce metastability in evolutionary search. *Physics Letters A*, 229(2):144–150, 1997.

[15] C. R. Reeves and J. E. Rowe. *Genetic algorithms: principles and perspectives*. Kluwer Academic Publishers, 2002.

[16] J. E. Rowe. Finding attractors for periodic fitness functions. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honovar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999)*, pages 557–563. Morgan Kaufmann, 1999.

[17] J. E. Rowe. Population fixed-points for functions of unitation. In W. Banzhaf and C. R. Reeves, editors, *Foundations of Genetic Algorithms*, volume 5, pages 69–84. Morgan Kaufmann, 1999.

[18] J. E. Rowe. A normed space of genetic operators with applications to scalability issues. *Evolutionary Computation*, 9(1):25–42, 2001.

[19] J. E. Rowe and N. F. McPhee. The effects of crossover and mutation operators on variable length linear structures. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, pages 535–542. Morgan Kaufmann, 2001.

[20] J. E. Rowe, M. D. Vose, and A. H. Wright. Group properties of crossover and mutation. *Evolutionary Computation*, 10(2):151–184, 2002.

[21] J. E. Rowe, M. D. Vose, and A. H. Wright. Structural search spaces and genetic operators. *Evolutionary Computation*, 12(4), 2004.

[22] G. Rudolph. *Convergence properties of evolutionary algorithms*. Kovacs, 1997.

[23] J. Shapiro. Statistical mechanics theory of genetic algorithms. In L. Kallel, B. Naudts, and A. Rogers, editors, *Theoretical aspects of evolutionary computation*, pages 87–108. Springer, 2001.

[24] C. R. Stephens and H.Waelbroeck. Schemata evolution and building blocks. *Evolutionary Computation*, 7(2):109–124, 1999.

[25] M. D. Vose. *The simple genetic algorithm*. MIT Press, 1999.

[26] M. D. Vose and A. H. Wright. Stability of vertex fixed points and applications. In L. D. Whitley and M. D. Vose, editors, *Foundations of Genetic Algorithms*, volume 3, pages 103–114. Morgan Kaufmann, 1995.

[27] C. O. Wilke. *Evolutionary dynamics in time-dependent environments*. Shaker Verlag, 1999.

[28] A. H. Wright and J. E. Rowe. Continuous dynamical system models of steady-state genetic algorithms. In W. N. Martin and W. M. Spears, editors, *Foundations of Genetic Algorithms*, volume 6, pages 209–226. Morgan Kaufmann, 2001.

[29] A. H. Wright, J. E. Rowe, R. Poli, and C. R. Stephens. Bistability in a gene pool ga with mutation. In K. De jong, R. Poli, and J. E. Rowe, editors, *Foundations of Genetic Algorithms*, volume 7, pages 63–80. Morgan Kaufmann, 2003.

Chapter 5

Co-evolution and extinction

5.1 Introduction

A *co-evolutionary* system is an evolutionary system in which the fitness of an individual (or *strategy*, to use the game-theoretic term) depends upon the current state of the population. That is, the success or failure of an individual at any time depends upon which other individuals co-exist with it. Such systems have been used in a number of evolutionary optimisation algorithms (see for example [3, 5]). In this paper, we will consider what happens to such systems under the repeated (generational) application of proportional selection, using an infinite population model. Implications for finite populations will be demonstrated empirically.

Previous analytical work on co-evolutionary computational systems has stemmed from evolutionary game theory (see [4] for a general introduction). This body of work typically models such systems in continuous time, using differential equations. For generational implementations, a discrete-time analysis is needed. Along these lines, Ficici and Pollack [2, 1] have adapted ideas from this field and applied them to computational systems. We extend their analysis in this paper.

Following the notation of Michael Vose [6] for simple genetic algorithms, we will represent the set of all possible individuals (or strategies) by

$$\Omega = \{0, 1, \ldots, n-1\}$$

We represent populations as vectors

$$p = (p_0, p_1, \ldots, p_{n-1})$$

where p_k is the proportion of the population utilising strategy $k \in \Omega$. Population vectors are elements of the *simplex*

$$\Lambda = \left\{ x \in \mathbb{R}^n : x_k \geq 0 \text{ for all } k \text{ and } \sum_k x_k = 1 \right\}$$

We will denote the fitness of the elements of Ω by a vector $f \in \mathbb{R}^n$ where f_k is the current fitness of $k \in \Omega$. Since in a co-evolutionary system, fitness depends on the population, we have an

operator $C : \Lambda \to \mathbb{R}^n$ such that $C(p)$ is the fitness vector corresponding to population $p \in \Lambda$. For convenience, we will assume that $C(p)$ always gives a positive fitness function for any $p \in \Lambda$.

Some co-evolutionary systems make use of two (or more) separate populations [3]. The fitness of individuals in one population is determined by the composition of the other population and vice-versa.We will show how to fit such systems into our general framework in section 5.5.

Let $\mathcal{G} : \Lambda \to \Lambda$ be the operator that maps one population to the next.[1] Since we have only fitness proportional selection:

$$\mathcal{G}(p) = \frac{\text{diag}(f)p}{f^T p} \tag{5.1}$$

where $f = C(p)$. This gives us a discrete-time *replicator* equation.

The use of this framework for describing coevolutionary systems was previously reported in [2]. However, that paper was descriptive rather than analytical.

5.2 The fixed-point theorem

We now characterise the fixed-points of equation 5.1. To start with, we make the simplifying assumption that we are only concerned with fixed-points that are complete *mixed strategies*. That is, we are looking for fixed-points p with $p_k \neq 0$ for all $k \in \Omega$.

Theorem 1 *Suppose that $C : \Lambda \to \mathbb{R}^n$ is invertible with inverse C^{-1}. Also assume that there is a fixed-point p of equation 5.1 with $p_k \neq 0$ for all $k \in \Omega$. Then*

$$p = C^{-1}(\lambda \mathbf{1})$$

where $\mathbf{1}$ is the vector containing all ones, and λ is a normalising factor which ensures that $\sum_k p_k = 1$.

Proof

If p is a fixed-point as described then

$$\mathcal{G}(p) = p$$

and so

$$p = \frac{\text{diag}(f)p}{\lambda}$$

where we have set $\lambda = f^T p$, the average fitness of the population. Then

$$\begin{aligned} \lambda p &= \text{diag}(f)p \\ &= \text{diag}(p)f \\ &= \text{diag}(p)C(p) \end{aligned}$$

[1] In the infinite population limit, this map is deterministic — see [6] for details.

Since we have assumed that $p_k \neq 0$ for all k, it follows that diag(p) is invertible. Thus

$$C(p) = \lambda \text{diag}(p)^{-1} p = \lambda \mathbf{1}$$

and so, since C is also assumed to be invertible

$$p = C^{-1}(\lambda \mathbf{1})$$

□

We can also calculate the fitness function at such a fixed-point:

Corollary 2 *If the conditions of the previous theorem hold, then the fitness function at the fixed-point p is $\lambda\mathbf{1}$. That is, the fitness function is constant at the fixed-point.*

Proof

$$f = C(p) = C(C^{-1}(\lambda \mathbf{1})) = \lambda \mathbf{1}$$

□

In the above theorem we were assuming that $p_k \neq 0$ for all k. We now relax this assumption. Let $A \subseteq \Omega$ be a set of strategies which we will assume have become extinct, so that $p_k = 0$ for all $k \in A \subset \Omega$. Then we consider the operator

$$C_A : \Lambda_{n-|A|} \longrightarrow \mathbb{R}^{n-|A|}$$

where $\Lambda_{n-|A|}$ is the projection of Λ onto those dimensions where p is non-zero, given by projecting C onto that space. If C_A is invertible, then we will get another fixed-point (by following the same argument as applied in the proof of the theorem). In this case, the fitness function will be constant over those strategies that have not become extinct.

Since there are 2^n possible ways of selecting some of the components of p to be zero, we have potentially got 2^n sources of fixed-points (depending on the invertibility of the projected operator C_A). One of these choices will be the origin, which we can safely ignore. However, any of the other choices could give rise to fixed-points (possibly more than one, since there might be more than one choice for the inverse of C_A). The main point of this paper is the fact that there are (potentially) a lot of possible fixed-points to co-evolutionary systems. We will illustrate the use of the theorem, and this conclusion, with a series of simple examples in the following sections.

5.3 Linear fitness operators

Suppose we have a co-evolutionary system in which each individual in a population interacts with each other individual, and thereby accrues some fitness. That is, if an individual playing strategy i interacts with an individual playing strategy j, it accrues some fixed amount of fitness $F(i, j)$. If the fitness of each individual is the sum of all these contributions, then

$$f_i = \sum_j F(i, j) r_j$$

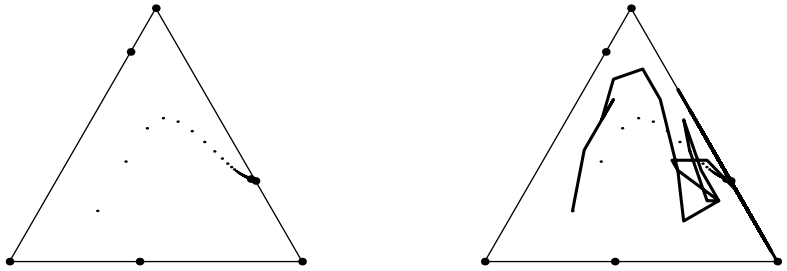


Figure 5.1: The fixed-points of a simple linear co-evolutionary system. An infinite population trajectory is shown (left). Compare this with a typical run of an actual GA (right).

where r_j is the number of individuals playing strategy j. Since proportional selection is invariant with respect to the fitness being scaled, we can divide this through by the population size to get

$$f_i = \sum_j F(i,j)p_j$$

The operator $C : \Lambda \to \mathbb{R}^n$ is in this case a linear operator, and is defined by a matrix

$$C_{i,j} = F(i,j)$$

Our theorem states that we can find the fixed-points of equation 5.1 as follows:

1. Choose some (or none) of the strategies to be extinct. Let this set be $A \subset \Omega$.
2. Project C onto the remaining space. This is easy for linear C, just remove the rows and columns from the matrix C corresponding to each $k \in A$.
3. If the resulting matrix C_A is invertible, then calculate $C_A^{-1}\mathbf{1}$ where $\mathbf{1}$ contains the right number of ones.
4. Normalise the result so that the components sum to one.
5. The normalisation factor is the fitness of each non-extinct strategy at that fixed-point.
6. Repeat for all other combinations of extinct strategies.

Let's work through an example with three strategies. We expect to get a maximum of $2^3 - 1 = 7$ fixed-points (ignoring 0). Consider the matrix:

$$C = \begin{pmatrix} 3 & 20 & 10 \\ 10 & 5 & 30 \\ 8 & 10 & 6 \end{pmatrix}$$

This matrix is invertible and

$$C^{-1} = \begin{pmatrix} -0.0796 & -0.0059 & 0.1622 \\ 0.0531 & -0.0183 & 0.0029 \\ 0.0177 & 0.0383 & -0.0546 \end{pmatrix}$$

Multiplying C^{-1} by $(1,1,1)$ and normalising gives the fixed-point $(0.6616, 0.3257, 0.0127)$ corresponding to $\lambda = 8.626$, which is the constant fitness value at that fixed-point.

Now we must consider what happens when $p_0 = 0$. The projection of C is

$$\begin{pmatrix} 5 & 30 \\ 10 & 6 \end{pmatrix}$$

which is also invertible. Multiplying the inverse by $(1,1)$ and normalising gives us the fixed-point $(0, 0.828, 0.172)$ corresponding to fitness value $\lambda = 9.31$. Note that this is the fitness value for strategies 1 and 2. The fitness for the extinct strategy 0 is different, but irrelevant.

When $p_1 = 0$ a similar process gives us the fixed-point $(0.444, 0, 0.556)$ corresponding to $\lambda = 6.89$. And when $p_2 = 0$ we get the fixed-point $(0.6818, 0.3182, 0)$ with $\lambda = 8.41$.

You can also check that $(1,0,0)$, $(0,1,0)$ and $(0,0,1)$ are fixed-points with corresponding average fitness 3, 5 and 6 respectively. We therefore have seven fixed-points for this system, all of which could, in fact, correspond to actual finite populations. Only one of the fixed-points, however, contains copies of all three strategies.

These fixed-points, together with a typical evolutionary trajectory, are shown in figure 5.1. An actual run of a coevolutionary GA (population size 25) is also shown. It can be seen that, as soon as the population gets near the pair of fixed-points that are close to each other, one strategy immediately becomes extinct. The population then walks randomly along the edge of the simplex until a second strategy becomes extinct. The evolution finishes at the corner corresponding to the remaining strategy.

5.4 A non-linear example

We now look at a simple example in which the fitness of an individual depends in a non-linear way on the composition of the population. Consider the map:

$$C(x,y,z) = (x^2 + y, z(1+x)^2, x)$$

so that the fitness function at a population p is $C(p_0, p_1, p_2)$. This function is invertible with

$$C^{-1}(x,y,z) = (z, x - z^2, \frac{y}{(1+z)^2})$$

Assuming $p_k > 0$ for all k, our fixed-point is given by $C^{-1}(\lambda \mathbf{1})$, where λ must be chosen so that the point is in the simplex. That is, λ must satisfy

$$\lambda^4 - 2\lambda^2 - \lambda + 1 = 0$$

This equation has two complex and two real roots. The only one which, on normalising, produces a fixed-point inside the simplex is $\lambda = 0.5249$, giving a fixed-point

$$(0.5249, 0.2494, 0.2257)$$

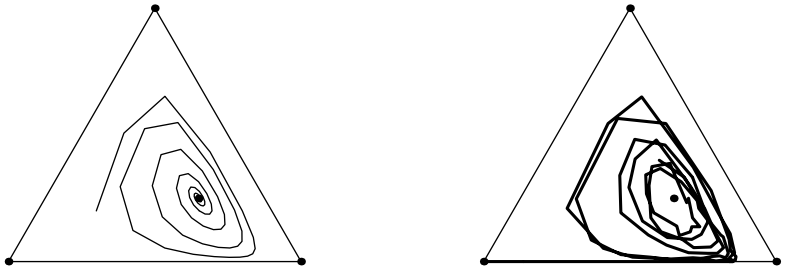


Figure 5.2: The fixed-points of a non-linear co-evolutionary system. An infinite population trajectory is shown (left). Compare this with a typical run of an actual GA (right).

It can be shown that the only other fixed-points are the vertices of the simplex. These points, together with a trajectory of the infinite population model, are shown in figure 5.2. Also shown is a typical run of a GA on this problem (population size 400). It can be seen that after circling the fixed-point for a number of generations, the population accidentally hits the edge of the simplex (that is, one strategy becomes extinct). After that, it isn't long before the system converges to the corner, with only one remaining strategy.

The other possible values of λ are:

$$\begin{aligned} \lambda &= 1.49 \\ \lambda &= -1.0075 - 0.5131\mathbf{i} \\ \lambda &= -1.0075 + 0.5131\mathbf{i} \end{aligned}$$

These all lead to fixed-points outside the simplex. If one imagines that the dynamics are extended to the whole of complex space, then such points can have an effect on the dynamics, even within the simplex. Such a possibility is discussed (in the context of the simple genetic algorithm) in chapter 8 of [6].

5.5 Multiple populations

Multiple population systems can be analysed in a similar way to single populations. For suppose we have m disjoint populations $p^{(0)}, \ldots, p^{(m-1)}$, and the fitness function in any one population depends upon the composition of one of the other populations. For example, let the fitness $f^{(k)}$ of population $p^{(k)}$ depend upon the composition of population $p^{(j)}$, via some operator $C_{(k)}$. That is

$$f^{(k)} = C_{(k)}(p^{(j)})$$

Then when the entire system is at a fixed-point

$$p^{(k)} = \frac{\mathrm{diag}(f^{(k)})p^{(k)}}{\lambda}$$

Following the same argument as above we get

$$\lambda p^{(k)} = \text{diag}(p^{(k)})C_{(k)}(p^{(j)})$$

and therefore

$$p^{(j)} = C_{(k)}^{-1}(\lambda \mathbf{1})$$

allowing the fixed-point to be calculated.

However, the population dynamics of the infinite population model is rarely that straightforward for multiple populations. For example, suppose we have two disjoint populations. The first is a population of predators, and the second a population of prey. The predators have two strategies: they can try to run fast (Speed), or they can develop good eyesight (Sight). The prey also have two strategies: they can also try to run (Run), or they can be camouflaged (Hide). We construct the payoff matrices by asking what happens if an individual from one population meets an individual from the other. The payoff matrix for the predator is:

	Run	Hide
Sight	4	10
Speed	10	2

The payoff for the prey is given by:

	Sight	Speed
Run	10	3
Hide	1	10

Applying the above method to these matrices tells us that the fixed-point for the predators is $(0.4375, 0.5625)$ whereas for the prey it is $(0.571, 0.429)$. That is, there is a slight preference for Speed in the predators, and for Run in the prey at this fixed-point. However, the actual dynamics of the system do not converge to this fixed-point! Rather, they converge to an oscillatory pattern (typical of predator-prey interactions [4]). This is illustrated in figure 5.3. A finite population, of course, will run the risk of extinction whenever the number of representatives of a strategy approaches zero.

A two-population approach is sometimes used to try to generate an *arms race* between sets of problems and their proposed solutions (or solution methods). A classic example is the evolution of sorting networks [3], in which one population contains examples of lists to be sorted, and the other contains networks for performing the sorting procedure. The idea is that as the solutions get more sophisticated, so the problems get harder. This in turn reinforces the need for even more sophisticated solutions. One can similarly imagine a high-level Genetic Programming algorithm working along these lines. One population contains a list of requirements (or formal specifications) while the other provides a set of component software services, each of which may meet some subset of the requirements. The goal here is to evolve a subset of services that meet all the requirements in an efficient manner. A third interpretation of this system is as a model

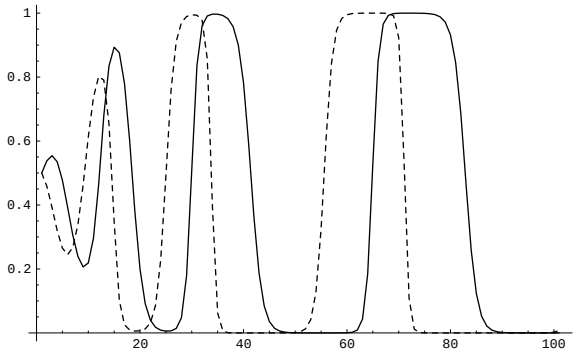


Figure 5.3: The dynamics of predator-prey interactions. Solid line: proportion of the predator strategy Sight. Dashed line: proportion of the prey strategy Hide.

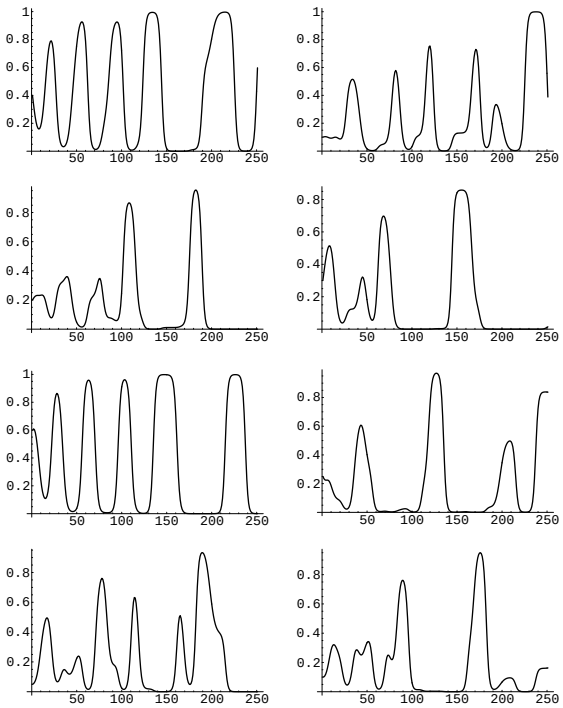


Figure 5.4: The dynamics of the population of problems (left) and solutions (right) for the particular initial condition: $p = (0.1, 0.7, 0.1, 0.1), s = (0.6, 0.25, 0.05, 0.1)$.

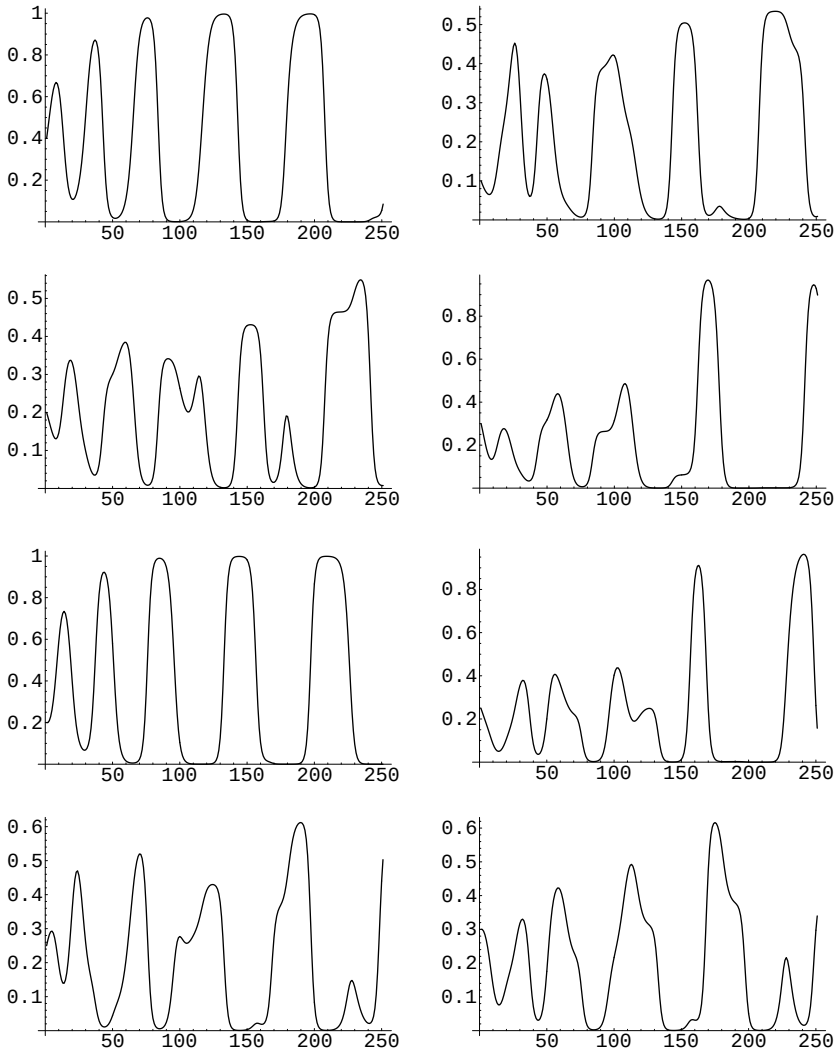


Figure 5.5: The dynamics of the population of problems (left) and solutions (right) for the particular initial condition: $p = (0.4, 0.1, 0.2, 0.3), s = (0.2, 0.25, 0.25, 0.3)$.

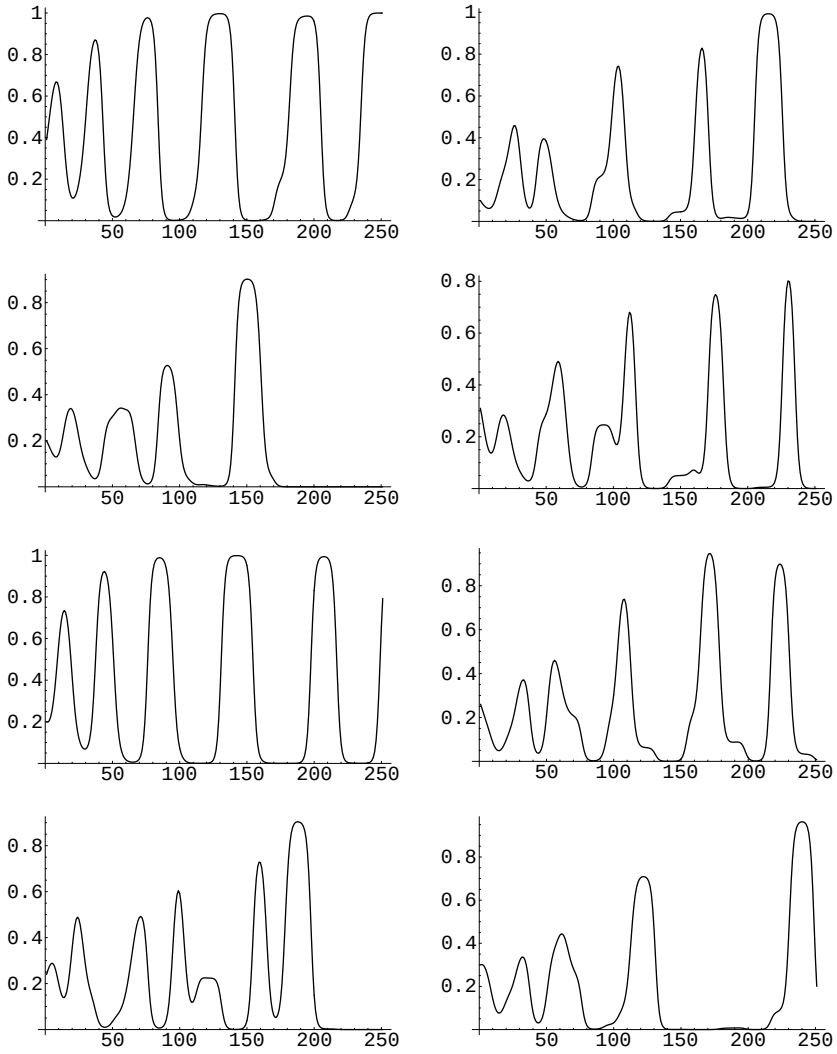


Figure 5.6: The dynamics of the population of problems (left) and solutions (right) for the particular initial condition: $p = (0.39, 0.1, 0.2, 0.31), s = (0.2, 0.26, 0.24, 0.3)$.

of the immune system. The "problems" here represent the invading bacteria, and the "solutions" are the antigens, each of which can destroy one or more invaders.

We can model such a system abstractly as follows. We have a finite set of problems and another set of solutions. Each solution can solve some subset of the problems. Fitness is assigned by randomly drawing a solution and a problem from their respective populations. If the solution solves the problem, then it gets to survive and reproduce, otherwise it dies. Similarly, if the problem is solved, then it dies — otherwise it remains and reproduces. We therefore have evolutionary pressure in favour of good solutions and for difficult problems.

Let p be the population of problems and s the population of solutions. The fitness function to be applied to p is given by Cs where C is the matrix

$$C_{i,j} = \begin{cases} 2 & \text{if } i \text{ solves } j \\ 1 & \text{otherwise} \end{cases}$$

Similarly, the fitness function applied to s is Dp where

$$D_{i,j} = \begin{cases} 1 & \text{if } j \text{ is solved by } i \\ 2 & \text{otherwise} \end{cases}$$

Let us consider a simple example with four problems and four solutions, as follows:

$$C = \begin{pmatrix} 2 & 1 & 1 & 1 \\ 1 & 2 & 2 & 1 \\ 1 & 1 & 2 & 2 \\ 1 & 2 & 1 & 2 \end{pmatrix} \qquad D = \begin{pmatrix} 1 & 2 & 2 & 2 \\ 2 & 1 & 2 & 1 \\ 2 & 1 & 1 & 2 \\ 2 & 2 & 1 & 1 \end{pmatrix}$$

Applying the method described above, we find the fixed-points of the two populations:

$$p = (0.4, 0.2, 0.2, 0.2) \qquad s = (0.4, 0.2, 0.2, 0.2)$$

The fitness at this fixed-point is 1.4 for the population of solutions and 1.6 for the population of problems. This seems to indicate that the first solution is the most important, because it is the only one that can solve the first problem (which is the most difficult). However, it again turns out that this fixed-point is unstable, and the dynamics of the system are far more interesting. In fact, it seems that the dynamics are chaotic. They are very sensitive to the initial conditions, and produce unpredictable behaviour — for example, see figures 5.4, 5.5 and 5.6. The initial conditions for the last two figures are very close, yet the long-term behaviour is quite different.

Such chaotic behaviour has a profound effect on finite population behaviour. For relatively small populations, there is a high risk of extinction, as soon as the proportion of a strategy reaches zero (see figure 5.7. Larger populations can avoid this risk (for a time), but the small amounts of noise due to sampling effects can quickly move the dynamics away from that given by the infinite population model (see figure 5.8). Ultimately, however, the final state will inevitably involve the extinction of many strategies. This can be seen in figure 5.9. The system eventually converges (on this particular run) to the state:

$$p = (0, 1, 0, 0) \qquad s = (0, 1, 0, 0)$$

On different runs, the finite population system ends in different states, though it often takes a long time for the penultimate strategy to become extinct.

5.6 Conclusions

We have given a method for calculating the (infinite population) fixed-points of a discrete-time co-evolutionary system based on fitness-proportional selection. One of the main points of the paper is that there may be many such fixed-points. It is known that these points (even ones outside the simplex, including complex ones) can have an influence over the behaviour of finite populations. One should therefore expect to see, in actual experiments, that the population falls into one of many possible states, often degenerate in nature (in the sense that some of the strategies may have died out).

In the case of multiple populations, we can sometimes find fixed-points using an extension of the method given. However, in even relatively simple cases, convergence to a fixed-point is not guaranteed: indeed the dynamics can act in a chaotic manner.

Bibliography

[1] Sevan G. Ficici and Jordan B. Pollack. Effects of finite populations on evolutionary stable strategies. In L. Darrell Whitley, editor, *Proceedings of the 2000 Genetic and Evolutionary Computation Conference*. Morgan Kaufmann Publishers, 2000.

[2] Sevan G. Ficici and Jordan B. Pollack. A game-theoretic approach to the simple coevolutionary algorithm. In Marc Schoenauer, Kalyanmoy Deb, Guenter Rudolph, Xin Yao, Evelyne Lutton, Juan Julian Merelo, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature VI*. Springer Verlag, 2000.

[3] W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D*, 42:228–234, 1990.

[4] Josef Hofbauer and Karl Sigmund. *Evolutionary games and population dynamics*. Cambridge University Press, 1998.

[5] Mitchell A. Potter and Kenneth A. De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1):1–29, 2000.

[6] Michael D. Vose. *The Simple Genetic Algorithm: Foundations and Theory*. MIT Press, 1999.

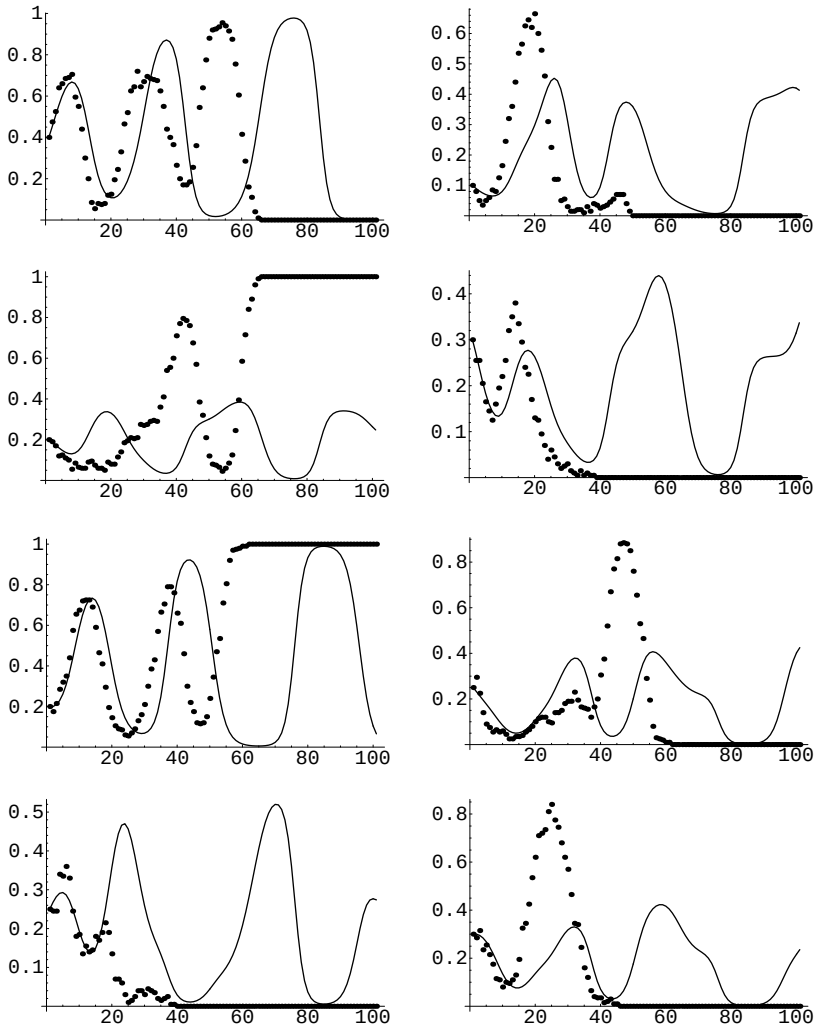


Figure 5.7: Finite population dynamics (population size = 200, shown as dots) of problems (left) and solutions (right) for the particular initial condition: $p = (0.4, 0.1, 0.2, 0.3), s = (0.2, 0.25, 0.25, 0.3)$. Notice that with such a small population, strategies are prone to extinction.

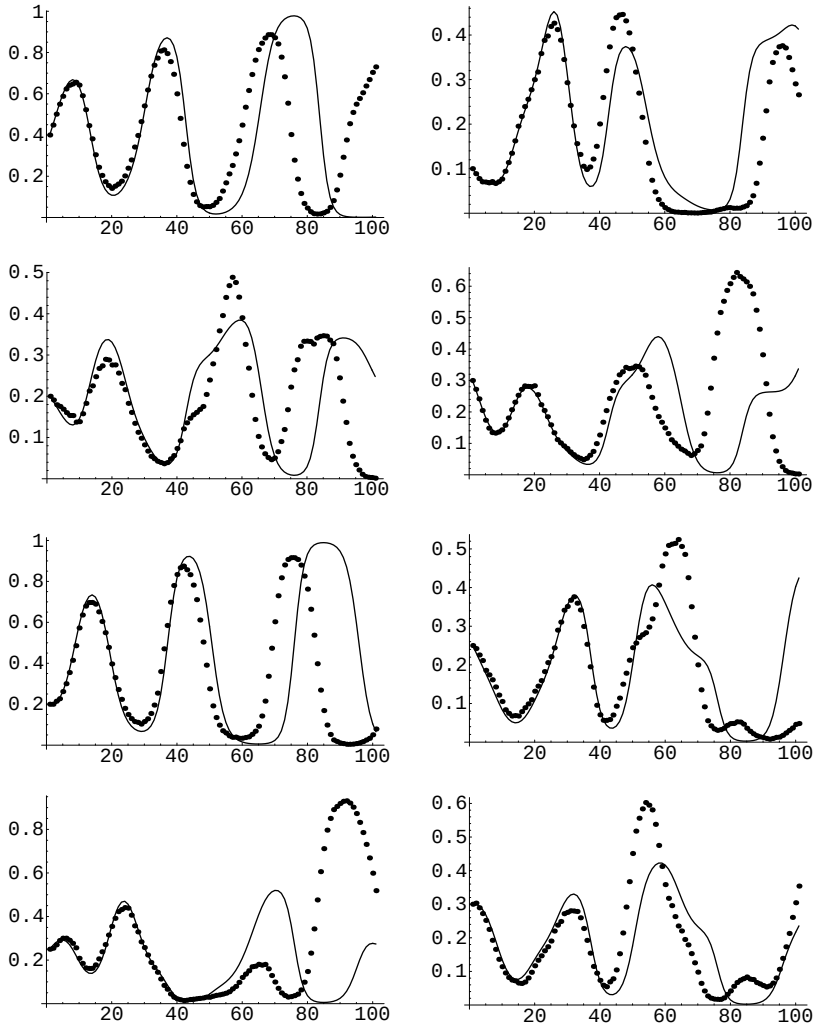


Figure 5.8: Finite population dynamics (population size = 5000, shown as dots) of problems (left) and solutions (right) for the particular initial condition: $p = (0.4, 0.1, 0.2, 0.3), s = (0.2, 0.25, 0.25, 0.3)$. The larger population size means that risk of extinction is reduced. However, sampling noise quickly moves the dynamics away from the infinite population model.

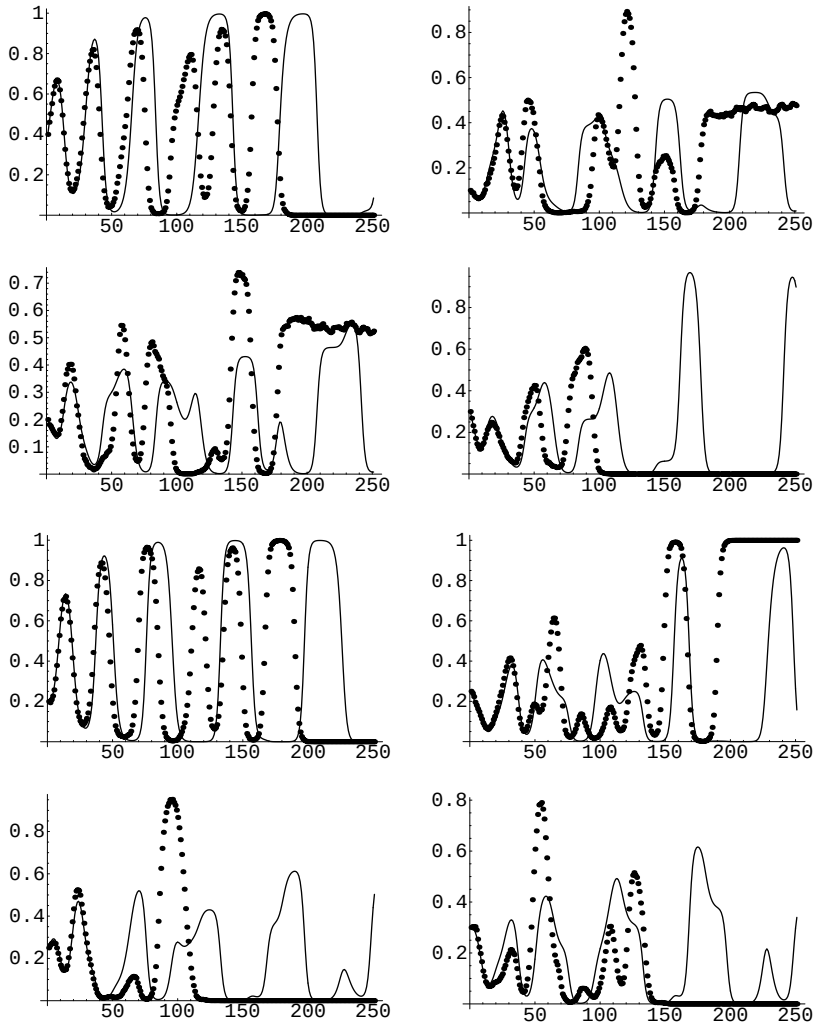


Figure 5.9: Long-term finite population dynamics (population size = 5000, shown as dots) of problems (left) and solutions (right) for the particular initial condition: $p = (0.4, 0.1, 0.2, 0.3), s = (0.2, 0.25, 0.25, 0.3)$.

Chapter 6

Variable-sized structures

This chapter will be published as: Mitavskiy, B. and Rowe, J.E. (2005). A schema-based version of Geiringer's Theorem for nonlinear Genetic Programming with homologous crossover. To appear in Foundations of Genetic Algorithms, Vol. 8.

6.1 Introduction

Geiringer's classical theorem (see [3]) is an important part of GA theory. It has been cited in a number of papers: see, for instance, [9], [10], [16] and [17]. It deals with the limit of the sequence of population vectors obtained by repeatedly applying the crossover operator $\mathcal{C}(p)_k = \sum_{i,j} p_i p_j r_{(i,\, j\to k)}$ where $r_{(i,\, j\to k)}$ denotes the probability of obtaining the individual k from the parents i and j after crossover. In other words, it speaks to the limit of repeated crossover in the case of an infinite population. In [6], a new version of this result was proved for ***finite*** populations, addressing the limiting distribution of the associated Markov chain, as follows. Let $\Omega = \prod_{i=1}^{n} A_i$ denote the search space of a given genetic algorithm (intuitively A_i is the set of alleles corresponding to the i^{th} gene and n is the chromosome length). Fix a population P consisting of m individuals with m being an even number. P can be thought of as an m by n matrix whose rows are the individuals of the population P. Write

$$P = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}.$$

Notice that the elements of the i^{th} column of P are members of A_i. Continuing with the notation used in [9], denote by $\Phi(h, P, i)$ where $h \in A_i$ the proportion of rows, say j, of P for which $a_{ji} = h$. In other words, let $R_h = \{j \,|\, 1 \leq j \leq m \text{ and } a_{ji} = h\}$. Now simply let $\Phi(h, P, i) = \frac{|R_h|}{m}$. The classical Geiringer theorem (see [3] or, [9] for modern notation) says that if one starts with a population P of individuals and runs a genetic algorithm (GA) in the absence of selection and mutation (crossover being the only operator involved) then, in the

"long run", the frequency of occurrence of the individual $(h_1,\ h_2, \ldots, h_n)$ before time t, call it $\Phi(h_1,\ h_2, \ldots, h_n,\ t)$, approaches independence:

$$\lim_{t\to\infty} \Phi(h_1,\ h_2, \ldots, h_n,\ t) = \prod_{i=1}^{n} \Phi(h, P, i).$$

Thereby, Geiringer's theorem tells us something about the limiting frequency with which certain elements of the search space are sampled in the long run, provided one uses crossover alone. In [9] this theorem has been generalized to cover the cases of variable-length GA's and homologous linear genetic programming (GP) crossover. The limiting distributions of the frequency of occurrence of individuals belonging to a certain schema under these algorithms have been computed. The special conditions under which such a limiting distribution exists for linear GP under homologous crossover have been established (see theorem 9 and section 4.2.1 of [9]). In [6] a rather powerful extension of the finite population version of Geiringer's theorem has been established. It was also shown how the finite population versions of these results given in [3] and in [9] are special cases of the generalized Geiringer theorem proved in [6]. In the current paper we shall use the recipe described in [6] to derive a version of Geiringer's theorem for nonlinear GP with homologous crossover (see section 6.3 or [7] for a detailed description of how nonlinear GP with homologous crossover works) which is based on Poli's hyperschemata (see section 6.3 or [7]). The first step in this procedure is to describe the search space and the appropriate family of reproduction transformations so that the resulting GP algorithm is bijective and self-transient in the sense of definition 5.2 of [6]. Then the generalized Geiringer theorem (theorem 5.2 of [6]) as well as corollaries 6.1 and 6.2 of [6] apply. A simplified version of the necessary details is presented in the next section.

6.2 General Framework

In this section we introduce the necessary framework needed to state the schema-based version of Geiringer theorem for GP [1]. First, we describe how a general evolutionary search algorithm works. Let Ω denote a set, which we shall refer to as a *search space*. Denote by $\mathcal{F}$ a family of functions on Ω^2 (every $F \in \mathcal{F}$ is simply a function $F : \Omega^2 \to \Omega^2$). Intuitively, $\mathcal{F}$ is the family of *reproduction transformations*. A typical evolutionary search algorithm works by cycling through a sequence of steps such as selection, reproduction (crossover) and asexual reproduction (mutation). Geiringer theorem deals only with the reproduction steps. In the current paper we shall concentrate only on the bisexual reproduction step which is described in a general setting below (for a more detailed description see [6] and [5]):

A given population $P = (x_1, x_2, \ldots, x_m)$ with $x_i \in \Omega$ is taken as an input. The individuals in P are partitioned into pairs according to some probability distribution $\wp$ on the set of all partitions of the set $\{1,\ 2, \ldots, m\}$ into 2-element subsets. (For simplicity of presentation we assume that m is an even integer. This assumption can be safely ignored and all of the results

[1] The theorem presented in this section is a special case of the extended Geiringer theorem established in [6], yet it is general enough for most applications

would still hold. See [6] for the details.) We shall assume that $\wp$ assigns a positive probability to every possible partition. Although this assumption can be weakened, we shall not bother in the current presentation. For instance, the couples could be

$$Q_1 = (x_{i_1^1},\, x_{i_2^1}),\, Q_2 = (x_{i_1^2},\, x_{i_2^2}),\ \ \ldots\ ,\, Q_j = (x_{i_1^j}, x_{i_2^j}),\ \ldots\ ,\, Q_{\frac{m}{2}} = (x_{i_1^{\frac{m}{2}}}, x_{i_2^{\frac{m}{2}}}).$$

Transformations $T_j \in \mathcal{F}$ are chosen independently according to some probability distribution p on the family $\mathcal{F}$. (The general result holds even under the assumption that the probability distributions are different for every j. However, in practice, this distribution is usually fixed.) Once the choices are made, replace the pairs Q_j with the pairs $T_j(y_{i_1^j},\, y_{i_2^j})$. This way a new population $P' = (y_1,\, y_2, \ldots\, , y_m)$ is obtained.

Remark 1 Given some pair of individuals $(x,\, y) \in \Omega^2$, it is quite possible that there are two or more transformations, let's say for the sake of concreteness F_1 and $F_2 \in \mathcal{F}$, with $F_1 \neq F_2$ but $F_1(x,\, y) = F_2(x,\, y)$.

We now consider the following Markov chain: The states of this Markov chain are all populations [2]. For two populations $\mathbf{x}$ and $\mathbf{y}$ the transition probability of going from $\mathbf{x}$ to $\mathbf{y}$, $p_{\mathbf{x}\to\mathbf{y}}$ is the probability that population $\mathbf{y}$ is obtained from the population $\mathbf{x}$ after a single reproduction step. (Evidently $p_{\mathbf{x}\to\mathbf{y}}$ depends on $\mathcal{F}$, p, and $\wp$) Denote by $p^n_{\mathbf{x},\,\mathbf{y}}$ the probability that population $\mathbf{y}$ is obtained from $\mathbf{x}$ upon the completion of n reproduction steps (this is the n^{th} power of the Markov transition matrix defined above).

Definition 2 We say that a given algorithm is bijective and self-transient if the following conditions hold:

1. Every transformation $T \in \mathcal{F}$ is bijective (i. e. one-to-one and onto).
2. $\mathbf{1} \in \mathcal{F}$ and $p(\mathbf{1}) > 0$. [3] (Here $\mathbf{1} : \Omega^2 \to \Omega^2$ denotes the identity map.)

We shall consider the following relation on the set of all populations:

Definition 3 Fix an evolutionary algorithm $\mathcal{A}$ with a reproduction step as described above. Fix populations $\mathbf{x}$ and $\mathbf{y}$. We shall write $\mathbf{x} \xrightarrow{\mathcal{A}} \mathbf{y}$ if $p^n_{\mathbf{x},\,\mathbf{y}} > 0$ for some n.

The following facts have been established in [6]. Appendix A of [6] reveals the mathematics behind all of the facts listed in the remainder of this section.

Proposition 4 *If a given algorithm $\mathcal{A}$ is bijective and self-transient then $\xrightarrow{\mathcal{A}}$ is an equivalence relation.*

Definition 5 Given a population $P \in \Omega^m$ denote by $[P]_{\mathcal{A}}$ the equivalence class of the population P under the equivalence relation $\xrightarrow{\mathcal{A}}$.

[2] In the current presentation a population is an ordered m-tuple, i. e. an element of Ω^m. Lothar Schmitt used this representation in some of his work (see [14] and [15]).

[3] This condition may be weakened but we want to make the presentation as simple as possible to follow

When a given algorithm $\mathcal{A}$ starts running with the initial population P and reproduction is the only step performed, thanks to proposition 4, only the populations in $[P]_{\mathcal{A}}$ may occur with nonzero probability. It makes sense, therefore, to restrict the state space of our Markov chain to include only the elements of the equivalence class $[P]_{\mathcal{A}}$. We shall call such a Markov process "the Markov chain initiated at P". The generalized Geiringer theorem of [6] tells us something nice about this Markov chain:

Theorem 6 *Let $\mathcal{A}$ denote a bijective and self-transient algorithm. Then the Markov chain initiated at some population $P \in \Omega^m$ is irreducible and its unique stationary distribution is the uniform distribution on $[P]_{\mathcal{A}}$.*

The classical versions of Geiringer theorem, such as the ones established in [3] and in [9] are stated in terms of the "limiting frequency of occurrence" of a certain element of the search space. The following definitions, which also appear in [6], make these notions precise in the general setting:

Definition 7 We define the characteristic function $\mathcal{X} : \Omega^m \times \mathcal{P}(\Omega) \to \mathbb{N} \cup \{0\}$ as follows: $\mathcal{X}(P, S) =$ the number of individuals of P which are the elements of S. (Recall that $P \in \Omega^m$ is a population consisting of m individuals and $S \in \mathcal{P}(\Omega)$ simply means that $S \subseteq \Omega$.)

Example 8 For instance, suppose $\Omega = \{0, 1\}^n$, $P = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{pmatrix}$ and $S \subseteq \Omega = \{0, 1\}^n$ is determined by the Holland schema $(*, 1, *, 1, *)$. Then $\mathcal{X}(P, S) = 3$ because exactly three rows of P, the 1^{st}, the 2^{nd}, and the 5^{th} are in S.

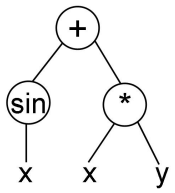
Definition 9 Fix an evolutionary algorithm $\mathcal{A}$ and an initial population $P \in \Omega^m$. Let $P(t)$ denote the population obtained upon the completion of t reproduction steps of the algorithm $\mathcal{A}$ in the absence of selection and mutation. For instance, $P(0) = P$. Denote by $\Phi(S, P, t)$ the proportion of individuals from the set S which occur before time t. That is, $\Phi(S, P, t) = \frac{\sum_{s=1}^{t} \mathcal{X}(P(s), S)}{tm}$. (Notice that tm is simply the total number of individuals encountered before time t. The same individual may be repeated more than once and the multiplicity contributes to Φ.) Denote by $\mathcal{X}(\Box, S) : \Omega^m \to \mathbb{N}$ the restriction of the function $\mathcal{X}$ when the set S is fixed (the notation suggests that one plugs a population P into the box).

Intuitively, $\Phi(S, P, t)$ is the frequency of encountering the individuals in S before time t when we run the algorithm starting with the initial population P.

6.3 Nonlinear Genetic Programming (GP) with Homologous Crossover.

In genetic programming, the search space, Ω, consists of the parse trees which usually represent various computer programs.

Example 10 A typical parse tree representing the program $(+(\sin(x), *(x, y)))$ is drawn below:



Since computers have only a finite amount of memory, it is reasonable to assume that there are finitely many basic operations which can be used to construct programs and that every program tree has depth less than or equal to some integer L. Under these assumptions Ω is a finite set. We may then define the search space as follows:

Definition 11 Fix a signature $\Sigma = (\Sigma_0, \Sigma_1, \Sigma_2, \ldots, \Sigma_N)$ where Σ_i's are finite sets [4]. We assume that $\Sigma_i \neq \emptyset$ for some i and $|\Sigma_j| \neq 1 \, \forall \, j$ [5]. The search space Ω consists of all parse trees having depth at most L. Interior nodes having i children are labelled by the elements of Σ_i. The leaf nodes are labelled by the elements of Σ_0.

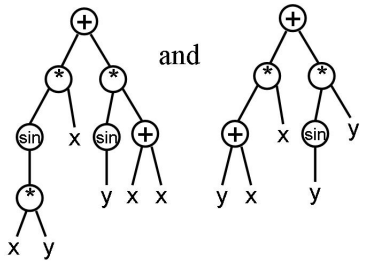
In order to study the appropriate family of reproduction (crossover) transformations with the aim of applying the generalized Geiringer theorem, it is most convenient to exploit Poli's hyperschemata ([7] for a more detailed description).

Definition 12 A Poli's hyperschema is a rooted parse tree which may have two additional labels for the nodes, namely $\#$ and $=$ signs (it is assumed, of course, that neither one of these denotes an operation). $=$ sign may label any interior node v of the tree. Since v does occur in the tree, we must have $|\Sigma_i| > 0$.) The $\#$ sign can only label a leaf node. A given Poli's hyperschema represents the set of all programs whose parse tree can be obtained by replacing the $=$ signs with any operation of the appropriate arities and attaching any program trees in place of the $\#$ signs. Different occurrences of $\#$ or $=$ may be replaced differently. We shall denote by S_t the set of programs represented by a hyperschema t.

A couple of programs fitting the hyperschema $(+(=(\#, x), *(\sin(y), \#))$ are shown below:

[4] Intuitively Σ_i is the set consisting of i-ary operations and Σ_0 consists of the input variables. Formally this does not have to be the case though.

[5] The assumption that $|\Sigma_j| \neq 1 \, \forall \, j$ does not cause any problems since we are free to select any elements from the search space that we want. On the other hand, this assumption helps us to avoid unnecessary complications when dealing with the poset of Poli's hyperschemata later



In order to model the family of reproduction (crossover) transformation in a way which makes it obvious that GP is a bijective and self-transient algorithm, we shall introduce a partial order on the set of all Poli's hyperschema which will make it into a complete lattice (every two elements have a least upper bound). The notion of the least upper bound will be also used to define the *common region* (see [8] for an alternative description of the notion of a common region).

Definition 13 Denote by $\mathcal{O}$ the set of all basic operations which can be used to construct the programs and by $\mathcal{V}$ the set of all variables. Put the following partial order, $\preceq$, on the set $\mathcal{O}\cup\mathcal{V}\cup\{=, \#\}$:

1. $\forall\, a,\, b \in \mathcal{O} \cup \mathcal{V}$ we have $a \preceq b \iff a = b$.
2. $\forall\, a \in \mathcal{O}$ we have $a \preceq =$.
3. $\forall\, a \in \mathcal{O} \cup \mathcal{V}$ we have $a \preceq \#$.
4. $= \preceq =$, $\# \preceq \#$ and $= \preceq \#$.

It is easy to see that $\preceq$ is, indeed a partial order. Moreover, every collection of elements of $\mathcal{O} \cup \mathcal{V} \cup \{=, \#\}$ has the least upper bound under $\preceq$. We are now ready to define the partial order relation on the set of all Poli's hyperschemata:

Definition 14 Let t_1 and t_2 denote two Poli's hyperschemata. We say that $t_1 \geq t_2$ if and only if the following two conditions are satisfied:

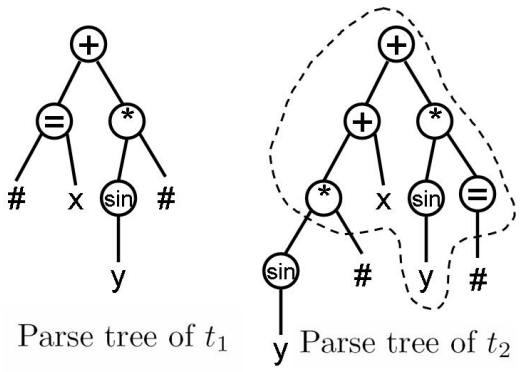
1. the tree corresponding to t_1 when all of the labels are deleted is a subtree of the tree corresponding to t_2 with all of the labels deleted.

2. Every one of the labels (which represents an operation or a variable) of t_1 is $\succeq$ the label of the node in the corresponding position of t_2.

Example 15 For instance, the hyperschema

$$t_1 = (+(=(\#,\, x)), *(\sin(y),\, \#)) \geq t_2 = (+(+(*(\sin(x), y),\, x)), *(\sin(y),\, =(\#)))$$

Indeed, the parse trees of t_1 and t_2 appear on the picture below:



When all the labels in the dashed subtree of the parse tree of t_2 are deleted one gets the tree isomorphic to that obtained from t_1 by deleting all the labels. Thus condition 1 of definition 14 is satisfied. To see that condition 2 is fulfilled as well, we notice that the labels of t_1 are $\preceq$ to the corresponding labels of the dashed subtree of t_2: Indeed, we have $+ \succeq +$, $= \succeq +$, $* \succeq *$, $\# \succeq *$, $x \succeq x$, $\sin \succeq \sin$, $\# \succeq =$ and $y \succeq y$.

Again it is easy to check that $\geq$ is, indeed, a partial order relation on the collection of Poli's hyperschemata. Proposition 16 below tells us even more:

Proposition 16 *Any given collection of Poli's hyperschemata has the least upper bound under* $\geq$.

Proof: Denote by $\mathcal{S}$ a given collection of Poli's hyperschemata. We provide an algorithm to construct the least upper bound of $\mathcal{S}$ as follows: Copies of all the trees in $\mathcal{S}$ are recursively jointly traversed starting from the root nodes to identify the parts with the same shape, i. e. the same arity in the nodes visited. Recursion is stopped as soon as an arity mismatch between corresponding nodes in some two trees from $\mathcal{S}$ is present. All the nodes and links encountered are stored. This way we obtain a tree. It remains to stick in the labels. Each one of the interior nodes is labeled by the least upper bound of the corresponding labels of the trees in $\mathcal{S}$. The label of a leaf node is a variable, say x, if all the labels of the corresponding nodes of the trees in $\mathcal{S}$ are x (which implies that they are leaf nodes themselves). In all other cases the label of the leaf node is the $\#$ sign. It is not hard to see that this produces the least upper bound of the collection $\mathcal{S}$ of parse trees. □

It was pointed out before, that programs themselves are Poli's hyperschemata. The following fact is almost immediate from the explicit construction of the least upper bound carried out in the proof of proposition 16:

Proposition 17 *A given Poli's hyperschema t is the least upper bound of the set S_t of programs determined by t.*

From proposition 17 it follows easily that $\geq$ is order isomorphic to the collection of subsets determined by the Poli's hyperschemata:

Proposition 18 *Let t and s denote Poli's hyperschemata. Denote by S_t and S_s the subsets of the search space determined by the hyperschemata t and s respectively. Then $t \geq s \iff S_t \supseteq S_s$.*

There is another type of schemata which is useful to introduce in order to define the family of reproduction (crossover) transformations:

Definition 19 A shape schema is just a rooted ordered tree. If $\tilde{t}$ is a given shape schema then $S_{\tilde{t}}$ is just the set of all programs whose underlying tree when all the labels are deleted is precisely $\tilde{t}$. Given a Poli's hyperschema s, we shall denote by $\tilde{s}$ the underlying shape schema of s, i. e. the tree obtained by deleting all the labels in s.

The notion of a common region which is equivalent to the one defined below also appears in [8]:

Definition 20 Given two Poli's hyperschemata t and s we define their common region to be the underlying shape schema of the least upper bound of t and s.

Definition 21 Fix a shape schema $\tilde{t}$. We shall say that the set $C_{\tilde{t}} = \{(a,\, b) \,|\, a,\, b$ are program trees and $\tilde{t}$ is the common region of a and $b\}$ is a component corresponding to the shape $\tilde{t}$.

Notice that sets determined by the shape schemata partition the search space:

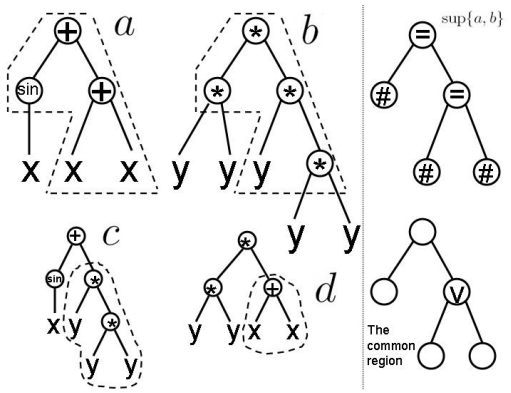
Remark 22 Notice that $\Omega^2 = \bigcup_{\tilde{t} \text{ is a shape}} C_{\tilde{t}}$. Moreover, $C_{\tilde{t}} \cap C_{\tilde{s}} = \emptyset$ for $t \neq s$. (This is so because least upper bounds in a poset are uniquely determined and so the function sending $(a,\, b) \to \sup(a,\, b) \to$ the underlying shape of $\sup(a,\, b)$ is well defined. But then the sets $C_{\tilde{t}}$ are simply the pre-images under a function of singleton subsets of the set of all shapes and, hence, form a partition of Ω^2.)

We now proceed to define the family of reproduction transformations. Our goal is to introduce a family of functions on Ω^2 in such a way that each one of them is easily seen to be bijective (see theorem 6 and definition 2). The idea is to define these transformations on each of the components first:

Definition 23 Fix a shape schema $\tilde{t}$. Fix a node, v of $\tilde{t}$. A one-point partial homologous crossover transformation $T_v : C_{\tilde{t}} \to C_{\tilde{t}}$ is defined as follows: For given $(a,\, b) \in C_{\tilde{t}}$ let $T_v(a,\, b) = (c,\, d)$ where c and d are obtained from the program trees of a and b as follows: First identify the node v in the parse trees of a and b respectively. Now obtain the pair $(c,\, d)$ by swapping the subtrees of a and b rooted at v. (This procedure is described in detail in [8] and it is also illustrated in the example below). Let $\mathcal{F}_{\tilde{t}} = \{T_v \,|\, v$ is a node of $\tilde{t}\}$ denote the family of all partial homologous one-point crossover transformations associated to the shape $\tilde{t}$.

The following example illustrates the concepts in definitions 19, 20 and 23:

Example 24 In the upper left part of the picture parse trees of the two sample programs a and b are shown. Then on the upper right one can see the least upper bound of a and b. On the lower right the underlying tree of the least upper bound of a and b is drawn. According to definition 20, this tree is precisely the common region of the programs a and b. The isomorphic subtrees inside both, a and b, are emphasized inside the dashed areas:



A node v is selected inside the common region. The pair of children $(c, d) = T_v(a,\ b)$ appear on the lower left of the picture above. The subtrees rooted at v which are swapped during crossover are emphasized inside the dashed area.

Remark 25 One does need to show that for $(a,\ b) \in C_{\tilde{t}}$ we have $T_v(a,\ b) \in C_{\tilde{t}}$. A rigorous argument can be given as follows: Clearly $T_v : C_{\tilde{t}} \to \bigcup_{\tilde{t} \text{ is a shape}} C_{\tilde{t}}$ is a well-defined map. Moreover, since v is a node of the least upper bound of a and b and the pair $(c,\ d)$ is obtained simply by swapping the corresponding subtrees rooted at v, we get $s = \sup\{c, d\} \leq \sup\{a, b\}$. Now consider the transformation $F_v : C_{\tilde{s}} \to \bigcup_{\tilde{t} \text{ is a shape}} C_{\tilde{t}}$ and notice that, by definition, we have $F_v(c,\ d) = (a,\ b)$. But then, according to the reasoning above, we have $\sup\{c, d\} \leq \sup\{a, b\}$. Thereby, we get $\sup\{c, d\} \leq \sup\{a, b\} \leq \sup\{c, d\} \implies \sup\{c, d\} = \sup\{a, b\} \implies \tilde{t} = \tilde{s}$. This shows that T_v does, indeed, map into $C_{\tilde{v}}$. Moreover, in the process, we have also observed a couple of very important facts:

1. $T_v \circ T_v = \mathbf{1}_{C_{\tilde{t}}}$ where $\mathbf{1}_{C_{\tilde{t}}}$ denotes the identity map on $C_{\tilde{t}}$. This shows, in particular, that T_v is a bijection.
2. T_v preserves the least upper bounds: $\sup\{a,\ b\} = \sup T_v(a,\ b)$.

We are finally ready to define the family of reproduction transformations on the search space Ω of all programs:

Definition 26 For every shape schema $\tilde{t}$ fix a node $v_{\tilde{t}}$ of $\tilde{t}$. Define a one point crossover transformation $T_{\{v_{\tilde{t}}\}_{\tilde{t} \text{ is a shape schema}}} : \Omega^2 \to \Omega^2$ to be the set-theoretic union of all partial crossover transformations of the form $T_{v_{\tilde{t}}}$. More explicitly, this means that whenever a given pair $(a,\ b) \in \Omega^2$ we must have $(a,\ b) \in C_{\tilde{s}}$ for a unique shape schema $\tilde{s}$ (since, according to remark 22, Ω^2 is a disjoint union of components corresponding to various shapes). But then $T_{\{v_{\tilde{t}}\}_{\tilde{t} \text{ is a shape schema}}}(a,\ b) = T_{v_{\tilde{s}}}(a,\ b)$. Denote by $\mathcal{F}$ the family of all crossover transformations together with the identity map on Ω^2. For simplicity of notation we shall denote the transformations in $\mathcal{F}$ by plain English letters: T, F etc., keeping in mind that every such transformation is determined by making choices of partial crossover transformations on every one of the components.

Remark 27 Thanks to remark 25, everyone of the crossover transformations in the family $\mathcal{F}$ is bijective (since it is a union of bijections on the pieces of a partition). It follows now that the generalized Geiringer theorem (theorem 6) applies to the case of homologous GP.

Remark 28 It is also possible to model uniform GP crossover (this type of crossover is examined in detail in [8]) in the analogous manner. All of the results established in the current paper apply to this case without any modification.

6.4 The Statement of the Schema-Based Version of Geiringer's Theorem for Non-linear GP under Homologous Crossover.

As mentioned before, the schema-based version of Geiringer's theorem for non-linear GP is stated in terms of Poli's hyperschemata.

Definition 29 A Poli's hyperschema of order i is a Poli's hyperschema which has exactly i nodes whose label is not a $\#$ or an $=$ sign.

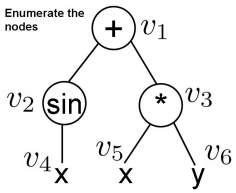
A configuration schema is a 0-order Poli's hyperschema (i.e a hyperschema which has only the equal signs in the interior nodes and $\#$ signs in the leaf nodes.)

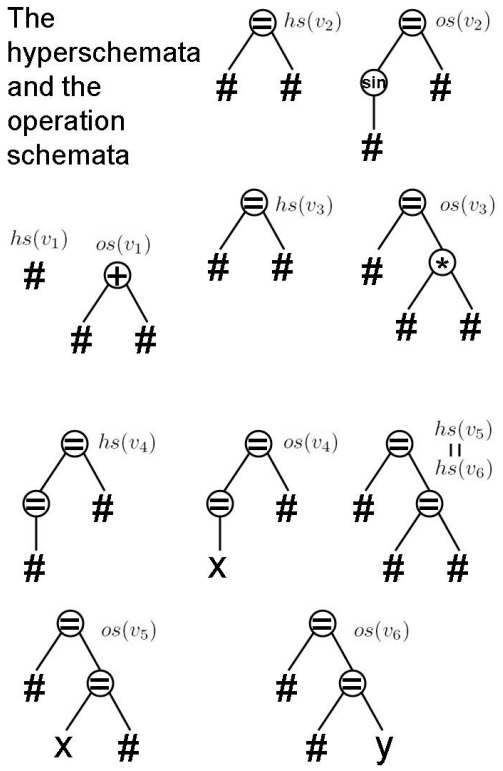
An operation schema is a Poli's hyperschema of order 1 (i. e. a hyperschema which has exactly one node whose label is not a $\#$ or an $=$ sign).

Fix an individual (a parse tree) $\mathbf{u} \in \Omega$. Let v denote any node of $\mathbf{u}$. Let $B(v)$ denote the branch of the shape schema of $\mathbf{u}$ from the root down to the node v. Let $B^+(v) = B(v) \cup \{w \mid w$ is a child of some node z of B with $z \neq v\}$. Now define $cs(v)$ to be the configuration schema whose underlying shape schema is $B^+(v)$. Let o denote an operation or a variable (an element of Σ_i for some i between 0 and N). Now obtain the operation schema os_o from $cs(v)$ by attaching the node labelled by o in place of the $\#$ sign at the node corresponding to v of $cs(v)$. Unless v is the leaf node of $\mathbf{u}$, all the children of this new node are the leaf nodes of os_o labelled by the $\#$ sign. When o is the operation (or the variable) labelling the node v of $\mathbf{u}$, we shall write $os(v)$ instead of os_o.

Notice that if v is a root node then $cs(v)$ is just the schema which determines the entire search space, i. e. the parse tree consisting of a single node labelled by the $\#$ sign. Example 30 illustrates definition 29.

Example 30 Below we list all of the configuration schemata and operation schemata for the individual of example 10:





Recall from definition 7 that $\mathcal{X}(P, S)$ denotes the number of individuals in the population P which are the elements of $S \subseteq \Omega$. The following definition makes it more convenient to state the schema-based version of Geiringer's theorem:

Definition 31 Given a Poli's hyperschema H, we shall write $|H(P)|$ instead of $\mathcal{X}(P, S_H)$ (see definition 12) to denote the number of individuals (counting repetitions) in the population P fitting the hyperschema H.

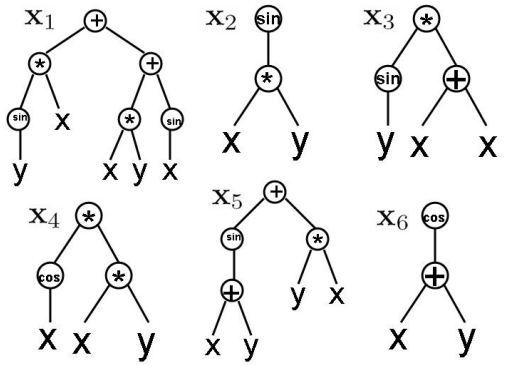
We can now finally state the Geiringer's theorem for non-linear GP under homologous crossover:

Theorem 32 *Fix an initial population $P \in \Omega^m$ and an individual $\mathbf{u} \in \Omega$. Suppose every pair of individuals has a positive probability to be paired up for crossover and every transformation in $\mathcal{F}$ has a positive probability of being chosen* [6]*. Then the limiting frequency of occurrence of a given individual* $\mathbf{u}$*,*

$$\lim_{t\to\infty} \Phi(\mathbf{u},\, P,\, t) = \prod_{v \text{ is a node of } \mathbf{u}} \frac{|os(v)(P)|}{|cs(v)(P)|}.$$

Example 33 To illustrate how theorem 32 can be applied in practice, suppose we are interested in computing the frequency of encountering the individual $\mathbf{u}$ from examples 10 and 30 when the initial population of 6 individuals pictured below is chosen:

[6] These conditions can be slightly relaxed, but we try to present the main idea only



The number of individuals in P fitting the operation schema $os(v_1)$ is 2 (these are $\mathbf{x}_1$ and $\mathbf{x}_5$) while every individual fits the configuration schema $cs(v_1)$. Therefore $\frac{|os(v_1)(P)|}{|cs(v_1)(P)|} = \frac{2}{6} = \frac{1}{3}$. 4 individuals, namely $\mathbf{x}_1$, $\mathbf{x}_3$, $\mathbf{x}_4$ and $\mathbf{x}_5$ fit $cs(v_2) = cs(v_3)$, among these only 2 individuals, namely $\mathbf{x}_3$ and $\mathbf{x}_5$, fit $os(v_2)$ and 2 individuals, $\mathbf{x}_4$ and $\mathbf{x}_5$ fit $os(v_3)$ so that $\frac{|os(v_2)(P)|}{|cs(v_2)(P)|} = \frac{|os(v_3)(P)|}{|cs(v_3)(P)|} = \frac{2}{4} = \frac{1}{2}$. Individuals $\mathbf{x}_3$, $\mathbf{x}_4$ and $\mathbf{x}_5$ fit the configuration schema $cs(v_4)$ while only $\mathbf{x}_4$ fits the operation schema $os(v_4)$ so that $\frac{|os(v_4)(P)|}{|cs(v_4)(P)|} = \frac{1}{3}$. $\mathbf{x}_1$, $\mathbf{x}_3$, $\mathbf{x}_4$ and $\mathbf{x}_5$ fit $cs(v_5) = cs(v_6)$. Among these only $\mathbf{x}_3$ and $\mathbf{x}_4$ fit $os(v_5)$ while only $\mathbf{x}_4$ fits $os(v_6)$ so that $\frac{|os(v_5)(P)|}{|cs(v_5)(P)|} = \frac{2}{4} = \frac{1}{2}$ and $\frac{|os(v_6)(P)|}{|cs(v_6)(P)|} = \frac{1}{4}$. Thereby, according to theorem 32, we obtain:

$$\lim_{t\to\infty} \Phi(\mathbf{u},\, P,\, t) = \prod_{i=1}^{6} \frac{|os(v_i)(P)|}{|cs(v_i)(P)|} =$$

$$= \frac{1}{3} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{3} \cdot \frac{1}{2} \cdot \frac{1}{4} = \frac{1}{288}.$$

Roughly speaking, this means that if we run GP starting with the population P pictured above, in the absence of mutation and selection (crossover being the only step) for an infinitely long time, the individual $\mathbf{u}$ will be encountered on average 1 out of 288 times.

Example 34 Notice that linear GP (or, equivalently, variable length GA) as described in [9] is a special case of nonlinear GP when $\forall\, i > 1\ \Sigma_i = \emptyset$ and Σ_0 and $\Sigma_1 \neq \emptyset$. Indeed, the elements of such a search space are parse trees such that every interior node has exactly one child and the depth of the tree is bounded by some integer N. One can think of such a tree as a sequence of labels $(a_1,\, a_2, \ldots, a_n)$, the first label affiliated with the root node, second label with the child of the root node and so on. The label a_n is affiliated with the leaf node. This gives us a one-to-one correspondence, call it ϕ between the search space for nonlinear GP in our specific case when $\forall\, i > 1\ \Sigma_i = \emptyset$ while Σ_0 and $\Sigma_1 \neq \emptyset$ and the search space for linear GP which preserves crossover. The following types of schemata have been introduced in [9]:

Definition 35 The schema $H = (*^{i-1},\, h_i,\, \#)$ represents the subset $S_H = \{\mathbf{x} = (x_1,\, x_2, \ldots, x_l) \,|\, l > i \text{ and } x_i = h_i\}$. In words, S_H is simply the set of all individuals whose length is at least $i+1$ and whose i^{th} allele is h_i.

Definition 36 The schema $H = (*^i,\, \#)$ represents the subset

$$S_H = \{\mathbf{x} = (x_1,\, x_2, \ldots, x_l) \,|\, l > i\}.$$

In words, S_H is simply the subset of all individuals whose length is at least $i + 1$.

Definition 37 The schema $H = (*^{i-1}, h_i)$ represents the subset

$$S_H = \{\mathbf{x} = (x_1,\, x_2, \ldots, x_i) \,|\, x_i = h_i\}$$

of the search space which is simply the set of all individuals of length exactly equal to i whose i^{th} (last) allele is h_i.

The reader may check that under the correspondence ϕ the configuration schemata correspond to the schemata $H_i = (*^i,\, \#)$ for $i \geq 1$, operation schemata correspond to the schemata of the form $H = (*^{i-1},\, h_i,\, \#)$ and of the form $H = (*^{i-1}, h_i)$ for $i > 1$. Finally, the hyperschema $t_{(1,\,1)}$ corresponds to the schema $H = (h_1,\, \#)$. Fix a population $P \in \Omega^m$. Recall that we denote by $|H|$ the number of individuals in P which fit the schema H counting repetitions. Also recall from definition 9 that $\Phi(S_H,\, P,\, 1) = \frac{|H|}{m}$ denotes the fraction of the number of individuals of P which fit the schema H. To abbreviate the notation we shall write $\Phi(H,\, P,\, 1)$ instead of $\Phi(S_H,\, P,\, 1)$. Fix an individual $\mathbf{u} = (h_1,\, h_2, \ldots, h_n) \in \Omega$. Theorem 32 tells us that

$$\lim_{t\to\infty} \Phi(\mathbf{u},\, P,\, t) = \frac{|(h_1,\, \#)|}{m} \cdot \Big(\prod_{i=1}^{n-2} \frac{|(*^i,\, h_{i+1},\, \#)|}{|(*^i,\, \#)|}\Big) \cdot \frac{|(*^{n-1},\, h_n)|}{|(*^{n-1},\, \#)|} =$$

$$= \frac{|(h_1,\, \#)|}{m} \cdot \Big(\prod_{i=1}^{n-2} \frac{\frac{|(*^i, h_{i+1}, \#)|}{m}}{\frac{|(*^i, \#)|}{m}}\Big) \cdot \frac{\frac{|(*^{n-1}, h_n)|}{m}}{\frac{|(*^{n-1}, \#)|}{m}} =$$

$$= \Phi(h_1,\, \#) \cdot \Big(\prod_{i=1}^{n-2} \frac{\Phi(*^i,\, h_{i+1},\, \#)}{\Phi(*^i,\, \#)}\Big) \cdot \frac{\Phi(*^{n-1},\, h_n)}{\Phi(*^{n-1},\, \#)} =$$

$$= \Phi(*^{n-1},\, h_n) \cdot \frac{\prod_{i=n-2}^{0} \Phi(*^i,\, h_{i+1},\, \#)}{\prod_{i=n-1}^{1} \Phi(*^i,\, \#)} = \Phi(*^{n-1},\, h_n) \cdot \prod_{i=n-1}^{i=1} \frac{\Phi(*^{i-1},\, h_i,\, \#)}{\Phi(*^i,\, \#)}$$

which is precisely the formula obtained in [9].

6.5 How Do We Obtain Theorem 32 from Theorem 6?

The following couple of corollaries from [6] are useful in obtaining the schema-based versions of Geiringer theorem for various evolutionary algorithms. Throughout, we shall denote by $\varrho_{[P]_{\mathcal{A}}}$ the uniform probability distribution on the set $[P]_{\mathcal{A}}$ (see definition 5).

Corollary 38 *Fix a bijective and self-transient algorithm $\mathcal{A}$ and an initial population $P \in \Omega^m$. Fix a set S of individuals in Ω ($S \subseteq \Omega$). Then $\lim_{t\to\infty} \Phi(S,\, P,\, t) = \frac{1}{m} E_{\varrho_{[P]_\mathcal{A}}}(\mathcal{X}(\square,\, S))$ (here $E_{\varrho_{[P]_\mathcal{A}}}(f)$ denotes the expectation of the random variable f with respect to the uniform distribution on the set $[P]_\mathcal{A}$).*[7]

To state the next corollary which brings us one step closer to deriving results similar in flavor to Geiringer's original theorem we need one more, purely formal, assumption about the algorithm:

Definition 39 We say that a given algorithm $\mathcal{A}$ is regular if the following is true: for every population $P = (x_1,\, x_2, \ldots,\, x_m) \in \Omega^m$ and for every permutation $\pi \in \mathcal{S}_m$, the population obtained by permuting the elements of P by π, namely $\pi(P) = (x_{\pi(1)},\, x_{\pi(2)}, \ldots,\, x_{\pi(m)}) \in [P]_\mathcal{A}$. In words this says that the equivalence classes $[P]_\mathcal{A}$ are permutation invariant.

Remark 40 Definition 39 is only needed because our description of an evolutionary search algorithm uses the ordered multi-set model. This makes the generalized Geiringer theorem (theorem 6) look nice (the stationary distribution is uniform on $[P]_\mathcal{A}$). A disadvantage of the multi-set model is that it allows algorithms which are not regular. If we were to use the model of [17] where the order of elements in a population is not taken into account (a reasonable assumption since most evolutionary algorithms used in practice are, indeed, regular) then the Generalized Geiringer theorem would have to be modified accordingly since the stationary distribution of the corresponding Markov process would be different from uniform (it is not difficult to compute it though since the corresponding Markov chain is just a "projection" of the one used in the current paper).

Corollary 41 *Fix a regular bijective and self-transient algorithm $\mathcal{A}$ and an initial population $P \in \Omega^m$. Denote by $\varrho_{[P]_\mathcal{A}}$ the uniform probability distribution on $[P]_\mathcal{A}$ (see definition 5). Fix a set S of individuals in Ω ($S \subseteq \Omega$). Then $\lim_{t\to\infty} \Phi(S,\, P,\, t) = \varrho_{[P]_\mathcal{A}}(\mathcal{V}_S)$ where*

$$\mathcal{V}_S = \{P \,|\, P \in [P]_\mathcal{A} \text{ and the } 1^{st} \text{ individual of } P \text{ is an element of } S\}.$$

Corollaries 38 and 41 are proved in section 6 of [6]. When deriving schema-based versions of Geiringer theorem for a specific algorithm the following strategy may be implemented: Continuing with the notation in corollaries 38 and 41, suppose we are given a nested sequence of subsets of the search space: $S_1 \supseteq S_2 \supseteq \ldots \supseteq S_n$. According to corollary 41,

$$\lim_{t\to\infty} \Phi(S_n,\, P,\, t) = \varrho_{[P]_\mathcal{A}}(\mathcal{V}_{S_n}) = \frac{|\mathcal{V}_{S_n}|}{|[P]_\mathcal{A}|} = \frac{|\mathcal{V}_{S_n}|}{|\mathcal{V}_{S_{n-1}}|} \cdot \frac{|\mathcal{V}_{S_{n-1}}|}{|[P]_\mathcal{A}|} =$$

$$= \frac{|\mathcal{V}_{S_n}|}{|\mathcal{V}_{S_{n-1}}|} \cdot \frac{|\mathcal{V}_{S_{n-1}}|}{|\mathcal{V}_{S_{n-2}}|} \cdot \ldots \cdot \frac{|\mathcal{V}_{S_2}|}{|\mathcal{V}_{S_1}|} \cdot \frac{|\mathcal{V}_{S_1}|}{|[P]_\mathcal{A}|} =$$

$$= \varrho_{[P]_\mathcal{A}}(\mathcal{V}_{S_1}) \cdot \prod_{j=0}^{n-2} \frac{|\mathcal{V}_{S_{n-j}}|}{|\mathcal{V}_{S_{n-j-1}}|} = \frac{1}{m} E_{\varrho_{[P]_\mathcal{A}}}(\mathcal{X}(\square,\, S)) \cdot \prod_{j=0}^{n-2} \frac{|\mathcal{V}_{S_{n-j}}|}{|\mathcal{V}_{S_{n-j-1}}|}$$

[7] Throughout the paper, whenever a limit is involved, the equality is meant to hold for almost every infinite sequence of trials.

Notice that $\frac{|\mathcal{V}_{S_j}|}{|\mathcal{V}_{S_{j-1}}|}$ is just the proportion of populations in $[P]_{\mathcal{A}}$ whose first individual is a member of S_j inside the set of populations in $[P]_{\mathcal{A}}$ whose first individual is a member of S_{j-1}.

Corollary 42 *Fix a regular, bijective and self-transient algorithm $\mathcal{A}$ and an initial population $P \in \Omega^m$. Fix a nested sequence of subsets $S_1 \supseteq S_2 \supseteq \ldots \supseteq S_n$ of individuals in Ω ($S_1 \subseteq \Omega$). Then $\lim_{t\to\infty} \Phi(S_n,\, P,\, t) = \frac{1}{m} E_{\varrho_{[P]_{\mathcal{A}}}}(\mathcal{X}(\square,\, S)) \cdot \prod_{j=0}^{n-2} \frac{|\mathcal{V}_{S_{n-j}}|}{|\mathcal{V}_{S_{n-j-1}}|}$ where, as before, $\mathcal{V}_S$ denotes the set of all populations whose first individual is a member of S for a given subset $S \subseteq \Omega$.*

Denote by $\mathcal{A}$ a given GP algorithm. Fix an individual $x \in \Omega$. In order to apply corollary 42, we may choose a descending chain of Poli's hyperschemata $t_1 \geq t_2 \geq \ldots \geq t_n = x$. Fix an initial population P. To avoid putting many subscripts, we shall write $\mathcal{V}_t$ instead of $\mathcal{V}_{S_t}$ for the set of all populations in $[P]_{\mathcal{A}}$ (see definition 3) whose 1st individual is a member of S_t (the set of individuals determined by the hyperschema t). Now fix an individual $x \in \Omega$. In order to construct the desired sequence of nested hyperschemata, we assign the following numerical labelling to the nodes of the parse tree of $\mathbf{u}$: The nodes are labelled by the pairs of integer coordinates. The first coordinate shows the depth of the tree and the second coordinate shows how far to the right a given node at the depth specified by the first coordinate is located. Notice, for instance, that the root node is labelled by the coordinates $(1,\, 1)$. We also introduce the following lexicographic linear ordering on the set of coordinate pairs:

Definition 43 $(a,\, b) \leq (c,\, d)$ if and only if either $a \leq c$ or ($a = c$ and $b \leq d$).

It is well known and easy to verify that this defines a linear ordering.

Definition 44 Given a pair of coordinates $(i,\, j)$, denote by $\uparrow (i,\, j)$ the immediate successor of $(i,\, j)$ under the lexicographic ordering defined above. Explicitly,

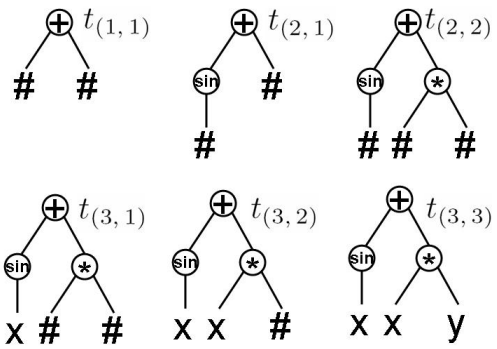
$$\uparrow (i,\, j) = \begin{cases} (i+1,\, 1) \text{ if } (i,j) \text{ labels the rightmost node of } \mathbf{u} \text{ at depth } i \\ (i,\, j+1) \text{ otherwise} \end{cases}$$

We obtain the desired nested sequence of hyperschemata for the given individual $\mathbf{u}$ recursively in the following manner:

Definition 45 Define $t_{(1,1)}$ to be the hyperschema whose root node has the same label (operation) and arity as that of the root node of $\mathbf{u}$. All children of the root node are the leaf nodes labelled by the $\#$ sign. Once the hyperschema $t_{(i,j)}$ has been constructed, we obtain the hyperschema $t_{\uparrow(i,j)}$ by attaching the node of $\mathbf{u}$ with coordinate $\uparrow (i,j)$ in place of the $\#$ sign at coordinate $\uparrow (i,j)$ to the parse tree of $t_{(i,j)}$. Unless this node, call it v, is a leaf node of $\mathbf{u}$, all children of this new node are the leaf nodes of $t_{\uparrow(i,j)}$ labelled by the $\#$ sign.

We illustrate the construction with an explicit example:

Example 46 Below, the nested sequence $t_{(1,\,1)} \geq t_{(2,\,1)} \geq t_{(2,\,2)} \geq t_{(3,\,1)}, \geq t_{(3,\,2)} \geq t_{(3,\,3)}$ corresponding to the program of example 10 is drawn explicitly:



The formula for the limiting frequency of occurrence of a given program u in corollary 42 involves the ratios of the form $\frac{\mathcal{V}_{t_{\uparrow(i,j)}}}{\mathcal{V}_{t_{(i,j)}}}$. It turns out that these ratios can be expressed nicely in terms of the presence of certain configuration and operation schemata in the initial population P:

Definition 47 Given a program tree $\mathbf{u}$ and the corresponding nested sequence $t_{(1,1)} \geq t_{(2,1)} \geq \ldots \geq t_{(i,j)} \geq t_{\uparrow(i,j)}, \geq \ldots \geq t_{(l,k)} = \mathbf{u}$ of hyperschemata as in definition 45, for every $(i,\, j) \neq (l,\, k)$, denote by $cs_{(i,\,j)}$ ($os_{(i,\,j)}$) the configuration schema $cs(v)$ (operation schema $os(v)$) where v is the node of $\mathbf{u}$ with coordinate $\uparrow (i,\, j)$.

Example 48 Continuing with examples 10 and 30 notice that for the individual in these examples we have $cs_{(1,1)} = cs_{(2,1)} = cs(v_2) = cs(v_3)$ while $os_{(1,1)} = os(v_2)$ and $os_{(2,1)} = os(v_3)$ (see example 30), $cs_{(2,2)} = cs(v_4)$ while $os_{(2,2)} = os(v_4)$ and $cs_{(3,1)} = cs_{(3,2)} = cs(v_5) = cs(v_6)$ while $os_{(3,1)} = os(v_5)$ and $os_{(3,2)} = os(v_6)$.

The following "orbit description" lemma is the reason for introducing configuration and operation schemata: We prove the lemma under the following special assumption:

Definition 49 We say that a population P is special with respect to the individual $\mathbf{u}$ if for every node v of $\mathbf{u}$ and for every operation (or variable) o we have $|os_o(P)| \leq 1$ where os_o is obtained from $cs(v)$ by means of attaching the operation o at the leaf node of $cs(v)$ corresponding to v as described in definition 29.

Definition 49 basically requires that no 2 operations (or variables) occurring in P at the specified location are the same. It turns out that the orbit description lemma stated below is a lot more convenient to prove under this special assumption. The general case will then follow by introducing enough extra labels for the operations and variables involved and then deleting the extra labels.

Lemma 50 *Fix an initial population P and a program $\mathbf{u} \in \Omega$ Assume that the population P is special with respect to the individual $\mathbf{u}$. Suppose every pair of individuals has a positive probability to be paired up for crossover and every transformation in $\mathcal{F}$ has a positive probability of being chosen* [8]. *Consider the sequences of hyperschemata $t_{(1,1)} \geq t_{(2,1)} \geq \ldots \geq$*

[8] These conditions can be slightly relaxed, but we try to present the main idea only

$t_{(i,j)} \geq t_{\uparrow(i,j)}, \geq \ldots \geq t_{(l,k)} = \mathbf{u}$, $\{cs_{(i,j)} \,|\, (i,\, j)$ *is a coordinate of* $\mathbf{u},\ (i,\, j)$ *is not the maximal coordinate* $\}$ *and* $\{os_{(i,j)} \,|\, (i,\, j)$ *is a coordinate of* $\mathbf{u},\ (i,\, j)$ *is not the maximal coordinate* $\}$ *corresponding to the individual* $\mathbf{u}$. *For a given hyperschema* t, *denote by* $|t(P)|$ *the number of individuals in* P *which fit the hyperschema* t *counting repetitions. Suppose* $\forall$ *non-maximal pairs of coordinates* $(i,\, j)$ *we have* $|os_{(i,j)}(P)| \neq 0$ *and* $|t_{(1,\,1)}(P)| \neq 0$. *Then it is true that* $\forall\,(i,\, j)\ \frac{\mathcal{V}_{t_{\uparrow(i,j)}}}{\mathcal{V}_{t_{(i,j)}}} = \frac{1}{|cs_{(i,j)}(P)|}$.

Proof: The key idea is to observe the following fact:

Claim: Fix a coordinate $(i,\, j)$. Fix any two operation schemata os_1 and os_2 which are obtained from $cs_{(i,j)}$ by attaching either a variable or an operation at the node $(i,\, j)$. Suppose $\exists$ individuals in P fitting both, os_1 and os_2. Then $|\mathcal{V}_{t_{(i,\,j)}} \cap \mathcal{V}_{os_1}| = |\mathcal{V}_{t_{(i,\,j)}} \cap \mathcal{V}_{os_2}|$. *Proof:* Consider the map $F : [P]_{\mathcal{A}} \to [P]_{\mathcal{A}}$ defined as follows: Given a population, say $Q \in [P]_{\mathcal{A}}$, notice that $\exists$ an individual, say $\mathbf{x}_1$, in Q fitting the operation schema os_1 (due to the way crossover is defined, the number of individuals fitting the operation schema $os_1(Q)$ is the same in every population $Q \in [P]_{\mathcal{A}}$). Moreover, such an individual is unique since we assumed that all operations appearing in the individuals of P are distinct. Likewise, $\exists$ unique individual in Q, say $\mathbf{x}_2$ fitting the operation schema os_2. Pair up individuals $\mathbf{x}_1$ and $\mathbf{x}_2$ and pair up the rest of the individuals arbitrarily for crossover. Select the crossover transformation T_v where v is the node with coordinate $(i,\, j)$ for the pair $(\mathbf{x}_1,\, \mathbf{x}_2)$ and choose the identity transformation for the rest of the pairs. Now let $F(Q)$ be the population obtained upon the completion of the reproduction step described above (notice that $F(Q) \in [P]_{\mathcal{A}}$ by definition of $[P]_{\mathcal{A}}$). Notice also that F is its own inverse (i. e. $F \circ F = \mathbf{1}_{[P]_{\mathcal{A}}}$). This tells us, in particular, that F is bijective. Moreover, it is clear from the definitions that $F(\mathcal{V}_{t_{(i,\,j)}} \cap \mathcal{V}_{os_1}) \subseteq \mathcal{V}_{t_{(i,\,j)}} \cap \mathcal{V}_{os_2}$ and, likewise, $F(\mathcal{V}_{t_{(i,\,j)}} \cap \mathcal{V}_{os_2}) \subseteq \mathcal{V}_{t_{(i,\,j)}} \cap \mathcal{V}_{os_1}$. The desired conclusion follows at once. □

Now observe that $t_{\uparrow(i,j)} = t_{(i,j)} \cap os_{(i,j)}$ so that $\mathcal{V}_{t_{\uparrow(i,j)}} = \mathcal{V}_{t_{(i,\,j)}} \cap \mathcal{V}_{os_{(i,\,j)}}$ and $t_{(i,j)} = \bigcup_{o \text{ is an operation or a variable}} (t_{(i,j)} \cap os_o)$ where os_o is obtained from $cs_{(i,j)}$ by attaching the operation (or variable) o at the node $\uparrow (i,\, j)$. Therefore we also have $\mathcal{V}_{t_{(i,\,j)}} = \bigcup_{o \text{ is an operation or a variable,}} (\mathcal{V}_{t_{(i,\,j)}} \cap \mathcal{V}_{os_o})$. Since operations can not appear or disappear from a population during crossover, $\mathcal{V}_{os_o} \neq \emptyset \implies \exists$ an individual in P fitting the operation schema os_o. Thus the only sets of the form $\mathcal{V}_{t_{(i,\,j)}} \cap \mathcal{V}_{os_o}$ which may possibly contribute to the union above are these for which $\exists$ an individual in P fitting the operation schema os_o. According to the claim above, all such sets contribute exactly the same amount. Moreover, by assumption $os_{(i,\,j)}(P) \neq \emptyset$, and so we have $|\mathcal{V}_{t_{(i,\,j)}}| = n \cdot |\mathcal{V}_{t_{(i,\,j)}} \cap \mathcal{V}_{os_{(i,\,j)}}| = n \cdot |\mathcal{V}_{t_{(i,\,j)} \cap os_{(i,\,j)}}| = n \cdot |\mathcal{V}_{t_{\uparrow(i,\,j)}}| \implies \frac{|\mathcal{V}_{t_{\uparrow(i,\,j)}}|}{|\mathcal{V}_{t_{(i,\,j)}}|} = \frac{1}{n}$ where n is the number of operation schemata of the form os_o for which $\exists$ an individual in P fitting the operation schema os_o and the last implication holds under the condition that $|\mathcal{V}_{t_{(i,\,j)}}| \neq 0$. This condition is, indeed satisfied. (Suppose not. Let $(a,\, b)$ denote the smallest coordinate such that $|\mathcal{V}_{t_{(a,\,b)}}| = 0$. Notice that $(a,\, b) \neq (1,\, 1)$ since $|\mathcal{V}_{t_{(1,\,1)}}| \neq 0$. (By assumption $\exists$ an individual, say $\mathbf{x}$, in P fitting the hyperschema $t_{(1,\,1)}$. Even if $\mathbf{x}$ is not the 1^{st} individual of P, by performing crossover of $\mathbf{x}$ with the 1^{st} individual of P at the root node one gets a population $Q \in \mathcal{V}_{t_{(1,\,1)}}$.) But then $(a,\, b) = \uparrow (i,\, j)$ for some coordinate $(i,\, j)$ and according to the equation above we have $|\mathcal{V}_{t_{(i,\,j)}}| = n \cdot |\mathcal{V}_{t_{\uparrow(i,\,j)}}| = n \cdot |\mathcal{V}_{t_{(a,\,b)}}| = 0$ which contradicts the minimality of

the coordinate $(a,\, b)$. So we conclude that $|\mathcal{V}_{t_{(i,\,j)}}| \neq 0$) Thereby we have $\frac{|\mathcal{V}_{t_{\uparrow(i,\,j)}}|}{|\mathcal{V}_{t_{(i,\,j)}}|} = \frac{1}{n}$. But $cs_{(i,\,j)} = \bigcup_{o \text{ is an operation or a variable}} os_o \implies cs_{(i,\,j)}(P) = \bigcup_{o \text{ is an operation or a variable}} os_o(P)$. Since we assumed that all of the operations and variables are distinct, $\exists$ at most one individual in P fitting the operation schema os_o and it now follows that $|cs_{(i,\,j)}(P)| =$ the number of operation schemata of the form os_o such that $os_o(P) \neq \emptyset$ which is precisely the number n. We finally obtain $\frac{|\mathcal{V}_{t_{\uparrow(i,\,j)}}|}{|\mathcal{V}_{t_{(i,\,j)}}|} = \frac{1}{|cs_{(i,\,j)}|}$ which is precisely the conclusion of the lemma. □

Remark 51 Given an individual $\mathbf{u}$ and a population P consisting of m individuals, observe that the number of individuals fitting the hyperschema $t_{(1,\,1)}$ is the same in every population from $[P]_{\mathcal{A}}$, i. e. $\forall Q \in [P]_{\mathcal{A}}$ we have $|t_{(1,\,1)}(Q)| = |t_{(1,\,1)}(P)| = 1$. It follows immediately now that $\frac{1}{m} E_{\varrho_{[P]_{\mathcal{A}}}}(\mathcal{X}(\square,\, S_{t_{(1,\,1)}})) = \frac{1}{m}$.

We now combine corollary 42, remark 51 and lemma 50 to obtain the following special case of Geiringer theorem for nonlinear GP under homologous crossover in case when all of the operations appearing in the individuals of the initial population P are distinct:

$$\lim_{t\to\infty} \Phi(\mathbf{u},\, P,\, t) = \frac{1}{m} \cdot \prod_{(i,\,j) \text{ is not the maximal coordinate of } \mathbf{u}} \frac{1}{|cs_{(i,\,j)}(P)|} =$$

$$= \prod_{v \text{ is a node of } \mathbf{u}} \frac{1}{|cs(v)(P)|}$$

(recall that when v is the root node of $\mathbf{u}$, $cs(v)$ determines the entire search space, and so $\frac{1}{|cs(v)(P)|} = \frac{1}{m}$) To obtain the general case, suppose we are given an initial population P. Fix a node v of $\mathbf{u}$ and consider the set of operations $\mathcal{O}(v) = \{o \,|\, |os_o(P)| \geq 1\}$ where os_o is obtained from $cs(v)$ as in definition 29. For every node v of $\mathbf{u}$ and for every operation (or variable) $o \in \mathcal{O}(v)$ fix an enumeration $\mathbf{x}^o_1,\, \mathbf{x}^o_2, \ldots, \mathbf{x}^o_{|os_o(P)|}$ of the individuals in P fitting the operation schema $os_o(P)$. Relabel the operation o occurring in the node v of $\mathbf{x}^o_i$ by the formally different operation $(o,\, i)$ (i. e. by the ordered pair $(o,\, i)$ whose first element is the operation o itself and the second element is the index telling us in which individual of P the operation o labels the node v). After all of the relabelling is complete we obtain a new population P' which is special with respect to the individual $\mathbf{u}$ in the sense of definition 49. Formally speaking, we expand our signature $\Sigma = (\Sigma_1,\, \Sigma_2, \ldots, \Sigma_N)$ as in definition 11 by adding the operations (variables) $(o,\, i)$ into Σ_j where j is the arity of the operation o. This gives us a new signature $\Sigma^* = (\Sigma^*_1,\, \Sigma^*_2, \ldots, \Sigma^*_N)$ where

$$\Sigma^*_j = \{o \,|\, o \in \Sigma_j \text{ and } o \notin \bigcup_{v \text{ is a node of } \mathbf{u}} \mathcal{O}(v)\} \cup$$

$$\cup \{(o,\, i) \,|\, o \in \mathcal{O}(v) \text{ for some } v \text{ and } 1 \leq i \leq |os_o(P)|\}.$$

Denote by Ω^* the search space induced by the signature Σ^*. The natural projection maps $p_j : \Sigma^*_j \to \Sigma_j$ sending $0 \to o$ when $o \notin \bigcup_{v \text{ is a node of } \mathbf{u}} \mathcal{O}(v)$ and $(o,\, i) \to o$ when $o \in \mathcal{O}(v)$ for some node v of $\mathbf{u}$, induce the natural "deletion of the extra labels" projection of the search spaces $\varphi : \Omega^* \to \Omega$ where the individual $\varphi(\mathbf{w}) \in \Omega$ is obtained from the individual $\mathbf{w} \in \Omega^*$ by

replacing the label of every node w of $\mathbf{w}$ with $p_j(w)$ where j is the arity of the node w. It is easily seen that the natural projection φ commutes with the crossover transformations in the sense that for any individuals $\mathbf{x},\, \mathbf{y} \in \Omega^*$ and for any crossover transformation $T \in \mathcal{F}$ (see definition 26) we have $\varphi(T(\mathbf{x},\, \mathbf{y})) = T(\varphi(\mathbf{x}),\, \varphi(\mathbf{y}))$. [9] Notice also that the population P can be obtained from the population P' by applying the natural projection φ to every individual of P'. Therefore, running the algorithm with the initial population P is the same thing as running the algorithm with the initial population P' and reading the output by applying the natural projection φ. The special case does apply to the population P', as mentioned above, and so we have

$$\lim_{t\to\infty} \Phi(\mathbf{u},\, P,\, t) = \sum_{\mathbf{w}\in\varphi^{-1}(\mathbf{u})} \lim_{t\to\infty} \Phi(\mathbf{w},\, P,\, t) = \sum_{\mathbf{w}\in\varphi^{-1}(\mathbf{u})} \prod_{v \text{ is a node of } \mathbf{w}} \frac{1}{|cs(v)(P)|}.$$

Notice that $\mathbf{w} \in \varphi^{-1}(\mathbf{u})$ precisely when the underlying shape schema of $\mathbf{w}$ is the same as that of $\mathbf{u}$, call this shape schema $t_{\mathbf{u}}$, and the label of every node v of $\mathbf{w}$ is $(o,\, i)$ where o is the label of the node v of $\mathbf{u}$. According to the way the population P' was introduced, there are precisely $|os(v)(P)|$ such labels (see also definition 29). We can then identify the preimage $\varphi^{-1}(\mathbf{u})$ with the set $\prod_{j=1}^{K}\{i \,|\, 1 \le i \le |os(v_j)|\}$ of ordered K-tuples of integers where K is the number of nodes in the parse tree of $\mathbf{u}$ and $v_1, v_2, \ldots, v_K$ is any fixed enumeration of the nodes of $\mathbf{u}$, in the following manner: The identification map $\imath : \prod_{j=1}^{K}\{i \,|\, 1 \le i \le |os(v_j)(P)|\} \to \varphi^{-1}(\mathbf{u})$ sends a given ordered K-tuple $(i_1,\, i_2, \ldots, i_K)$ into the tree $\mathbf{w} = \imath((i_1,\, i_2, \ldots, i_K))$ whose underlying shape schema is $t_{\mathbf{u}}$ and the label of a node v_j of $\mathbf{w}$ is $(o_j,\, i_j)$ where o_j is the label of the node v_j in the parse tree of $\mathbf{u}$. We finally obtain:

$$\lim_{t\to\infty} \Phi(\mathbf{u},\, P,\, t) = \sum_{\mathbf{w}\in\varphi^{-1}(\mathbf{u})} \prod_{v \text{ is a node of } \mathbf{w}} \frac{1}{|cs(v)(P)|} =$$

$$= \sum_{(i_1, i_2, \ldots i_K)\in\prod_{j=1}^{K}\{i \,|\, 1\le i\le |os(v_j)|\}} \prod_{v \text{ is a node of } \mathbf{u}} \frac{1}{|cs(v)(P)|} =$$

$$= \sum_{i_1=1}^{|os(v_1)(P)|} \sum_{i_2=1}^{|os(v_2)(P)|} \cdots \sum_{i_K=1}^{|os(v_K)(P)|} \prod_{v \text{ is a node of } \mathbf{u}} \frac{1}{|cs(v)(P)|}$$

$$= \prod_{j=1}^{K} \sum_{i_j=1}^{|os(v_j)(P)|} \frac{1}{|cs(v_j)(P)|} = \prod_{v \text{ is a node of } \mathbf{u}} \frac{|os(v)(P)|}{|cs(v)(P)|}$$

which is precisely the assertion of theorem 32.

[9] Of course, formally speaking, the two transformations T involved in the equation above are distinct, as they have different domains (Ω^* and Ω respectively), but they are determined by the same set of shape schemata and the same choice of nodes for crossover so we denote them by the same symbol.

6.6 Conclusions

In the current paper we applied the methods developed in [6] to obtain a schema-based version of Geiringer's theorem for non-linear GP with homologous crossover. The result enables us to calculate exactly the limiting distribution of the Markov chain associated with the evolution of a finite (fixed size) population under the action of repeated crossover. This is an extension of the results for fixed and variable length strings given in [6] for finite populations. The infinite population versions are given by the classical Geiringer theorem (in the case of fixed length strings) and the generalization given in [9] (for variable length strings). The corresponding infinite population result for non-linear GP is not yet established, although it seems to follow from the embedding theorems of [5] together with the corresponding result of [9] for linear GP. We are currently working on this issue. What is not known, is under which general conditions does the finite population result imply a corresponding limit in the infinite population case. This remains an open question.

Bibliography

[1] Booker, L. (1993). Recombination distributions for genetic algorithms. In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 29-44, Morgan Kaufmann.

[2] Coffey, S. (1999) An Applied Probabilist's Guide to Genetic Algorithms. *A Thesis Submitted to The University of Dublin for the degree of Master in Science*.

[3] Geiringer, H. (1944). On the probability of linkage in Mendelian heredity. *Annals of Mathematical Statistics*, 15:25-57.

[4] Mitavskiy B. (2004). Crossover Invariant Subsets of the Search Space for Evolutionary Algorithms. *Evolutionary Computation*, 12(1): 19-46.

[5] Mitavskiy B. (2003). Comparing Evolutionary Computation Techniques via Their Representation. In Eric Cantú-Paz *et al*. editor, *Proceedings of the Genetic and Evolutionary Computation (GECCO) Conference*, Vol. 1, pages 1196-1209, Springer-Verlag.

[6] Mitavskiy B. (recently submitted). A Generalization of Geiringer Theorem for a wide class of evolutionary search algorithms. *Evolutionary Computation*. http://www.math.lsa.umich.edu/∽bmitavsk/

[7] Poli, R. (2000). Hyperschema Theory for GP with One-Point Crossover, Building Blocks, and Some New Results in GA Theory. In R. Poli, W. Banzhaf, and *et al*., editors, *Genetic Programming, Proceedings of EuroGP'2000*, Springer-Verlag

[8] Poli, R. and Langdon, W. (1998). On the search Properties of Different Crossover Operators in Genetic Programming.

[9] Poli, R., Stephens, C., Wright, A., Rowe, J. (2002). A Schema-Theory-Based Extension of Geiringer's Theorem for Linear GP and variable-length GAs under Homologous Crossover. *FOGA 2002*

[10] Rabani, Y., Rabinovich, Y., Sinclair, A. (1995) A Computational View of Population Genetics. In *Annual ACM Symposium on the Theory of Computing* pages 83-92.

[11] Rosenthal, Jeffrey (1995). Convergence Rates for Markov Chains. *SIAM Review* 37(3): 387-405.

[12] Rowe, J., Vose, M., and Wright, A. (2002). Group properties of crossover and mutation. *Evolutionary Computation*, 10(2): 151-184.

[13] Rowe, J., Vose, M., and Wright, A. (to appear) Structural Search Spaces and Genetic Operators. *Evolutionary Computation*.

[14] Schmitt, L. (2001) "Theory of Genetic Algorithms." *Theoretical Computer Science*, 259: 1-61.

[15] Schmitt, L. (2004) "Theory of Genetic Algorithms II: Models for Genetic Operators over the String-Tensor representation of Populations and Convergence to Global Optima for Arbitrary Fitness Function under Scaling." *Theoretical Computer Science*, 310: 181-231.

[16] Spears, W. (2000) "The Equilibrium and Transient Behavior of Mutation and Recombination." *Foundations of Genetic Algorithms 6 (FOGA-6)*.

[17] Vose, M. (1999). The simple genetic algorithm: foundations and theory. *MIT Press, Cambridge, Massachusetts*.