Genetic Algorithms with Implicit Memory

Robert P. B. Morris

Submitted in partial fulfilment of
the requirements for the degree of
Master of Philosophy

De Montfort University

June 2011

Abstract

Genetic Algorithms with Implicit Memory

Robert P. B. Morris, June 2011

This thesis investigates the workings of genetic algorithms in dynamic optimisation problems where fitness landscapes materialise that are identical to, or resemble in some way, landscapes previously encountered. The objective is to appraise the performances of the various approaches offered by the GAs. Approaches specifically tailored for different kinds of dynamic environment lie outside the remit of the thesis.

The main topics that are explored are: genetic redundancy, modularity, neutral evolution, explicit memory, and implicit memory. It is in the matter of implicit memory that the thesis makes the majority of its novel contributions. It is demonstrated via experimental analysis that the pre-existing techniques are deficient, and a new algorithm – the pointer genetic algorithm (pGA) – is expounded and assessed in an attempt to offer an improvement. It is shown that though it outperforms its rivals, it cannot attain the performance levels of an explicit memory algorithm (that is, an algorithm using an external memory bank).

The main claims of the thesis are that with regard to memory, the pre-existing implicit-memory algorithms are deficient, the new pointer GA is superior, and that because all of the implicit approaches are inferior to explicit approaches, it is explicit approaches that should be used in real-world problem solving.

Contents

Abstract 1

Contents 2

List of Figures 5

1 Introduction **9**

1.1 Genetic Redundancy 11

1.2 Defining Genetic Redundancy 12

2 Literature Review **14**

2.1 Modularity 14

2.1.1 Modularity Generalised 16

2.1.2 Alternative Phenotypes 17

2.2 Explicit Memory 20

2.3 Implicit Memory 21

2.3.1 The Structured Genetic Algorithm 22

2.3.2 Dominance and Polyploidy 23

2.4 Other Works 30

2.4.1 The Dual Genetic Algorithm 30

2.4.2 Strategies for Games 31

2.4.3 Variable-Length Representations 33

2.5 Summary 34

3 Evaluation of Pre-Existing Memory Algorithms 35

3.1 Success Criteria for Memory Algorithms 38

3.2 Two Cycling Landscapes 39

3.3 Three and Four Cycling Landscapes 43

3.4 Introducing One-Off Landscapes 48

3.5 On the Structured Genetic Algorithm 51

3.6 On Dominance and Polyploidy 54

3.7 Summary 57

4 The Pointer Genetic Algorithm 58

4.1 Description of the Algorithm 59

4.2 Test Results 62

4.2.1 Two Cycling Landscapes 64

4.2.2 Three and Four Cycling Landscapes 68

4.2.3 Introducing One-Off Landscapes 73

4.3 Analysis of Results 77

4.3.1 Continuous Pointer-Mutation 77

4.3.2 Triggered Pointer-Hypermutation 83

4.4 Alternative Configurations 88

4.4.1 Sequential Pointer Mutation 88

4.4.2 Environmental Pointer Control 89

4.4.3 An Independent pGA 90

4.5 Summary 92

5 Conclusion **93**

5.1 Future Work 96

References 98

Appendices 103

List of Figures

2.1 Three representations where one genotype can map to two phenotypes 18

2.2 An example genotype of the structured genetic algorithm 22

2.3 The Ng-Wong dominance matrix for diploidy 24

2.4 A polyploid representation featuring a mask 26

2.5 A phenotype mapping of a variable-length-representation GA 33

3.1 The domination relationships used in the tests 37

3.2 Performance of the standard GA across two cycling landscapes 39

3.3 Performance of the explicit memory GA across two cycling landscapes 40

3.4 Performance of the structured GA across two cycling landscapes 40

3.5 Performance of the structured GA across two cycling landscapes, no hyper . . . 41

3.6 Performance of the diploid GA across two cycling landscapes, two matrices . . 41

3.7 Performance of the diploid GA across two cycling landscapes, six matrices . . . 42

3.8 Performance of the standard GA across three cycling landscapes 42

3.9 Performance of the standard GA across four cycling landscapes 43

3.10 Performance of the explicit memory GA across three cycling landscapes 43

3.11 Performance of the explicit memory GA across four cycling landscapes 44

3.12 Performance of the structured GA across three cycling landscapes 44

3.13 Performance of the structured GA across four cycling landscapes 45

3.14 Performance of the structured GA across four cycling landscapes, no hyper . . 45

3.15 Performance of the diploid GA across three cycling landscapes, six matrices . . 46

3.16 Performance of the diploid GA across three cycling landscapes, aligned matrices 46

3.17 Performance of the diploid GA across three cycling landscapes, 14 matrices . . 47

3.18 Performance of the diploid GA across four cycling landscapes, six matrices . . . 47

3.19 Performance of the standard GA across mixed landscapes, 25% random 48

3.20 Performance of the standard GA across mixed landscapes, 67% random 48

3.21 Performance of the explicit memory GA across mixed landscapes, 25% random 49

3.22 Performance of the explicit memory GA across mixed landscapes, 67% random 49

3.23 Performance of the structured GA across mixed landscapes, 25% random . . . 50

3.24 Performance of the structured GA across mixed landscapes, 67% random . . . 50

3.25 Performance of the diploid GA across mixed landscapes, 25% random 51

3.26 Performance of the diploid GA across mixed landscapes, 67% random 51

3.27 Fittest structured GA genotypes before and after a landscape change 53

3.28 Fittest diploid GA genotypes before and after a landscape change 54

4.1 The genotype of the pointer GA 59

4.2 Performance of a randomised pointer GA across two cycling landscapes 63

4.3 Performance of a 2-Ch pointer GA across two cycling landscapes, PMR 10% . . 64

4.4 Performance of an 8-Ch pointer GA across two cycling landscapes, PMR 10% . 65

4.5 Performance of a 2-Ch pointer GA across two cycling landscapes, PMR 50% . . 65

4.6 Performance of an 8-Ch pointer GA across two cycling landscapes, PMR 50% . 66

4.7 Performance of a 2-Ch pointer GA across two cycling landscapes, hyper PM . . 66

4.8 Performance of an 8-Ch pointer GA across two cycling landscapes, hyper PM . 67

4.9 Performance of a 3-Ch pointer GA across three cycling landscapes, PMR 10% . 67

4.10 Performance of a 9-Ch pointer GA across three cycling landscapes, PMR 10% . 68

4.11 Performance of a 3-Ch pointer GA across three cycling landscapes, PMR 50% . 68

4.12 Performance of a 9-Ch pointer GA across three cycling landscapes, PMR 50% . 69

4.13 Performance of a 3-Ch pointer GA across three cycling landscapes, hyper PM . 69

4.14 Performance of a 9-Ch pointer GA across three cycling landscapes, hyper PM . 70

4.15 Performance of a 6-Ch pointer GA across four cycling landscapes, PMR 10% . 70

4.16 Performance of a 20-Ch pointer GA across four cycling landscapes, PMR 10% . 71

4.17 Performance of a 6-Ch pointer GA across four cycling landscapes, PMR 50% . 71

4.18 Performance of a 20-Ch pointer GA across four cycling landscapes, PMR 50% . 72

4.19 Performance of a 6-Ch pointer GA across four cycling landscapes, hyper PM . 72

4.20 Performance of a 20-Ch pointer GA across four cycling landscapes, hyper PM . 73

4.21 Performance of a pointer GA across mixed landscapes, 25% random, PMR 10% 74

4.22 Performance of a pointer GA across mixed landscapes, 25% random, PMR 50% 74

4.23 Performance of a pointer GA across mixed landscapes, 25% random, hyper PM 75

4.24 Performance of a pointer GA across mixed landscapes, 67% random, PMR 10% 75

4.25 Performance of a pointer GA across mixed landscapes, 67% random, PMR 50% 76

4.26 Performance of a pointer GA across mixed landscapes, 67% random, hyper PM 77

4.27 Dominant pGA genotypes before and after landscape change, memory acquired 78

4.28 Two dominant pGA genotypes 800 generations apart, memory retained 79

4.29 Dominant pGA genotypes before and after landscape change, memory recalled 80

4.30 Dominant pGA genotypes before and after landscape change, no acquisition . . 80

4.31 Two dominant pGA genotypes 560 generations apart, memory lost 81

4.32 Dominant pGA genotypes before and after landscape change, no recall 81

4.33 Dominant pGA genotypes at the end of epochal and non-epochal runs 82

4.34 Two dominant pGA genotypes 1900 generations apart, memory retained 84

4.35 A pair of one-dimensional fitness landscapes 87

4.36 Sample performance of the independent pGA 91

Chapter 1

Introduction

The majority of the problems to which genetic algorithms (and optimisation algorithms in general) are applied do not change over time, so the only challenge is to find one or more appropriate solutions in a stationary search space. However, there are many problems that vary over time, giving algorithms an additional challenge. This kind of optimisation, together with optimisation in uncertain environments, has been a topic of interest in recent years in the evolutionary computing community (Yang, Ong & Jin 2007).

Although the modes of landscape change can in principle be endless, researchers (e.g. Trojanowski & Michalewicz (1999), pp. 4–6) tend to categorise them (firstly) as gradual or periodic. In gradually changing problems, the topographical features of the search space (which from the genetic algorithm point of view, can be the genotype space or the phenotype space) shift and stretch over time. For example, peaks and troughs may move along the axes, or they may rise or fall in height. In periodically changing problems, on the other hand, the fitness landscape changes instantaneously into something significantly different. For example, a peak may move a long way away, or a new peak may appear.

A further useful distinction can be made within the realm of periodically/epochally changing environments. These can be viewed as offering a succession of landscapes, and the distinction is between those environments where the landscapes are always new, and those where some or all of the past landscapes reappear, cyclically or otherwise. And with the latter should also be considered those where landscapes reappear that *resemble* past landscapes

somehow. It is clear that in cases of recurring landscapes, it is advantageous for a genetic algorithm to exploit stored information about past landscapes in order to ease and/or hasten its (re)adaptation to them should they return.

The aim of the present thesis is to identify and appraise genetic algorithms that accomplish that. The core research question is this: how and how well do the algorithms under consideration improve their adaptation to familiar fitness landscapes? The research objective is to supply satisfactory descriptions of every algorithm or algorithm-type considered, which involves either relaying results from the literature, or testing and analysing algorithms and then presenting the results. In every case, the success criterion is the credibility of the evidence-based analysis.

The main claims of the thesis lie in the realm of implicit memory, and can be summarised as follows.

- Demonstration of the ineffectiveness of previous implicit memory algorithms (the sGA and the polyploid GAs)
- Introduction of a new implicit memory algorithm (the pGA)
- Demonstration of the relative effectiveness of the pGA, including the concept of passive convergence.
- Constructive criticism of the concept of implicit memory in general

The remainder of the thesis is organised as follows. Firstly, the concept of genetic redundancy is clarified, because it is through that – and through neutral evolution – that most of the algorithms considered in the thesis fundamentally operate. After that comes the literature review chapter, which mainly covers modularity and implicit memory.

The claims made on behalf of the pre-existing implicit memory algorithms were felt to be unconvincing, so they have been investigated experimentally, the results being presented in Chapter 3. Analyses, as well as comparison to an explicit memory algorithm, make the contribution to knowledge that those algorithms are indeed defective with respect to memory behaviour.

In Chapter 4, a new implicit memory algorithm dubbed the *pointer genetic algorithm* (pGA) is defined, described, and compared to its rivals. In a further contribution to knowledge, it is shown to be superior to the other implicit memory algorithms, although inferior to the explicit memory algorithm. The shielding of memories from the genetic operators and the novel concept of *passive convergence* are identified as the central reasons for the pGA's success.

The thesis concludes in Chapter 5. The various genetic algorithms that belong to the research topic are summarised, most significantly those in relation to memory. The key conclusion is that for the problem environments considered, explicit memory algorithms ought to be used in preference to implicit memory algorithms, on account of the self-imposed deficiencies of the latter.

1.1 Genetic Redundancy

As stated earlier, the basis of improved adaptability of GA populations to recurring landscapes is the exploitation of stored information. An important distinction that is made here with respect to genotypes, is between external, or *explicit* storage, and internal, or *implicit* storage (Branke 2001). External storage is the simpler of the two, and is exemplified by explicit memory algorithms, such as that of Ramsey & Grefenstette (1993) where good genotypes were saved in an external memory bank and re-inserted into the population when they were needed. Another, less direct way to use stored information is in *prediction*, for example in chapter 7 of Simões (2010) where linear regression and Markov chains were used to predict when and how landscapes would change.

In GAs where the storage is wholly or partly implicit, there is invariably genetic/genotypic redundancy, commonly defined as a situation where the genotype space is bigger than the phenotype space. This is visible in the genotype to phenotype mappings that appear in the subsequent chapters. Before looking at the meaning of the term *genetic redundancy* in more detail, it will be briefly considered how close an approximation to implicit memory can be attained *without* genetic redundancy.

No redundancy whatsoever means that there is a one-to-one mapping between genotypes and phenotypes. Because there is no storage space for memories, the only way a population

can quickly adapt to a returning landscape is if the two optimal genotypes are extremely similar, in spite of dissimilarity between their associated phenotypes. For example, in a GA where the genotypes are 7-bit bitstrings and the phenotypes are the decimal forms, if the first landscape contains a fitness spike at phenotype = 0 and the second landscape contains one at phenotype = 64, then the optimal genotypes are 0000000 and 1000000 respectively. A single mutation of the first bit switches individuals between these two very different phenotypes, so in effect, the population can evolve to quickly adapt to the past landscapes when they return.

Other scenarios can be envisaged, but what they all have in common is that it is nothing more than 'convenient' search-space topographies that make high performance levels possible. Because such search spaces are so unrealistic, the notion of redundancy-free implicit memory can be discarded for practical purposes.

1.2 Defining Genetic Redundancy

Definitions of genetic/genotypic redundancy in the literature vary. For example, from Rothlauf & Goldberg (2003) there is this: "Representations are redundant if the number of genotypes exceeds the number of phenotypes" (p1) but later on, this: "in general a representation is redundant if [on] average one accessible phenotype is represented by more than one genotype" (p5). And from Nowak, Boerlijst, Cooke & Maynard Smith (1997) there is this: "Genetic redundancy means that two or more genes are performing the same function and that inactivation of one of these genes has little or no effect on the biological phenotype" (p1). Because of the central role redundancy plays in many algorithms discussed in this thesis, it is felt worthwhile to provide a particular definition of it, namely:

> A representation contains genotypic redundancy iff there is at least one phenotype which is produced by at least two genotypes.

Note: the word *phenotype* here means 'that to which a genotype maps.' If the representation under consideration features an intermediate stage in the mapping – for example, the mediation layer of polyploid GAs (Section 2.3.2), or the substring concatenation of the structured GA (Section 2.3.1), or the transliteration phase of the dual GA (Section 2.4.1) –

then the product of *that* stage corresponds to the "phenotype" of the present section.

In itself this definition of redundancy is unremarkable. The interest is in the fact that it ignores the relative size of the genotype space. It is true that in most redundant representations, the genotype space is a collection of subsets each mapping to one phenotype – making the genotype space larger than the phenotype space – *but this need not always be the case.* A trivial way to create a representation with genetic redundancy but with a genotype space that is *smaller* than the phenotype space is to take a typical redundant representation and embed it in another one where the additional genotypes each map to many phenotypes. The following example illustrates the idea.

> The genotype is three bits. The phenotype is produced in the following way. If the decimal form of the bitstring is 0–3, the phenotype is that divided by 2 and rounded down. If it is 4–7, the phenotype is that number plus ten randomly chosen decimal places.

The genotype space in that example is of size 8, whereas the phenotype space is of size 40 billion, and yet there is genetic redundancy in there. This proves that the ratio of the two spaces should not be used as the sole indicator of genotypic or phenotypic redundancy.

To summarise, the goal of the preceding sections was to specify the meaning of the term *genetic redundancy* as it applies to the algorithms considered in the thesis. It was necessary because – as has been shown – other definitions could potentially imply that genetic redundancy is absent when in fact it is present.

Chapter 2

Literature Review

This literature review aims to survey the different means by which genetic algorithms have been enhanced to enable them to adapt quickly to new fitness landscapes that are the same as, or resemble, previous landscapes. It is not concerned with enhancements that are tailored to landscapes that change gradually (e.g. Angeline (1997)) nor those that are tailored to environmental dynamism in general (e.g. Grefenstette (1992)).

Section 2.1 concerns modular phenotypes, and as an underlying mechanism, neutrality in the genotype space. In these works the algorithms evolve individuals that are primed for new landscapes *similar* to previous ones, rather than for immediate re-convergence onto returning optima.

Section 2.2 concerns explicit memory, and Section 2.3 concerns implicit memory, the largest topic of the thesis. The algorithms that have made claims about implicit memory capacity – the *structured GA* and the *polyploid GAs* – are described thoroughly, with an emphasis on their memory handling.

Section 2.4 concerns a handful of other, miscellaneous algorithms that one way or another offer solutions to the dynamic optimisation problems of interest to this thesis.

2.1 Modularity

Kashtan & Alon (2005) investigated the origin of *modularity* in evolutionary systems. They

observed in their introduction that phenotypes produced by computational evolution tend to be non-modular, whereas their biological and engineered counterparts often display modularity. The explanation for this is that computationally evolved solutions (i.e. phenotypes) are highly specialized to their problem, and that modular solutions are less optimal. GAs are not driven to introduce modularity, and even if it somehow arose, it would be broken down in favour of more robust, non-modular alternatives.

The key idea of Kashtan & Alon (2005) is to evolve solutions in an environment that oscillates between two fitness landscapes whose optimal solutions are structurally similar. They evolved logic circuits and neural networks, and in the case of the circuits, the two target functions were:

$$\text{G1} = (\text{X XOR Y}) \text{ AND } (\text{Z XOR W})$$
$$\text{G2} = (\text{X XOR Y}) \text{ OR } (\text{Z XOR W})$$

They refer to this as *modularly varying goal* (MVG) evolution, the opposite being *fixed-goal* (FG) evolution. In the logic-circuit experiments, the genotypes encoded arrangements of NAND gates and the specific connections between them, giving rise to 4-input-1-output circuits. They first evolved circuits for G1 alone, and obtained accurate but non-modular solutions. They then evolved circuits in an environment where every 20 generations, the goal flipped. They obtained solutions that were modular, differing only in 2 connections and resembling what a human engineer would design, and they observed that after an environmental change, it took "about five generations" (p4) to get to the alternative configuration.

Another experiment they performed was a repeat of the above but with randomly generated – as opposed to structurally similar – target functions. They found that the solutions were repeatedly non-modular, and as they put it, the individuals seemed to "forget" the previous goals and start again from scratch.

Something to remark in respect of the successful evolution in Kashtan & Alon (2005) is the nature of the transitions between the (two) optimal genotypes. Even though they were only 2 mutations apart, and could reach one another in around 5 generations, they were banking on at least one of them in the population experiencing the right mutation sequence. The deleterious effects of other mutations will have made that sequence more likely to occur,

but it should not be inferred from this that in similar scenarios, close genotypes (in terms of Hamming distance) can always be expected to find one another. Generally speaking, whether or not it will happen is dependent on the fitness landscape(s) and the GA's parameters.

2.1.1 Modularity Generalised

Parter, Kashtan & Alon (2008) continued the work of Kashtan & Alon (2005) in an investigation into *facilitated variation* in evolution (Kirschner & Gerhart 2005). FV is the idea that natural organisms have to some extent evolved their evolvability such that the genetic changes they undergo are beneficial.

Having re-stated the findings of that earlier paper, Parter et al. (2008) then discussed the phenotypic neighbourhood of the evolved solutions, and *neutral networks* in the genotype space. Neutral networks are networks of genotypes that share a single phenotype, and are so called because an individual can traverse them without experiencing positive or negative fitness changes. Neutrality is widely considered to aid evolution (Kimura 1983) because of how it increases the number of phenotypes that a given individual can reach without having to cross fitness valleys. Shackleton, Shipman & Ebner (2000), in one of numerous papers on the subject, showed that extensive neutral networks are needed to significantly improve optimisation, and that too little or the wrong kind of neutrality does not help.

The following excerpt from Parter et al. (2008) is highly significant.

> FG populations are known to evolve toward the center of the neutral network [...]. Thus the FG organisms are more robust to genetic mutations and their phenotypic neighborhood exhibits a lower degree of variation than the MVG organisms. [...] In contrast, MVG organisms seem to be located at the edge of the neutral network that is closest to the neutral networks of the previously seen goals. This implies that temporally varying environments push populations towards special regions of the neutral network. (p4)

This notion provides a sound explanation for the behaviour of not just those specific algorithms, but of all the algorithms contained in this thesis. Having said that, however, it

is a very low-level explanation, and a more thorough understanding of how the algorithms operate requires higher-level explanation as well.

The innovation from Parter et al. (2008) through which they generalise from the previous work is their conception of "modularity language" (p5). For the case of the logic circuits, they defined this as those functions of the form

$$G = (X\ a\ Y)\ b\ (Z\ c\ W)$$

where *a, b* and *c* are any logic gates (not their terminology). They found that ready-evolved MVG populations were much better prepared for new landscapes in the same language as the previous ones, than were similarly prepared FG populations; when the two populations were mixed and forced to compete, the MVGs took over the population in 75% of the runs.

The paper then went on to consider the ramifications for FV. As far as the present thesis is concerned, the message from these projects is that a genetic algorithm can improve its adaptation to new fitness landscapes that are the same as, or in the same "language" as, previous ones, if it can be induced to produce modular phenotypes. To be so induced, the landscapes must have common or proximate regions of high fitness.

This may be thought of as pseudo implicit memory.

2.1.2 Alternative Phenotypes

Worgan & Mills (2008) modelled part of the *Alternative Phenotypes Hypothesis* (APH), which they summarised in their introduction as follows:

> [The] APH, put forward by West Eberhard (2003), puts across the view that phenotypic plasticity in the form of condition-sensitive phenotype expression (i.e. alternative phenotypes) is key in a sequence of evolutionary processes that lead to organic novelty, and in turn to speciation and higher macroevolutionary events.

Of the seven stages that comprise the framework of the hypothesis, Worgan & Mills (2008) focused on the third, namely that: "Novel alternative phenotypes evolve and stably persist in the population." To do this, they extended and modified Kashtan & Alon (2005). They

took their representation and made two changes: they enabled the chromosomes to vary in length, and they added integers in [0..2] to the genes, where a 0 meant the gene (gate) was always expressed, but a 1 or 2 meant that it was only expressed in environment one or two respectively.

They found that when the environment oscillated between two target logic functions, systems with landscape-specific genes performed well, whilst those with no such alternative phenotypes performed badly. What happened in the former case was that the individuals evolved to high fitness in both landscapes, sometimes using the same genes in the two landscapes, but also taking advantage of the capacity to express selectively. In the latter case – the control – the individuals were compelled to use the same genes in both landscapes, so were incapable of solving both problems simultaneously.

Additionally, Worgan & Mills (2008) found that when the environments oscillated at a lower frequency (that is, when the populations spent more time in each landscape before it changed) the fitness levels were lower and more erratic.

It is pertinent at this point in the thesis to consider how multiple phenotypes are deterministically drawn from a single genotype at different times. There are essentially three degrees of overlap between the different sections of the genotype that must be used; Figure 2.1 shows them for the case of two phenotypes.

A: <111111222222>

B: <111bbb222iii>

C: <iiibbbbbbiii>

Figure 2.1: Three representations where one genotype can supply one of two phenotypes. '1' genes contribute to phenotype 1, '2' genes to phenotype 2, 'b' genes to both, and 'i' genes (actually introns) to neither. Note that the symbols are not alleles.

In representation A, the two sections do not overlap at all; in B, they half overlap; and in C, they overlap completely. The advantage of A is that the sections are fully independent of one another. This means that they can both be optimised no matter what genes are required. For example, if they were bitstrings and the optimal configurations were all zeroes and all ones respectively, then the genotype could evolve to <000000111111>. This would clearly be

impossible in example C.

The disadvantage of A is that it contains non-expressing genes that have to spend periods of time without selective pressure on them. Taking the half-ones-half-zeroes example, if an optimal population was left in the first landscape for a long time, the genes for the second landscape (i.e. the ones) would drift and corrupt.

The situation in C (and also B) is complicated by a dependence on the particulars of the two landscapes. At one extreme, if the two landscapes are identical, there is no problem, but at the other extreme – such as with all zeroes and all ones – there is a serious problem, in that co-optimisation is impossible. In general, for co-optimisation or at least an approximation to it to be possible, there must be points of reasonably high fitness in the first landscape's genotype space whose equivalent points in the second landscapes's genotype space are also of reasonably high fitness. This is a repetition of what has been said earlier with regard to modularity and neutrality, and it was known by Worgan & Mills (2008) who wrote:

> [...] selection favours genotypes that express phenotypic traits that can contribute to high fitness in both environments, such that the population will move to a portion of the fitness landscape that overlaps. This can only be the case when the environments share a significant portion of their structure – and the target functions chosen by Kashtan and Alon have exactly this property. (p722)

The disadvantage of representation C is thus the dependence for success on 'convenient' fitness landscapes. An advantage is that no gene will have the selective pressure taken off it, but in that there lies another disadvantage: the selective pressure might be applied in the wrong direction. If the population spends too much time in one environment, it may find itself unfit when it returns to the other. For a population to be successful in a situation like this, it must oscillate fairly rapidly between the two landscapes. (This is demonstrated in Chapter 3 with a different algorithm.)

Incidentally, overlap can occur on intra-generational as well as inter-generational scales; Wu & Lindsay (1996) studied a GA where the building blocks for the phenotypes could "float" along the genotype, often overlapping. The implications are similar in both situations.

2.2 Explicit Memory

Branke (1999) reviewed the utilisation of memory in GAs, and he described *explicit memory* as an approach in which "specific information is stored and reintroduced into the population at later generations" (p2). In section 3 of that paper ("General Thoughts about Memory" pp. 3–4) he put aside implicit memory, and discussed the various aspects of explicit-memory implementation. The following three design decisions were highlighted:

1. When and which individuals should be stored in the memory?
2. How many individuals should be stored in the memory and which should be replaced to make space for new individuals?
3. Which individuals should be retrieved when from the memory and reinserted into the population?

What this section of that paper captures is that, in effect, most explicit memory GAs are the same, and that the difference between them is the exact configuration and choice of parameters and operators. The algorithms presented by Louis & Xu (1996), Eggermont, Lenaerts, Poyhonen & Termier (2001), Bendtsen & Krink (2002), Acan & Tekol (2003) and Simões (2010) testify to this observation. And given the sure and simple nature of their operation, it is not surprising that in all the works, the experimental results are favourable.

Not all explicit memory algorithms are the same, however, because there are some that do not use a memory bank *per se,* and there are others that use the stored information in a different way than straightforward reinsertion. Of the first kind there are *multiple population* algorithms, for example by Branke, Kauler, Schmidt & Schmeck (2000) and Klinkmeijer, de Jong & Wiering (2006), where one of several populations is being run at any given time, the others holding onto fit individuals from previous fitness landscapes, awaiting their return. Like the regular explicit-memory algorithms, these also produced good results for similarly simple reasons.

Of the second kind of alternative, an example is the memory-based random-immigration scheme (Yang 2005). Here, random immigrants are inserted into the population in the usual

way, but instead of being randomly generated, their alleles are biased towards those of individuals stored in a memory bank.

The present thesis accepts the positive claims made about explicit-memory genetic algorithms, but nevertheless, one of them (Eggermont et al. 2001) was coded and tested, and is included in the comparisons between algorithms presented in Chapter 3.

2.3 Implicit Memory

The subsection in Branke (1999) that covers implicit memory opens with the following passage:

> An evolutionary algorithm that uses representations containing more information than necessary to define the phenotype (i.e. redundant representations) basically has some memory where good (partial) solutions may be stored and reused later as necessary.
>
> We call this kind of memory *implicit* because it is left to the EA to find a way to use it appropriately. (p1)

This description is somewhat vague because the classification itself is equally vague, beyond the idea that the memories are stored in the genotypes, and therefore directly handled by selection. The biggest problem is defining what constitutes a memory, or a partial memory, and another problem area is how – and how *well* – these memories are acquired, retained, and recalled. With regard to memory recollection, for example, it has to be decided whether the process must be totally implicit, or whether external components (e.g. dominance matrices) are permitted to be involved.

A very broad definition could embrace every algorithm covered in the present literature review bar the explicit-memory ones, and a very narrow definition could exclude all but the *pointer genetic algorithm* introduced in Chapter 4. The fact of the matter is that the term has been in use for over a decade in the literature, with certain types of algorithm repeatedly being described as implicit-memory, and the rest never being described as such. The present thesis finds it adequate to follow the prevailing usage of the term, and because the interest is in performances and behaviours, and not taxonomy, no attempt will be made at a formal

binary classification (implicit vs. non-implicit) nor a ternary one (implicit vs. pseudo-implicit vs. non-implicit).

In accordance with the foregoing statement, the algorithms covered in this subsection are only those which are commonly described as possessing implicit-memory capacity. Experimental analysis of representative example algorithms is supplied later, in Chapter 3.

2.3.1 The Structured Genetic Algorithm

Dasgupta & McGregor (1992*b*), Dasgupta & McGregor (1992*a*), and Dasgupta (1994) presented the structured genetic algorithm (sGA). It was specifically designed for nonstationary optimisation problems, and is claimed to perform well in both gradually and periodically changing environments. The representation is as follows.

The genotype is divided into substrings of equal length, each of which has a meta-bit assigned to it. The meta-bits form a higher-level string, and this string may itself be divided into substrings, to each of which would be assigned additional meta-bits ('meta-meta-bits' perhaps). There is thus a tree structure of meta-bits, where the lengths of the substrings and the total number of layers are chosen by the programmer. Every meta-bit, irrespective of which layer it is in, affects the phenotype mapping in the following way: if it is 1, the substring below it is *active* (or "on"); if it is 0, the substring below it is *passive* (or "off"). When all the meta-bits in the tree have made their contribution, some of the bottom-level substrings will be active and the rest will be passive. The concatenation of the active substrings is what is used to build the phenotype.

Figure 2.2 gives an example of an sGA chromosome, where there are 3 layers of 2-bit substrings. In that example, the bitstring <0110> is the concatenation that finally maps to the phenotype.

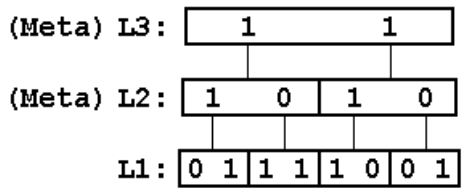

Figure 2.2: An example of an sGA genotype

It is obvious that all other things being equal, the length of the concatenated string can vary from zero bits to the full length of the bottom layer. Because most GA phenotypes are built using a fixed number of genes, this represents a serious problem. The authors knew this, and so added a stage in their own implementations where any individuals with under- or over-long concatenations (referred to as "chimeras") that appeared, were immediately replaced.

In their experiments, the authors found that the sGA performed well in a dynamic Knapsack problem and in a moving-parabola problem.

The question of the merits of the sGA is investigated in Chapter 3, so instead of repeating that material here, there will now just be an uncritical and idealised verbal summary of how its memory works.

While converged, an sGA population behaves like a standard GA population, but with more mutation, thanks to meta-bit mutations substituting one substring in the bottom layer for another. This mutating does not displace the population from whatever optimum it may be on.

If the fitness landscape changes, mutants that otherwise would have died, now often survive, so hitherto passive substrings may now persist in the population. If the landscape is a returning one, and the population had been converged on an optimum in it when it was incumbent, then optimal genes may have survived in passive substrings. These substrings, when they reappear, may guide the population back to that returned optimum. There, the population re-converges, with optimal genes from the recently departed landscape having been switched from active to passive, awaiting that landscape's return.

Finally, there have been a few applications of the sGA; Nasaroui, Dasgupta & Pavuluri (2002) applied a "soft" version of the sGA to a web-mining task, and Bellas, Becerra & Duro (2009) wrote a GA to evolve topologies and weights for artificial neural networks using an sGA-like representation.

2.3.2 Dominance and Polyploidy

Dozens of algorithms of the kind covered in the present subsection have been published, and if they were all to be discussed, this part of the literature review would unjustifiably take up

a disproportionately large percentage of the thesis as a whole. It was therefore decided to discuss just an exemplifying selection of the papers, with the aim of capturing their essence.

In the majority of GA representations, the genotypes are *haploid*, meaning that there is only one gene at each locus. In *polyploid* representations, on the other hand, the number of genes per locus is greater than one. In the common case of there being two, the genotype is *diploid*.

Dominance refers to the way in which the genes in a polyploid genotype affect one another, per locus, in the production of what is sometimes called the *mediation type*, which then maps to the phenotype. The units that compose a mediation type are referred to as *mediate units* in this thesis; the term 'mene' could potentially be coined, by analogy with *gene* and *phene,* but this could be confused with *mean*. Another possibility would be 'wene'.

The diploid dominance matrix from Ng & Wong (1995) is reproduced in Figure 2.3 to illustrate the preceding ideas. Note that the (binary) allele alphabet has been extended – a common feature of polyploid GAs.

	o	0	i	1
o	0	0	?	1
0	0	0	0	?
i	?	0	1	1
1	1	?	1	1

Figure 2.3: The dominance matrix from Ng & Wong (1995)

The matrix in Figure 2.3 shows how every possible pair of alleles is processed during the mapping to the mediation type. The alleles '0' and '1' are designated *dominant*, and the alleles 'o' and 'i' are designated *recessive 0* and *recessive 1* respectively. When a dominant allele is paired with a recessive allele, if they are different, then the dominant allele goes into the mediation type. When two alleles of equal status but different values (e.g. 'o' plus 'i') are paired, in this matrix a random mediate unit is generated.

Note that with dominance matrices in general, the ratio of mediate units over the space of all possible inputs can be biased. The common sense advice to programmers is to make the ratio as unbiased as possible unless they know that an imbalance would be beneficial.

To utilise the traits of dominance and polyploidy GAs in dynamic optimisation problems, changes are made to the dominance matrix and/or the genes. In the case of Ng & Wong (1995) it was this: if the fitness of any individual dropped by a given amount between adjacent generations, its dominant genes were made recessive and its recessive genes were made dominant.

Before reviewing the papers, an idealised and uncritical description of how memory works in these algorithms is offered. Experimental results and analysis are presented in Chapter 3.

The matrix in Figure 2.3 is used, as is the fitness-drop rule quoted earlier, but instead of genes being changed, the matrix itself is changed to invert the domination relationship.

The environment oscillates between two fitness landscapes, where the optimal mediation-types are 0011 and 0101 respectively. In the first landscape, the population is taken over by this genotype: <0o, 0i, 1o, 1i>. The dominance matrix maps this genotype to <0011>, which is precisely the optimal mediation-type.

The second landscape now appears, causing fitnesses to drop. The matrix changes in consequence, such that recessive alleles now dominate dominant alleles. In the new regime, the genotype that was optimal previously now maps to <0101>. This is precisely the optimal mediation-type of the new landscape.

Those individuals survive for the entire run, and the population stays optimally fit. Recessive genes provide the storage space for the memories, and the dominance matrix is the recall mechanism.

It was rather fortunate that the dominance matrices were such that a mediation type optimal to both landscapes was producible, and it was also fortunate that dominated genes were not corrupted by mutation before becoming expressive again.

The review of selected dominance-and-polyploidy works now follows.

Goldberg & Smith (1987) were the first to study dominance mechanisms in a nonstationary environment. They compared a traditional haploid representation, a simple diploid representation in which 1s always dominated 0s, and the tri-allelic diploid representation from an earlier paper (Hollstien 1971). The dominance matrix was able to evolve in the tri-allelic case. Their environment comprised two landscapes whose global optima were 01111101111111111

and 01101101111111011.

The haploid case was predicable: the algorithm had to recommence searching after every landscape change. In the simple diploid case, the individuals were able to store *parts* of the other landscape's optimal genes, so the performance lever was higher. The reason full optima could not be stored was that only 0s could be stored, and to go from the second optimum to the first, an individual would have needed 1s in memory.

In the tri-allelic case, the population, having visited both optima, was able to re-discover them after almost every landscape change.

Collingwood, Corne & Ross (1996) were the first to study polyploidy in the context of evolutionary computing. Their genotype contained p chromosomes and a "mask" shown in Figure 2.4.

Mask:	0	0	0	1	1	1	2	2	2
Chrom[0]:	**a**	**a**	**a**	a	a	a	a	a	a
Chrom[1]:	b	b	b	**b**	**b**	**b**	b	b	b
Chrom[2]:	c	c	c	c	c	c	**c**	**c**	**c**
Phenotype:	a	a	a	b	b	b	c	c	c

Figure 2.4: Example of the basic polyploid representation from Collingwood et al. (1996), for ploidy = 3, showing the genotype (mask + chromosomes) and the phenotype.

The mask controlled the dominance relationship between the alleles at every locus, and every individual had its own mask. Mask genes could mutate like any other gene, so if an individual had different alleles in locus i, a mutation of mask-gene i was comparable to a mutation in the currently-expressing chromosome-gene i. Crossover operated both between and within chromosomes.

Collingwood et al. (1996) tested populations of individuals with p from 1–10 on two problems: One-Max, with binary chromosomes, and *Indecisive(K),* with alleles in $[0..K]$. The population sizes were varied to hold the total number of genes constant, for fairness of comparison. In the One-Max tests, when mutation was turned off, the polyploid populations all outperformed the haploid, but when mutation was turned on, it was the other way round. The superiority of polyploidy in the absence of regular mutation stemmed from its implicit di-

versity; whereas the haploid populations would prematurely converge and thereafter be stuck, the polyploid ones would, during the similarly converged state, be able to introduce previously non-expressing genes. The *inferiority* of polyploidy in the default mutation-on setting came from the same source – too many alleles were expressing, which in a nearly-all-1s population, were often unwanted 0s.

The Indecisive(K) problem (created in Collingwood et al. (1996)) is this. There are $K+1$ alleles, and every individual will have some number of each of them in its genotype. The largest such number becomes the individual's fitness – and if the most common allele is *not* K, 1 is then subtracted. This means that for a genotype length of L, there are K local optima of fitness $L-1$, and one global optimum of fitness L. The challenge is to get to the global optimum, which a standard GA population can be expected to do $\frac{1}{K+1}$ of the time.

The haploid populations performed as expected, and the polyploid ones performed better. Crudely speaking, the polyploid populations did better because they 'mixed things up' more in the early-to-middle parts of the runs. In the haploid case, there would be an early struggle between the groups with different majority-alleles, after which the population would evolve towards one of the optima. But in the polyploid cases, the expressing of hitherto non-expressing alleles would prolong this struggle, giving the K alleles more opportunity to benefit from their slight superiority.

The authors concluded that polyploidy is sometimes helpful, sometimes not, depending on the problem. And it is evident from their work that stationarity does not always imply a bad polyploid performance.

The concept of *additive diploidy* was first applied in evolutionary computing by Ryan (1996) and Ryan (1997). Here the dominance matrix is dispensed with, and the genes are instead aggregated for the mediate units. To take the example from Ryan (1996), there could be four alleles: 2, 3, 7, 9. There would be two of these at each locus, which would be summed: a sum of less than 10, and a 0 is taken; otherwise, a 1 is taken. This example scheme contains 16 permutations, exactly half of which yield 0s, so it is unbiased.

Ryan (1996) compared an additive representation to the one from Hollstien (1971) in a two-state environment with maximally different optima. He found the additive to be better

for low-frequency landscape changes, but the tri-allelic for high frequencies. He then tested an additive version with *polygenic inheritance,* which performed the best. It should be noted that mutation was not used in these tests.

There was no dynamism in any of the mappings in Ryan (1996), so the reasonably good adaptivity after landscape changes was made possible by recessive alleles, which crossover reactivated. To make this happen in algorithms like these, crossover must be able to cut within loci, because otherwise the population would have to depend on ordinary mutations like a standard GA. Something else that can improve their adaptability is "forced mutations" (Ryan 1997), which were in fact additional neutral mutations.

Lewis, Hart & Ritchie (1998) compared three algorithms – haploid, additive, ordinary dominance – with and without the capacity to change the dominance mapping (for which in the haploid case hypermutation was used instead). They found that none of them performed particularly well without such capacities, but that with them they worked well.

When the better algorithms were put in environments where the landscape oscillated between the same two states, the one with ordinary dominance performed best; whereas the other two regularly re-adapted from scratch, it managed to effectively memorise most of both landscape's optimal genes, so its fitness levels stayed high throughout. But when the new landscapes were always unseen, the other two performed similarly and the ordinary dominance algorithm struggled. What it possessed in special-case memory capacity it evidently lacked in post-change adaptability.

Perhaps the most interesting finding by Lewis et al. (1998) was that an ordinary haploid GA with post-change hypermutation – cf. Cobb (1990) – can compete with its more complicated rivals, meaning that (to quote from their concluding sentence) "the case for implementing a diploid mechanism as opposed to a simple mutation operator may be weakened."

Other non-encouraging results were obtained by Yilmaz & Wu (2003), who compared haploidy and diploidy in integer representations of the *Travelling Salesman Problem* (TSP). Across many parameters, the haploid version performed better, and the authors concluded that in that particular TSP, diploidy had little to offer.

Yang (2006) defined a generalised dominance scheme for diploid genotypes in which the

number of alleles can be chosen arbitrarily, and non-determinism in the mapping can be switched on or off. It is essentially an additive scheme, where the sum of the two alleles in a given locus creates the mediate unit. In the case of two alleles summing to $C + 1$, where C is the cardinality (i.e. the allele range), a random bit can be chosen (non-deterministic) or the magnitude of the smaller allele decides it (deterministic).

Yang (2006) performed many tests and drew two interesting conclusions. Firstly, that the larger the cardinality, the better the performance. Secondly, that non-determinism worsens the performance, chiefly because optimal mediate units are sometimes only expressed as such half the time. It should be stated that the range of performances was generally not very wide, so the differences in settings are not that important.

Saito & Hamagami (2010) designed a diploid representation where the alleles were real numbers and where as well as having a probabilistic dominance mapping between the genotype and the mediation type (Kominami & Hamagami 2007), the phenotype was produced from the mediation type in a neural-network-like manner. Every mediate unit had a connection to every phene, and every connection had a weight. Like a neural network firing, each phene was calculated by summing the weighted contributions of the mediate units.

The algorithm exploited this arrangement by dynamically updating the weights in every generation, drawing the values towards those that were associated with the fittest individuals. Consequently, the population was able to track moving optima – which was incidentally also the case in Kominami & Hamagami (2007), where they only had the probabilistic dominance mapping.

Lastly, there have been several applications of these techniques, including the following. Most noteworthy of all, Hillis (1990), in his celebrated co-evolution paper, used a diploid representation to evolve sorting strategies. Fonteix, Bicking, Perrin & Marc (1995), as well as comparing haploid and diploid algorithms on several test problems, applied them on a real hydrodynamical problem. Massebeuf, Fonteix & Kiss (1999) used a diploid GA to discover the "optimal zones" in a multi-objective optimisation problem concerning food granulation. Wu, Ho & Wang (2000) used a diploid GA to schedule the activities of a particular hydro-thermal system, and Karakoc, Soke & Kavak (2007) used one to evolve dynamic code allocation for a particular kind of network.

It should be noted that memory behaviour *per se* was not called for in these applications.

2.4 Other Works

This last section of the literature review describes a small miscellaneous group of algorithms that each possess a capacity for improved adaptation to familiar fitness landscapes.

2.4.1 The Dual Genetic Algorithm

The DGA (Collard, Escazut & Gaspar 1996) was created as an improvement to the standard GA in dynamic optimisation problems. Its novel features are its genotype to phenotype mapping and its *mirroring operator*, which work as follows.

Instead of using the whole bitstring-genotype to build the phenotype, there is a "transliteration phase" whereby if the first bit is a 0, the rest of the bitstring is taken, but if it is a 1, the *complement* of the rest of the bitstring is taken. So for example, the genotype <$\dot{0}$ 001>, where the meta bit is dotted, would be transliterated into <001>, whereas the genotype <$\dot{1}$ 001> would become <110>.

In the mirroring operation, some small fraction (typically 1%) of the individuals in the population have all their bits flipped during the creation of the next generation. These individuals are genotypically inverted, but because of the mapping, they are phenotypically unchanged. The important consequence of this is that during periods of convergence, the population continuously maintains a small amount of diversity, thanks to the fact that the inverted minority are of equal fitness to the converged majority.

In every other regard, the DGA is coded and run like a standard GA, with the meta bit manipulated like any other gene, and thus in the hands of selection.

The key benefit of the DGA lies in what its authors refer to as its *dualism*: its maintenance of two complementary genotypes during convergence. When a converged standard-GA population suddenly finds itself in a new fitness landscape, the lack of diversity and (usually) low mutation rate hamper its exploration. But when there is dualism, crossover is able to generate diversity by recombining complementary genotypes in the first generation, and subsequently

their offspring. This diversity 'kick starts' the new genetic searching that is necessary for optimisation.

Collard et al. (1996) showed that the DGA outperforms the standard GA in dynamic optimisation. Having said this, results from Grefenstette (1992), where the *random immigration* technique is used, are similarly positive. What happens there is that a few completely new genotypes (random immigrants) are inserted into every generation as a diversity-maintaining measure. This, or the insertion of complements ('complementary immigrants'), is simpler to implement than dualism.

Regarding rapid adaptation to returning landscapes, the DGA can do this in just one particular kind of environment: when there are two alternating landscapes that contain optima whose genotypes are complementary, or at least near-complementary. Here, the individuals can reach a state where they are effectively implicitly-storing the other optimal genotype, and a mutation of the meta bit after a landscape change causes that other genotype to be restored. Experimental confirmation of this phenomenon may be found in Collard et al. (1996), p8.

Two variants of the DGA have been published: the folding GA (Gaspar & Collard 1997), and the primal-dual GA (Yang 2003).

2.4.2 Strategies for Games

In one of the earliest and most influential instances of the GA production of game-theoretic strategies, Axelrod (1987) evolved them for the *iterated prisoner's dilemma* (IPD), in the setting of a tournament. His representation (which has been re-used by others, e.g. Darwen & Yao (1995), and in a simplified form in Ishibuchi & Namikawa (2005)) held a binary look-up chart of next moves – 'C' for co-operate, 'D' for defect – based on the last 3 turns. There are 4 possible outcomes to a turn, so the look-up chart comprised $4^3 = 64$ entries. If, at any point in the game, the last 3 moves had been for example CC-DC-CD – which in binary may be represented as 00-10-01 – then the individual would look to the 9th (001001 binary = 9 decimal) gene for the instruction of what to do next. For the first 3 moves of a game, the player has no history to refer to, so the genotype had added to it a 'fake history' for that purpose. Those additional 6 genes made the total genotype length 70 bits. (The strategies

that emerged from the evolutionary process were generally of the TIT-FOR-TAT form.)

A peculiar aspect of GAs like these where a genotype's fitness is obtained in conjunction with other genotype(s), rather than being obtained via some separate fitness function, is that for each individual, the rest of the population *is* the environment. This makes the environment necessarily dynamic, and any given individual's fitness depends on which other individuals currently exist.

For the case of the IPD, the phenotypes are the played games, and they are produced by a traversal of two genotypes, whereby the developmental function 'hops' from gene to gene as the game progresses. Different opponents compel a given player to use only a particular subset of its genes, and it is hoped that by playing enough – and sufficiently varied – opponents, players can evolve good responses for most of the realistic in-game scenarios. It is in this regard that this representation is effectively one that prepares its genotypes for returns to familiar fitness landscapes. If fit individuals from an early generation survive through other generations (with different opponents) until a time when opponents similar to those earlier on re-evolve, then in effect they have returned to an old fitness landscape, and it is possible that the genes relevant to those returning opponents will have survived for re-use.

But as Darwen & Yao (1995) showed, this cannot be counted on, and "glitches" (to use their term) are always possible. They wrote this on the problem:

> There may be genetic drift: that is, since there is no selection pressure one way or the other on this part of the genotype, random mutations will fill this part of the genotype with garbage, since its contents will make no difference to fitness. (p10)

In the matter of the measurement of genotypic convergence, Morris & Watson (2008) evolved strategies for a different game using a representation like that of Axelrod (1987), and offered the following piece of advice to GA programmers whose genotypes contain many genes that go for long periods of time without selective pressure on them.

The absence of selective pressure means that the alleles go into drift, and when the population converges one of two things can happen. Firstly, these genes may *hitchhike* with the expressing genes, so the convergence is both genotypic and phenotypic; but secondly, the population may converge phenotypically whilst simultaneously not genotypically. Because of this

second possibility, if convergence is to be used as a stopping criterion (or for any other reason) then it should ideally be somehow phenotypic and not genotypic. This avoids situations where algorithms run on indefinitely beyond the times they should have stopped, wasting time and also money if the fitness evaluations are expensive.

2.4.3 Variable-Length Representations

Yu, Wu, Lin & Schiavone (2003) designed a genetic algorithm with a variable-length representation (VLR) to solve a dynamic task-scheduling problem. Every pair of genes in the genotype coded for a (task, processor) pair, and thanks to the crossover operator, the genotype length was variable (albeit capped at an upper limit). Their example of a phenotype mapping is reproduced in Figure 2.5.

<(3,0)(1,3)(2,2)(1,2)(3,0)(4,3)(5,2)(0,0)(2,1)>

↓

For processor 0:	task 3, task 0
For processor 1:	task 2
For processor 2:	task 2, task 1, task 5
For processor 3:	task 1, task 4

Figure 2.5: The example of a genotype to phenotype mapping in Yu et al. (2003), pp. 3–4.

The phenotype is produced by reading along the genotype from left to right and assigning the tasks to the processers one by one. Any repeated genes are ignored. Mutation was applied everywhere, and one-point crossover was allowed to cut and splice between any pairs.

The authors ran the algorithm for 1500 generations at a time, and switched from one to the other of two fitness functions after every 100 generations. The difference between the fitness functions lay in the order that the tasks had to be performed in, the one being the opposite of the other. They compared their algorithm to a fixed-length GA and found theirs to perform better.

The most noteworthy finding of Yu et al. (2003) is what happened after landscape changes. Whereas in previous similar work (e.g. Burke, Jong, Grefenstette, Ramsey & Wu (1998)),

longer individuals had prospered in those circumstances, here it was the *shorter* ones that were favoured by selection. In the previous VLR cases, when a population entered a new fitness landscape, there was a need for new good genes, and the representations were such that longer genotypes – by virtue of having more genetic material – tended to have more such genes. In Yu et al. (2003), however, the representation was more regular, with solutions usually being specific to their landscape. The longer a genotype, the more specialised it was, so after a landscape change, the shorter genotypes found themselves less unfit and more adaptable. It was thus the shorter genotypes that spread through the population post-change, growing longer as they evolved towards the new optima.

The GA in Yu et al. (2003) seems essentially to be designed for improved adaptation to *any* new fitness landscape. It merits a mention in the present thesis, however, because – as the example genotypes they display on p10 show – the population can carry building blocks that are fit in both landscapes, on account of which it could be claimed that it is in an implicit-memory GA.

2.5 Summary

The genetic algorithms from the literature whose individuals have the specific ability to adapt quickly to familiar fitness landscapes, have been reviewed. Section 2.1 covered the way in which modularity can facilitate this, and Section 2.4 covered some miscellaneous representations that enable it to a limited degree. Sections 2.2 and 2.3 covered memory methods *per se,* which are the most prominent approaches to the optimisation problems of the thesis. It is worth remarking that for all but the explicit memory approaches, genetic redundancy is a key property of the representation, and at bottom, successful adaptation is achieved by the exploitation of neutrality in the genotype space.

The explicit memory approaches seem well founded, but the implicit memory approaches seem to be very limited in their applicability and efficacy. It is necessary in accordance with the research aim to ascertain the effectiveness of GAs with memory capabilities, so to that end, Chapter 3 presents an experimental investigation into the techniques described in Sections 2.2 and 2.3.

Chapter 3

Evaluation of Pre-Existing Memory Algorithms

This chapter presents test results and analyses of representative versions of the genetic algorithms that have been claimed to possess memory capacity. Results from standard GA runs are also included to provide a baseline for performances. It is found that the explicit memory algorithm performs very well, and that the implicit memory algorithms perform badly.

The algorithms assessed are the following four (each of whose C code is appended):

1. Standard GA (Holland 1975)
2. GA with a case-based memory (Eggermont et al. 2001)
3. Structured GA (Dasgupta & McGregor 1992*b*)
4. Dominance-and-diploidy GA

The landscape topography was taken from Morrison & de Jong (1999). The particular implementation used here consisted of a set of stationary landscapes, each having 5 floating-point dimensions, and each having 5 cones along those dimensions. The fitness values were calculated using a direct extension into 5 dimensions of this equation from Morrison & de Jong (1999) (p2049):

$$f(X,Y) = max_{i=1,N}[H_i - R_i \times \sqrt{(X - X_i)^2 + (Y - Y_i)^2}]$$

(X_i, Y_i) is the location of the centre of the ith cone, H_i is the height of the ith cone, and R_i is a factor that controls the width of the ith cone.

In the present case, every H was 10, the axes ranged between 0.0 and 25.0, and the R factor was (after tweaking) set to 4.0. To get the 5 co-ordinates, the chromosome was split into 5 equal parts and each one was converted into a decimal number and then re-scaled. These settings created landscapes that lay in the *Goldilocks region* for genetic search – that is, the optima were neither too easy nor too hard to find.

Multi-dimensional multi-cone landscapes were used because they are simple to implement, they do their job, they are popular with GA researchers, and because other prominent candidates were found to have deficiencies during preliminary tests. For example, needle-in-a-haystack problems, because of their extreme simplicity, caused certain phenomena to be missed. And the 01-knapsack problem (which is very popular) was too easy, in that individuals could rapidly become fit, so it was very difficult to distinguish between fitness rises due to memory recollection and fitness rises due to ordinary evolution.

The following parameters were common to all four algorithms. (As with the landscapes, they were not chosen on the basis of some deep underlying principle, but because *something* had to be chosen, and they produced satisfactory results.) The population size was 200 and the genotype length – or in the cases of the sGA and the diploid, the mediation-type length – was 30 or 60. The landscapes changed every 200 generations, and the runs lasted 3000 generations, or 15 epochs. The pairwise crossover probability was 60%, the per-gene mutation probability was 1/(genotype length $\times$ 2), and standard proportionate selection was used.

The following parameters were specific to one algorithm only. For (2), the explicit-memory GA, the memory bank could store 10 genotype patterns. For (3), the structured GA, there were 2 layers, the top layer containing 10 genes of which 5 were always active, and the bottom layer containing 60 or 120 genes of which 30 or 60 respectively were always active. Unless otherwise stated, the top layer was hypermutated when the landscape changed. And for (4), the diploid GA, it was the dominance matrix, not the genes, that was changed when the landscape changed, and the interpretations of conflicting alleles were as shown in Figure 3.1.

	1	2;	3	4	5	6;	7	8	9	10	11	12	13	14;	15	16
01	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
0i	0	1	0	1	1	0	0	1	0	1	0	1	0	1	1	0
o1	1	0	1	0	1	0	1	0	1	0	0	1	0	1	1	0
oi	1	0	0	1	0	1	1	0	0	1	0	1	1	0	1	0

Figure 3.1: The domination relationships between conflicting alleles in the tests

Matrices 1 and 2 (the first two columns on the left in Figure 3.1) are the salient relationships, 1 being the default case and 2 being the complement of the default case. Matrices 3–6 are like 1 and 2 in that the 0:1 ratio across the four pairings is unbiased at 2:2. Matrices 7–14 contain biased ratios of 3:1, and the last two matrices do not really offer a domination relationship at all, only one allele ever expressing.

The range of matrices used in the tests is stated along with the data, and was always 2, 6, or 14.

The last thing to discuss in relation to the algorithms is the matter of detecting landscape-change. This is an important component of many of the algorithms covered in the present thesis, and in spite of the fact that it is often taken for granted, it is a non-trivial task. Richter (2009) investigated it, and identified two individual-using methods: *population-based* and *sensor-based*. In population-based methods, fitness changes (typically reductions) on the part of members of the evolving population are used to make the judgement; in sensor-based methods, additional individuals that are unaffected by the genetic operators are deployed and used in the same way.

It is easy to imagine scenarios where these methods fail, for example if the landscape changes in places where there are no individuals. And conversely there are possible scenarios where a false positive could occur, for example if the fitnesses of every point in the search space all fell by the same amount. Richter (2009) was aware of the preceding problems, having written the following:

> [...] the changes in the dynamic fitness landscape have to exceed a certain severity so that they can be detected. Clearly, in a practical context we are mainly

> interested in detecting changes that are in principle detectable, i.e. the fitness landscapes are dynamically distinguishable after each change, meaning that the number of points [at which the fitness changes] must be sufficiently large. (p1614)

The only 100% sure way to always be able to detect any environmental change would be to have sensor individuals at every point in the search space. This would be practically impossible or at least infeasible in most cases, and even if it were not, the fact of its being possible would render a GA unnecessary because the problem space could then be searched exhaustively.

The finding of Richter (2009) was that population-based methods are superior when the changes are easier to detect, and that sensor-based methods are superior when the changes are harder to detect.

The intention for the present tests was to use a combination of population- and sensor-based methods, but it was found in preliminary runs that even that combination was sometimes unreliable. It was therefore decided that by default, the algorithms would 'cheat' by being told by the program whenever change occurred. This enabled the memory-handling to be seen as clearly as possible; the alternative would have been situations where memory-recollection failure could be misinterpreted as change-detection failure.

3.1 Success Criteria for Memory Algorithms

Before presenting the test results and assessing the implicit memory algorithms, the criteria for success will here be clarified. There are three things that an algorithm must do in order to qualify as an effective memory algorithm: (1) *it must acquire memories;* that is, while or shortly after being on a given landscape, it must obtain genetic information relating to that landscape. The location and format of the stored information may vary. (2) *it must retain memories;* that is, in the time between the departure of a given landscape and its return, the stored information particular to that landscape must not be lost or critically corrupted. It should not be assumed that these periods of time will be short. And (3) *it must restore memories;* that is, when a landscape returns for which there is stored information, that information must be restored to re-express in the population.

Success is judged here on the basis of experimental results, and analysis of the low-level behaviour of the algorithms. It should be borne in mind that a memory system that satisfies these criteria no better than, or worse than, a standard GA, does not qualify as an effective memory algorithm even if it possesses features for it.

3.2 Two Cycling Landscapes

In every double figure in this and the subsequent sections, the Y-axes show the population-wide mean fitnesses, the X-axes show time in steps of four generations, and the (effective) genotype lengths are 30 and 60 in the top and bottom plots respectively.

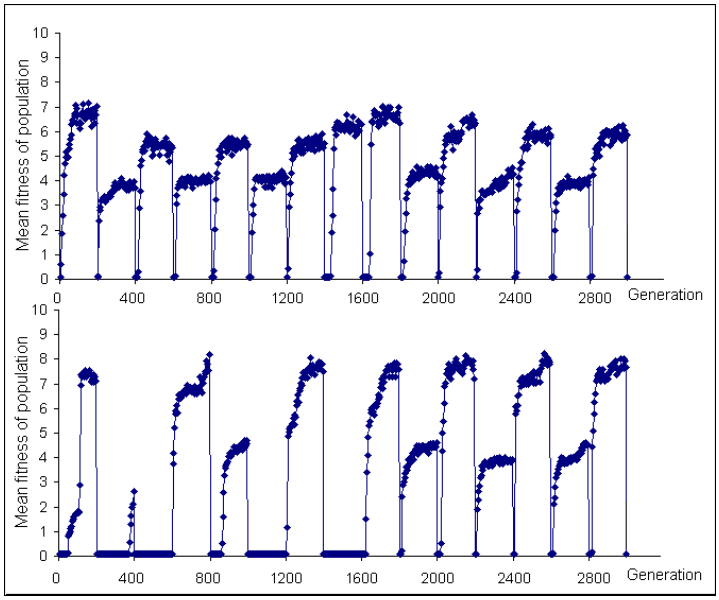

Figure 3.2: Performance of the standard GA across two cycling landscapes

The first environment to be considered is the simplest one within the remit of the thesis, where there are two landscapes that swap places every 200 generations. Figure 3.2 shows how the standard GA performed in it.

The standard GA, having no special mechanisms for coping with environmental change, adapted to every new landscape from scratch. The narrow gaps between the curves in Figure 3.2 communicate that fact.

Figure 3.3 shows how the explicit memory GA performed in this simplest of environments. It is clear from the immediacy of the restorations of high fitness (evidenced by the lack of

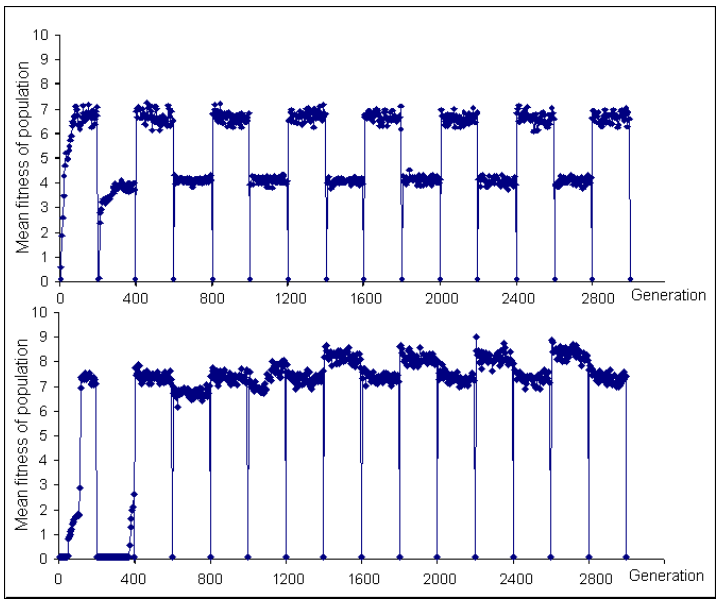

Figure 3.3: Performance of the explicit memory GA across two cycling landscapes

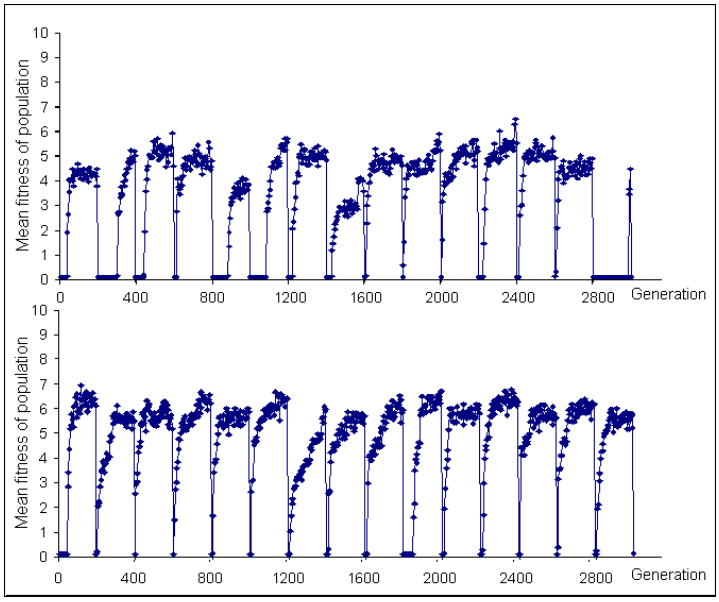

Figure 3.4: Performance of the structured GA across two cycling landscapes

gaps between curves) that the technique of reinserting genotypes from a memory bank was effective.

Figures 3.4 and 3.5 show how the structured GA performed, with and without top-level hypermutation respectively. It behaved like a standard GA in that it recommenced searching in every returning landscape, but it was much more consistent in its ability to re-attain high fitness. The other thing to note is that the version without hypermutation of its top-level

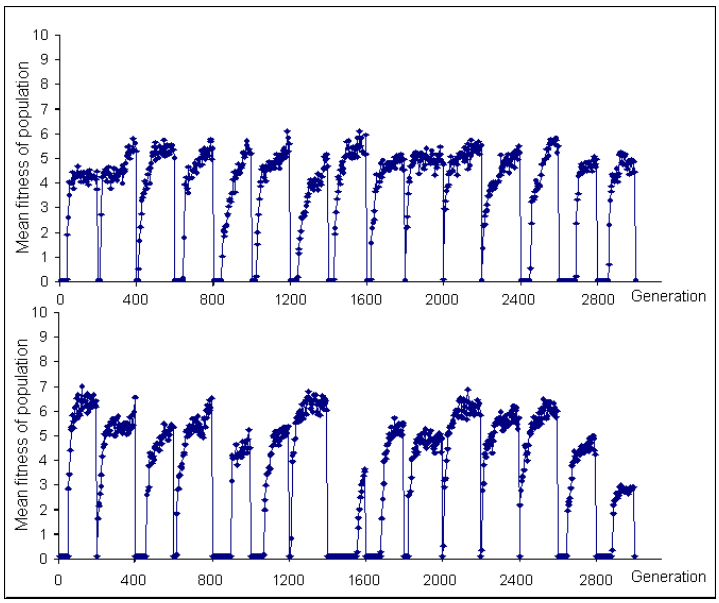

Figure 3.5: Performance of the structured GA across two cycling landscapes, without top-level hypermutation

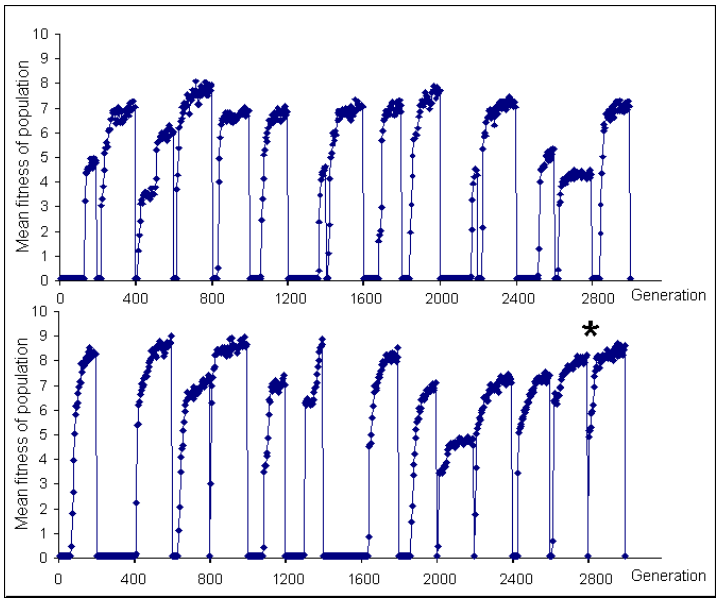

Figure 3.6: Performance of the diploid GA across two cycling landscapes, two matrices

genes performed only slightly worse than the default version.

Figures 3.6 and 3.7 show how the diploid GA performed, for two and six available dominance matrices respectively. A new matrix was chosen randomly whenever landscape change was detected. In both cases the algorithm performed worse than the standard GA, regularly recommencing search from scratch and sometimes failing to rediscover any cone.

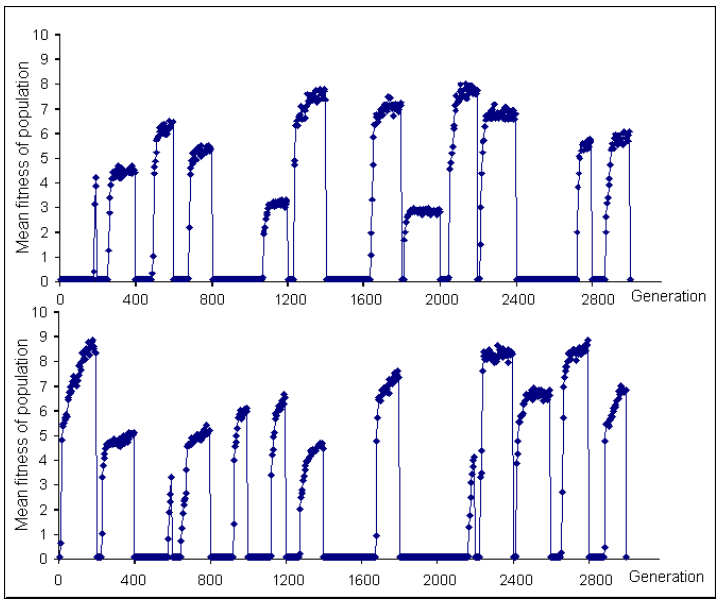

Figure 3.7: Performance of the diploid GA across two cycling landscapes, six matrices

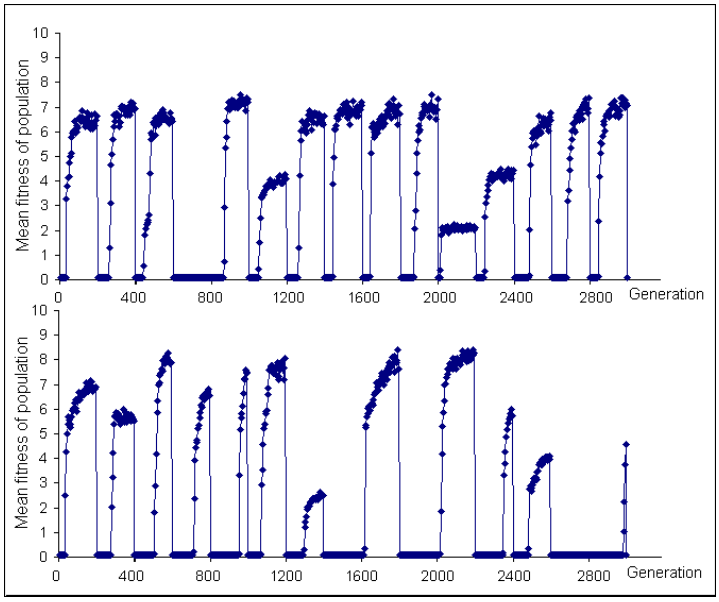

Figure 3.8: Performance of the standard GA across three cycling landscapes

Analysis of the key aspects of all the experiments in this chapter is provided after the results have been displayed, in Sections 3.5 and 3.6. However, it is worth remarking at this early stage that the implicit memory algorithms seem to have fallen at the first – and easiest – hurdle.

3.3 Three and Four Cycling Landscapes

Figures 3.8 and 3.9 show the performances of the standard GA with three and four cycling landscapes respectively. In both situations the algorithm performed a little worse than with only two landscapes, usually taking longer to find the cones.

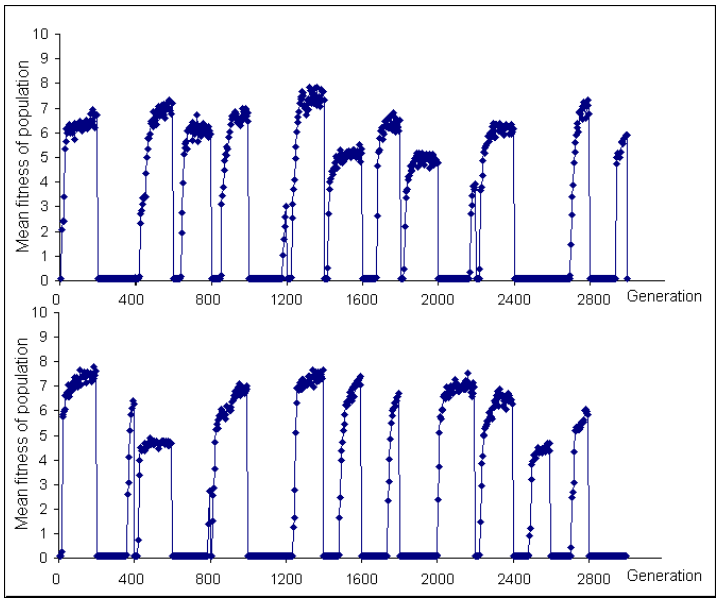

Figure 3.9: Performance of the standard GA across four cycling landscapes

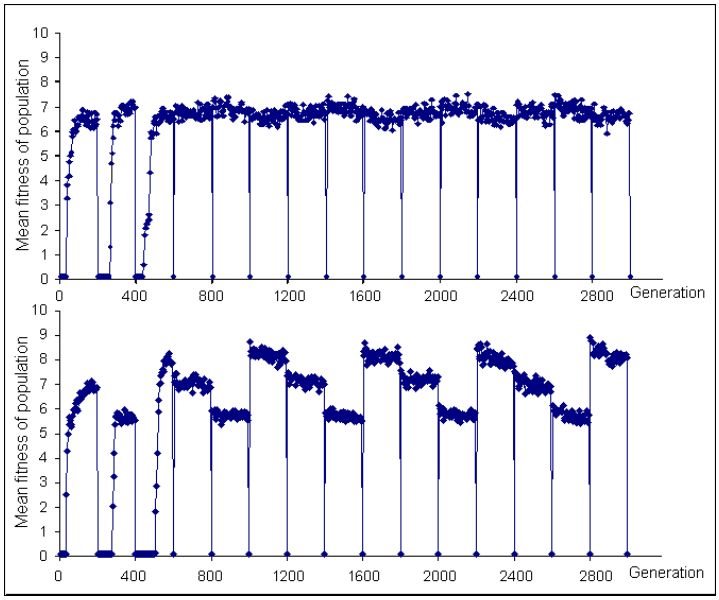

Figure 3.10: Performance of the explicit memory GA across three cycling landscapes

Figures 3.10 and 3.11 show the performances of the explicit memory GA with three and four cycling landscapes respectively. The situation there is a straightforward extension of the

situation with two landscapes. That is, once the population had found an optimum, it was stored in the memory bank and successfully reinstated whenever its landscape returned. The only thing worth noting is something in the bottom plot of Figure 3.11, starred.

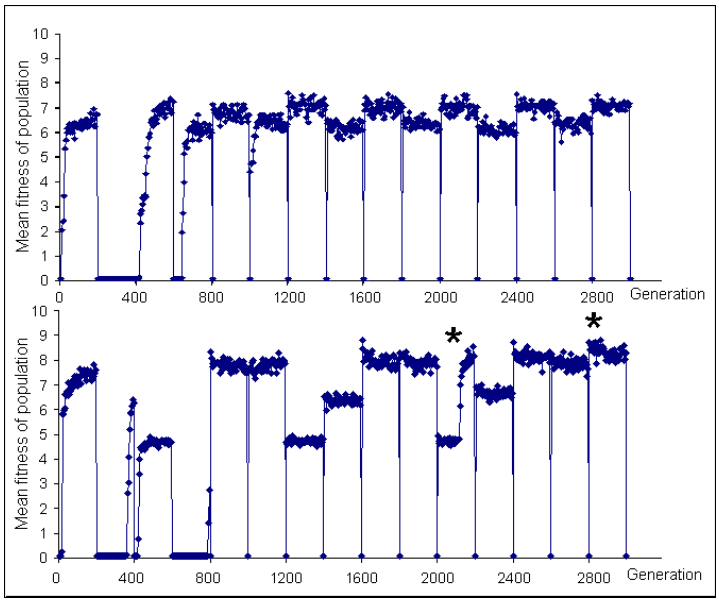

Figure 3.11: Performance of the explicit memory GA across four cycling landscapes

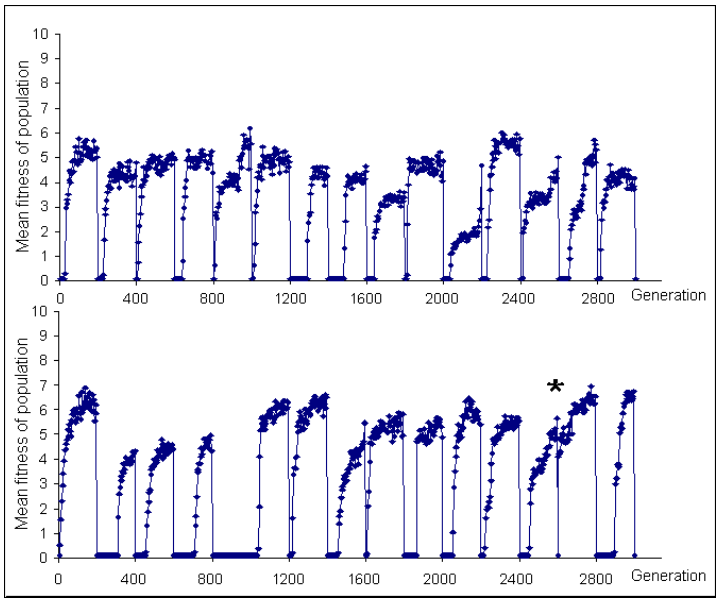

Figure 3.12: Performance of the structured GA across three cycling landscapes

There, while in landscape 3 of 4, an individual discovered further dimensions of the cone, and the population relocated. The last peak in that plot is the last manifestation of the 3rd landscape, in which the population returned to that new location. This shows that the memory bank was restoring not just landscape-specific information, but the best of that information.

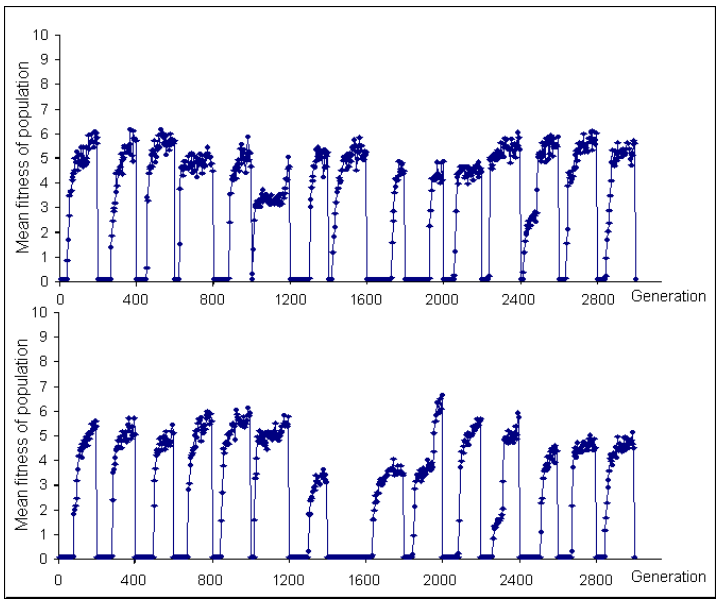

Figure 3.13: Performance of the structured GA across four cycling landscapes

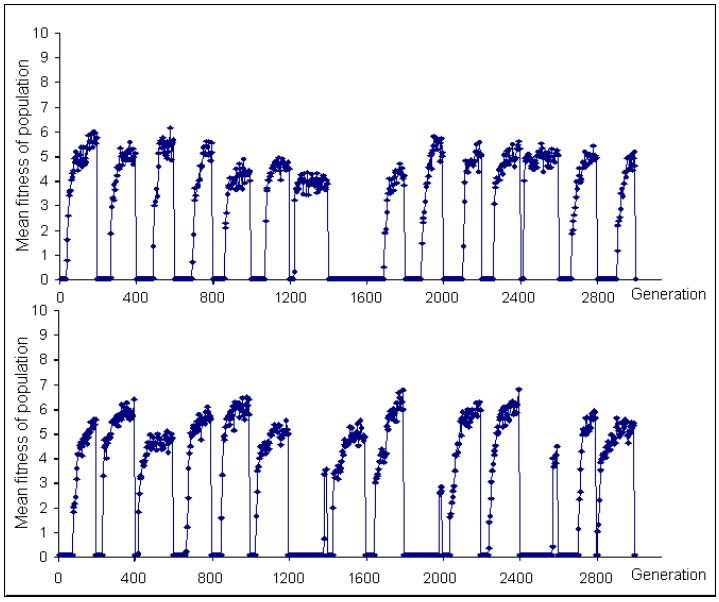

Figure 3.14: Performance of the structured GA across four cycling landscapes, without top-level hypermutation

Figures 3.12, 3.13, and 3.14 show the performances of the sGA with three and four cycling landscapes. It imitated the standard GA in that it performed slightly worse in these harder environments, still effectively treating every returning landscape as new. And there is again similarity between the performances with and without top-level hypermutation.

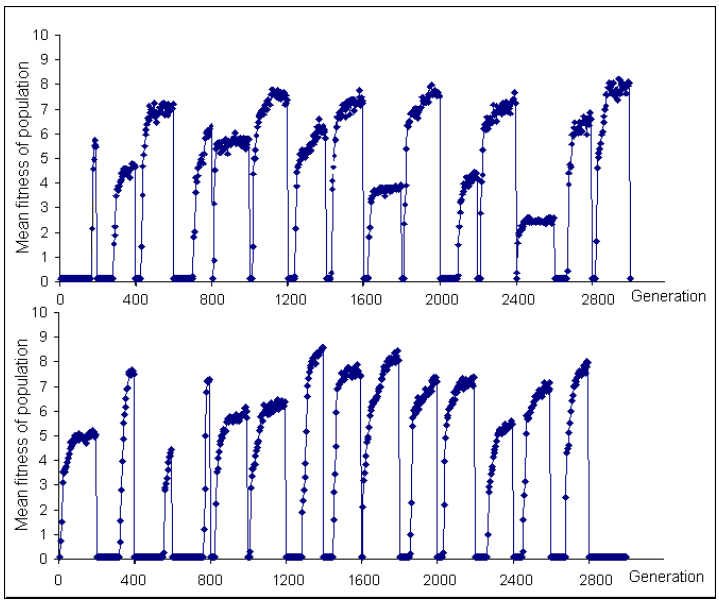

Figure 3.15: Performance of the diploid GA across three cycling landscapes, six matrices

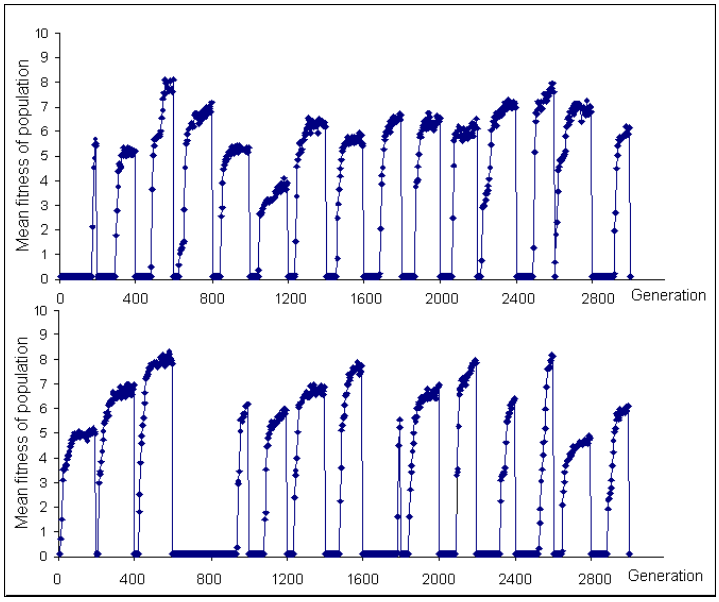

Figure 3.16: Performance of the diploid GA across three cycling landscapes, six matrices, the same matrix used with each landscape

Figures 3.15 and 3.16 show the performances of the diploid GA with six matrices for three cycling landscapes, for randomly chosen and landscape-aligned matrices respectively. Figure 3.17 shows the performances with fourteen matrices for three cycling landscapes, and Figure 3.18 shows the performances with six matrices for four cycling landscapes.

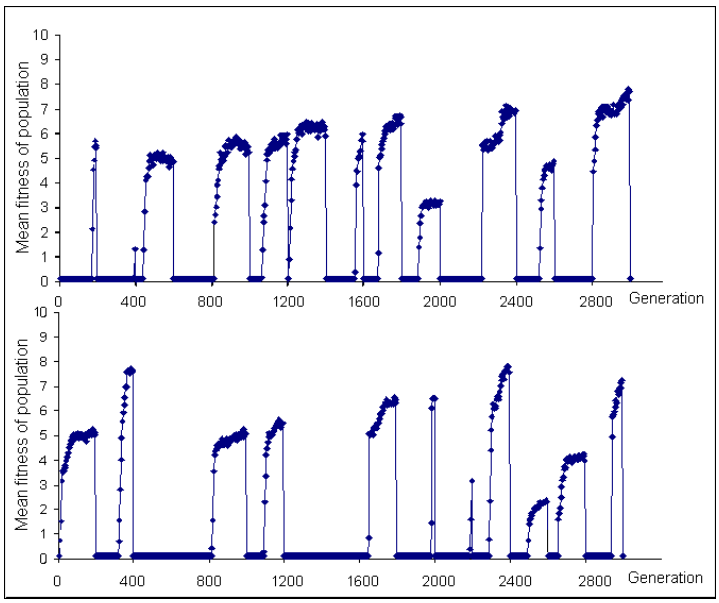

Figure 3.17: Performance of the diploid GA across three cycling landscapes, fourteen matrices

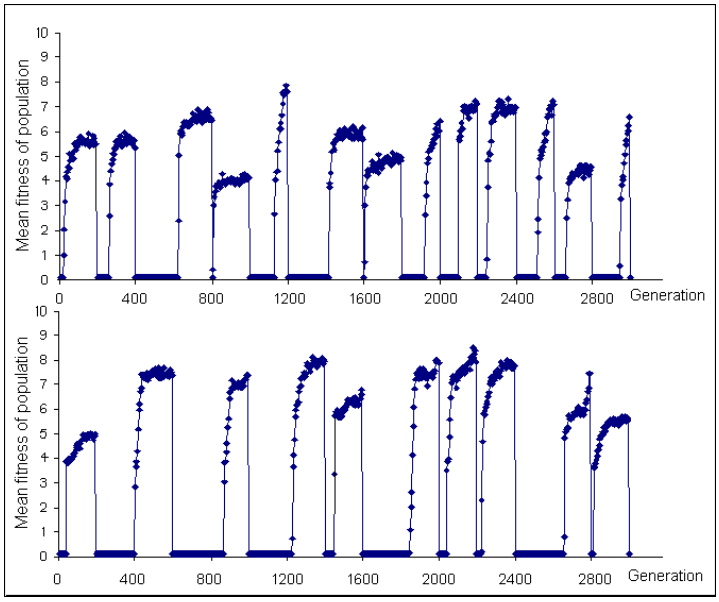

Figure 3.18: Performance of the diploid GA across four cycling landscapes, six matrices

Inspection of th ese figures, as well as those for two landscapes, reveals essentially the same performance every time. That is, the population takes several generations to find a cone in every landscape, and sometimes it does not find one at all.

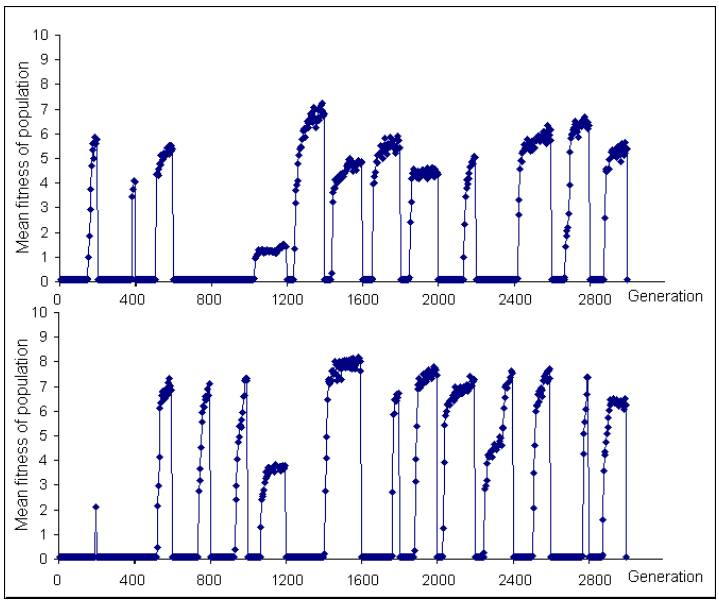

Figure 3.19: Performance of the standard GA across mixed landscapes, 25% random

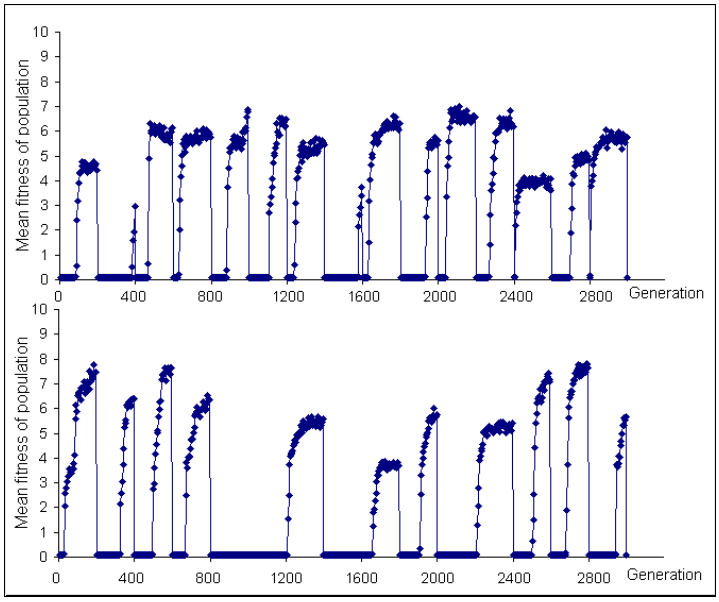

Figure 3.20: Performance of the standard GA across mixed landscapes, 67% random

3.4 Introducing One-Off Landscapes

The previous two sections contain results for environments comprising closed sets of cycling landscapes. In this section, results are presented for environments of two returning landscapes plus some proportion of random 'one-off' landscapes. That proportion was set to either 25% or 67%, so in the 25% case, 75% of the landscapes used in the runs were taken from a subset of two, and the other 25% were randomly generated on demand, never to reappear. The

annotations **1**, **2**, and **R** in the figures show which was the current landscape.

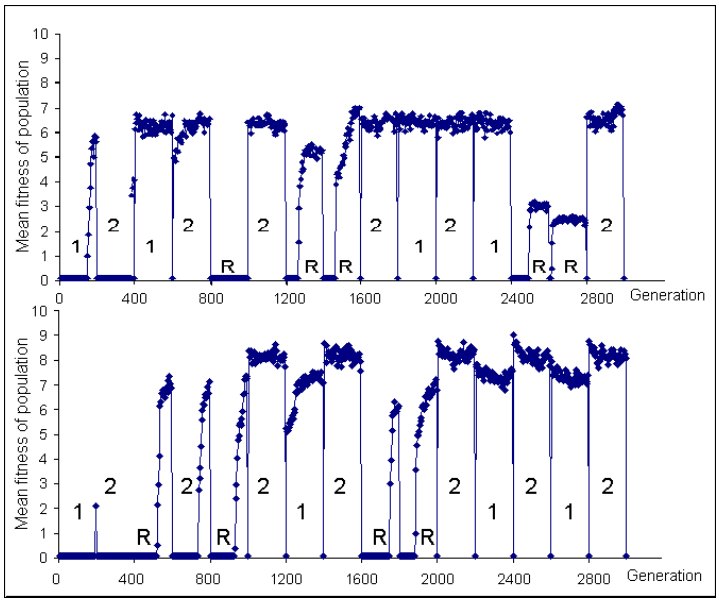

Figure 3.21: Performance of the explicit memory GA across mixed landscapes, 25% random

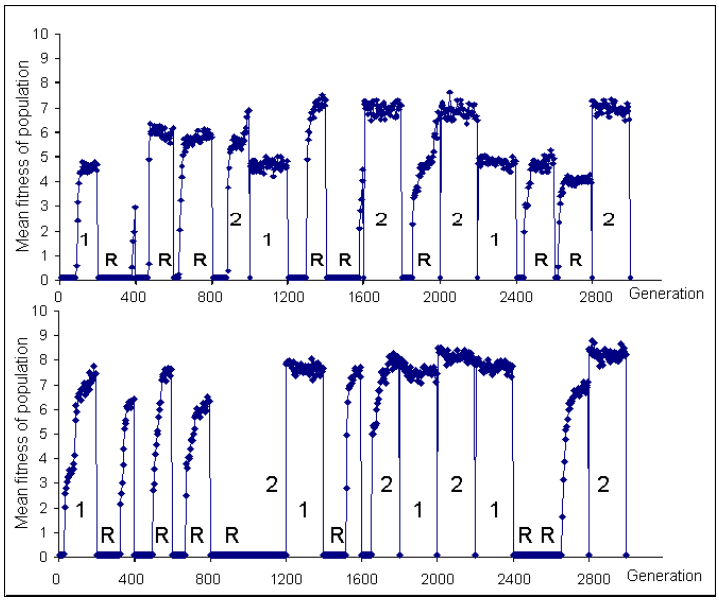

Figure 3.22: Performance of the explicit memory GA across mixed landscapes, 67% random

Figures 3.19 and 3.20 show the performances of the standard GA when one-off landscapes appeared in 25% and 67% of the epochs respectively. There is little difference between these and its previous performances.

Figures 3.21 and 3.22 show the performances of the explicit memory GA when one-off landscapes appeared in 25% and 67% of the epochs respectively. It is clear that the usual

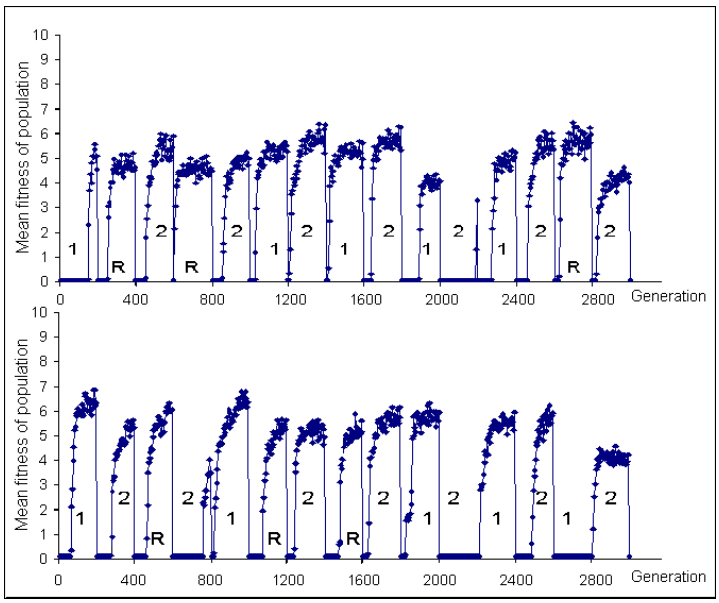

Figure 3.23: Performance of the structured GA across mixed landscapes, 25% random

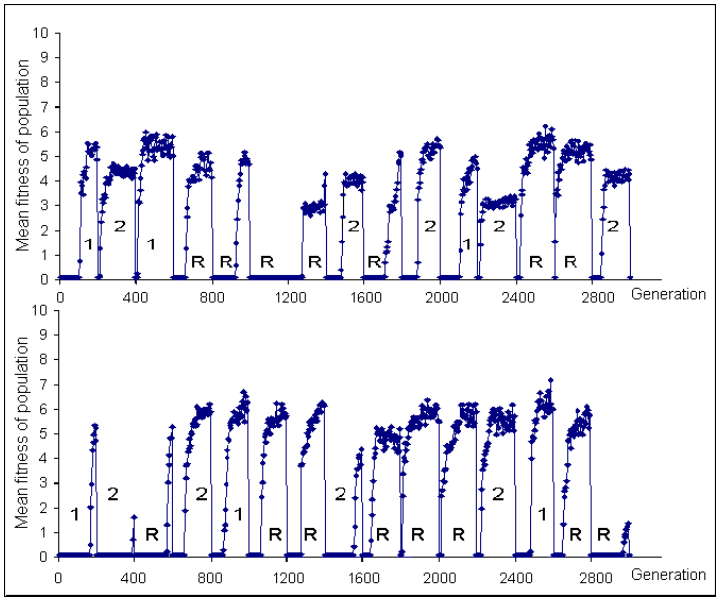

Figure 3.24: Performance of the structured GA across mixed landscapes, 67% random

behaviour of unfailingly restoring optimal genotypes was present, and that the interruptions by one-off landscapes did not undermine this, in spite of the fittest genotypes from those landscapes temporarily spending time in the memory bank.

Figures 3.23 and 3.24 show the performances of the sGA when one-off landscapes appeared in 25% and 67% of the epochs respectively. Again this algorithm resembles the standard GA in the negligible difference between these and its previous performances.

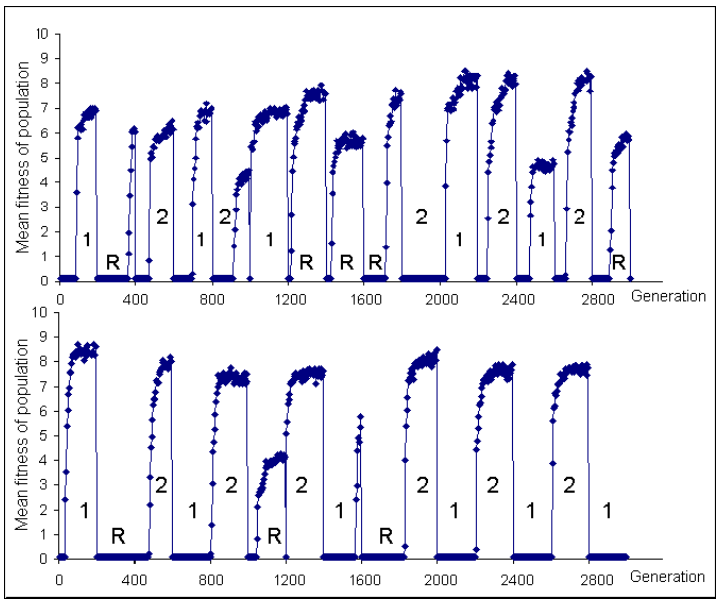

Figure 3.25: Performance of the diploid GA across mixed landscapes, 25% random

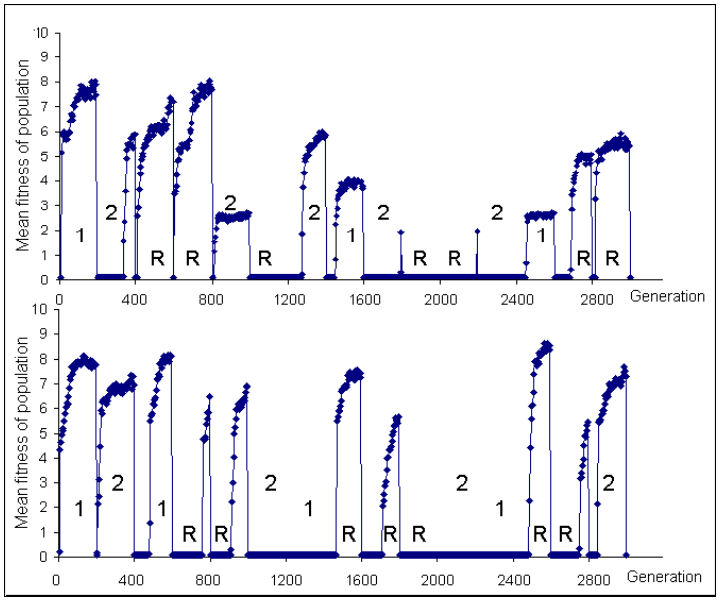

Figure 3.26: Performance of the diploid GA across mixed landscapes, 67% random

The corresponding results for the diploid GA – in Figures 3.25 and 3.26 – also show performances qualitatively similar to the previous ones.

3.5 On the Structured Genetic Algorithm

The structured genetic algorithm was claimed to offer improved adaptability (with respect to the standard GA) to environmental dynamism, and memory capacity (Dasgupta & McGregor

1992*b*, Dasgupta & McGregor 1992*a*, Dasgupta 1994). The first claim is of secondary interest to the present thesis, but it is worth observing that comparison between plots for the sGA and the standard GA in the same environment, does not detract from that claim.

With regard to the memory claim, however, every one of the fourteen plots concerning the structured GA, when compared to any one of the standard GA or explicit-memory GA plots, points to the same conclusion. In terms of the criteria in Section 3.1, the structured GA is a poor memory algorithm. Its failure to restore memories is clear from the plots, where it resembles the standard GA in its unpreparedness for returning landscapes. This contrasts with the behaviour of the explicit memory GA, which immediately reconverges in returning landscapes thanks to its immediate memory restoration.

The situation regarding the acquisition and (short term) retention of memories is more subtle, and it is necessary to examine the low-level behaviour of selected genotypes to see what happens. Figure 3.27 shows the fittest genotype in the population from the bottom plot in Figure 3.12 (page 44) at three points in the run: 5 generations before a landscape change, 5 generations after it, and 195 generations after it (that is, 5 generations before another change). The transition is starred in the figure. This particular transition was chosen because it seemed the most successful instance of memory behaviour by the algorithm, and so represents the best available 'advertisement' for it.

The way the sGA can memorise a bottom-level substring is by changing it from active/on to inactive/off. The first and third active substrings in the fittest genotype 5 generations before the change, were the only two of the five to be made inactive after the change. This in itself provides half of the explanation of the algorithm's deficiency, because it is clear by looking from left to right from the other three substrings, that they each evolved away from the previously fit patterns. By exposing to the genetic operators genes that should have been preserved, the algorithm effectively forgot them.

The first and third active substrings were thus the only two that were memorised in the present example. However, by looking from their leftmost to their rightmost configurations, it is clear that like those other substrings, they changed until they became unrecognisably different. This was not because of selective pressure, but because of genetic drift; mutation and crossover induced genetic changes which the algorithm had no means of preventing. These

	5 Before		5 After		195 After
on	**000001101111**	off	011001101111	off	001100001100
off	100111110100	off	010111100101	on	**010011000101**
off	011000100010	off	011000010010	off	001011101111
on	**011111111001**	on	**110111111001**	off	101101111000
off	100001000001	on	**100001000000**	on	**000101000000**
off	001011100100	on	**001011100100**	on	**101111000100**
off	110110101001	off	110110101001	off	001111101011
on	**111111110110**	off	111111110110	off	010011011110
on	**001111010011**	on	**111111010011**	on	**100110110011**
on	**110110110110**	on	**000111110110**	on	**110110101110**
	(Fitness = 7.92)		(Fitness = 6.94)		(Fitness = 9.75)

Figure 3.27: The fittest structured GA genotype 5 generations before, 5 after, and 195 after, the landscape change that is starred in Figure 3.12

"silent genetic changes" (Dasgupta (1994), p3), which geneticists may identify as *Muller's ratchet* (Muller 1964), represent the second half of the reason why the sGA is deficient as a memory algorithm. By failing to shield memorised substrings from the genetic operators, it loses them.

Bellas et al. (2009), who designed an sGA-like algorithm, wrote the following, which agrees with the analysis here.

> Obviously, if the cycles [i.e. periods of environmental stationarity] are very long, there comes a point where probabilistically the information that is being preserved in the unexpressed parts of the chromosome will tend to degrade and be lost (p2140)

To summarise this section, whereas the structured GA offers improved adaptability in nonstationary environments, its memory capability is poor, because it usually fails to acquire or retain memories. The idealistic description of its memory behaviour in Section 2.3.1 is unrealistic in the extreme.

3.6 On Dominance and Polyploidy

Similarly to the structured GA, various polyploid GAs have been claimed to be superior to the standard GA in dynamic optimisation, both generally and with respect to memory. The results presented earlier in this chapter suggest, however, that polyploidy does not offer any advantages. Returning optima were seldom rediscovered immediately (cf. the explicit memory GA) or even after a small number of generations, and instead it was usual for the algorithm to require many generations to locate them from scratch. Like the sGA, in terms of the success criteria given earlier, the sample diploid algorithm is an ineffective memory algorithm. And again like the sGA, it is only the failure to restore memories that is visible in the plots, and to understand the behaviour surrounding acquisition and retention, it is necessary to inspect individual genotypes.

5 generations before the change (fitness = 9.28)

o1-o0-o1-00-1o-1o-1o-0i-**1i**-i1-**io-1i-io**-ii-0o-**oi-0o**-io-**o1**-io–

11-io-oo-io-01-00-ii-o1-o0-1i-oo-**o0**-11-1o-i1-**oo**-io-**10**-01-oo–

10-11-1i-i0-1i-1i-0o-**i1-i1-01-11**-10-ii-**01-oi-11**-ii-o0-1i-11.

00000001110101000000–

10001010010010100110–

11111101111111011011.

5 generations after the change (fitness = 6.80)

o0-0i-o1-00-1o-1o-1o-0i-**i1**-i1-**oo-11-ii**-ii-0o-**1i-oi**-io-**o0**-io–

i1-io-**0o-oo-i1**-00-ii-o1-o0-1i-oo-**oi**-11-1o-i1-**o0**-io-**00**-01-oo–

oo-oo-01-10-ii-0i-0o-**ii-io-i0-1i**-10-ii-**i1-o1-1i**-ii-o0-1i-11.

00101110110111011101–

11001011010111101000–

00001001101011111011.

Figure 3.28: The fittest diploid GA genotype and phenotype 5 generations before and 5 generations after the landscape change that is starred in Figure 3.6

Figure 3.28 shows the fittest genotype in the population from the bottom plot in Figure 3.6 (page 41) at two points in the run: 5 generations before a landscape change and 5 generations

after it. The transition is starred in the figure. Similarly with the sGA analysis, this transition was chosen because it seemed to represent the most successful instance of memory behaviour by the algorithm, representing the best available 'advertisement' for it.

There are 60 loci and 60 corresponding mediate bits for the phenotype. The mediate bits that the two individuals did not have in common are underlined, as it is they that represent the difference between the optima. The genes that map to those mediate bits are also underlined.

The way a diploid genotype can pass smoothly from one optimum to another, is by a change in the dominance matrix (or equivalent mutation of all the genes) that has the following effect, per locus: if the alleles must map to the same mediate unit as before, they continue to do so, but if they must map to a different unit, then the matrix henceforth interprets them in that way. The fourth pair of alleles in the genotypes in Figure 3.28, <00>, give an example of an unchanged mapping, and the third pair, <o1>, give an example of a changed mapping.

Those two pairs of alleles highlight the diploid memorisation method. Redundant genes are the storage space for memories, and it is by a change in their interpretation (via a dominance relation or equivalent forced mutation) that they are restored to re-express. This is true for every polyploid GA, irrespective of the number of alleles or the number of genes per locus.

There is a critically important algorithmic detail on which this particular method depends – the coupling of landscapes with dominance matrices, which occurs by default in toy two-landscape two-matrix settings. If a given genotype that carries optimal genes for a given landscape returns to that landscape, the new dominance matrix must interpret the alleles such that the new phenotype gets to the returned optimum. Other researchers who have raised this issue are Kominami & Hamagami (2007), who wrote: "The suitable dominance map according to the environment change depend[s] on the heuristics" (p2), and Miorandi & Yamamoto (2008), who wrote something similar in a bullet-point summary of implicit memory (p4).

It seems evident that in order to get the mapping right, the algorithm must be able to identify every landscape, and have a specific matrix for it. It is telling that not a single one of the published polyploid algorithms even so much as attempted this. The closest anyone came was Saito & Hamagami (2010) with their adaptive mediation-type to phenotype mapping

(page 31), but this was a current-optimum *tracker* rather than an N-optima memoriser.

Furthermore, even if it assumed that a polyploid GA has a competent heuristic for handling the matrices (which could resemble the methods presented in chapter 7 of Simões (2010), for example), there are additional problems, connected to the central issues of memory acquisition and retention. Firstly, how can the genes be annealed into configurations that suit the matrices? There is only selective pressure on the mediate units, not the allele combinations behind them, so for example if the allele pair <0i> is desired, ordinary selection has no preference between <0i>, <0o>, and <00> (assuming the default mapping).

Secondly, assuming that a genotype optimal across multiple landscapes arises, how can it persist without corrupting? How can it resist Muller's ratchet? Figure 3.28 shows how extensive the corruption problem actually is, over the space of 10 generations. The allele pairings that changed are emboldened, and number 29 out of 60. Kominami & Hamagami (2007) were aware of this problem as well, and summarised it thus: "It is difficult to reuse the individuals adapted to the past environment because the GA process selects the best adaptive gene on the current environment condition [in every] generation" (p2).

The literature on nonstationary optimisation sometimes makes the terms *dominance and diploidy* and *implicit memory* seem interchangeable, but the reality, in terms of the present thesis, is that these algorithms are deficient with regard to memory. It is clear from all the plots that the exemplifying diploid algorithm was incapable of remembering previous solutions, and it is clear from a combination of those plots and subsequent analysis of its low-level behaviour and evolution that it was incapable of memorising in the first place.

And a further blow to the implicit-memory claim of polyploid algorithms can be delivered by the consideration that their memory handling method depends on both redundant genes – which are implicit – and dominance matrices – which are explicit. They ought perhaps to be called quasi-implicit memory algorithms.

Regarding the applicability of dominance and polyploidy, the following passages are pertinent.

> Given the evidence available so far, it can be assumed that the multiploid representations may be useful in periodically changing environments where it is sufficient

> to remember a few states and where it is important to be able to return to previous states quickly. The applicability to problems without periodicity and more than a few re-occurring states is at least questionable. (Branke 1999), p2

And having compared a diploid mechanism to a standard GA with enhanced mutation, Lewis et al. (1998) ended their paper thus:

> However, there is little difference in performance between this [diploid] GA and a simple haploid GA which undergoes heavy mutation when a decrease in fitness is observed between evaluations. [...] the case for implementing a diploid mechanism as opposed to a simple mutation operator may be weakened, given that diploid schemes require more storage space and extra evaluations to decode genotype into phenotype.

To conclude this section, dominance and polyploidy GAs are defective memory algorithms whose exceptional performances are confined to certain oscillating two-landscape environments. They resemble the structured GA in their chronic inability to acquire, retain, or restore memories, and there is little to recommend them.

3.7 Summary

This chapter reported experimental analyses of representative implementations of GAs that have been claimed to possess memory capabilities. The explicit memory GA that was tested supported the claims made about explicit memory in the literature, but this was not the case with the two implicit memory GAs (the sGA and a diploid GA). In a new contribution to the domain of heuristic nonstationary optimisation, the chapter demonstrated how and why the pre-existing implicit memory techniques are deficient.

In reaction to these findings, the following chapter makes an additional contribution to the domain by presenting a novel implicit memory algorithm that outperforms the defective alternatives.

Chapter 4

The Pointer Genetic Algorithm

This chapter introduces a new implicit-memory GA called the *pointer genetic algorithm* (pGA). It was created to discover if effective memory behaviour can be achieved via implicit memory, in the light of the failure of the previous implicit memory GAs. Its design is based on intuitions regarding how a successful implicit-memory GA should be constructed and how it should operate.

Section 4.1 supplies a full description of the pGA, including the rationales behind the design decisions. The section concludes with an idealised description of its operation, and references to some other algorithms from the literature with similar features.

Section 4.2 contains the test results of the pGA, which are conducted in the same conditions as, and are directly comparable to, those contained in Chapter 3. The results are analysed in Section 4.3, together with examination of the algorithms' behaviours. It is found that the pGA – particularly the hypermutational version – clearly outperforms the other implicit memory algorithms, whilst being outperformed itself by the explicit memory GA.

The chapter closes in Section 4.4 with descriptions of some speculative alternative versions of the pGA, largely inspired in reaction to certain limitations of the algorithm.

4.1 Description of the Algorithm

The simplest and most direct way a GA population can re-converge on a returned optimum is by the re-introduction of the genotype for it, in one step and in one piece. If instead building blocks of the genotype are re-introduced across multiple individuals, then it cannot be guaranteed that they will recombine in a single individual.

If an implicit memory GA is to have the ability to do this, then the most obvious way to implement it is by having a genotype comprising multiple chromosomes, where the chromosomes are structurally identical. To produce a given phenotype, one chromosome would be chosen to map to it (cf. representations where there are multiple *non-identical* chromosomes in the genotype, e.g. Hinterding (1997) and Mayer & Spitzlinger (2003)). If the chromosomes were not separated, there would necessarily be overlap (discussed in Chapter 2) which can be problematic in that it can restrict and/or mislead evolution.

Parsimony suggests that the choice of which chromosome expresses should be deterministic and related to the current state of the environment. More specifically, the genotype as a whole should be handled like an explicit-memory GA's memory bank, being left alone during stationary periods and accessed when changes are detected.

Because implicit memory algorithms ideally execute their behaviours internally, the chromosome handling method ought to be incorporated into the representation. The simplest way to do this is to have a gene that decodes to an integer that represents the ordinal position of one of the chromosomes in the genotype. By analogy with the C programming language and others, this gene shall be called a *pointer,* hence the algorithm itself being called the *pointer genetic algorithm* (pGA). (The lowercase P distinguishes it from a score of other algorithms called the PGA.) Figure 4.1 illustrates the representation, for n chromosomes and a pointer in $[1..n]$.

<Pointer, Chromosome 1, Chromosome 2, ..., Chromosome n >

Figure 4.1: The genotype of the pointer GA

At any given time, the chromosome that is active may be referred to as the *active chromosome* (ACh), with every other chromosome a *passive chromosome* (PCh).

A major problem with the other implicit-memory GAs is their failure to carry memory-bearing genes through foreign landscapes. By allowing mutation and crossover to manipulate those genes, the algorithms allowed them to drift and corrupt (evidenced in Chapter 3). To prevent this from happening in the pGA, it is decreed that *passive chromosomes are never mutated or crossed over.* Active chromosomes, on the other hand, are treated like standard GA chromosomes.

It is worth clarifying at this point how exactly crossover occurs in the pGA. What happens in effect is that the AChs of the two individuals are extracted, crossed over in isolation, and then replaced, taking no account of their absolute locus positions. The following diagram gives an example of one-point crossover:

<1, aaaa, bbbb> + <2, cccc, dddd>

⇓

<1, aadd, bbbb> and <2, cccc, ddaa>

The last issue to cover in this section is the control of the pointer values, which is key to the behaviour of the algorithm. It is shown in the remaining sections of this chapter that hypermutation of the pointers when the landscape changes is the best method, and that the simpler method of using a constant rate is less effective.

As with the earlier descriptions of the structured GA (Section 2.3.1) and dominance and polyploidy (Section 2.3.2), the present section now provides an idealistic description of how the pGA operates, with three chromosomes and via hypermutation of the pointers, in a cycling three-landscape environment.

An individual with pointer = 1 discovers the optimum in the first landscape, and spreads to take over the population. When the second landscape arrives, every individual's pointer mutates to 2 or 3, putting every copy of the previous optimum's genes in passive first-chromosomes. An individual with pointer = 2 discovers the optimum in the second landscape and takes over the population. When the third landscape arrives, the pointers all mutate to

1 or 3, putting the second optimum's genes in passive second-chromosomes. Fortuitously, an individual with pointer = 3 discovers the third optimum, so a situation arises where the population is dominated by individuals with the first optimum's genes in their first chromosomes, the second in their second, and the third in their third. When the first landscape returns, the pointers all mutate to 1 and 2, so the individuals with pointer = 1 promptly take over the population. Henceforth the population moves cyclically from optimum to optimum, flawlessly, until the end of the run.

As a postscript to this section, some other representations from the literature that contain something comparable to the pGA pointer will be mentioned.

First of all, the structured GA. The tree of genes above the bottom level that determines which substrings from that level are active, looks like a complicated pointer. However, the complication of the on-off signals that are transmitted down the tree, together with the fact that the final substrings constitute building blocks rather than whole chromosomes, weaken the pointer analogy.

Secondly, the modularity-primed representation in Parter et al. (2008), which was covered in Section 2.1.1, yielded the following, quoted from the paper.

> We find that the rapid adaptation to previously seen goals [...] is facilitated by key positions in the genome that can stabilize a desired sub-structure or module among other potential outcomes. We term these positions 'genetic triggers', since they can trigger a large and prepared phenotypic response. (p5)

Two differences between those genetic triggers and pGA pointers are that the triggers governed modules rather than whole chromosomes, and moreover that they emerged spontaneously rather than being deliberately coded-in and controlled.

Lastly, Jakobi (1996), in an investigation into neutral evolution within open-ended evolution, described "a simple encoding scheme that precludes local fitness maxima" (p3). There, it was permitted to add genes to the genotype – including bits that controlled whether given segments were "on" or "off" – such that it was possible in principle for any genotype to eventually evolve to be able to mutate to the global maximum in steps of monotonically increasing

fitness. The bits and their subordinate segments are respectively comparable at best to the pGA pointer and its chromosomes, and at worst to the sGA and its substrings.

The differences between that system and the pGA's are that the arrangement is expected to be produced by evolution rather than being coded in (cf. the previously mentioned genetic triggers), and that it is intended for single static landscapes, having therefore no mechanisms to cope with environmental dynamism.

4.2 Test Results

This section contains results of tests of the pointer genetic algorithm. The testing environment was identical to that used with the other memory algorithms (outlined in Chapter 3) and the parameters the pGA has in common with those algorithms were set to the same values. This enables direct comparison between the results for every algorithm under consideration, for every environment.

Two numbers of chromosomes were used in the cycling landscape runs, roughly corresponding to 'just enough' and 'plenty' in which to store memories. Specifically, there were 2 or 8 chromosomes for 2 landscapes, 3 or 9 for 3 landscapes, and 6 or 20 for 4 landscapes. The issue of the number of chromosomes is discussed in subsection 4.3.2 after the results have been presented.

In the pGA plots, the numbers printed along the tops of the curves indicate which pointer value was dominant in the population during that 200-generation epoch. Two numbers joined by a hyphen represent a transition between an early dominant pointer and a late dominant pointer. Where there are no such numbers, it is because there was either no dominant pointer, or no meaningful correlation between pointer and landscape.

The following three versions of the pGA were fully tested: (1) with a constant PMR of 10%; (2) with a constant PMR of 50%; and (3) with hypermutation of 90%, on average, of the pointers when the landscape changed. Regarding constant PMR, any given rate lies on a spectrum between 0% – where the algorithm is equivalent to a standard GA with useless introns – and 100% – where in effect there is random chromosome selection in every phenotype mapping. The rates of 10% and 50% were chosen for the tests because it was found in

preliminary runs that very high rates induce erratic behaviour and poor performances. Figure 4.2 gives an example of this, for a PMR that corresponded to random pointer choice.

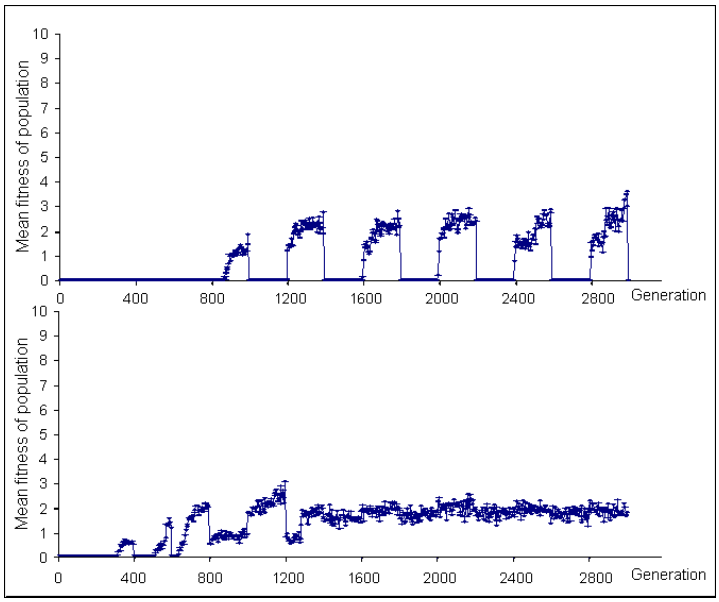

Figure 4.2: Performance of an eight-chromosome pointer GA across two cycling landscapes, with 87.5% pointer mutation

The things to note in Figure 4.2 are the inability of the population to find any of the second landscape's cones in the top plot, and the incoherent behaviour that emerged in the bottom plot, where the landscape changes are barely discernible.

The versions with constant PMR did not use hypermutation – their ongoing pointer mutation being their form of pointer control – so it was only the hypermutating version that depended on landscape change detection. As with the other algorithms covered in Chapter 3, this was bypassed in the program in order to highlight the memory behaviour.

Regarding the 90% probability of hypermutating, a rate of 100% can appear preferable because that guarantees *passive convergence* (described in Section 4.3) and thereby perfect memory acquisition. However, if 100% hypermutation is wrongly triggered – for example because of a mistake by a landscape change detection method – it pacifies every active chromosome, potentially ejecting the population from an optimum. A slightly reduced rate alleviates this problem by letting a small minority of the chromosomes remain active – and able to re-dominate the population – whilst still enabling extensive memorisation.

The pGA is subject to the same success criteria – set out in Chapter 3 – as were the other memory algorithms. It is assessed in Section 4.3, after its results have been presented in the intervening subsections.

4.2.1 Two Cycling Landscapes

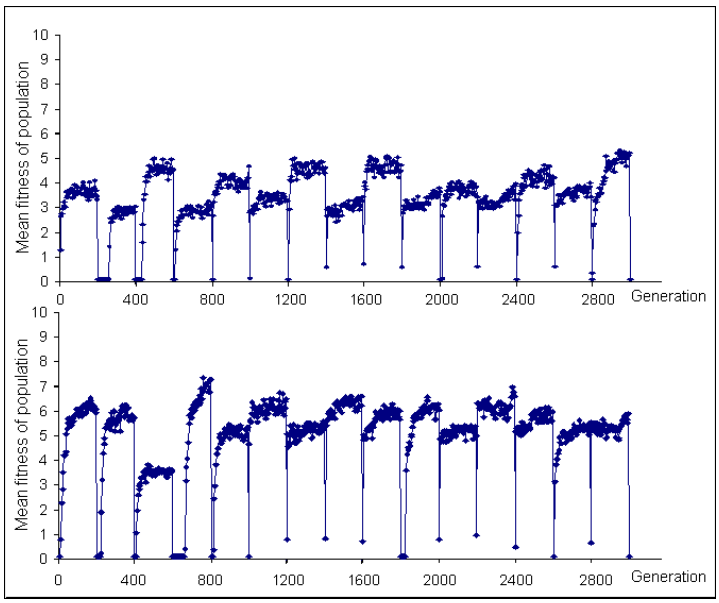

Figure 4.3: Performance of a two-chromosome pointer GA across two cycling landscapes, for 10% pointer mutation

Figures 4.3 and 4.4 show how the pGA with 10% PMR and with two and eight chromosomes respectively performed in the simple oscillating two-landscape environment. With two chromosomes, it behaved like a standard GA during the first four epochs, but by then the population had evidently memorised the two optima it had discovered, and for the remainder of the runs it rapidly reconverged on them when they returned.

With eight chromosomes (Figure 4.4) the population with the shorter genotypes (top plot) performed like the two-chromosome versions, albeit not finding much of the first landscape's cones. The longer-genotype population, however, performed as well as the explicit memory GA, as seen in Figure 3.3 (page 40). Optimal genes for the first landscape were stored in chromosome 8, and those for the second landscape were stored in chromosome 1.

Figures 4.5 and 4.6 show how the pGA with 50% PMR performed in the two-landscape environment. The two-chromosome version performed like a standard GA through both runs,

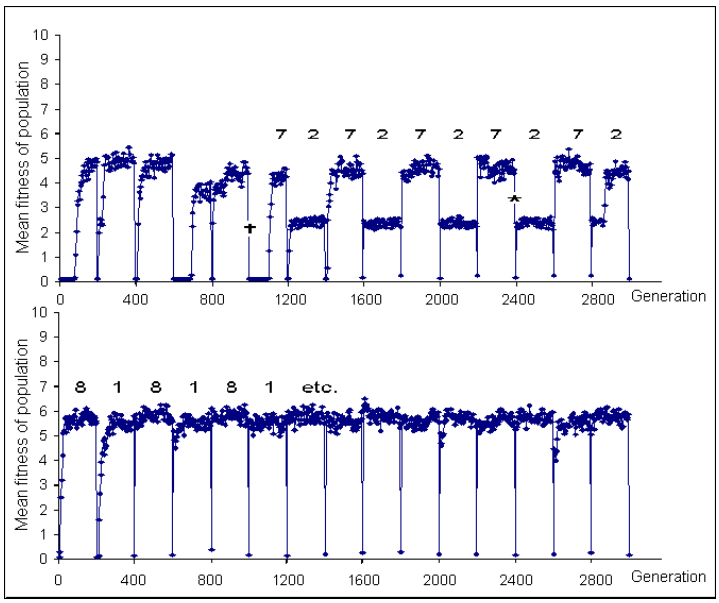

Figure 4.4: Performance of an eight-chromosome pointer GA across two cycling landscapes, for 10% pointer mutation

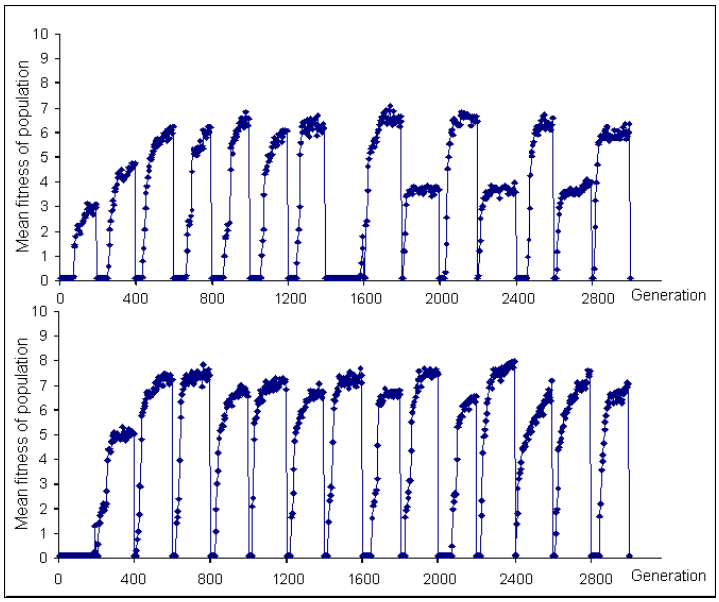

Figure 4.5: Performance of a two-chromosome pointer GA across two cycling landscapes, for 50% pointer mutation

but the eight-chromosome version did not. With the longer genotypes, it performed very badly, but with the shorter genotypes, it demonstrated memory behaviour (bottom plot of Figure 4.6). However, *three* chromosomes – 2, 4, and 7 – were used by the population instead of two, and it is visible by means of the dominant-chromosome numbering that there was instability in the transitions between epochs, in terms of the locations of the memories.

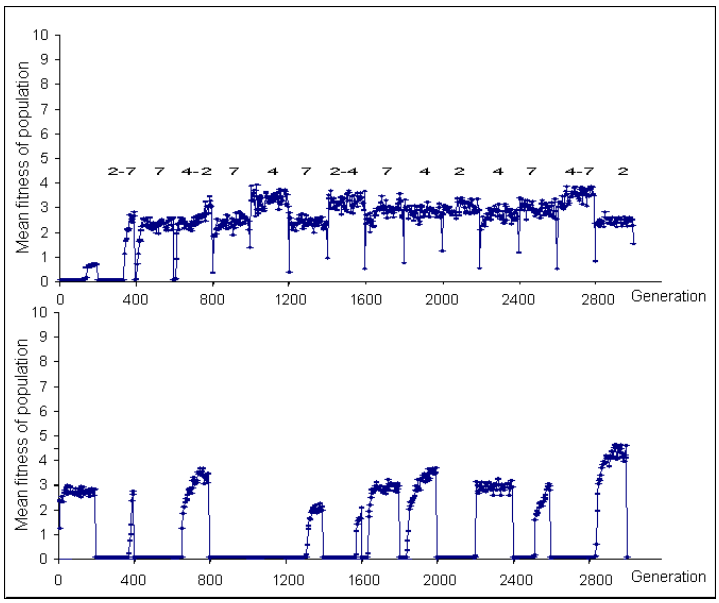

Figure 4.6: Performance of an eight-chromosome pointer GA across two cycling landscapes, for 50% pointer mutation

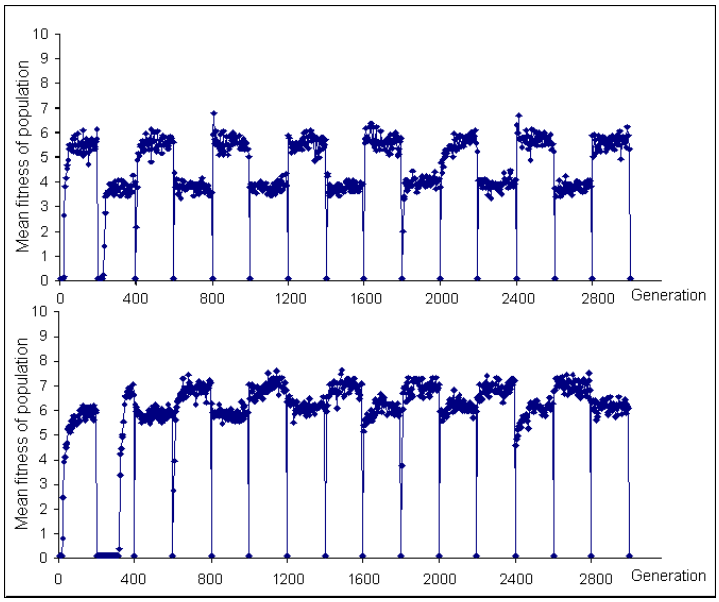

Figure 4.7: Performance of a two-chromosome pointer GA across two cycling landscapes, for triggered pointer-hypermutation

Figures 4.7 and 4.8 show how the pGA with triggered hypermutation of the pointers performed in the oscillating two-landscape environment.

In all four plots in Figures 4.7 and 4.8, it can be seen that the hypermutating pGA performed similarly to the explicit memory GA, with immediate reconvergence onto returning

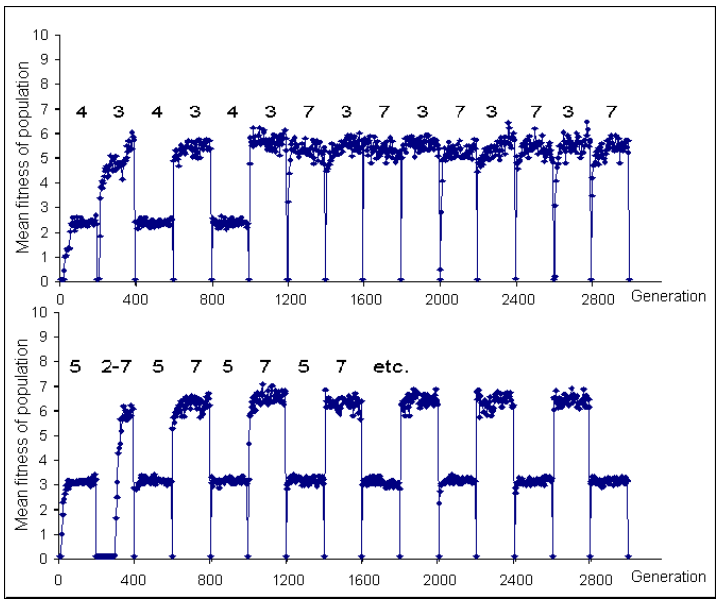

Figure 4.8: Performance of an eight-chromosome pointer GA across two cycling landscapes, for triggered pointer-hypermutation

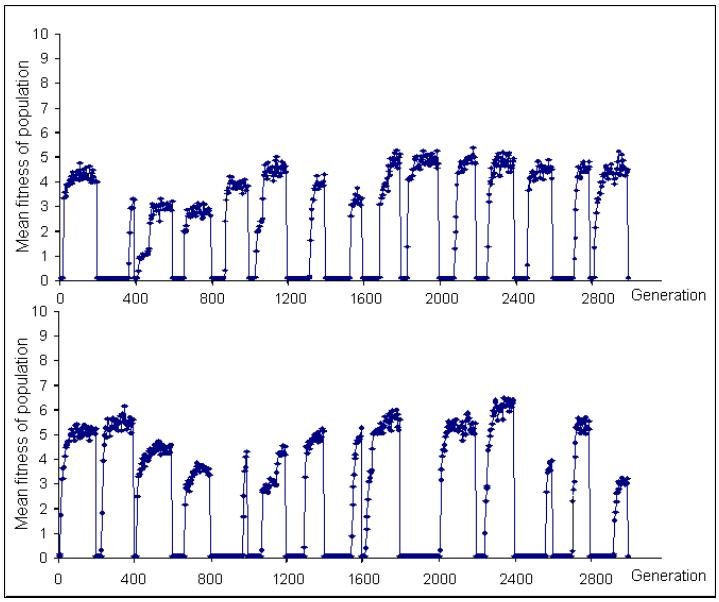

Figure 4.9: Performance of a three-chromosome pointer GA across three cycling landscapes, for 10% pointer mutation

optima. The quality of the optima, however, was lower than for the explicit memory GA, particularly in the bottom plot in Figure 4.8. It is interesting to note that a similar situation arose in the plot above it (for the shorter genotypes) but a change in the memory-bearing chromosome occurred that led to the acquisition of a better optimum.

4.2.2 Three and Four Cycling Landscapes

Figures 4.9 and 4.10 show how the 10% PMR pGA performed for three cycling landscapes, with three and nine chromosomes.

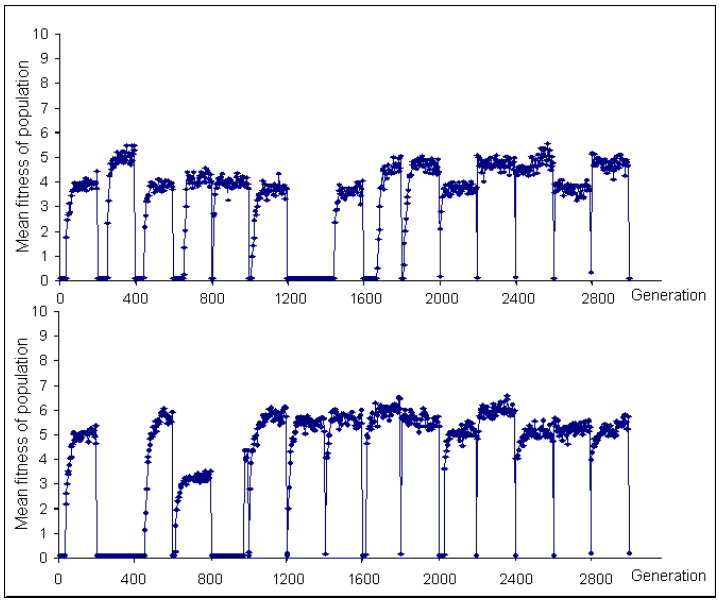

Figure 4.10: Performance of a nine-chromosome pointer GA across three cycling landscapes, for 10% pointer mutation

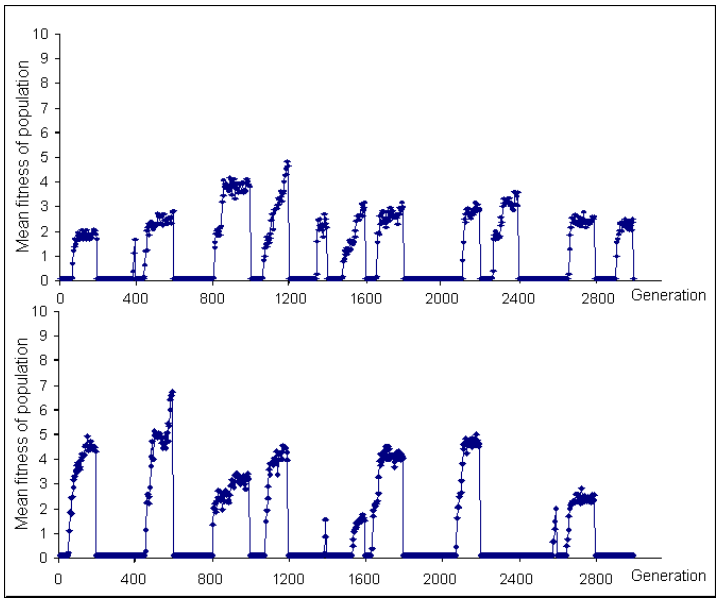

Figure 4.11: Performance of a three-chromosome pointer GA across three cycling landscapes, for 50% pointer mutation

With three chromosomes, the 10% PMR pGA performed badly, optimising in every returning landscape from scratch. The situation was the same with nine chromosomes during the first 6–10 epochs, but then memory behaviour finally appeared.

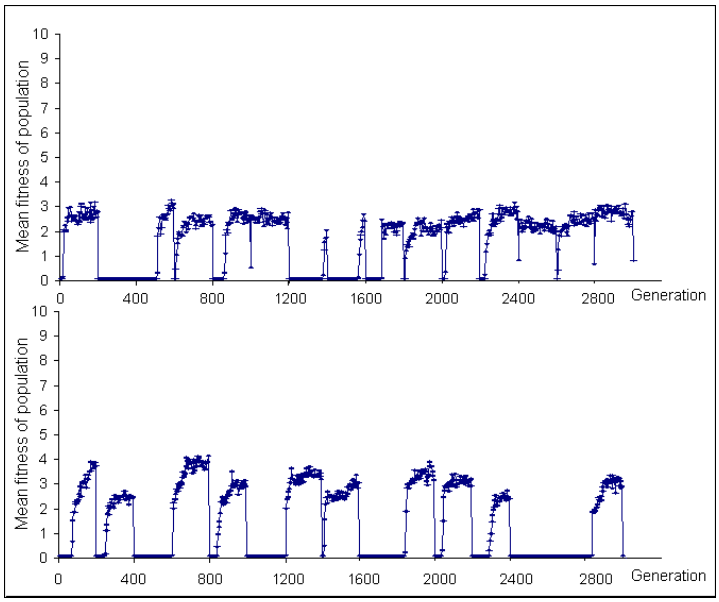

Figure 4.12: Performance of a nine-chromosome pointer GA across three cycling landscapes, for 50% pointer mutation

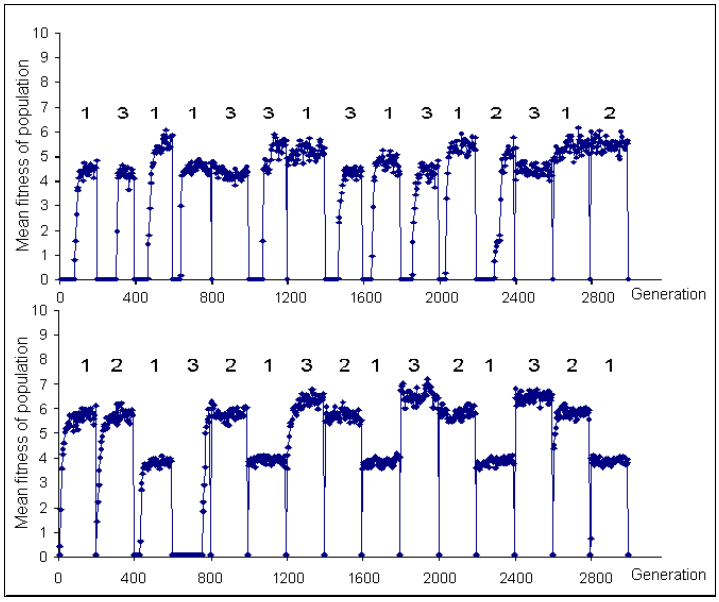

Figure 4.13: Performance of a three-chromosome pointer GA across three cycling landscapes, for triggered pointer-hypermutation

Figures 4.11 and 4.12 show how the 50% PMR pGA performed for three landscapes. With the exception of some memory recollection in the last epochs in the top plot of Figure 4.12,

its performances were poor.

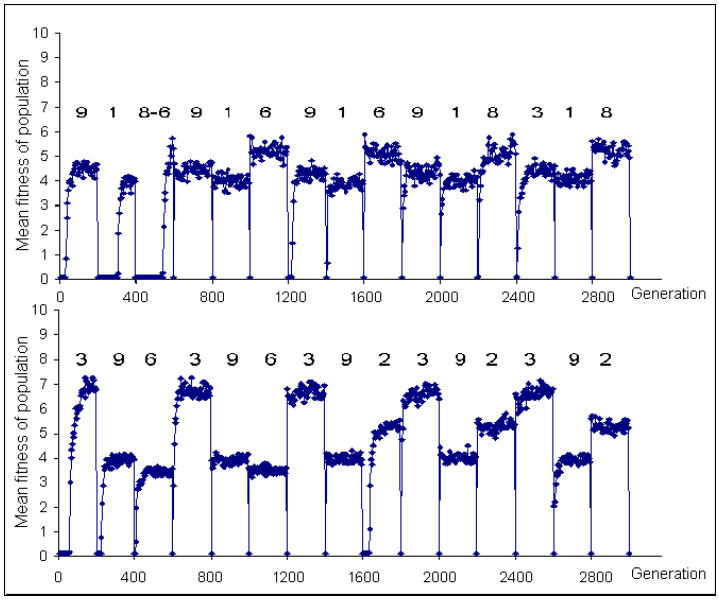

Figure 4.14: Performance of a nine-chromosome pointer GA across three cycling landscapes, for triggered pointer-hypermutation

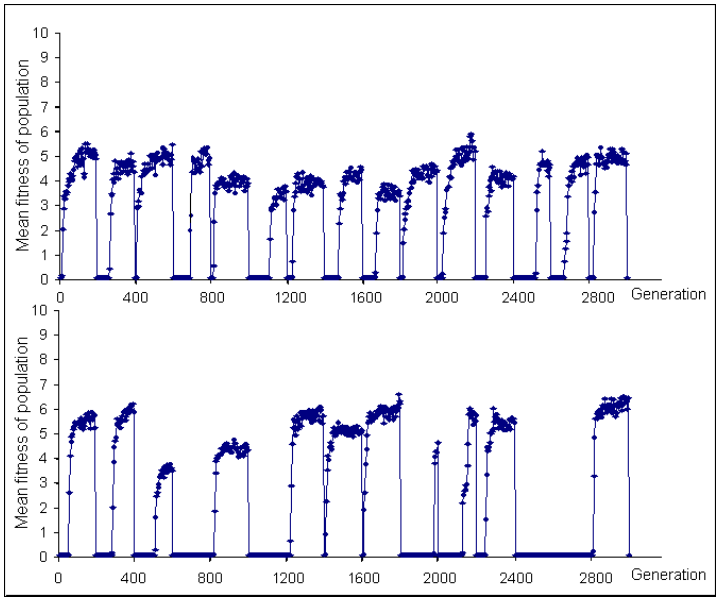

Figure 4.15: Performance of a six-chromosome pointer GA across four cycling landscapes, for 10% pointer mutation

Figures 4.13 and 4.14 show how the hypermutating pGA performed with three cycling landscapes. With three chromosomes, the shorter genotypes only occasionally achieved memory behaviour, but the longer genotypes came to successfully memorise all three optima, after each of them had appeared three times. In this regard the algorithm eventually matched the

explicit memory GA's high performance (Figure 3.10, page 43).

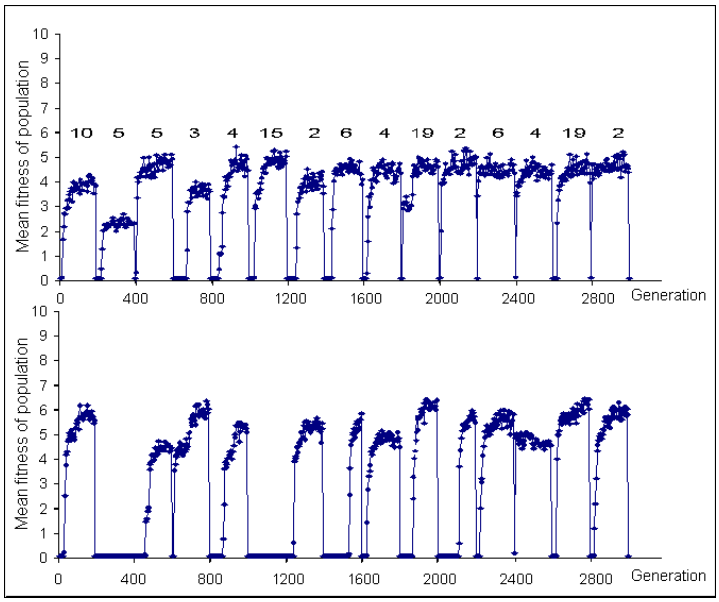

Figure 4.16: Performance of a twenty-chromosome pointer GA across four cycling landscapes, for 10% pointer mutation

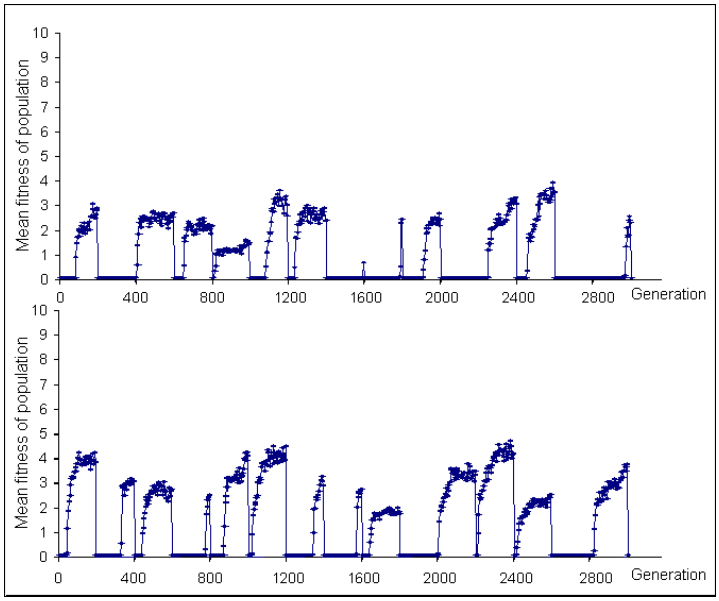

Figure 4.17: Performance of a six-chromosome pointer GA across four cycling landscapes, for 50% pointer mutation

With nine chromosomes, the hypermutating pGA needed only one visit to each landscape to memorise an optimum, and the curves in Figure 4.14 are consistent with effective memory behaviour. However, examination of the dominant-pointer sequences reveals that previously favoured chromosomes were sometimes usurped by others of similar fitness.

Figures 4.15, 4.16, 4.18, and 4.16 show how the constant PMR pGA performed for four cycling landscapes. For both rates, when the number of available chromosomes was low, the performances were poor, being worse for the PMR of 50%. When the number of chromosomes was higher, however, memory behaviour materialised near the ends of the runs.

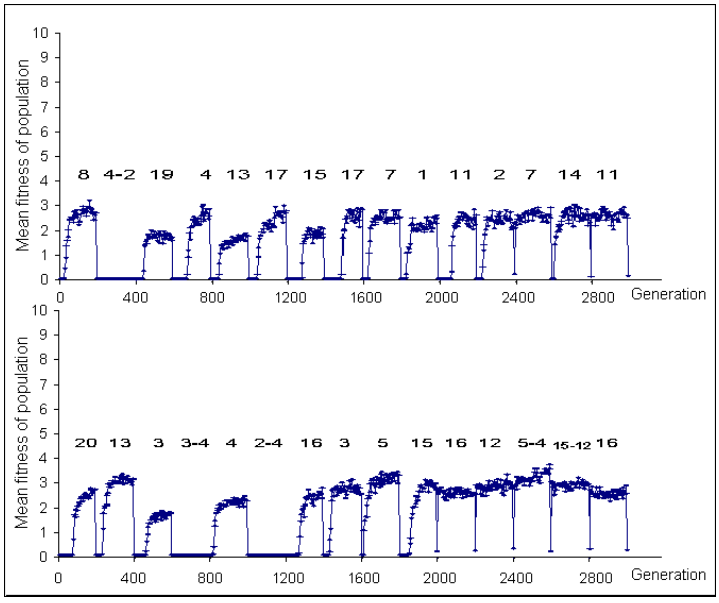

Figure 4.18: Performance of a twenty-chromosome pointer GA across four cycling landscapes, for 50% pointer mutation

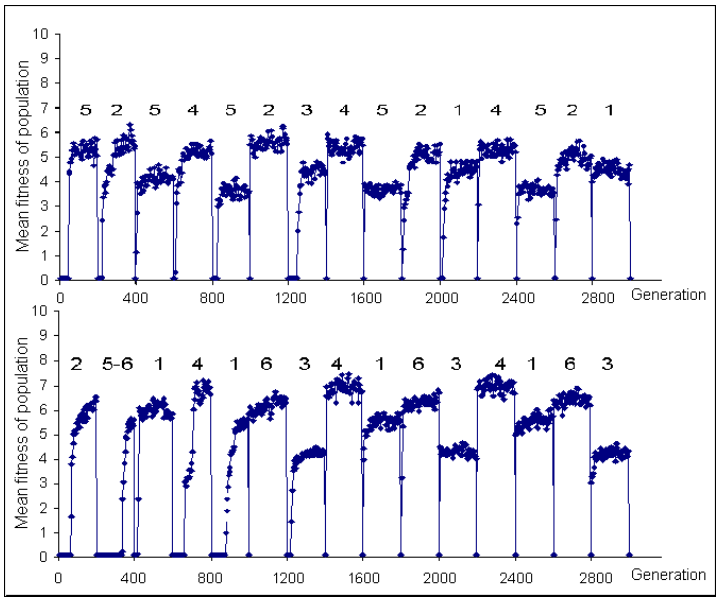

Figure 4.19: Performance of a six-chromosome pointer GA across four cycling landscapes, for triggered pointer-hypermutation

In the top plot in Figure 4.16, it can be seen that correlation arose between landscapes and

chromosomes, but that when chromosome 19 should have caused reconvergence immediately in the last return to landscape 2, there was actually a delay. Such a delay never occurred in any of the runs with the explicit memory GA.

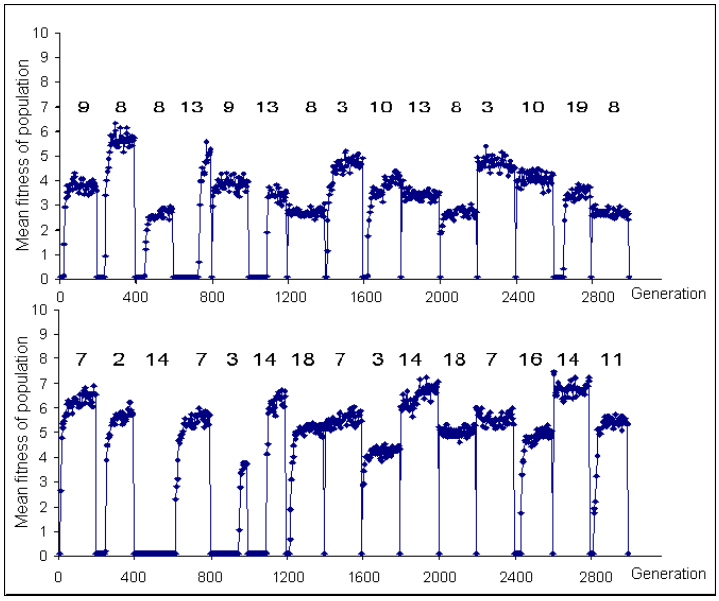

Figure 4.20: Performance of a twenty-chromosome pointer GA across four cycling landscapes, for triggered pointer-hypermutation

Figures 4.19 and 4.20 show how the hypermutating pGA performed with four cycling landscapes. Like the versions with constant PMR, these performances are similar to those with three landscapes. It was the norm rather than the exception for the hypermutating pGA to quickly attain high fitness in returning landscapes, although there were occasional instances of failed memory recollection, and it was not uncommon for the dominant chromosome in a given landscape to change seemingly arbitrarily.

4.2.3 Introducing One-Off Landscapes

Figure 4.21 shows how an eight-chromosome pGA with a 10% PMR performed in the environment composed of two recurring landscapes plus on-demand one-off landscapes designed to appear in 25% of the epochs. There were isolated instances of successful memory recollection (in the second half of the top plot, and in the 4th and 8th transitions in the bottom plot) but usually the population sought the optima from scratch, be they random or returned.

Figure 4.22 shows how an eight-chromosome pGA with a 50% PMR performed in the same

25%-random environment. Although there were isolated instances of successful recollection, there were also instances of failing to find any cone at all in several epochs.

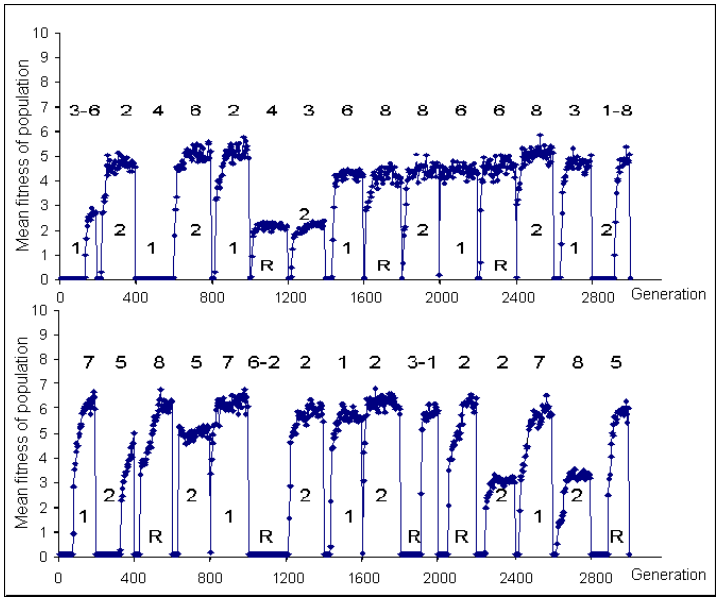

Figure 4.21: Performance of an eight-chromosome pointer GA across mixed landscapes, 25% random, for 10% pointer mutation

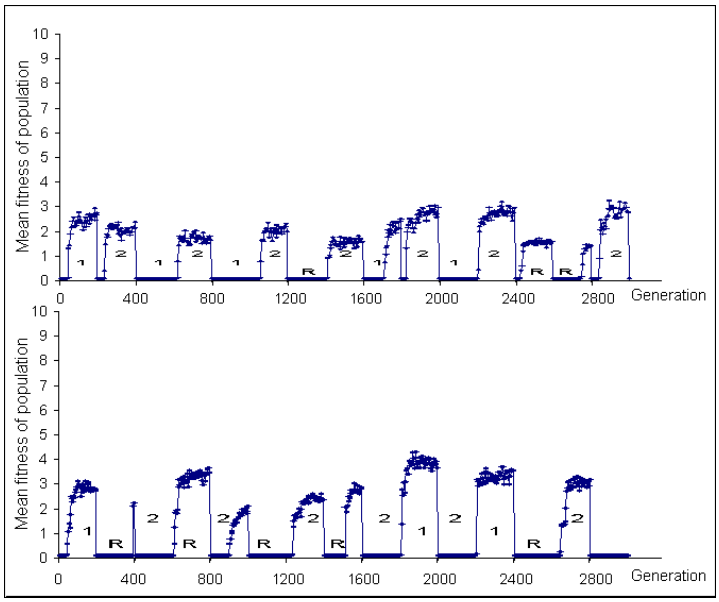

Figure 4.22: Performance of an eight-chromosome pointer GA across mixed landscapes, 25% random, for 50% pointer mutation

Figure 4.23 shows how an eight-chromosome pGA with triggered pointer-hypermutation performed in that environment. In spite of the difficulty the population had in optimising in the first landscape in the bottom plot, the memory behaviour was generally successful. The

infrequent random landscapes did not break up the recollection patterns that are characteristic of this algorithm.

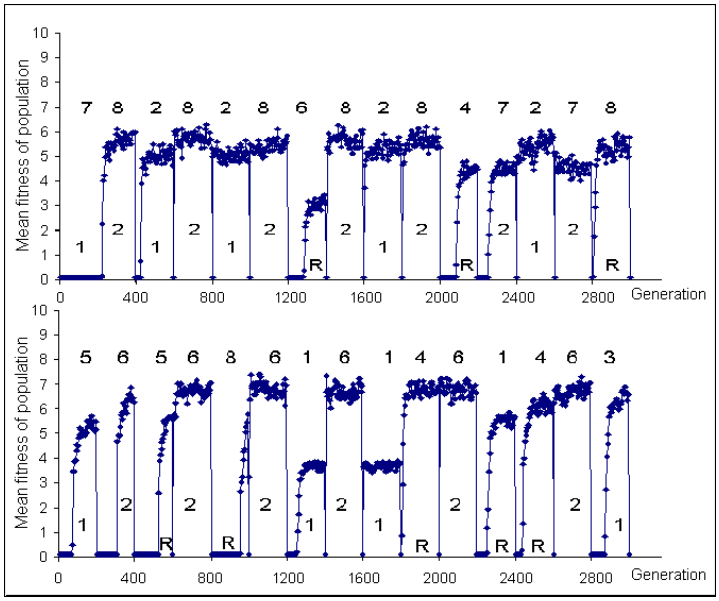

Figure 4.23: Performance of an eight-chromosome pointer GA across mixed landscapes, 25% random, for triggered pointer-hypermutation

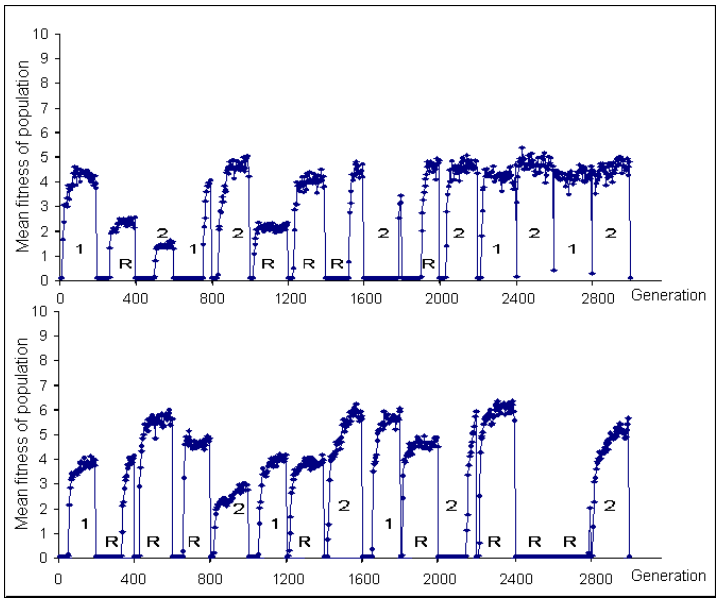

Figure 4.24: Performance of an eight-chromosome pointer GA across mixed landscapes, 67% random, for 10% pointer mutation

Figures 4.24 and 4.25 show how the pGAs with constant PMRs performed in the environments where random landscapes were programmed to appear in 67% of epochs on average. It was invariably the case that the many random landscapes prevented memorisation from

occurring. The latter part of the top plot in Figure 4.24 is exceptional thanks to a fortuitous absence of random landscapes. The performance of the pGA with 50% PMR was particularly bad, worst of all near the end where four consecutive landscapes passed without any evolutionary progress.

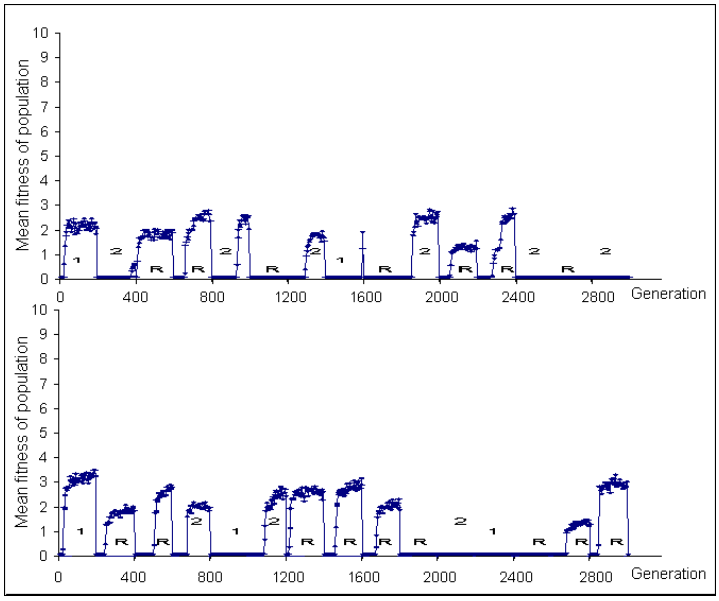

Figure 4.25: Performance of an eight-chromosome pointer GA across mixed landscapes, 67% random, for 50% pointer mutation

Figure 4.26 shows how the hypermutating pGA performed in the 67%-random environment. The algorithm performed very well, almost matching the explicit memory GA (Figure 3.22, page 49). In both plots, it can be seen that fit genes for the second landscape were retained throughout the run, in chromosome 1 in the top plot, and in chromosome 4 in the bottom plot. The bottom plot also shows the best example of memory retention by any algorithm tested in the present thesis bar the explicit memory GA – fit genes for the first landscape persisted in chromosome 3 through eight epochs (= 1,600 generations). That feat was not repeated in the top plot, however, because the memory-bearing chromosome was recycled in the very next landscape, where it came to store the other landscape's optimum.

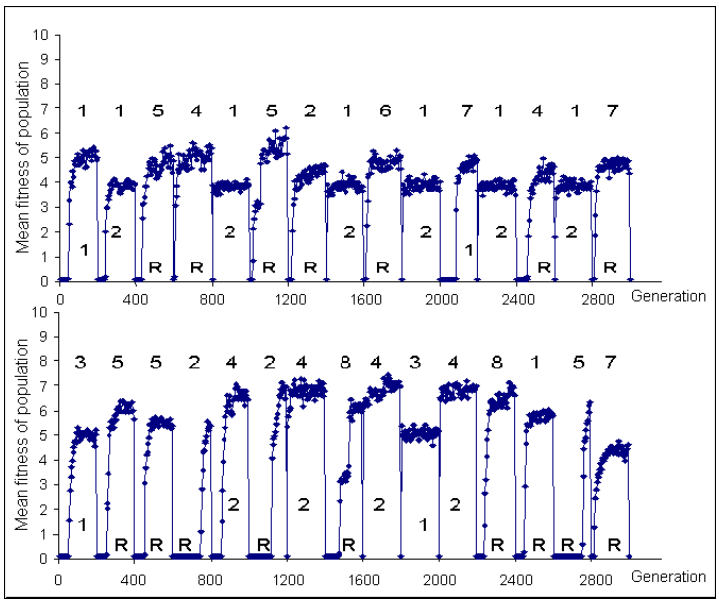

Figure 4.26: Performance of an eight-chromosome pointer GA across mixed landscapes, 67% random, for triggered pointer-hypermutation

4.3 Analysis of Results

The results plots by themselves – when compared to their corresponding plots in Chapter 3 – indicate the pGA is much superior to the other implicit memory algorithms (the structured GA, the diploid GA) but inferior to the explicit memory GA, with regard to memory behaviour. This assessment is made based on the visible evidence of rapid reconvergence on returning optima, which is precisely what happens in an effective memory algorithm.

In the present section, various aspects of the behaviour of the pGA implementations are analysed. The aim of doing this is to complete the exposition of the algorithm, and thereby to let it be fairly compared to its rivals. The pGA with a constant pointer-mutation rate is considered first, followed by the hypermutating version.

4.3.1 Continuous Pointer-Mutation

The first thing to remark is that the low mean-fitnesses achieved by the pGA with 50% PMR are misleading; the fittest individuals were similarly fit to those in the other algorithms. For example, whereas the mean fitnesses in the last epoch in the top plot in Figure 4.6 are around 2.4, the fittest individuals were 7.9; and in the bottom plot of Figure 4.18 the mean is around

2.5 while the fittest were around 7.0.

The mean fitnesses were lowered by the 50% of the population who, in any given generation, had had their pointers mutated in the previous generation. To elaborate, if half the individuals in the population were on an optimum, those individuals would represent the majority of the population in the next generation – thanks to the proportionate selection regime – but then half of them would be pointer-mutated, giving them new active chromosomes. Those new chromosomes tended to be different from what they had replaced, hence the genotype's fitness dropped.

In spite of being inferior to the hypermutational pGA in terms of memory behaviour across all the runs taken together, the versions with constant PMR did sometimes demonstrate effective memory behaviour. Converged genotypes from the run represented in the top plot of Figure 4.4 (page 65) are used to support explanations of how memories were acquired, retained, and recalled.

<u>1100001001000010110110000110</u>10,1011011000111010101111000110 01,1001100011101100101111111011011,0011111001000101010101111001110101101111101000000001001010 11,1110010000111111101001101111 01,1100101111110011001111111000011,0100101000011001100111010001111100001001100010110110000110 10,1011011000111010101111000110 01,1001100011101100101111111011011,0011111001000101010101111001110101101111101000000001001010 11,<u>0011011000100111100010000110 01</u>,1100101111110011001111111000011,0100101000011001100111010001111

Figure 4.27: The dominant pGA genotype 20 generations before and 20 generations after the first landscape change in Figure 4.4, where the ACh is underlined

In the first epoch shown in the plot, an individual with chromosome-1 as the active chromosome discovered a cone and dominated the population. Figure 4.27 shows the fittest genotype in the population 20 generations before and 20 generations after the first transition from landscape one to landscape two. The upper ACh (underlined) mapped to the aforementioned cone, and when the landscape changed, that chromosome lost its fitness. The ongoing 10% pointer mutation pacified 10% of those chromosomes in the next generation, and (approximately) geometrically decreasing percentages in the subsequent generations.

An individual with chromosome 6 as the ACh found the slope of a cone ten generations into the second epoch, and proceeded to dominate the population. The individual had the genes for the first landscape's cone – one of which was mutated (emboldened in the figure) – in its

now-passive chromosome 1, thanks to a pointer mutation during the previous ten generations. When this individual filled the population with copies of itself, every PCh *hitchhiked* with its fit ACh, creating a situation where every chromosome was converged. The process of passive chromosomes becoming converged is termed *passive convergence* by the present thesis.

It is by pointer mutation that the pGA commits gene patterns to memory, and it is by facilitating passive convergence that the pGA retains memories. The necessity of passive convergence of memories can be appreciated by considering its absence, that is, a scenario where there are individuals in the population who do not carry the memory in a PCh. Genetic search is performed in the AChs, and it is essentially a matter of luck which individual will discover the next cone. If a memory-bearing individual discovers the cone, the memory passively converges, but if a non-bearer discovers the cone, the passive convergence that occurs drives the memory-bearing PChs out of the population.

Passive convergence of memories in itself is not enough to assure their persistence. The disabling of mutation and crossover in PChs enables memories to persist in them, because otherwise, mutation would gradually corrupt them and crossover would fragment them. The total absence of any shielding of memory-bearing genes contributed to the failure of the structured GA and the diploid GA to retain memories (Chapter 3).

011010110011000011111001111000,1111010001111010101111101110001,0011111001010001111111111010000,0011010000001100100110000111010110101100001000101110011111000,0000101000101010000110011100000,01110111000000101011110111101,011101110110001010011100010111

011010110011000011111001111000,1111010001111010101111101110001,0011111001010001111111111010000,0011010000001100100110000111010110101100001000101110011111000,0000101000101010000110011100000,01110111000000101011110111101,011101110110001010011100010111

Figure 4.28: The dominant pGA genotype in generations 1980 and 2780 in Figure 4.4, where the ACh is underlined

Figure 4.28 gives an example of a successfully persisting pGA memory. It is the second chromosome in either genotype, and it was passive in generations 1800–2000, 2200–2400 and 2600–2800. In the time considered it alternated between the active and passive states without undergoing any genetic change.

The third aspect of effective memory behaviour is the recollection of memories. In the pGA, this is provoked by pointer mutation, which is also what provokes memory acquisition.

When a population returns to a landscape for which it has fit genes in PChs, pointer mutations that activate those chromosomes thereby create the opportunity for them to spread again through the population. Figure 4.29 shows an example of that for the memory from Figure 4.28.

011010110011000011111001111000,111101100111101010011101110001,001111100101000111111111010000,001101000000110010011000011101
011010110000100010111001111000,000010100010101000011001110000,011101110000001010111101111011,011101110110001010011100010111

011010110011000011111001111000,111101100111101010111101110001,001111100101000111111111010000,001101000000110010011000011101
011010110000100010111001111000,000010100010101000011001110000,111101110000001010111101011011,011101110110001010011100010111

Figure 4.29: The dominant pGA genotype 20 generations before and 20 generations after the generation-2400 landscape change in Figure 4.4, starred, where the ACh is underlined

Instances of success having been showcased, instances of failed memory behaviour by the constant-PMR pGA are now covered. It is important to note from the figures that failure was more frequent than success – which was not the case with the hypermutating version.

011010110011000011111001111000,101101100011101010111100011001,011011100101001101011011010011,100110100110010000010000000000
010011100111101010111101110001,111011010011000101010111001011,010111110000110110011111101101,010010100001100110011101000111

011010110011000011111001111000,101101100011101010111100011001,001111100101000111111111010000,001101000000110010011000011101
011010110000100010111001111000,000010100010101000011001110000,011101110000001010111101000111,010010100001100110011101000111

Figure 4.30: The dominant pGA genotype 20 generations before and 180 generations after the generation-1000 landscape change in Figure 4.4, crucifix, where the ACh is underlined

Figure 4.30 shows an example of a memory failing to be acquired; the bottom left chromosome should have been the same as the one underlined in the top half of the figure, but it is 10 bits different. What happened was that after the environmental change occurred, it took the population about 100 generations to find a cone. During those 100 generations, AChs carrying the previous landscape's genes went into genetic drift, with the extent of the drifting related to the activity level of the genetic operators, and on how long they spent as PChs. The individual that eventually discovered the next cone was carrying a heavily (10/30 genes) corrupted memory, so when that 'memory' converged along with the rest of its genotype, any accurate remaining copies elsewhere in the population were purged.

1100001001100010110110000011010,1011011000111010101111000011001,1001100011101100101111111011011,0011111001000101010101111001110101101111101000000000100101011,0011011000100111100010000011001,1100101111110011001111111000011,0100101000011001100111010001110110101100011000011111001111000,1011011000111010101111000011001,0110111001010011010110110100111,1001101001100100000100000000000000110010000001010111101111011,1110110100110001010101110010110,0101111100001101100111111101101,0100101000011001100111010001111

Figure 4.31: The dominant pGA genotype in generations 220 and 780 in Figure 4.4, where the ACh is underlined

Figure 4.31 concerns the memory shown earlier in Figure 4.27, acquired during the first epoch in the run. Even though it was not recalled when its landscape returned, it offers an example of long-term (i.e. multiple-epoch) memory retention by the constant-PMR pGA. The genotype in the top half of Figure 4.31 contains the memory pattern in the first chromosome (top left). It was held in the population for a few hundred generations, before being lost during the epoch between generations 600 and 800. Its absence from the genotype in the bottom half of the figure testifies to that.

Memory retention failure in the pGA happens in essentially the same way as memory acquisition failure. That is, by a non-memory-bearing individual taking over the population after the passive convergence of the memory has been undermined by genetic searching induced by environmental change.

Figure 4.32 shows the first epoch where the afore discussed memory failed to be recalled. When the population re-entered the returning first landscape for the first time, genetic search in AChs fortuitously discovered a new cone very quickly. The re-emerging memories – of a different cone – therefore found themselves competing with other fit individuals, as opposed to unfit individuals whom they would expect to displace. The struggle between these two rival groups was won by the bearers of the new cone's genes, so the memory from the first epoch was not recalled. It should be noted, however, that the old memory did remain stored.

1100001001100010110110000011010,1011011000111010101111000011001,1001100011101100101111111011011,0011111001000101010101111001110101101111101000000000100101011,0101011100101011100010000111001,1100101111110011001111111000011,0100101000011001100111010001110110101100011000011111001111000,1011011000111010101111000011001,1001100011101100101111111011011,0011111001000101010101111001110101101111101000000000100101011,1001101101100111010011001100110,1100101111110011001111111000011,0100101000011001100111010001111

Figure 4.32: The dominant pGA genotype 20 generations before and 20 generations after the first return of the first landscape in Figure 4.4, where the ACh is underlined

The memory behaviour of the pGA with a continuous constant pointer-mutation rate has now been described. Before turning to the more successful version with triggered hypermutation of the pointers, an additional problem associated with a constant PMR will be raised. It is separate from the foregoing analysis because the epochs of 200-generation landscapes used in the tests did not discernibly induce it.

1100110101100011010010111011 10,01111101000010010101101001 1101,01111111101110010110000011 1001,01110100100000101000001111101100011011100111011111111101101 01,00110000111111111101110001 1010,01110100100000101000001111101 1,11010100011000001110011101000 01
01111100010010101000101111101 01,01011000000101110110110001 1011,01011000000111110100110001 1011,01011110000101011100110010101001011000000101110111100011011,01011000000101110100110001 1011,01011000000101110100110001 1011,00111010100001101110110011110

Figure 4.33: The dominant pGA genotype at the end of the top run in Figure 4.6, and the genotype for a re-run without landscape change

Ongoing pointer mutation moves chromosomes in and out of the ACh evolution of the population. Sometimes the pointer mutants promptly disappear, but otherwise they survive for some number of generations. If the population is converged or close to being converged, crossover between a pointer mutant and an individual from the majority group has the effect of making the mutant's ACh more similar to that of the majority. In other words, these pointer mutants are 'bred into' the majority group, in what is in effect a process of population-wide homogenisation.

Figure 4.33 gives an example of this homogenising. The top plot shows the dominant genotype at the end of the run represented in the top plot of Figure 4.6, for a pGA with 50% pointer mutation. The bottom plot shows the dominant genotype at the end of a repeat run where the first landscape was the only landscape. It can be seen that one particular gene pattern effectively spread through all the chromosomes. Although there is a small degree of variation between the chromosomes, the intra-genotypic diversity that forms the basis of successful pGA behaviour was lost.

The experimental results suggest that this problem is less severe than those pertaining to memory behaviour directly, but it is a problem nevertheless. The most obvious way to solve it would be to disable the crossover operator, but this would have the effect of reducing the evolutionary search power of the population.

4.3.2 Triggered Pointer-Hypermutation

Like the pGA with constant PMR in the previous subsection, the pGA with hypermutation of the pointers will be examined with respect to memory acquisition, retention, and recollection. The experimental results show that it is the superior form of the pGA, and the present subsection will explain why and how that is the case.

The first thing to consider is memory acquisition by the population. This is achieved in the pGA by the pacifying of chromosome(s) containing the genes to be memorised, and subsequent *passive convergence* of those chromosomes. With continuous pointer mutation, the pacifying is performed in instalments proportional to the PMR, and desirable passive convergence only occurs if an individual with the unspoilt memory in a PCh discovers the next optimum. With triggered hypermutation of the pointers, on the other hand, the situation is far better. The vast majority of the memories (90% in the tests) are pacified immediately, avoiding the possibility of corruption, and making passive convergence extremely likely. The only way a correctly timed instance of hypermutation can fail to cause memorisation is if one or more of the tiny minority of un-mutated individuals persist and discover the next optimum.

In the 224 landscape-transitions represented in the eight figures for the hypermutating pGA, there were only 5 instances of the aforesaid phenomenon, namely: the 3rd and 5th transitions in the top plot of Figure 4.13, the 2nd transition in the top plot of Figure 4.20, and the 1st and 2nd transitions in the top and bottom plots respectively of Figure 4.26. Setting the hypermutation rate to 100% would have prevented those misfortunes, but the side effect of that rate in a pGA using landscape change detection would be that in the event of an erroneous detection, the entire population would be ejected from the current optimum. A tiny minority of un-pointer-mutated individuals are a safeguard against that risk, so their (unnecessary) inclusion in the present tests was deemed worthwhile because a real-world pGA would include them.

There is an aspect of pGA memory acquisition which despite not occurring in the experiments, exists as a realistic possibility. It does not pose a problem, but it is worth mentioning for completeness. When the populations converged in the runs, they were dominated by a single, highly fit ACh, which was often memorised. The possibility exists however that two

(or more) distinct AChs of similarly high fitness could arise simultaneously. It is likely that in this situation, there would be two competing factions in the population, and that a *random walk* would eventually enable one of them to purge the other and dominate the population. The ramification of a two-or-more-faction population is that if the landscape changes while they are still competing, passive convergence of a single memory could not be induced; both gene patterns would be memorised, and it is likely that one of them would later disappear as a consequence of another random walk and/or of domination of the population by a genotype carrying the rival memory.

Because it is a chromosome's fitness that is usually its valued trait, and not its genetic configuration, the loss of one or more fit chromosomes in favour of another similarly fit chromosome is not a problem.

Regarding the retention of memories, the hypermutating pGA resembles the constant PMR pGA in that it attains this via passive convergence of the memories. And it is common to both versions that the threat of losing memories comes from pointer mutations exposing them to the genetic operators and causing corrupted versions of them to subsequently fill the population. But whereas this threat was serious in most of the constant-PMR runs, it was less so in the corresponding hypermutational runs. This was a simple consequence of the fact that the triggered pointer mutations were only executed in the transitions between epochs, rather than continuously.

0101110101000010110011110000010101110000110001000101000001 10,1100111110101111001010011010011100011111010011010010011 01000
<u>1111111100001111111111100001111001100000001010001110111110 1</u>,1100010110111011001110010001100000001001001000001100000 10110
1101110011000110011001001101111001110010001010110010001011 00,0100011101011100000101111111000000001111100011000010110 10100
0100110100111111010100001000110010000011111111011111100 10001,0110011100000110111111010011000111100111101100100001 0001100

0101110101000010110011110000010101110000110001000101000001 10,0100010000100110100100111100010100111101011100101001111 1110
<u>1111111100001111111111101011111100111000000101001110 01111101</u>,0011100111111001110001111100001000010001100011010000000 10000
1111000000101010111100101000010101001101110110000010100 0010,0100011101011100000101111111000000001111100011000010110 10100
0100110100111111010100001000110010000011111111011111100 10001,0111111110110100001101010101111111101000010100010111001 01101

epoch were retained in memory for 8 epochs before re-expressing when the first landscape eventually returned. Figure 4.34 shows the dominant genotype in the population during the first and (much later) second epoch of the first landscape. It is clearly the same chromosome in both instances; 55 of the 60 genes are the same, and the 5 that are different made only tiny contributions in the binary-to-decimal phenotype mapping – in fact such genes often effectively drift during periods of convergence because of how little selective pressure falls on them, so they can almost be disregarded.

The third and final memory behaviour to be considered is recollection. Like the pGA with constant PMR, the hypermutational pGA performs this via pointer mutation when the relevant landscape returns; extensive pointer mutation 'throws the doors open' on the PChs, with the goal of reactivating any memories. Reactivated memory-bearing chromosomes are expected to be fitter than the other chromosomes, and consequently to take over the population during the coming generations.

The probability of a given memory being reactivated from a *fully converged* pGA population depends on three parameters: the pointer mutation rate, the population size, and the number of chromosomes in the genotype. The number of copies of the memory that can be expected to be reactivated by a triggered hypermutation is given by the following equation:

$$\textit{Number of copies} = \textit{PMR} \times \textit{Population size} \,/\, \textit{Number of chromosomes}$$

For the example of the runs with random landscapes included, the PMR was 0.9, the population size was 200, and there were 8 chromosomes in the genotype, so the number of copies expected was 23.

That equation describes an expectation, so the number of copies that is exposed varies in practice. It is important that at least one of each PCh be reactivated because the algorithm does not know how many memories there are nor where they are stored. It is therefore sensible when configuring a pGA to make the population size much larger than the number of chromosomes in the genotype, as this raises the probability. It is also sensible to have a high PMR, although with this parameter there is a trade-off between maximising the benefits of the mutations, and minimising the risk of ejections from optima.

Similarly to the ratio of the population size to the number of chromosomes, the ratio of

the number of chromosomes to the number of expected fitness landscapes is important. A working principle of the pGA is that memories are stored in successive separate chromosomes as they are acquired. (Although it would have been possible in principle to make them share chromosomes – cf. Section 2.1 – this would have needlessly complicated matters, also reducing the efficacy and applicability of the algorithm.) An obvious idea is that if the number of landscapes is known, the number of chromosomes should be set to that, for economy. The ACh sequences in Figure 4.7 and in the bottom plot of Figure 4.13 support that idea, but the flawed sequence in the top plot of Figure 4.13 – and the sequences in the other figures, especially Figure 4.20 – show that in fact, the issues of how many chromosome to have, and of the correspondence between chromosomes and memories, are uncertain and problematic. The crux of the problem is the fact that the algorithm does not regulate in which chromosomes optima are discovered during genetic search. Two similar negative ramifications of that will now be discussed.

The first problem manifested in the run shown in the bottom plot of Figure 4.20. A cone was discovered in the first epoch and memorised in the normal way, but when the population entered the fourth landscape in the fourth epoch, a reactivated chromosome carrying the memory of the first landscape's cone underwent several generations of evolution before discovering a new cone there. In effect, this chromosome was recycled for the fourth landscape, the old memory being overwritten in the process. It was not the case that the two cones were highly proximate – allowing one memory to serve both landscapes – because when the first landscape returned in the next epoch, the chromosome dropped to minimal fitness.

The second problem appears not to have manifested in any of the runs, owing to the equally narrow widths of the cones and the high multidimensional distances between them – features designed to make the transitions between landscapes suitably difficult for the population. (Had they been easier, the distinction between rapid adaptation to a returning landscape due to ordinary genetic search, and rapid adaptation through memory recall, would have been obfuscated.) Fortunately, the problem can be adequately conveyed by a thought experiment using the bespoke fitness landscapes shown in Figure 4.35.

If a pGA population in the top landscape discovers the global optimum on the left, that optimum will be memorised when the epoch ends. But if the population enters the bottom

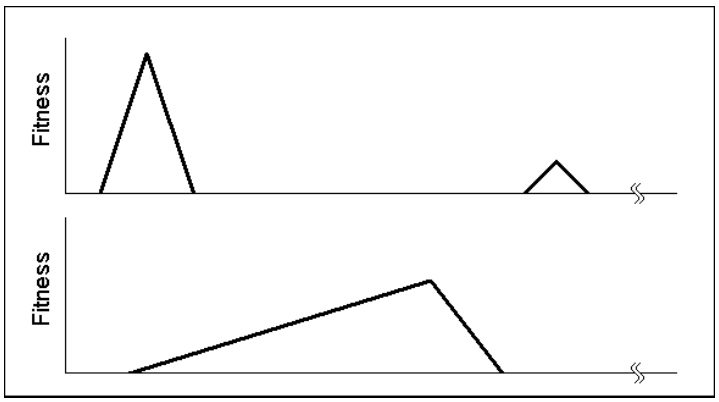

Figure 4.35: A pair of one-dimensional fitness landscapes, where the omitted regions represent the majority of the dimension, and contain no points of above-minimal fitness

landscape at a later time in the run, chromosomes carrying the memory from the top landscape will find themselves on the slope of that landscape's optimum. If the rest of the population has minimal fitness (being distributed along the region omitted from the figure) those chromosomes will climb the slope until they reach the summit, taking over the population. This means that the memory from the top landscape has been recycled in the bottom landscape, and because the new optimum has no overlap with the previous optimum, on their return to the top landscape, the individuals would have to rediscover the old optimum from scratch. Furthermore, because the recycled memory location is closer to the local optimum on the right than the global optimum on the left, the population would be more likely to discover the local optimum. The topographies of those landscapes thus induce a situation where a desirable memory is very likely to be lost, and the population transferred from a global to a local optimum.

To summarise the foregoing material, the unregulated and probabilistic manner by which the default pGA acquires and restores memories, allows memory-bearing chromosomes to be detrimentally recycled. Sometimes that recycling happens after a period of standard genetic searching, and at other times it happens rapidly due to overlapping optima. It should be noted, however, that the second type of recycling is not a problem in general, but only in certain cases. Indeed, it can sometimes be helpful – for example, if the cone in the bottom landscape of Figure 4.35 were horizontally reversed, it would induce a trapped population from the top landscape to relocate to the global optimum there.

It is pertinent to state at this point what would happen with the explicit memory GA in these recycling scenarios. The key fact is that when it inserts a memory back into the population, it retains it in the bank as well. Thus, no matter what evolutionary trajectory the reinserted individual follows, the original gene pattern will not be lost, and in the event of the reinserted individual discovering a new optimum, this would simply be added to the memory bank alongside the original.

4.4 Alternative Configurations

The versions of the pGA that have been tested and analysed are representatives of a potentially very large family. To have generated results for additional versions would probably have inflated the thesis without significantly altering the conclusions, so this work was not carried through. Instead, in this last section of Chapter 4, some of those variants are described and discussed, with a focus on addressing limitations that were identified in the experiments. The coding and testing of these algorithms is categorised as future work.

4.4.1 Sequential Pointer Mutation

In this pGA variant, the pointers are all initially set to 1, and instead of being mutated randomly thereafter, they are incremented by 1, modulo the number of chromosomes.

The idea here is that in environments of cycling landscapes, memories of the first landscape go into the first chromosomes, memories of the second landscape go into the second chromosomes, and so on. After every landscape has been visited, each genotype possesses a neatly ordered set of memories.

The first thing to say about this system is that it only works well if there is a closed set of landscapes that always cycle in the same order, and where the number of chromosomes is the same as – or at least an integer multiple of – the number of landscapes in the set. In any other environment, memory recollection would be compromised, and there would inevitably be a large amount of chromosome recycling.

And even if the environment is suitable and the necessary synchronicity between the point-

ers and the landscapes is attained, a single mistake by the landscape change detector could have fatal consequences – by moving the pointers and the landscapes out of phase, memory recollection would be rendered impossible, at least until the synchronicity be reattained or the optima all be rediscovered.

4.4.2 Environmental Pointer Control

Here the pointer values are not set randomly or on the basis of algorithmic variables, but on the basis of the current fitness landscape. The following two mechanisms are suggested:

1. Take a hypermutating pGA. At regular intervals, evaluate the fitnesses of *every chromosome* in one, some, or all of the genotypes in the population. If the genotype(s) contain any chromosome whose fitness is sufficiently high in the current landscape, manually mutate the pointer to activate that chromosome.

2. Obtain a means of identifying landscapes, and use this to control the pointers accordingly.

The outstanding advantage of the first mechanism would be an increased reliability of memory recollection; in the extreme case, if every chromosome in every genotype was evaluated in every generation, memory recollection would be absolutely guaranteed. The enhancement of chromosome restoration would also increase the number of instances of recycling, which can be useful as well as damaging.

Additionally, the chromosome check-ups would allow the hypermutation rate to be set higher – even to 100% – because situations where the population gets ejected from optima would be rectified in the next check-up.

It is predicted that this variant would be an improvement on the hypermutating pGA, in terms of performance. But this improvement would come at the cost associated with the extra fitness evaluations. The fitness function is generally the most time-consuming and expensive part of a GA, so whereas this variant could seem unfailingly superior to the ordinary hypermutating pGA in testing, in (hypothetical) real-world applications it could actually be excessively wasteful.

The second proposed mechanism could produce the perfect pGA: chromosomes would be perfectly matched to landscapes, and the risks of failing to acquire, retain, and recall memories would be reduced to zero. In fact, in terms of its workings, this algorithm would resemble an explicit memory GA to a high degree.

But there is a weakness here, namely the landscape identifier. To paraphrase what was said in Chapter 3 on this topic, landscape identification is a deeply challenging problem, and unless information is known in advance pertaining to the topographies that will be encountered, or accurate prediction is possible, infallible identification cannot be depended on.

It is probably best for landscape identifiers to be incorporated into other variants of the pGA, rather than being used alone in an aspiration to perfection.

4.4.3 An Independent pGA

Every previously considered version of the pGA depends on external input – from a random number generator, a landscape change detector, or a landscape identifier – for its pointer control. In this last subsection it will be speculated how a mechanism independent of external/explicit input could purposefully control the pointers. The pGA that is described here that possesses such an internalised/implicit mechanism, may be thought of as an 'ultra implicit memory' GA.

To begin with, the source of the pointer control must be chosen. It has to come from within the population, so the candidates are: the pointer distribution, the composition of the AChs, the composition of the PChs, and the composition of the genotype as a whole. Those compositions can be difficult to measure and also difficult to interpret, whereas the pointer distribution is very simple, so this stands out as the candidate to choose.

The pointer distribution can serve as an indicator of ACh convergence, because while the population is still diverse (across the AChs) and still searching for optima, the distribution is varied, but when an individual discovers an optimum and dominates the population, the distribution becomes uniform. It would therefore be possible to use the statistical variance of the pointer distribution as a numerical indicator of ACh convergence, with the rising of the variance above some threshold signifying the convergence. (It should be remarked that ACh

convergence is possible without pointer convergence, and vice versa.)

It was shown in Section 4.3 how triggered hypermutation of the pointers is the better way to control them, so combining that fact with the aforementioned aspect of the pointer distribution, leads to the idea that in the present pGA variant, hypermutation be triggered by pointer convergence. In every other regard, the algorithm would be handled like the hypermutating pGA.

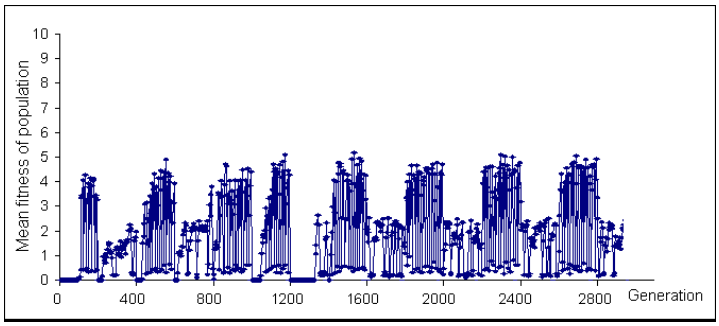

Figure 4.36: A sample performance of the proposed independent pGA with two 30-gene chromosomes for two landscapes, with crossover disabled

To glimpse the behaviour of this algorithm, and as a proof of concept for it, the pGA C-program was modified to support it. Figure 4.36 offers an example performance that can be compared to corresponding results from Chapters 3 and 4. The only important difference in parameters is that here, crossover was disabled. This was because preliminary runs with crossover enabled produced results comparable to those for random pointer control (see Figure 4.2).

The first thing to note in Figure 4.36 is the rapid and extreme oscillations in mean fitness over time, in the form of highly compressed saw-tooth waves. What was happening was that having discovered a cone, the population was repeatedly having to re-converge on it, because when each convergence was attained, it provoked a pointer hypermutation that dispersed it.

It can be discerned that the plot came to resemble the bottom plot of Figure 4.8, in that the population was successfully alternating between two cones. The independent pGA population achieved this thanks to the small minority of un-pointer-mutated individuals that was left after each hypermutation, who were able to re-dominate. And when the landscape changed, the last hypermutation caused passive convergence as in the regular hypermutational pGA.

The independent pGA, with its claim to ultra implicit memory behaviour, may be able to achieve performance levels somewhere between those of the constant PMR pGA and those of the hypermutating pGA. It is an interesting algorithm that would probably exhibit several interesting behaviours, but its peculiar method of pointer control, its unstable population dynamics, and its problem with crossover, detract from its useful applicability.

However, there is a further aspect of the independent pGA that could improve both its own performances and those of other, more straightforward pGA implementations. During a period of search when the individuals were of similarly low fitness, the pointer distribution would vary, and it is very likely that (near) random walks would gradually distort the pointer distribution and eventually cause them to converge. Such 'random pointer convergence' would reactivate passive chromosomes, so if the population had unknowingly re-entered a landscape for which it had memories, they would now be recalled.

The thesis suggests that the strategy of hypermutating the pointers when they converge *randomly* could be used in conjunction with a landscape change detector to improve the likelihood of successful memory recollection by the pGA.

4.5 Summary

This chapter introduced, defined, tested, and analysed a novel implicit memory algorithm called the pointer genetic algorithm. This was done in response to the failings of the other implicit memory algorithms, which had been shown to be deficient in Chapter 3. The pGA has been demonstrated to outperform the other implicit memory GAs and achieve successful memory behaviour, in various problem environments. But at the same time, it was not as effective as an explicit memory GA.

The exposition of the pGA and of its behaviours are among the main contributions to knowledge of the thesis. It is hoped that the domain of implicit memory may now be better understood.

The ramifications of this and the previous chapters' findings are discussed in the next chapter, the conclusion of the thesis.

Chapter 5

Conclusion

This thesis investigated the application of genetic algorithms in a specific class of nonstationary problem environments, namely those where the environmental change is epochal and where fitness landscapes appear that are identical to or at least resemble previous landscapes. The research aim was to discover and report how GAs can exploit familiarity of landscapes, and to appraise the techniques involved. Before summarising the thesis as a whole and drawing the conclusions, the original contributions will be listed. The memory contributions constitute the main claims of the thesis.

- Clarification of the concept of genetic redundancy
- Synthesis of disparate research projects under the umbrella theme of the thesis (including material on chromosome overlap and the measurement of convergence)
- Demonstration of the ineffectiveness of previous implicit memory algorithms (the sGA and the polyploid GAs)
- Introduction of a new implicit memory algorithm (the pGA)
- Demonstration of the relative effectiveness of the pGA, including the concept of passive convergence.
- Constructive criticism of the concept of implicit memory in general

It was necessary to specify an exact meaning of the term *genetic redundancy* because some previous definitions were either too biology-centric, or were inaccurate thanks to being based on the relative sizes of the genotypes and phenotype spaces. The definition supplied by the present thesis avoids situations where representations containing redundancy – and associated phenomena – could fail to be classified as such.

In the literature, several pertinent works were found. From theoretical biology came some GAs designed to investigate the emergence of modularity in evolution. Parter et al. (2008) found that putting a population in an oscillating two-landscape environment where the target functions were modularly related, induced the evolution of modular phenotypes. The evolved individuals were able to rearrange their modules by mutation to rapidly adapt to new landscapes with target functions formable from those same modules.

And there is an assortment of GAs – mostly written to evolve strategies for games – whose individuals have the potential to improve their adaptation to returning landscapes because of how the phenotypes are developed. In these representations (e.g. Axelrod (1987)) there is genetic redundancy, and it is the developmental process that chooses which genes are used. The ramification is that a given gene can make a positive fitness contribution in one landscape, lie dormant during subsequent epochs, and then, when the original landscape returns, contribute once again to its genotype's fitness.

The most numerous and the most important GAs that fall within the remit of the thesis are those that (claim to) include memory capacity. These have been categorised as explicit – where memories are stored externally (in memory banks or in separate populations) – or implicit – where memories are stored in the genotypes. The work that has already been done with explicit memory, including that of the present thesis, demonstrates that straightforward external memory handling significantly improves the performances of GAs in the type of environment under consideration.

It is in the domain of implicit memory that the thesis makes the majority of its contributions to knowledge. The memory claims made in the literature – one from the structured GA (Dasgupta & McGregor 1992*b*), the others from the polyploid GAs (e.g. Goldberg & Smith (1987)) – were doubted, so exemplifying algorithms were coded and tested along with a standard GA and an explicit memory GA. The observed mean fitnesses of the populations

throughout the runs, together with analyses of key genotypes, revealed that the claims of implicit memory were unfounded. The algorithms were incapable of acquiring or retaining memories of past optima, and despite offering enhanced genetic diversity, were hard to distinguish from the standard GA performance-wise.

A new implicit memory GA – called the pointer GA – was expounded in an attempt to find if effective memory behaviour can be attained within the constraints of implicit memory. By putting the pGA through the same testing as was used with the other algorithms, it was found that it can acquire, retain, and recall memories – the three vital attributes of an effective memory algorithm. Having said this, it did not perform as well as the explicit memory GA.

The pGA genotype contains multiple chromosomes, one of which maps to the phenotype at any given time, and it is the capacity to switch between chromosomes – by means of the pointer, a meta gene – that enables the pGA to acquire and recall memories.

A novel design feature of the pGA that enables it to retain memories is the non-application of the genetic operators across the passive chromosomes. Exposing memory-bearing genes to the operators – which happens in the other implicit memory GAs – causes them to be gradually destroyed, so it must not occur. The genetic operators are only applied to the active chromosomes in the pGA genotype, and it is in them that genetic search proceeds.

It is inferred from the findings of the thesis that an effective implicit memory algorithm must have an active/passive partition in its genotype, with the active section hosting the evolution, and the passive section hosting the memories. It is extremely difficult to imagine how genetic search and memory storage can both exist in the same genotype otherwise.

This problem resembles what Watson (2002) called the *inherent tension of innovation and reproductive fidelity*. His solution was to enable a higher level of organisation in the entity, with fidelity at the lower level and innovation at the higher level. In a sense, this is also what happens in the pGA, if the chromosomes are considered low-level units and the pointer is considered a high-level controller.

The other factor that contributes to the success of the pGA is what the thesis termed *passive convergence*, which is when a memory pattern in a passive chromosome spreads across the whole population. This is achieved by pacifying all or most copies of a converged active

chromosome, or alternatively via the known phenomenon of hitchhiking. It is a necessary occurrence because in its absence, a non-memory-bearing individual could later dominate the population, purging the memories.

In the light of the various solutions to the research question that have been elucidated, the thesis makes the following engineering claim. For the dynamic optimisation problems here considered, explicit memory approaches should be utilised (either in a pure form or augmented by diversity maintenance measures) and implicit memory approaches should be ignored. This message is based on the fact of the inferiority of implicit memory, which stems from the self-imposed constraint of having to internalise the memories. This unnecessary constraint introduces risk and uncertainty into the tasks of acquiring, retaining, and recalling memories – at the cost of increased algorithmic complexity and overheads.

The place where implicit memory has merit is where the idea for it was inspired in the first place, namely in biological genotypes. It should be remarked that the simplistic argument "because it works for nature, it can work for us" is fallacious; despite many successful instances of copying nature (artificial neural networks, Velcro, the GA itself), it is easy to imagine applications and technologies that would be foolish, such as cameras structured like eyeballs, or aeroplanes with flapping wings.

Regarding the non-memory algorithms that were reviewed in the thesis, the engineering claim is not to their detriment because they were not designed to be memory algorithms *per se,* but to attain other goals. For example, the modularity GAs were designed to model scientific theories, and the game-playing strategies were designed for continuously changing player-based environments.

5.1 Future Work

On the basis of the findings of the thesis, the most worthwhile future work from a utilitarian point of view would be in developing new and improved explicit memory algorithms. The material in Yang et al. (2007) could provide a starting point for that, likewise the material in Simões (2010).

The thesis did not delve into explicit memory very deeply, but the following comments are offered to algorithm developers based on what results there were. Firstly, there is one respect in which a pure memory approach is weaker than a more general, diversity-based approach – by reliably restoring memories, the algorithm can restore an undesirable situation, for example a population being trapped in a local optimum. The low fitnesses associated with the second landscape in the top plot of Figure 3.3 (page 40) provide an example of that. A hybrid of explicit memory and random immigration could offer an improvement.

Secondly, there is no guarantee that a reinserted memory will re-dominate the population, because other fit individuals might crowd it out. This was discussed in relation to Figure 4.4 (page 65) in the analysis of the pGA with constant PMR. Possible solutions to this unlikely problem include inserting multiple individuals, and/or giving the inserted individual(s) temporarily higher fitness.

Thirdly, as a general idea, in the event that only a small minority of the landscapes will be familiar, it might not be worth using a memory algorithm at all.

In the domain of implicit memory, an interesting future line of work would be to develop the pGA variants described in Section 4.4, and/or to devise new variants. The goal would be to design a pGA – or some other implicit memory algorithm – that can perform as well as a standard explicit memory GA.

Bibliography

Acan, A. & Tekol, Y. (2003), 'Chromosome Reuse in Genetic Algorithms', *Genetic and Evolutionary Computation Conference* pp. 695–705.

Angeline, P. J. (1997), 'Tracking Extrema in Dynamic Environments', *Proceedings of the 6th International Conference on Evolutionary Programming* .

Axelrod, R. (1987), 'The Evolution of Strategies in the Iterated Prisoner's Dilemma', *Genetic Algorithms and Simulated Annealing* pp. 32–41.

Bellas, F., Becerra, J. A. & Duro, R. J. (2009), 'Using Promoters and Functional Introns in Genetic Algorithms for Neuroevolutionary Learning in Nonstationary Problems', *Neurocomputing* **72**, 2134–2145.

Bendtsen, C. N. & Krink, T. (2002), 'Dynamic Memory Model for Non-Stationary Optimization', *Proceedings of the Congress on Evolutionary Computation* .

Branke, J. (1999), 'Memory Enhanced Evolutionary Algorithms for Changing Optimization Problems', *Proceedings of the 1999 Congress on Evolutionary Computation* .

Branke, J. (2001), 'Evolutionary Approaches to Dynamic Optimization Problems – Updated Survey', *GECCO Workshop on Evolutionary Algorithms for Dynamic Optimization* .

Branke, J., Kauler, T., Schmidt, C. & Schmeck, H. (2000), 'A Multi-Population Approach to Dynamic Optimization Problems', *Adaptive Computing in Design and Manufacturing* pp. 299–308.

Burke, D. S., Jong, K. A. D., Grefenstette, J. J., Ramsey, C. L. & Wu, A. S. (1998), 'Putting More Genetics into Genetic Algorithms', *Evolutionary Computation* **6**, 387–410.

Cobb, H. G. (1990), An Investigation into the Use of Hypermutation as an Adaptive Operator in Genetic Algorithms Having Continuous, Time-Dependent Nonstationary Environments, Technical report, Navy Center for Applied Research in Artificial Intelligence.

Collard, P., Escazut, C. & Gaspar, A. (1996), 'An Evolutionary Approach for Time Dependent Optimization', *International Conference on Artificial Intelligence Tools* .

Collingwood, E., Corne, D. & Ross, P. (1996), 'Useful Diversity via Multiploidy', *Proceedings of the IEEE International Conference on Evolutionary Computing* pp. 810–813.

Darwen, P. J. & Yao, X. (1995), *On Evolving Robust Strategies for Iterated Prisoner's Dilemma*, Progress in Evolutionary Computation, Vol. 956, Springer, pp. 276–292.

Dasgupta, D. (1994), Optimisation in Time-Varying environments using Structured Genetic Algorithms, Technical report, University of Strathclyde.

Dasgupta, D. & McGregor, D. R. (1992*a*), 'Designing Application-Specific Neural Networks using the Structured Genetic Algorithm', *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks* .

Dasgupta, D. & McGregor, D. R. (1992*b*), 'Nonstationary Function Optimization using the Structured Genetic Algorithm', *Proceedings of Parallel Problem Solving from Nature* pp. 145–154.

Eggermont, J., Lenaerts, T., Poyhonen, S. & Termier, A. (2001), 'Raising the Dead; Extending Evolutionary Algorithms with a Case-Based Memory', *Proceedings of the Fourth European Conference on Genetic Programming* .

Fonteix, C., Bicking, F., Perrin, E. & Marc, I. (1995), 'Haploid and Diploid Algorithms, a New Approach for Global Optimization: Compared Performances', *International Journal of Systems Science* **26**, 1919–1933.

Gaspar, A. & Collard, P. (1997), 'Time Dependent Optimization with a Folding Genetic Algorithm', *9th International IEEE Conference on Tools with Artificial Intelligence* pp. 125–132.

Goldberg, D. E. & Smith, R. E. (1987), 'Nonstationary Function Optimization using Genetic Algorithms with Dominance and Diploidy', *2nd International Conference on Genetic Algorithms* pp. 59–68.

Grefenstette, J. J. (1992), 'Genetic Algorithms for Changing Environments', *Parallel Problem Solving from Nature 2* .

Hillis, W. D. (1990), 'Co-Evolving Parasites Improve Simulated Evolution as an Optimization Procedure', *Physica D* **42**, 228–234.

Hinterding, R. (1997), 'Self-Adaptation using Multi-Chromosomes', *Proceedings of the 4th IEEE International Conference on Evolutionary Computation* pp. 87–91.

Holland, J. H. (1975), *Adaptation in Natural and Artificial Systems*, University of Michigan Press.

Hollstien, R. B. (1971), Artificial Genetic Adaptation in Computer Control Systems, PhD thesis, University of Michigan.

Ishibuchi, H. & Namikawa, N. (2005), 'Evolution of Iterated Prisoner's Dilemma Game Strategies in Structured Demes under Random Pairing in Game Playing', *IEEE Transactions on Evolutionary Computation* **9**, 552–561.

Jakobi, N. (1996), 'Encoding Scheme Issues for Open-Ended Artificial Evolution', *PPSN* pp. 52–61.

Karakoc, M., Soke, A. & Kavak, A. (2007), 'Using Diploidy Genetic Algorithm for Dynamic OVSF Code Allocation in WCDMA Networks', *IEEE Radio and Wireless Symposium* pp. 15–18.

Kashtan, N. & Alon, U. (2005), 'Spontaneous Evolution of Modularity and Network Motifs', *PNAS* **102**.

Kimura, M. (1983), *The Neutral Theory of Molecular Evolution*, Cambridge University Press.

Kirschner, M. & Gerhart, J. C. (2005), *The Plausibility of Life*, Yale University Press.

Klinkmeijer, L. Z., de Jong, E. & Wiering, M. (2006), 'A Serial Population Genetic Algorithm for Dynamic Optimization Problems', *Proceedings of the 15th Belgian-Dutch Conference on Machine Learning* pp. 41–48.

Kominami, M. & Hamagami, T. (2007), 'A New Genetic Algorithm with Diploid Chromosomes by Using Probability Decoding for Non-Stationary Function Optimization', *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics* pp. 1268–1273.

Lewis, J., Hart, E. & Ritchie, G. (1998), 'A Comparison of Dominance Mechanisms and Simple Mutation on Non-Stationary Problems', *Parallel Problem Solving from Nature 5* .

Louis, S. J. & Xu, Z. (1996), 'Genetic Algorithms for Open Shop Scheduling and Re-scheduling', *ISCA Eleventh International Conference on Computers and their Applications* pp. 99–102.

Massebeuf, S., Fonteix, C. & Kiss, L. N. (1999), 'Multicriteria Optimization and Decision Engineering of an Extrusion Process Aided by a Diploid Genetic Algorithm', *Proceedings of the Congress on Evolutionary Computation* pp. 14–21.

Mayer, H. A. & Spitzlinger, M. (2003), 'Multi-Chromosomal Representations and Chromosome Shuffling in Evolutionary Algorithms', *Proceedings of the 2003 Congress on Evolutionary Computation* .

Miorandi, D. & Yamamoto, L. (2008), 'Evolutionary and Embryogenic Approaches to Autonomic Systems', *Interdisciplinary Systems Approach in Performance Evaluation and Design of Computer and Communication Systems* .

Morris, R. & Watson, T. (2008), 'Evolving Strategies for the Game Footsteps', *Proceedings of the 8th UK Workshop on Computational Intelligence* pp. 65–70.

Morrison, R. W. & de Jong, K. A. (1999), 'A Test Problem Generator for Non-Stationary Environments', *IEEE* pp. 2047–2053.

Muller, H. J. (1964), 'The Relation of Recombination to Mutational Advance', *Mutation Research* **1**, 2–9.

Nasaroui, O., Dasgupta, D. & Pavuluri, M. (2002), 'ssGA: a Soft Structured Genetic Algorithm, and its Application in Web Mining', *Proceedings of the Annual Meeting of the North American Fuzzy Information Processing Society* pp. 87–92.

Ng, K. P. & Wong, K. C. (1995), 'A New Diploid Scheme and Dominance Change Mechanism for Non-Stationary Function Optimisation', *Proceedings of the 6th International Conference on Genetic Algorithms* .

Nowak, M. A., Boerlijst, M. C., Cooke, J. & Maynard Smith, J. (1997), 'Evolution of Genetic Redundancy', *Nature, Vol 388* pp. 167–171.

Parter, M., Kashtan, N. & Alon, U. (2008), 'Facilitated Variation: How Evolution Learns from Past Environments to Generalize to New Environments', *PLoS Computational Biology* **4**.

Ramsey, C. L. & Grefenstette, J. J. (1993), 'Case-Based Initialization of Genetic Algorithms', *Fifth International Conference on Genetic Algorithms* pp. 84–91.

Richter, H. (2009), 'Detecting Change in Dynamic Fitness Landscapes', *IEEE Congress on Evolutionary Computation* pp. 1613–1620.

Rothlauf, F. & Goldberg, D. E. (2003), 'Redundant Representations in Evolutionary Computation', *Evolutionary Computation* pp. 381–415.

Ryan, C. (1996), 'The Degree of Oneness', *Proceedings of the ECAI Workshop on Genetic Algorithms* .

Ryan, C. (1997), 'Diploidy without Dominance', *Proceedings of the Third Nordic Workshop on Genetic Algorithms* .

Saito, T. & Hamagami, T. (2010), 'Adaptive Adjustment of Weight Parameters for Diploid Genetic Algorithm with a Network Structure', *WRI Global Congress on Intelligent Systems* **1**, 249–253.

Shackleton, M., Shipman, R. & Ebner, M. (2000), 'An Investigation of Redundant Genotype-Phenotype Mappings and their Role in Evolutionary Search', *IEEE Transactions on Evolutionary Computation* pp. 493–500.

Simões, A. B. (2010), Improving Memory-based Evolutionary Algorithms for Dynamic Environments, PhD thesis, University of Coimbra.

Trojanowski, K. & Michalewicz, Z. (1999), 'Evolutionary Algorithms for Non-Stationary Environments', *Intelligent Information Systems VIII* .

Watson, R. A. (2002), Compositional Evolution: Interdisciplinary Investigations in Evolvability, Modularity, and Symbiosis, PhD thesis, Brandeis University, M.A.

West Eberhard, M. (2003), *Developmental Plasticity and Evolution*, Oxford University Press.

Worgan, S. & Mills, R. (2008), 'Initial Modelling of the Alternative Phenotypes Hypothesis', *Proceedings of the 11th International Conference on Artificial Life* pp. 717–724.

Wu, A. S. & Lindsay, R. K. (1996), 'A Comparison of the Fixed and Floating Building Block Representations in the Genetic Algorithm', *Evolutionary Computation* .

Wu, Y.-G., Ho, C.-Y. & Wang, D. Y. (2000), 'A Diploid Genetic Approach to Short-Term Scheduling of Hydro-Thermal System', *IEEE Transactions on Power Systems* **15**, 1268–1274.

Yang, S. (2003), 'Non-Stationary Problem Optimization Using the Primal-Dual Genetic Algorithm', *The 2003 Congress on Evolutionary Computation* pp. 2246–2253.

Yang, S. (2005), 'Memory-Based Immigrants for Genetic Algorithms in Dynamic Environments', *GECCO* pp. 1115–1122.

Yang, S. (2006), 'On the Design of Diploid Genetic Algorithms for Problem Optimization in Dynamic Environments', *IEEE Congress on Evolutionary Computation* pp. 1362–1369.

Yang, S., Ong, Y. S. & Jin, Y., eds (2007), *Evolutionary Computation in Dynamic and Uncertain Environments*, Studies in Computational Intelligence, Springer.

Yilmaz, A. S. & Wu, A. S. (2003), 'A Comparison of Haploidy and Diploidy without Dominance on Integer Representations', *Proceedings of the 17th International Symposium on Computer and Information Sciences* pp. 242–248.

Yu, H., Wu, A. S., Lin, K. C. & Schiavone, G. (2003), 'Adaptation of Length in a Nonstationary Environment', *Genetic and Evolutionary Computation Conference* .

```
// A GA with a case-based memory (from the paper "Raising the Dead")

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "C:\Dev-Cpp\PhD\MersenneTwister.h"

const char
func  = 'C', /* C = cones, K = knapsack */
cheat = 'Y'; /* N = detect heuristically, Y = cheat, S = become standard GA */

const int
SEED      =   77,
GENS      = 3000,
POPSIZE   =  200,
MEMORY    =   10, /* the size of the memory bank */
GT_LEN    =   30,
ENVIRON   =    1, /* 1+2 = no random landscapes, 1+3 = cyclic */
RND_NEXT  =   25, /* % */
PERIOD    =  200, /* for cyclic landscapes */
SCAPES    =    2,
P_CROSS   =   60, /* percentage */
P_MUT     = GT_LEN * 2, /* 1/n */
NUM_DIMS  =    5,
NUM_CONES =    5,
KNAPSACK  =  100,
DIM_GS = GT_LEN / NUM_DIMS;

const float
AX_MAX    = 25.0,
R         =  4.0, /* the bigger R, the narrower the cones */
CEIL_FIT  = 10.0,
MIN_FIT   =  0.1,
B2F_MAX   = powf(2.0, DIM_GS);

float cone[SCAPES + 1][NUM_CONES][NUM_DIMS]; // last one = random
int item[SCAPES + 1][GT_LEN][2];             //  "

FILE *f = fopen("data.txt", "w");

struct chrom
{
   int gene[GT_LEN];
   float fit;
}
   pop[POPSIZE + 1], /* last one = landscape tester */
   temp[POPSIZE],
   bank[MEMORY],
   last;

MTRand rand_num;

int rndi(int max)
{
   if(max > 0) return rand_num.randInt(max);
   else        return 0;
}
float rndf(float min, float max)
{
   return rand_num.rand(max - min) + min;
}

int new_gene()
{
   return rndi(1);
}

float cone_fitness(struct chrom ch, int scp)
{
   int d, c;
   float rms[NUM_CONES], b2f[NUM_DIMS];
```

```
   // Divide the genes between the problem dimensions
   for(d = 0, c = 0; d < NUM_DIMS; d++)
   {
      b2f[d] = 0.0;
      for(int g = 0; g < DIM_GS; g++)
         b2f[d] += ch.gene[g + c] * powf(2.0, g);
      c += DIM_GS;
   }
   // Now scale the phenotype values (the nonexistent "AX_MIN" = 0.0)
   for(d = 0; d < NUM_DIMS; d++)
      b2f[d] = (b2f[d] / B2F_MAX) * AX_MAX;

   for(c = 0; c < NUM_CONES; c++)
   {
      rms[c] = 0.0;
      for(d = 0; d < NUM_DIMS; d++)
         rms[c] += (b2f[d] - cone[scp][c][d]) * (b2f[d] - cone[scp][c][d]);
      rms[c] = sqrt(rms[c]);
      rms[c] = CEIL_FIT - (R * rms[c]);
      if(rms[c] < MIN_FIT) rms[c] = MIN_FIT;
   }
   for(c = 1; c < NUM_CONES; c++) /* Put the highest one last */
      if(rms[c] < rms[c - 1])
         rms[c] = rms[c - 1];

   return rms[NUM_CONES - 1];
}

float sack_fitness(struct chrom ch, int scape)
{
   int wgt = 0;
   float val = 0.0;

   // Each gene corresponds to an item for the knapsack
   for(int g = 0; g < GT_LEN; g++)
      if(ch.gene[g] > 0)
      {
         wgt += item[scape][g][0];
         val += (float)item[scape][g][1];
      }

   if(wgt > KNAPSACK) return MIN_FIT;
   else               return val;
}

void set_scape(int i)
{
   if(func == 'C')
   {
      // For cones across multiple dimensions
      for(int c = 0; c < NUM_CONES; c++)
         for(int d = 0; d < NUM_DIMS; d++)
            cone[i][c][d] = rndf(0.0, AX_MAX);
   }
   else if(func == 'K')
   {
      // For the 01-knapsack problem
      for(int g = 0; g < GT_LEN; g++)
      {
         item[i][g][0] = rndi(KNAPSACK / 4) + 1; // Weight
         item[i][g][1] = rndi(10);               // Value
      }
   }
}

main()
{
   int p, g, r, scape, scape1, x1, x2, top, lowest, btop = -1, upto = 0;
   float rb, mean_fit, top_fit, low_fit, detect, prev_mean_fit;
   bool detection;
```

```
rand_num.seed(SEED);

for(scape = 0; scape < SCAPES; scape++)
   set_scape(scape);

for(p = 0; p <= POPSIZE; p++)
   for(g = 0; g < GT_LEN; g++)
      pop[p].gene[g] = new_gene();

// Initialise the memory bank
for(p = 0; p < MEMORY; p++)
   for(g = 0; g < GT_LEN; g++)
   {
      bank[p].gene[g] = -1;
      bank[p].fit = 0;
   }

// MAIN LOOP

for(int gen = 1, scape = 0, scape1 = 0; gen <= GENS; gen++)
{
   //printf("Gen %d, landscape %d\n", gen, scape + 1);

   // If it's time to change the landscape, do so
   if(SCAPES > 1 && gen % PERIOD == 0)
   {
      switch(ENVIRON)
      {
         case 2: /* Non-cyclic, and no random ones */
            scape = (scape + rndi(SCAPES - 2) + 1) % SCAPES;
            break;
         case 3: /* Cyclic, with random ones */
            if(RND_NEXT > rndi(99)) {
               scape = SCAPES;
               set_scape(scape); }
            else {
               scape1 = (scape1 + 1) % SCAPES; scape = scape1; }
            break;
         case 4: /* Non-cyclic, with random ones */
            if(RND_NEXT > rndi(99)) {
               scape = SCAPES;
               set_scape(scape); }
            else scape = (scape + rndi(SCAPES - 2) + 1) % SCAPES;
            break;
         case 1:
         default: /* Cyclic, and no random ones */
            scape = (scape + 1) % SCAPES;
            break;
      }
   }

   // Go through the pop working out the fitnesses
   mean_fit = 0.0; top_fit = 0.0;
   if(     func == 'C') low_fit = CEIL_FIT + 1.0;
   else if(func == 'K') low_fit = 10.1 * (float)GT_LEN;
   for(p = 0; p < POPSIZE; p++)
   {
      if(     func == 'C') pop[p].fit = cone_fitness(pop[p], scape);
      else if(func == 'K') pop[p].fit = sack_fitness(pop[p], scape);
      mean_fit += pop[p].fit;
      if(pop[p].fit > top_fit)
      {
         top = p;
         top_fit = pop[p].fit;
      }
      if(pop[p].fit < low_fit)
      {
         lowest = p;
         low_fit = pop[p].fit;
      }
```

```
}
mean_fit /= (float)POPSIZE;

// Output
if(gen % 4 == 0) fprintf(f, "%.3f\n", mean_fit);

/* "top" is the ID of the (joint) fittest individual from that gen'n.
   If it's believed that the landscape changed, add "last" to the memory
   bank, and replace the (joint) least fit individual from the current
   generation with the fittest (in the current landscape) individual
   from the bank. */

detection = false;
if(cheat == 'N')
{
   if(func == 'C') pop[POPSIZE].fit=cone_fitness(pop[POPSIZE],scape);
   else if(func=='K') pop[POPSIZE].fit=sack_fitness(pop[POPSIZE],scape);
   if(gen == 1) detect = pop[POPSIZE].fit;

   if(mean_fit / prev_mean_fit < 0.75 || pop[POPSIZE].fit != detect)
   {
      printf("Landscape change detected, gen %d\n", gen);
      detection = true;
   }
}
else if(cheat == 'S')
{
   // In effect, a standard GA is running
}
else /* if cheat = 'Y' */
{
   if(gen % PERIOD == 0)
   {
      printf("Landscape changed, gen %d", gen);
      if(ENVIRON == 3 || ENVIRON == 4) printf(", scp %d\n", scape);
      else putchar('\n');
      detection = true;
   }
}

if(detection == true)
{
   for(top_fit = 0.0, p = 0; p < MEMORY; p++)
      if(bank[p].gene[0] != -1)
      {
         if(func == 'C') bank[p].fit = cone_fitness(bank[p], scape);
         else if(func=='K') bank[p].fit=sack_fitness(bank[p],scape);
         if(bank[p].fit > top_fit)
         {
            btop = p;
            top_fit = bank[p].fit;
         }
      }
   if(btop != -1)
      pop[lowest] = bank[btop]; // Bank's best replaces pop's worst

   // Check that "last" isn't already in the memory bank
   for(x1 = 0, p = 0; p < MEMORY; p++)
   {
      rb = 0.0;
      for(g = 0; g < GT_LEN; g++)
         rb += abs(last.gene[g] - bank[p].gene[g]);
      if(rb == 0.0) x1 = 1;
   }
   if(x1 == 0)
   {
      bank[upto] = last; // The old last-best goes into the bank
      upto = (upto + 1) % MEMORY;
   }
}
last = pop[top]; // The new last-best is held onto
```

```
      detect = pop[POPSIZE].fit; // Update for the next check
      prev_mean_fit = mean_fit; //    "

      // Build up the next generation by proportionate selection

      for(p = 1; p < POPSIZE; p++) pop[p].fit += pop[p - 1].fit;

      for(p = 0; p < POPSIZE; p++)
      {
         rb = rndf(0.0, pop[POPSIZE - 1].fit);
         r = 0;
         while(rb > pop[r].fit) r++; // Roll the ball round the roulette wheel
         temp[p] = pop[r];
         //if(gen % PERIOD == 0 && r == lowest) printf("(good one in)\n");
      }
      // Copy them back into the pop[] struct
      for(p = 0; p < POPSIZE; p++)
         pop[p] = temp[p];

      // Apply the Genetic Operators

      for(p = 0; p < POPSIZE - 1; p += 2) /* Two at a time for (2 pt) c/o */
         if(P_CROSS > rndi(99))
         {
            x1 = rndi((GT_LEN / 2) - 2) + 1;
            x2 = GT_LEN - 2 - rndi((GT_LEN / 2) - 2);
            for(int x = x1; x < x2; x++)
            {
               g = pop[p].gene[x];
               pop[p].gene[x] = pop[p + 1].gene[x];
               pop[p + 1].gene[x] = g;
            }
         }

      for(p = 0; p < POPSIZE; p++) /* One at a time for mutation */
         for(g = 0; g < GT_LEN; g++)
            if(rndi(P_MUT) == 0)
               pop[p].gene[g] = new_gene();

   }
   fclose(f);
   if(printf("Done\n")) getchar();
}
```

```
// The Pointer Genetic Algorithm (pGA), by Bobby Morris, 2007-2011

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "MersenneTwister.h"
// http://www-personal.engin.umich.edu/~wagnerr/MersenneTwister.html

const char
FUNC  = 'C', /* C = cones */
PM    = 'H', /* H = hyper-PM pGA, C = constant PMR, I = ptr-var based */
CHEAT = 'Y'; // Only applicable to the H-pGA

const int
SEED     =   77,
POPSIZE  =  200,
GENS     = 3000,
PERIOD   =  200,
CHR_LEN  =   30,
NUM_CHRS =    2,
SCAPES   =    2,
P_CROSS  =   60,
ENVIRON  =    1,
RND_NEXT =   25, /* % */
DIMS     =    5,
CONES    =    5,
DIM_GS = CHR_LEN / DIMS,
P_MUT = (int)(float(CHR_LEN) * 2); /* 1/n */

const float
CONST_PMR =  0.1, /* fraction of POPSIZE, for PM = 'C' */
PEAK_FIT  = 10.0,
MIN_FIT   =  0.1,
GRAD      =  4.0, /* Controls the gradient/narrowness of the cones */
AX_MAX    = 25.0, /* Axes range from 0.0 to this */
B2F_MAX   = powf(2.0, DIM_GS),
PTR_MEAN  = (float)POPSIZE / (float)NUM_CHRS; // in variance calculation

int Ptally[NUM_CHRS];
float Cone[SCAPES + 1][CONES][DIMS]; // Last = random option

MTRand rand_num;

struct Genotype
{
   int ptr, gene[NUM_CHRS][CHR_LEN];
   float fit;
}
   Pop[POPSIZE + 1]; // Last = detector

int rndi(int max)
{
   if(max > 0) return rand_num.randInt(max);
   else        return 0;
}
float rndf(float min, float max)
{
   return rand_num.rand(max - min) + min;
}

float cone_fitness(int p, int achr, int scp)
{
   int d, c;
   float rms[CONES], b2f[DIMS];

   // Divide the genes between the problem dimensions
   for(d = 0, c = 0; d < DIMS; d++)
   {
      b2f[d] = 0.0;
      for(int g = 0; g < DIM_GS; g++)
         b2f[d] += Pop[p].gene[achr][g + c] * powf(2.0, g);
```

```
      c += DIM_GS;
   }
   // Scale the phenotype values (the nonexistent "AX_MIN" = 0.0)
   for(d = 0; d < DIMS; d++)
      b2f[d] = (b2f[d] / B2F_MAX) * AX_MAX;

   for(c = 0; c < CONES; c++)
   {
      rms[c] = 0.0;
      for(d = 0; d < DIMS; d++)
         rms[c] += (b2f[d] - Cone[scp][c][d]) * (b2f[d] - Cone[scp][c][d]);
      rms[c] = sqrt(rms[c]);
      rms[c] = PEAK_FIT - (GRAD * rms[c]);
      if(rms[c] < MIN_FIT) rms[c] = MIN_FIT;
   }
   // Put the highest one last
   for(c = 1; c < CONES; c++)
      if(rms[c] < rms[c - 1])
         rms[c] = rms[c - 1];

   return rms[CONES - 1];
}
float sack_fitness(int p, int achr, int scape) { /* Absent */ }

float variance() /* of the pointer tallies */
{
   float w, s = 0.0;
   for(int t = 0; t < NUM_CHRS; t++)
   {
      w = (float)Ptally[t] - PTR_MEAN;
      s += powf(w, 2.0);
   }
   s /= (float)NUM_CHRS;
   return s;
}

void cross(int a, int b)
{
   bool co = true;
   Genotype swap = Pop[a];
   switch(co)
   {
      case true: // Uniform, with P(swap) = 0.5
         for(int i = 0; i < CHR_LEN; i++)
            if(rndi(1) == 0)
            {
               Pop[a].gene[Pop[a].ptr][i] = Pop[b].gene[Pop[b].ptr][i];
               Pop[b].gene[Pop[b].ptr][i] = swap.gene[swap.ptr][i];
            }
         break;
      case false: // 1-point
         if(CHR_LEN > 1)
            for(int i = 0; i < 1 + rndi(CHR_LEN - 2); i++)
            {
               Pop[a].gene[Pop[a].ptr][i] = Pop[b].gene[Pop[b].ptr][i];
               Pop[b].gene[Pop[b].ptr][i] = swap.gene[swap.ptr][i];
            }
         break;
   }
}

void set_scape(int s)
{
   if(FUNC == 'K') { /* 01-knapsack problem */ }
   else
   {
      // Cones across multiple dimensions
      for(int c = 0; c < CONES; c++)
         for(int d = 0; d < DIMS; d++)
            Cone[s][c][d] = rndf(0.0, AX_MAX);
   }
```

```
}

/*****************************************************************************/

main()
{
   int i, j, p, r, top_id, top_ac = -1, scape, scape2;
   float rb, max_var, old_mean_fit, new_mean_fit = 0.0, top_fit, detect;
   Genotype temp[POPSIZE];

   FILE
   *f1 = fopen("meanfits.txt", "w"),
   *f2 = fopen("topachr.txt", "w"),
   *f3 = fopen("topgts.txt", "w"),
   *f4 = fopen("ptrvar.txt", "w");

   // Initialise

   rand_num.seed(SEED);

   for(scape = 0; scape < SCAPES; scape++)
      set_scape(scape);

   for(p = 0; p < POPSIZE + 1; p++)
   {
      Pop[p].ptr = rndi(NUM_CHRS - 1);
      for(i = 0; i < NUM_CHRS; i++)
         for(j = 0; j < CHR_LEN; j++)
            Pop[p].gene[i][j] = rndi(1);
      Pop[p].fit = 0.0;
   }

   for(i = 1; i < NUM_CHRS; i++) Ptally[i] = 0;
   Ptally[0] = POPSIZE;
   max_var = variance(); // The maximum possible pointer variance
   if(ENVIRON == 3) scape2 = 0;

 /*****************************************************************************
 ****************************  MAIN LOOP ******************************************
 *****************************************************************************/

   for(int gen = 1, scape = 0; gen <= GENS; gen++)
   {
      if(gen % 100 == 0)
         printf("Gen %d, sc %d, ach %d\n", gen, scape + 1, top_ac + 1);

      // If it's time to change the landscape, do so.
      if(SCAPES > 1 && gen % PERIOD == 0)
      {
         switch(ENVIRON)
         {
            case 2: // Non-cyclic, and no random ones
               scape = (scape + rndi(SCAPES - 2) + 1) % SCAPES;
               break;
            case 3: // Cyclic, with random ones
               if(RND_NEXT > rndi(99))
               {
                  scape = SCAPES;
                  set_scape(scape);
               }
               else
               {
                  scape2 = (scape2 + 1) % SCAPES;
                  scape = scape2;
               }
               break;
            case 4: // Non-cyclic, with random ones
               if(RND_NEXT > rndi(99))
               {
                  scape = SCAPES;
                  set_scape(scape);
```

```
         }
         else
            scape = (scape + rndi(SCAPES - 2) + 1) % SCAPES;
         break;
      case 1:
      default: // Cyclic, and no random ones
         scape = (scape + 1) % SCAPES;
         break;
   }
}

// Reset the variables
old_mean_fit = new_mean_fit;
new_mean_fit = 0.0;
top_fit = -1.0;
for(i = 0; i < NUM_CHRS; i++) Ptally[i] = 0;

// Now go through the population working out the fitnesses

for(p = 0; p < POPSIZE; p++)
{
   if(FUNC == 'C') Pop[p].fit = cone_fitness(p, Pop[p].ptr, scape);
   new_mean_fit += Pop[p].fit;
   Ptally[Pop[p].ptr]++;
   if(Pop[p].fit > top_fit) { top_id = p; top_fit = Pop[p].fit; }
}
new_mean_fit /= float(POPSIZE);
for(i = 0, j = 0; i < NUM_CHRS; i++)
   if(Ptally[i] > j) {
      top_ac = i;
      j = Ptally[i]; }

if(gen % 4 == 0) /* Output */
{
   fprintf(f1, "%.3f\n", new_mean_fit);
   //fprintf(f4, "%.1f\n", variance());
}
if(gen % (PERIOD / 2) == (PERIOD / 2) - 5)
   fprintf(f2, "%d\n", top_ac + 1);
if(gen % PERIOD == 150)
{
   fprintf(f3, "%d\n", gen);
   for(i = 0; i < NUM_CHRS; i++) {
      for(j = 0; j < CHR_LEN; j++)
         fprintf(f3, "%d", Pop[top_id].gene[i][j]);
      Pop[top_id].ptr == i ? fprintf(f3, " ACh\n") : fputc('\n', f3); }
   fputc('\n', f3);
}

// Build up the next generation by proportionate selection
for(p = 1; p < POPSIZE; p++) Pop[p].fit += Pop[p - 1].fit;
for(p = 0; p < POPSIZE; p++)
{
   rb = rndf(0.0, Pop[POPSIZE - 1].fit - MIN_FIT);
   i = 0;
   while(rb >= Pop[i].fit) i++; // Roll ball round the roulette wheel
   temp[p] = Pop[i];
}
// Copy them back into the Pop[] struct
for(p = 0, i = 0, j = 0; p < POPSIZE; p++)
   Pop[p] = temp[p];
// Prepare for any pointer-mutation
if(PM == 'I')
{
   // Look at the scaled pointer variance
   if(100.0 * (variance() / max_var) > 90.0) j = 90; // thresh = ?
   //if(j == 90) printf("Pointer convergence in gen %d\n", gen);
}
else if(PM == 'H')
{
   if(CHEAT == 'Y')
```

```
         {
            // Infallible detection (aka "cheating")
            if(gen % PERIOD == 0) j = 90;
         }
         else
         {
            // Look for a fitness drop
            if(FUNC=='C') Pop[POPSIZE].fit = cone_fitness(POPSIZE, 0, scape);
            if(gen == 1) detect = Pop[POPSIZE].fit;
            if((new_mean_fit/old_mean_fit<0.75) || (Pop[POPSIZE].fit!=detect))
            {
               printf("Change detected in gen %d\n", gen);
               j = 90;
            }
            detect = Pop[POPSIZE].fit;
         }
      }
      if(PM == 'C') i = int(CONST_PMR * (float)POPSIZE);
      else { for(p = 0; p < POPSIZE; p++) if(j > rndi(99)) i++; }

      /* Apply the Genetic Operators
         The above i is the number of individuals to be pointer mutated.
         Go through the first i pop members thus mutating them. */
      for(p = 0; p < i; p++)
         Pop[p].ptr = (Pop[p].ptr + rndi(NUM_CHRS - 2) + 1) % NUM_CHRS;

      /* Slots 0 to i-1 of Pop[] now hold pointer-mutants. The remainder have
         the standard genetic operators applied to them. */

      for(p = i; p < POPSIZE - 1; p += 2) /* Two at a time for crossover */
         if(P_CROSS > rndi(99))
            cross(p, p + 1);

      for(p = i; p < POPSIZE; p++) /* One at a time for mutation */
         for(j = 0; j < CHR_LEN; j++)
            if(rndi(P_MUT) == 0)
               Pop[p].gene[Pop[p].ptr][j] = (Pop[p].gene[Pop[p].ptr][j]+1)%2;
   }
   fclose(f1); fclose(f2); fclose(f3); fclose(f4);

   if(printf("\nPress enter to continue")) getchar(); // Hold the window open
}
```

// My implementation of the structured genetic algorithm (sGA) by D. Dasgupta

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "C:\Dev-Cpp\PhD\MersenneTwister.h"

const char
func  = 'C', // 'C' = cones, 'K' = knapsack
cheat = 'Y', // 'N' = do proper detection
hyper = 'Y'; // 'Y' = hypermutate top level

const int
SEED        =   77,
GENS        = 3000,
POPSIZE     =  200,
PERIOD      =  200,
SCAPES      =    2,
ENVIRON     =    1,
RND_NEXT    =   25,
TOP_LEV_GS  =   10,
TOP_GS_ON   =    5,
SUB_PER_TOP =    6, /* 6 or 12 in the runs */
NUM_DIMS    =    5,
NUM_CONES   =    5,
KNAPSACK    =  100,
P_CROSS     =   60, /* % */
CONCAT = TOP_GS_ON * SUB_PER_TOP,
DIM_GS = (int)(float(CONCAT) / (float)NUM_DIMS),
P_MUT_TOP = TOP_LEV_GS * 2, /* 1/n */
P_MUT_SUB = P_MUT_TOP * SUB_PER_TOP; /* 1/n */

const float
AX_MAX   = 25.0,
R        =  4.0, /* the bigger R, the narrower the cones */
CEIL_FIT = 10.0,
MIN_FIT  =  0.1,
B2F_MAX  = powf(2.0, DIM_GS);

float cone[SCAPES + 1][NUM_CONES][NUM_DIMS]; // Last = random (same below)
int item[SCAPES + 1][CONCAT][2]; // There are as many items as active subgenes

bool fortop[TOP_LEV_GS], pile[TOP_LEV_GS];

FILE
*f1 = fopen("data.txt", "w"),
*f2 = fopen("scapes.txt", "w"),
*f3 = fopen("gt.txt", "w");

struct chrom
{
   bool topgene[TOP_LEV_GS];
   int subgene[TOP_LEV_GS][SUB_PER_TOP];
   float fit;
}
   pop[POPSIZE + 1], tmp[POPSIZE];

MTRand rand_num;

int rndi(int max)
{
   if(max > 0) return rand_num.randInt(max);
   else        return 0;
}
float rndf(float min, float max)
{
   return rand_num.rand(max - min) + min;
}

void prep_fortop(int on)
{
```

```
   int i, r, x = TOP_LEV_GS - 1;

   if(on > TOP_LEV_GS) on = TOP_LEV_GS;
   else if(on < 0)     on = 0;

   for(i = 0; i < TOP_GS_ON;  i++) pile[i] = true;
   for(     ; i < TOP_LEV_GS; i++) pile[i] = false;

   for(i = 0; i < TOP_LEV_GS; i++)
   {
      r = rndi(x);
      fortop[i] = pile[r];
      for(int j = r; j < x; j++)
         pile[j] = pile[j + 1];
      x--;
   }
}
int new_gene() { return rndi(1); }

float cone_fitness(int p, int scp)
{
   int c, b, actv_gene[CONCAT];
   float rms[NUM_CONES], b2f[NUM_DIMS];

   // Get the active genes from the sublayer of the chromosome
   for(int t = 0, c = 0; t < TOP_LEV_GS; t++)
      if(pop[p].topgene[t] == true)
      {
         for(b = 0; b < SUB_PER_TOP; b++)
            actv_gene[b + c] = pop[p].subgene[t][b];
         c += SUB_PER_TOP;
      }
   /* The array "actv_gene" now contains a concat'n of the active subgenes
      It is to be divided between the problem dimensions */

   for(b = 0, c = 0; b < NUM_DIMS; b++)
   {
      b2f[b] = 0.0;
      for(int b2 = 0; b2 < DIM_GS; b2++)
         b2f[b] += actv_gene[b2 + c] * pow(2, b2); // powf ?
      c += DIM_GS;
   }
   // Now scale the phenotype values (the nonexistent "AX_MIN" = 0.0)
   for(b = 0; b < NUM_DIMS; b++)
      b2f[b] = (b2f[b] / B2F_MAX) * AX_MAX;

   for(c = 0; c < NUM_CONES; c++)
   {
      rms[c] = 0.0;
      for(b = 0; b < NUM_DIMS; b++)
         rms[c] += (b2f[b] - cone[scp][c][b]) * (b2f[b] - cone[scp][c][b]);
      rms[c] = sqrt(rms[c]);
      rms[c] = CEIL_FIT - (R * rms[c]);
      if(rms[c] < MIN_FIT) rms[c] = MIN_FIT;
   }
   for(c = 1; c < NUM_CONES; c++) /* Put the highest one last */
      if(rms[c] < rms[c - 1])
         rms[c] = rms[c - 1];

   return rms[NUM_CONES - 1];
}

float sack_fitness(int p, int scape)
{
   int c, b, actv_gene[CONCAT], wgt;
   float val;

   // Get the active genes from the sublayer of the chromosome
   for(int t = 0, c = 0; t < TOP_LEV_GS; t++)
      if(pop[p].topgene[t] == true)
      {
```

```
         for(b = 0; b < SUB_PER_TOP; b++)
            actv_gene[b + c] = pop[p].subgene[t][b];
         c += SUB_PER_TOP;
      }
   /* The array "actv_gene" now contains a concat'n of the active subgenes
      Each one corresponds to an item for the knapsack */

   for(wgt = 0, val = 0.0, b = 0; b < CONCAT; b++)
      if(actv_gene[b] > 0)
      {
         wgt += item[scape][b][0];
         val += (float)item[scape][b][1];
      }

   if(wgt > KNAPSACK) return MIN_FIT;
   else               return val;
}

void set_scape(int sc)
{
   if(func == 'C')
   {
      // For cones across multiple dimensions
      for(int c = 0; c < NUM_CONES; c++)
         for(int d = 0; d < NUM_DIMS; d++)
            cone[sc][c][d] = rndf(0.0, AX_MAX);
   }
   else if(func == 'K')
   {
      // For the 01-knapsack problem
      for(int k = 0; k < CONCAT; k++)
      {
         item[sc][k][0] = rndi(KNAPSACK / 4) + 1; // Weight
         item[sc][k][1] = rndi(10);               // Value
      }
   }
}

main()
{
   int p, t, b, r, scape, scape2, tx1, tx2;
   float rb, mean_fit, top_fit, prev_mean_fit = MIN_FIT, detect;
   bool swap;

   rand_num.seed(SEED);

   for(scape = 0; scape < SCAPES; scape++) set_scape(scape);

   for(p = 0; p < POPSIZE + 1; p++)
   {
      prep_fortop(TOP_GS_ON);
      for(t = 0; t < TOP_LEV_GS; t++)
      {
         pop[p].topgene[t] = fortop[t];
         for(b = 0; b < SUB_PER_TOP; b++)
            pop[p].subgene[t][b] = new_gene();
      }
   }

   // MAIN LOOP

   for(int gen = 1, scape = 0, scape2 = 0; gen <= GENS; gen++)
   {
      //printf("Gen %d, landscape %d\n", gen, scape + 1);

      // If it's time to change the landscape, do so.
      if(SCAPES > 1 && gen % PERIOD == 0)
         switch(ENVIRON) {
            case 2: /* Non-cyclic, and no random ones */
               scape = (scape + rndi(SCAPES - 2) + 1) % SCAPES;
               break;
```

```
      case 3: /* Cyclic, with random ones */
         if(RND_NEXT > rndi(99)) {
            scape = SCAPES;
            set_scape(scape); }
         else {
            scape2 = (scape2 + 1) % SCAPES; scape = scape2; }
         break;
      case 4: /* Non-cyclic, with random ones */
         if(RND_NEXT > rndi(99)) {
            scape = SCAPES;
            set_scape(scape); }
         else scape = (scape + rndi(SCAPES - 2) + 1) % SCAPES;
         break;
      case 1:
      default: /* Cyclic, and no random ones */
         scape = (scape + 1) % SCAPES;
         break; }

// Go through the pop working out the fitnesses
for(mean_fit = 0.0, top_fit = 0.0, p = 0; p < POPSIZE; p++)
{
   if(     func == 'C') pop[p].fit = cone_fitness(p, scape);
   else if(func == 'K') pop[p].fit = sack_fitness(p, scape);
   mean_fit += pop[p].fit;
   if(pop[p].fit > top_fit) { top_fit = pop[p].fit; r = p; }
}
mean_fit /= (float)POPSIZE;

// Output
if(gen % 4 == 0) fprintf(f1, "%.3f\n", mean_fit);
if(gen % PERIOD == 10) fprintf(f2, "%d ", scape);
if(gen == ?? || gen == ??) // Get genotypes for analysis
{
   for(t = 0; t < TOP_LEV_GS; t++)
   {
      fprintf(f3, "%d (", pop[r].topgene[t]);
      for(b = 0; b < SUB_PER_TOP; b++)
         fprintf(f3, "%d", pop[r].subgene[t][b]);
      fprintf(f3, ")\n");
   }
   fprintf(f3, "%.2f\n", pop[r].fit);
}

if(cheat == 'N')
{
   // Test for 'scape change using old:new fit. ratio AND detector ind
   if(     func == 'C') pop[POPSIZE].fit = cone_fitness(POPSIZE, scape);
   else if(func == 'K') pop[POPSIZE].fit = sack_fitness(POPSIZE, scape);
   if(gen == 1) detect = pop[POPSIZE].fit;
   if(mean_fit / prev_mean_fit < 0.75 || pop[POPSIZE].fit != detect)
   {
      printf("Change detected in gen %d\n", gen);
      if(hyper == 'Y')
         for(p = 0; p < POPSIZE; p++)
         {
            // Hypermutate the top-level genes (to help the algo!)
            prep_fortop(TOP_GS_ON);
            for(t=0; t < TOP_LEV_GS; t++) pop[p].topgene[t] = fortop[t];
         }
   }
   prev_mean_fit = mean_fit;
   detect = pop[POPSIZE].fit;
}
else /* if cheat == 'Y' */
{
   // Change "detection" is done infallibly by the program
   if(SCAPES > 1 && gen % PERIOD == 0)
   {
      printf("Change occurred in gen %d\n", gen);
      if(hyper == 'Y')
         for(p = 0; p < POPSIZE; p++) {
```

```
                prep_fortop(TOP_GS_ON);
                for(t=0; t < TOP_LEV_GS; t++) pop[p].topgene[t]=fortop[t]; }
         }
      }

      // Build up the next generation by proportionate selection
      for(p = 1; p < POPSIZE; p++) pop[p].fit += pop[p - 1].fit;

      for(p = 0; p < POPSIZE; p++)
      {
         rb = rndf(0.0, pop[POPSIZE - 1].fit);
         t = 0;
         while(rb > pop[t].fit) t++; // Roll the ball round the roulette wheel
         tmp[p] = pop[t];
      }
      // Copy them back into the pop[] struct
      for(p = 0; p < POPSIZE; p++)
         pop[p] = tmp[p];

      // Apply the Genetic Operators

      for(p = 0; p < POPSIZE - 1; p += 2) // 2 at a time for (2-pt) crossover
         if(P_CROSS > rndi(99))
         {
            tx1 = rndi((TOP_LEV_GS / 2) - 2) + 1;
            tx2 = TOP_LEV_GS - rndi((TOP_LEV_GS / 2) - 3) - 2;
            for(t = tx1; t < tx2; t++)
            {
               // NB, don't cross top-level genes coz of on-off restriction
               for(b = 0; b < SUB_PER_TOP; b++)
               {
                  pop[p].subgene[t][b] += pop[p + 1].subgene[t][b];
                  pop[p + 1].subgene[t][b]
                     = pop[p].subgene[t][b] - pop[p + 1].subgene[t][b];
                  pop[p].subgene[t][b] -= pop[p + 1].subgene[t][b];
               }
            }
         }

      for(p = 0; p < POPSIZE; p++) // One at a time for mutation
         for(t = 0; t < TOP_LEV_GS; t++)
         {
            if(rndi(P_MUT_TOP) == 0)
            {
               r = rndi(TOP_LEV_GS - 1);
               swap = pop[p].topgene[t];
               pop[p].topgene[t] = pop[p].topgene[r];
               pop[p].topgene[r] = swap;
            }
            for(b = 0; b < SUB_PER_TOP; b++)
               if(rndi(P_MUT_SUB) == 0)
                  pop[p].subgene[t][b] = (pop[p].subgene[t][b] + 1) % 2;
         }
   }
   fclose(f1);
   fclose(f2);
   fclose(f3);
   if(printf("Done\n")) getchar();
}
```

```
/* A Biallelic Dominance-and-Polyploidy GA

   For 2 genes per locus:
      Matrices 0-1 are the "standard" pair
      2-5, like 0-1, give an equal zero:one ratio
      6-13 give 3/4 zeroes or ones
      14-15 give the same thing every time
   For more than 2:
      Any dominant zeroes plus no dominant ones -> zero
       "     "      ones    "   "    "   zeroes -> one
      otherwise o = 0, i = 1 and the rounded mean is taken

   The dominance outputs are as follows (for rival alleles):
         0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
         ------------------------------------------------
   01    0  1  1  0  0  1  1  0  0  1  1  0  0  1  1  0
   0i    0  1  0  1  1  0  0  1  0  1  0  1  0  1  1  0
   o1    1  0  1  0  1  0  1  0  1  0  0  1  0  1  1  0
   oi    1  0  0  1  0  1  1  0  0  1  0  1  1  0  1  0 */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "C:\Dev-Cpp\PhD\MersenneTwister.h"

const char
func  = 'C', /* C = cones, K = knapsack */
cheat = 'Y', /* N = use heuristic landscape change detection method */
align = 'N'; // Y = (matrix = scape)

const int
SEED      =   77,
GENS      = 3000,
POPSIZE   =  200,
NUM_LOCI  =   30, /* GT length = 2 x this */
PER_LOCUS =    2, /* simple extraction for >2 */
MATRICES  =    2, /* between 1 and 16 (see top) */
ENVIRON   =    1,
RND_NEXT  =   25, /* %, for ENVIRON = 3 or 4 */
PERIOD    =  200,
SCAPES    =    2,
P_CROSS   =   60, /* percentage */
P_MUT     = NUM_LOCI * 4, /* 1/n */
NUM_DIMS  =    5,
NUM_CONES =    5,
KNAPSACK  =  100,
DIM_GS    = NUM_LOCI / NUM_DIMS;

const float
AX_MAX   = 25.0,
R        =  4.0, /* the bigger R, the narrower the cones */
CEIL_FIT = 10.0,
MIN_FIT  =  0.1,
B2F_MAX  = powf(2.0, DIM_GS);

float cone[SCAPES + 1][NUM_CONES][NUM_DIMS]; // last = random
int matrix, item[SCAPES + 1][NUM_LOCI][2];   //   "

FILE
*f = fopen("data.txt", "w"),
*f2 = fopen("gt.txt", "w"),
*f3 = fopen("mat.txt", "w");

struct chrom
{
   char gene[NUM_LOCI][PER_LOCUS];
   int extracted[NUM_LOCI];
   float fit;
}
   pop[POPSIZE + 1], tmp[POPSIZE];
```

```
MTRand rand_num;

int rndi(int max)
{
   if(max > 0) return rand_num.randInt(max);
   else        return 0;
}
float rndf(float min, float max)
{
   return rand_num.rand(max - min) + min;
}
char new_gene()
{
   char c;
   switch(rndi(3)) {
      case 1: c = 'o'; break;
      case 2: c = '1'; break;
      case 3: c = 'i'; break;
      case 0: default: c = '0'; }
   return c;
}

void extract(int p) /* For PER_LOCUS = 2 */
{
   char a, b;
   int e;

   for(int g = 0; g < NUM_LOCI; g++)
   {
      a = pop[p].gene[g][0];
      b = pop[p].gene[g][1];
      e = -1;

      if(a == b)
      {
         if(     a == '0' || a == 'o') e = 0;
         else if(a == '1' || a == 'i') e = 1;
      }
      else
      {
         if((a == '0' && b == 'o') || (a == 'o' && b == '0'))
         {
            e = 0;
         }
         else if((a == '0' && b == '1') || (a == '1' && b == '0'))
         {
            switch(matrix)
            {
               case 1:  e = 1; break; case 2:  e = 1; break;
               case 3:  e = 0; break; case 4:  e = 0; break;
               case 5:  e = 1; break; case 6:  e = 1; break;
               case 7:  e = 0; break; case 8:  e = 0; break;
               case 9:  e = 1; break; case 10: e = 1; break;
               case 11: e = 0; break; case 12: e = 0; break;
               case 13: e = 1; break; case 14: e = 1; break;
               case 15: e = 0; break; default: e = 0; // case 0
            }
         }
         else if((a == '0' && b == 'i') || (a == 'i' && b == '0'))
         {
            switch(matrix)
            {
               case 1:  e = 1; break; case 2:  e = 0; break;
               case 3:  e = 1; break; case 4:  e = 1; break;
               case 5:  e = 0; break; case 6:  e = 0; break;
               case 7:  e = 1; break; case 8:  e = 0; break;
               case 9:  e = 1; break; case 10: e = 0; break;
               case 11: e = 1; break; case 12: e = 0; break;
               case 13: e = 1; break; case 14: e = 1; break;
               case 15: e = 0; break; default: e = 0; // case 0
            }
```

```
         }
         else if((a == 'o' && b == '1') || (a == '1' && b == 'o'))
         {
            switch(matrix)
            {
               case 1:  e = 0; break; case 2:  e = 1; break;
               case 3:  e = 0; break; case 4:  e = 1; break;
               case 5:  e = 0; break; case 6:  e = 1; break;
               case 7:  e = 0; break; case 8:  e = 1; break;
               case 9:  e = 0; break; case 10: e = 0; break;
               case 11: e = 1; break; case 12: e = 0; break;
               case 13: e = 1; break; case 14: e = 1; break;
               case 15: e = 0; break; default: e = 1; // case 0
            }
         }
         else if((a == 'o' && b == 'i') || (a == 'i' && b == 'o'))
         {
            switch(matrix)
            {
               case 1:  e = 0; break; case 2:  e = 0; break;
               case 3:  e = 1; break; case 4:  e = 0; break;
               case 5:  e = 1; break; case 6:  e = 1; break;
               case 7:  e = 0; break; case 8:  e = 0; break;
               case 9:  e = 1; break; case 10: e = 0; break;
               case 11: e = 1; break; case 12: e = 1; break;
               case 13: e = 0; break; case 14: e = 1; break;
               case 15: e = 0; break; default: e = 1; // case 0
            }
         }
         else if((a == '1' && b == 'i') || (a == 'i' && b == '1'))
         {
            e = 1;
         }
      }
      pop[p].extracted[g] = e;
   }
}
void extract3(int p) /* For PER_LOCUS > 2 */
{
   char *a = new char[PER_LOCUS];
   int e, dom0, dom1;
   float s;

   for(int g = 0; g < NUM_LOCI; g++)
   {
      dom0 = 0; dom1 = 0; e = -1;
      for(int l = 0; l < PER_LOCUS; l++)
      {
         a[l] = pop[p].gene[g][l];
         if(a[l] == '0') dom0++;
         else if(a[l] == '1') dom1++;
      }

      if(dom0 > 0 && dom1 == 0)
         e = 0;
      else if(dom1 > 0 && dom0 == 0)
         e = 1;
      else /* take the mean (o = 0, i = 1) and round off */
      {
         s = 0.0;
         for(int l = 0; l < PER_LOCUS; l++)
         {
            if(a[l] == '1' || a[l] == 'i') s += 1.0;
         }
         s /= (float)PER_LOCUS;
         s < 0.5 ? e = 0: e = 1;
      }
      pop[p].extracted[g] = e;
   }
}
```

```
float cone_fitness(int p, int scp)
{
   int d, c;
   float rms[NUM_CONES], b2f[NUM_DIMS];

   // Get the extracted genotype
   PER_LOCUS == 2 ? extract(p): extract3(p);

   // Divide the extracted genes between the problem dimensions
   for(d = 0, c = 0; d < NUM_DIMS; d++)
   {
      b2f[d] = 0.0;
      for(int g = 0; g < DIM_GS; g++)
         b2f[d] += pop[p].extracted[g + c] * powf(2.0, g);
      c += DIM_GS;
   }
   // Now scale the phenotype values (the nonexistent "AX_MIN" = 0.0)
   for(d = 0; d < NUM_DIMS; d++)
      b2f[d] = (b2f[d] / B2F_MAX) * AX_MAX;

   for(c = 0; c < NUM_CONES; c++)
   {
      rms[c] = 0.0;
      for(d = 0; d < NUM_DIMS; d++)
         rms[c] += (b2f[d] - cone[scp][c][d]) * (b2f[d] - cone[scp][c][d]);
      rms[c] = sqrt(rms[c]);
      rms[c] = CEIL_FIT - (R * rms[c]);
      if(rms[c] < MIN_FIT) rms[c] = MIN_FIT;
   }
   for(c = 1; c < NUM_CONES; c++) /* Put the highest one last */
      if(rms[c] < rms[c - 1])
         rms[c] = rms[c - 1];

   return rms[NUM_CONES - 1];
}

float sack_fitness(int p, int scape)
{
   int wgt = 0;
   float val = 0.0;

   // Get the extracted genotype
   PER_LOCUS == 2 ? extract(p): extract3(p);

   // Each extracted gene corresponds to an item for the knapsack
   for(int g = 0; g < NUM_LOCI; g++)
      if(pop[p].extracted[g] > 0)
      {
         wgt += item[scape][g][0];
         val += (float)item[scape][g][1];
      }

   if(wgt > KNAPSACK) return MIN_FIT;
   else               return val;
}

void set_scape(int sc)
{
   if(func == 'C')
   {
      // For cones across multiple dimensions
      for(int c = 0; c < NUM_CONES; c++)
         for(int d = 0; d < NUM_DIMS; d++)
            cone[sc][c][d] = rndf(0.0, AX_MAX);
   }
   else if(func == 'K')
   {
      // For the 01-knapsack problem
      for(int k = 0; k < NUM_LOCI; k++)
      {
         item[sc][k][0] = rndi(KNAPSACK / 4) + 1; // Weight
```

```
         item[sc][k][1] = rndi(10);               // Value
      }
   }
}

main()
{
   int p, t, b, r, scape, scape2, tx1, tx2;
   float rb, mean_fit, prev_mean_fit = MIN_FIT, top_fit, detect;
   char swap;
   rand_num.seed(SEED);

   for(scape = 0; scape < SCAPES; scape++) set_scape(scape);

   for(p = 0; p < POPSIZE + 1; p++)
      for(b = 0; b < NUM_LOCI; b++)
         for(t = 0; t < PER_LOCUS; t++)
            pop[p].gene[b][t] = new_gene();

   // MAIN LOOP

   matrix = 0;
   for(int gen = 1, scape = 0, scape2 = 0; gen <= GENS; gen++)
   {
      if(gen % 100 == 0) printf("Gen %d, landscape %d\n", gen, scape + 1);

      // If it's time to change the landscape, do so
      if(SCAPES > 1 && gen % PERIOD == 0)
         switch(ENVIRON) {
            case 2: /* Non-cyclic, and no random ones */
               scape = (scape + rndi(SCAPES - 2) + 1) % SCAPES;
               break;
            case 3: /* Cyclic, with random ones */
               if(RND_NEXT > rndi(99)) {
                  scape = SCAPES;
                  set_scape(scape); }
               else {
                  scape2 = (scape2 + 1) % SCAPES; scape = scape2; }
               break;
            case 4: /* Non-cyclic, with random ones */
               if(RND_NEXT > rndi(99)) {
                  scape = SCAPES;
                  set_scape(scape); }
               else scape = (scape + rndi(SCAPES - 2) + 1) % SCAPES;
               break;
            case 1:
            default: /* Cyclic, and no random ones */
               scape = (scape + 1) % SCAPES;
               break; }

      // Go through the pop working out the fitnesses
      for(mean_fit = 0.0, top_fit = -1.0, p = 0; p < POPSIZE; p++)
      {
         if(     func == 'C') pop[p].fit = cone_fitness(p, scape);
         else if(func == 'K') pop[p].fit = sack_fitness(p, scape);
         mean_fit += pop[p].fit;
         if(pop[p].fit > top_fit) { r = p; top_fit = pop[r].fit; }
      }
      mean_fit /= (float)POPSIZE;

      // Output
      if(gen % 4 == 0) fprintf(f, "%.3f\n", mean_fit);
      if(gen % PERIOD == PERIOD - 5 || gen % PERIOD == 5)
      {  /* Output top genotypes for analysis */
         fprintf(f2, "sc%d m%d (%.2f) ", scape + 1, matrix + 1, pop[r].fit);
         for(b = 0; b < NUM_LOCI; b++) {
            for(t=0; t < PER_LOCUS; t++) fprintf(f2, "%c", pop[r].gene[b][t]);
            fputc('-', f2); }
         PER_LOCUS == 2 ? extract(r): extract3(r); fprintf(f2, "  ");
         for(b = 0; b < NUM_LOCI; b++) fprintf(f2, "%d", pop[r].extracted[b]);
         fputc('\n', f2);
```

```
      }

      if(cheat == 'N')
      {
         // Test for 'scape change using old:new fit. ratio AND detector ind
         if(     func == 'C') pop[POPSIZE].fit = cone_fitness(POPSIZE, scape);
         else if(func == 'K') pop[POPSIZE].fit = sack_fitness(POPSIZE, scape);
         if(gen == 1) detect = pop[POPSIZE].fit;
         if(mean_fit / prev_mean_fit < 0.75 || pop[POPSIZE].fit != detect)
         {
            printf("Change detected in gen %d\n", gen);
            // Change the dominance matrix
            if(align == 'Y' && MATRICES > SCAPES) matrix = scape;
            else matrix = (matrix + rndi(MATRICES - 2) + 1) % MATRICES;
         }
         prev_mean_fit = mean_fit;
         detect = pop[POPSIZE].fit;
      }
      else
      {
         // Change "detection" is done infallibly by the program
         if(SCAPES > 1 && gen % PERIOD == 0)
         {
            printf("Change occurred in gen %d\n", gen);
            if(align == 'Y' && MATRICES > SCAPES) matrix = scape;
            else matrix = (matrix + rndi(MATRICES - 2) + 1) % MATRICES;
            fprintf(f3, "%d ", matrix);
         }
      }

      // Build up the next generation by proportionate selection

      for(p = 1; p < POPSIZE; p++) pop[p].fit += pop[p - 1].fit;

      for(p = 0; p < POPSIZE; p++)
      {
         rb = rndf(0.0, pop[POPSIZE - 1].fit);
         t = 0;
         while(rb > pop[t].fit) t++; // Roll the ball round the roulette wheel
         tmp[p] = pop[t];
      }
      // Copy them back into the pop[] struct
      for(p = 0; p < POPSIZE; p++)
         pop[p] = tmp[p];

      // Apply the Genetic Operators

      for(p = 0; p < POPSIZE - 1; p += 2) /* Two at a time for (2-pt) c/o */
         if(P_CROSS > rndi(99))
         {
            tx1 = rndi((NUM_LOCI / 2) - 2) + 1;
            tx2 = NUM_LOCI - 2 - rndi((NUM_LOCI / 2) - 2);
            for(t = tx1; t < tx2; t++)
               for(b = 0; b < PER_LOCUS; b++)
               {
                  swap = pop[p].gene[t][b];
                  pop[p].gene[t][b] = pop[p + 1].gene[t][b];
                  pop[p + 1].gene[t][b] = swap;
               }
         }

      for(p = 0; p < POPSIZE; p++) /* One at a time for mutation */
         for(t = 0; t < NUM_LOCI; t++)
            for(b = 0; b < PER_LOCUS; b++)
               if(rndi(P_MUT) == 0)
                  pop[p].gene[t][b] = new_gene();
   }
   fclose(f); fclose(f2);
   fputc('\n', f3); fclose(f3);
   if(printf("Done\n")) getchar();
}
```