A Serial Population Algorithm for Dynamic Optimization Problems

Lars Zwanepol Klinkmeijer

2006

Concluding thesis for Cognitive Artificial Intelligence
Utrecht University

1st supervisor:
Dr. Marco A. Wiering
2nd supervisor:
Dr. Ir. Edwin D. de Jong
3rd supervisor:
Prof. Dr. Albert Visser

Utrecht University

Table of Contents

I dedicate this work to my mother Ruth.
For every drop of sweat I put into this thesis, she sweated and worried twice more

1. Introduction

To some deeply religious people the theory of evolution is a great evil, to them evolution denounces the importance or even the existence of God and, with that, destroys all decent morals, imposes the rule of the strongest and turns people into fascists or communists. Other people feel the need to merge their traditional religious view of the creation of life with those from the scientific community. The most recent effort is Intelligent Design (ID), where the basic idea of evolution, that selection and variation can cause shifts in the characteristics of a population is accepted but still calls for an Intelligent Designer as a start of it all. The people supporting ID try to keep their theories scientific but by incorporating an un-falsifiable, supernatural designer they fail to do so, as a recent court ruling pointed out (Jones 2005). Other people, most notably Richard Dawkins, see a much more positive influence on society from knowledge of the theory of evolution as it takes people away from unfounded superstition toward a more scientific and falsifiable worldview.

This thesis will not go into this snake pit of religion versus science, although the author's position on these viewpoints may somewhat be guessed. Instead it will focus on a useful, applicable form of evolution: Evolutionary Algorithms (EAs). EAs are computer programs which mimic the process of evolution. Over the years many different algorithms have been written. Particularly in the last twenty years, research in this field has boomed. In most cases these algorithms were intended to solve stationary problems or problems with multiple objectives. More recently, researchers have been applying EAs to problems that change over time. This is the focus of this thesis as well. We developed and tested a new algorithm and compared it to two other algorithms. We tested the algorithms on several dynamic, discontinuous optimization problems.

1.1 Basic Elements of Evolution

The developers of such Evolutionary Algorithms were inspired by natural evolution, by the adaptability of species to changes in the environment and the arms race between predators and their prey. One often cited example of this is the adaptation of the peppered moth during the onset of the industrial age in Victorian England. As the countryside of the industrializing areas grew darker caused by the increasing pollution, the coloration in the peppered moth population was darkening as well. At nighttime the moths are active and rest during the day. At that time they were preyed on by birds that use vision to locate their prey. Because the industrial pollution darkened their surroundings, the moths with darker wings became less conspicuous to predating birds than the lighter ones. This caused a shift in the gene frequency in the population towards darker wings. Similarly, as environmental policy came into effect making the air cleaner, the coloration in the peppered moth population became lighter again. This classic example nicely demonstrates the adaptive ability of the evolutionary process through natural selection; in

this case the selection is done by the predating birds. Nevertheless it can not be seen as proof that evolution can actually create new species. Something the more intellectual critiques on the theory of evolution will gladly point out. Proponents will most likely reply to this by saying that this form of adaptation, combined with geographical separation over a long period of time can create different species. But enough about this discussion and let us focus more on what this thesis is really about: evolutionary algorithms, particularly evolutionary algorithms for dynamic optimization tasks. To understand where the research field of EAs comes from we will first look at what evolution is, how does this adaptability come about and what is needed to make it function? These are important questions that need to be answered if we want to implement them into an algorithm. Here we distill three basic elements for evolution to function:

1. A population of structures that can carry information and that can be copied. *(reproduction)*
2. Methods for exchanging information and inducing new variations within the population. *(variation)*
3. A form of selecting individuals for reproduction. *(selection)*

So, for evolution to function, be it in the natural world or as a computer program, these basic elements need to be present. In this chapter we will consider how nature has resolved these issues.

1. Reproduction

If you don't reproduce, you won't have children who will have children. Since no organism lives forever it means that all currently living organisms stem from reproducing ancestors. But for the reproduction to be of any real value to the evolutionary process the offspring must resemble their parents. For this heredity we need some form of replicable, information carrying structure. DNA is obviously the information carrying structure we find in nature. All living and reproducing things, from viruses to vertebrate animals are built up from the information carried in their DNA and it is the DNA that is handed down through the generations. Each individual has its own unique genetic sequence[1] which defines in a very large part how we are built, what we can learn and even how we behave. Of course this end result, or phenotype to put it in biological terms, is influenced by many environmental factors as well but no matter how hard you try, you can never become a bird, grow a third eye or get an IQ of 500. The importance of reproduction for the evolutionary process has left its prints on the organisms that have evolved. Much of the behaviors of organisms is not merely aimed at survival but also aimed to fertilize or be fertilized.

2. Variation

For evolution to function it is vital for several reasons that the genetic code doesn't stay the same. Firstly, the diversity is essential so not all individuals in a population are susceptible to, for instance, one single virus. Secondly, changing the genetic code allows for adaptation on the long run. This allows plants and animals to spread to different

[1] With the exception of clones, twins, triplets etc. stemming from a single cell.

environments or adapt to climatic or environmental changes. There are several ways variation is preserved in nature.

- mutation
- exchanging genes

Mutation induces new variation within the genetic code. A mutation results from errors when genes are copied. Most mutations have a negative effect on the organism as it distorts the way it is built or functions. A high copying fidelity therefore is essential for offspring to survive but on some rare occasions the error may turn out to be either neutral or even positive.

Exchanging genes can be done in very different ways, mostly depending on the way organisms reproduce. Sexual reproducing organisms combine their genes when they create new offspring. Mammalian parents copy roughly fifty percent of their genes to each child thereby creating new phenotypes through new combinations. More on this in chapter three where we discuss a diploid genetic algorithm.

Asexual reproducing organisms such as some bacteria have the ability to exchange snips of genes during their lifetime through a process called ‘lateral gene transfer’. The frequency and the extent of the use of lateral gene transfer is not yet clear.

By recombining genes organisms have the ability to quickly spread the rare positive mutations through their population. Also, combining good genes from one parent with those from another may result in an even better individual.

3. Selection

Selection is what comes closest to what most people know about evolution, neatly captured is the sound bite “survival of the fittest”. This phrase was coined several years before the publication of Darwin’s ‘The Origin of Species’ by a philosopher named Herbert Spencer but became almost synonymous with evolution. Often this phrase is misinterpreted as survival of the *strongest* but instead should be read as survival of the *best adapted*.

All breeders understand the powerful combination of hereditary trades and selection. By picking the plants or animals that show the basic trades he/she likes the breeder can create a new organism that shows those features even stronger. Charles Darwin, a hobbyist pigeon breeder and naturalist, combined his breeder’s knowledge with things he noticed during his voyage on The Beagle. Most notably are his observations of the beaks of the finches on the different Galapagos Islands. The beaks had adapted to the differing food sources on the different islands. He saw that nature had been doing the breeder’s work by selecting the finches with the right type of beak-features and so adapting the populations on each of the islands to their surroundings. This ‘Natural Selection’ (as opposed to the breeder’s selection) caused the birds to adapt, to fit to their separate

environments. Natural selection is somewhat difficult to put into exact terms. There are so many different factors that play a role. Basically an organism has to survive to a reproductive age and produce fertile offspring who, just like their parents, survive to a reproductive age and produce fertile offspring. Any obstacle that has to be overcome for this cycle to continue can be seen as a part of the natural selection for that organism.

So, because all living organisms that exist today stem from creatures that had to survive, reproduce and adapt, it should be little surprising that they evolved behaviors that promote these very aspects of evolution. In general the skills an animal is forced to have can be summarized by the four F's: Feeding, fleeing, fighting and reproducing.

1.2 Relevance to Cognitive Artificial Intelligence

Now that we have gotten a bit of understanding of what natural evolution is, we will move on to the artificial version. This is something which is very much part of the research in the field of Artificial Intelligence. Part of the aim of Cognitive Artificial Intelligence is to understand what knowledge is, how it is represented and functions within natural organisms and, similarly, how we can represent and make it function in (computer) models. Evolutionary Algorithms are one way of representing and developing knowledge in such computer models. Also, Machine learning plays a major role within the computer science tract of AI. Therefore, EAs being part of Machine Learning is part of AI as well.

This thesis focuses on EAs as an engineering tool specifically for dynamic optimization. We have developed our own algorithm which we tested on tasks with several different settings. These results we compared with two other popular algorithms which were tested on the same tasks.

In this chapter we tried to give the reader a bit of an understanding of natural evolution since it is the source of inspiration for most of the work discussed in this thesis. In chapter two and three we discuss how EAs function, we look at some of its history and we look at how researchers have adapted them to work on dynamic optimization problems. We end chapter three by putting forward our own genetic algorithm designed to handle dynamic optimization problems. This algorithm, with the acronym SPA which stands for Serial Population Algorithm, was tested on several dynamic versions of the 0/1 knapsack problem together with two other GAs; Cobb's successful hypermutation algorithm and a version of a diploid genetic algorithm. Chapter four describes several characteristics of dynamic environments on which EAs have been used. The experimental setup is given in chapter five. In chapter six we discuss the results from the experiments followed by chapter seven where we look at what the results tells us about how SPA functions. We round it all up in chapter eight where we draw our conclusions on the strengths and weaknesses of SPA and propose some future work.

2. Evolutionary Algorithms

In the first chapter we stated the basic elements needed for evolution to function, furthermore we looked at the forms in which these elements can be found in nature. In this chapter we will discuss artificial evolution otherwise known as Evolutionary Algorithms (EAs). We will describe how EAs function and how the basic elements of evolution relate to the elements of EAs. Furthermore, we will look at some of the history of EAs and at some of the different classes that exist in the field of EAs.

2.1 General functioning of an Evolutionary Algorithm

An Evolutionary Algorithm consists of the following parts:

- a population of individuals or 'possible solutions'
- a method of evaluating these possible solutions
- a selection method
- operators to alter the possible solutions

Because we are trying to solve a problem, in this case an optimization problem, we refer to the individuals as possible solutions. Each individual in the population is represented by a genetic code which is translated to the problem by a mapping from the genotype to the solution space[2]. The aim of an EA is to improve the quality of the solutions over several timesteps called generations. A rough pseudo code for a general EA is given in figure 2.1. During each generation the EA starts out with a population of possible solutions or also referred to as individuals. Often the individuals in the first generation are generated at random but this is not always necessarily the case. Once the algorithm is running, the individuals in the next generation will be based on individuals from the current generation's population.

The individuals in the population are evaluated on the problem at hand, assigning a value to each individual depending on how well they perform during the evaluation. This value is commonly referred to as a fitness value. The evaluation process can be very simple and fast, as we will see when we discuss the Knapsack Problem in section 5.2 which we use in our experiments. Other tasks may take much longer to evaluate. One may have to wait on complex simulations to finish which may last for minutes or even days. In such cases you may want to avoid many evaluations and for that reason maintain small populations or use heuristics for initializing the first population.

[2] The solution space is the set of all possible solutions, both good and bad.

Once the individuals have been evaluated the algorithm starts selecting individuals whose descendants will form the next generation. Because usually there exists a genetic difference between the individuals in a population, individuals will show a difference in the values associated to them. Based on this difference some individuals will have a higher chance of being selected than others. When we have selected some good individuals we can start transforming them through recombination and mutation. Two individuals can be recombined to form one or more offspring. These offspring can then be mutated slightly. One may even find it desirable to use only recombination or only mutation but this is a designer's choice. When you have created sufficient offspring you can go to the next generation and run through the steps we've just discussed. Many of the decisions on the exact nature of these steps, such as the size of the population, the mutationrate, the kind of selection used etc. are all for the designer to be made.

It is important to notice that each EA is an artifact of the designer who by no means is bounded to any natural realistic form of how his EA functions. For instance he may even go as far as to incorporate Lamarckian evolution into his algorithm, something that is believed not to be possible in nature.

Generate initial population P_t
evaluate P_t
while(*stopping_criteria*)
    P'_t = **recombine**(P_t)
    P''_t = **mutate**(P'_t)
    F_t = **evaluate**(P''_t)
    P_{t+1} = **select**(P''_t , F_t)
    t=t+1
end

Figure 2.1 *pseudo code for a general EA*

We now will take a closer look at the workings of the various parts of an EA.

2.2 Representation

As stated above, by far most Evolutionary Algorithms consist of a population of individuals. These individuals are part of the representation space which itself is the set of all possible and legal genotypes. The way these individuals represent their solutions is dependent on several aspects:

- genetic representation
- the definition of the search problem
- the fitness function

2.2.1 GENETIC REPRESENTATION

The genetic representation can have several forms. Here we will mention the most common:

- Binary
  - Example: genotype: [0,0,1,0,1,0,1,1,0]
- Real values ($\mathbb{R}$)
  - Example: genotype: [10, 1.5, -2, 4.1, 531]

Many researchers in EA have used the binary coding for their representation (Calabretta, R., Galbiati, R., Nolfi, S. & Parisi, 1996, Golberg & Smith, 1987). Some even go as far as to transform real valued numbers into binary strings. This transformation is not necessarily needed and some researchers even believe it is better to use real valued representations when your problem has real values (Giraldez, Aguilar-Ruiz & Riquelme, 2003).

2.2.2 THE DEFINITION OF THE SEARCH PROBLEM

The definition of the search problem relates your genetic code to the search space. For instance, with the 0/1 dynamic knapsack problem you tell which gene belongs to which item and what the weight and value of that item is. The story that goes with this problem is the following: You have several objects, for instance gold nuggets, each of which has its own weight and value. You want to take along as much value as you possibly can, but because the total weight of the objects is more than your knapsack can handle, you cannot take them all. So now you have to make a choice: which items to take along and which item to leave behind. The definition of the search problem in this case tells you:

- which item is linked to which location on the binary genotype
- the weight and value of each item
- ‘1’in the genotype means put in knapsack and ‘0’ means leave out
- The goal of the problem: find a set of items that maximize the total value, limited by a maximal weight.

2.2.3 THE FITNESS FUNCTION

The fitness function, or objective function as it is also known, rates the individuals on their specific property or properties. In a sense it tells you how ‘good’ an individual is. In simple theoretical problems the fitness function can be very simple and straightforward. In the max-one problem for instance, where the genetic representation is a binary string and the fitness function is simply the summing of the number of ‘ones’ in the genotype. The problem definition is almost identical to the actual fitness function. For the 0/1 Knapsack Problem it is slightly more complex. Here the fitness function is a combination

of a penalty function combined with parts of the problem definition, namely the objective to find a set of items that maximize the total value, limited by a maximal weight.

In some cases, particularly the simplest theoretical problems it can be difficult to clearly distinguish the definition of the search problem from the fitness function. Take for instance the simple max-one problem. The genetic representation is binary and the objective is to get a binary string with as many ‘ones’ as possible. Here both the definition and the fitness function are simply counting the number of occurrences of ‘one’ in a string.

Because the selection process uses the outcome of the fitness function to decide which individuals to use to create the next generation, it is an essential part in guiding the path of evolution. If you reward the wrong characteristics of an individual, your EA will never end up with the solution you initially intended it to find. In the case of multi-objective optimization, where two or more, often opposing, goals are to be met, this can be especially difficult. Changing the balance of how important each goal is may give very different results.

2.2.4 STRUCTURE OF THE SEARCH SPACE

In most cases an Evolutionary Algorithm is supposed to find the global optimum in the search space without getting stuck on local, less good optima. To get a better understanding of what we mean when we talk about local and global optima it is useful to look at the structure of the search space, or fitness landscape. For this we will use some very low dimensional search spaces. If we have a two dimensional search space the fitness landscape could look like figure 2.2

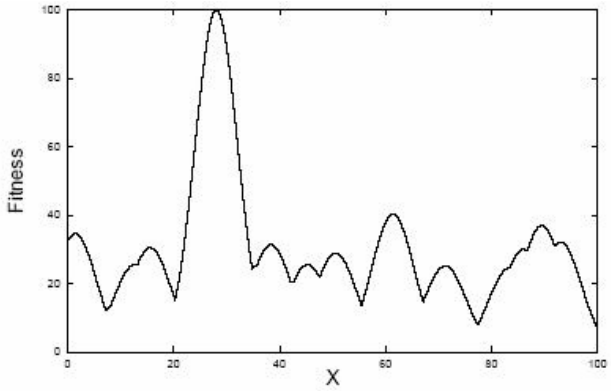

Figure 2.2 *a two dimensional search space, with only one gene ‘x’*

The genotype in this case has only one gene which can have a value ranging from ‘1’ to ‘100’. Let’s say we want to find the highest point in the landscape. There are several peaks to choose from but only one is the highest. All other peaks are optimal only for their local area; globally they are dominated by the global optimum.

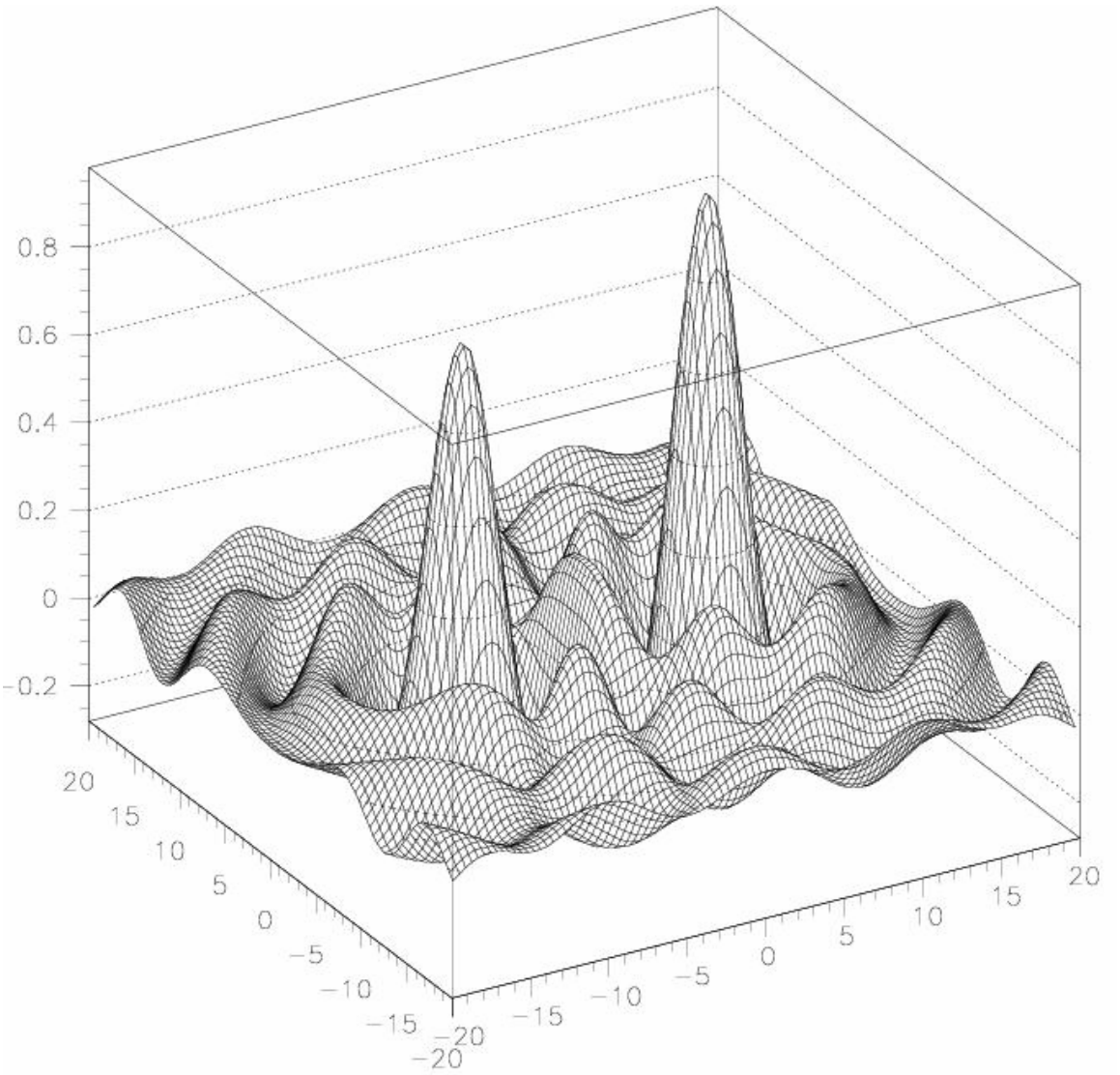

Figure 2.3 *a three dimensional fitness landscape* (image taken from http://nc25.troja.mff.cuni.cz/~soustruznik/GA.html)

In figure 2.3 we see an example of a landscape based on two genes. The values of these genes are set out against the x and z axis whereas the fitness is set out on the y axis. Here it is much clearer why it is called a fitness landscape. Again there is one peak that globally dominates all others.

The representation space does not necessarily have a one on one match with the solutions space. For instance, through using complex genotypes or complex expressions of the genotypes, we first create a phenotype from the genotype before relating it to a solution space. This may cause different genotypes to have the same location on the fitness landscape. An example of this is shown in figure 6.1 where a population of diploid individuals often converges to the same phenotype resulting in having the same fitness values.

2.3 Fitness-based Selection

Fitness-based selection is the driving mechanism that steers the population to a higher average fitness. Individuals are selected from the population to produce offspring. Because the selection is based on the genotype's fitness value, the higher a genotype's fitness is relative to the populations' average fitness, the more like it is for that genotype to have offspring in the next generation.

Here we will mention some of the most common forms of selection systems found in the field of EAs followed by a short explanation:

- Fitness proportionate selection
- Ranking selection
- Tournament selection

2.3.1 Fitness Proportionate Selection

Fitness proportionate selection is a rather literal implementation of the selection principle described above. The chance that an individual is selected is equal to the individual's fitness relative to the population's average. A common example of this form of selection is roulette wheel selection where each individual in the population occupies an area of the roulette wheel proportionate to its fitness. So whenever an individual has to be selected you spin the wheel and pick the individual where the ball lands. A well known problem with this form of selection is the loss of diversity in the population. If one individual is many times better than all others in the population it will occupy most of the roulette wheel, causing it to be selected almost every time. Such individuals are referred to as a 'supersolution' and are generally seen a problem for EAs because after a few generations they take over the population resulting in a loss in variation.

Maintaining diversity in a population is vital to an EA for two reasons:

- Parallel search
- Selection pressure

Identical individuals occupy the same point in the search space and therefore do not search the space in parallel, loosing some of the power of an EA.

When individuals have similar fitness values there is less selection pressure causing the algorithm to drift randomly through the search space. This slows down the speed with which the algorithm will find the solution.

2.3.2 RANKING SELECTION

With ranking selection the occurrences of supersolutions are eliminated. Here we no longer select based upon their relative fitness but proportionate to their rank in the population. So the individual with the highest fitness gets the highest rank and accordingly the highest chance of being selected. The second best gets a slightly less high chance of selection and so on.

2.3.3 TOURNAMENT SELECTION

With tournament selection, every time you want to select one or two individuals, you hold a small tournament. The size of the tournament may vary depending on the amount of selection pressure the designer wants. With a tournament size of for instance four, you randomly select four individuals from the population. This random selection does not look at the fitness of an individual so each has an equal chance of being selected. Once we have the four individuals we order them according to fitness and we keep the best. A smaller tournament size, which has low selection pressure, means that low fit individuals have a better chance of surviving. Although this comes at the cost of slower finding the optimum it does maintain a higher diversity and so increases the odds of finding the actual optimum. The nice thing about tournament selection is, is that with only one parameter, namely the tournament size, one can easily control the selection pressure and thereby the amount of variation in the population.

2.4 Recombination

The learning ability of an EA is based for a large part on the fact that it uses a population of individuals to search the solution space in parallel and by selecting from the population better solutions to be combined to find possible better solutions. One example of recombination in EAs is one-point crossover. Say, you have two good individuals I_A and I_B. When the genes that score well for I_A are in the first portion of the genotype and for I_B in the second part cutting the two genes in half and mending the beginning of I_A with the end of I_B together could potentially result in a child I_C that has an even higher fitness.

Solve Max-One problem
I_A: [**1,1,1,0,0,0**] fitness = 3
I_B: [*0,1,0,0,1,1*] fitness = 3

Cut after third number and mend the pieces
I_C 1: [**1,1,1**,*0,1,1*] fitness = 5
I_C 2: [*0,1,0*,**0,0,0**] fitness = 1

Figure 2.2 Example of one-point crossover.

It is not necessary that the genes are cut exactly in half, as we did in the example. The important thing is that the two genes are cut at the same position. This position can be

anywhere on the chromosome and is chosen at random. One-point crossover is not the only form of recombination in EAs, the designer may want to select more than one place to cut the genes (n-point crossover) or swap some elements between the genes (uniform crossover).
As you can tell from this example recombining the two parents does not necessarily result in a child with a higher fitness. The same goes for mutating an individual. In fact: most changes tend to result in worse individuals. It is for this reason that many algorithms incorporate a child population size that is much larger than the original parent population size. From this larger child population the best individuals will be selected to go to the next generation.

So if:
Parent population size = N
Child population size = 3N
N children will be selected to form the new parent population.

2.5 Mutation

Mutation is used in all EAs. Some EAs only use it as a background operator for introducing new genetic material, relying mostly on crossover and a large population to find solutions (Goldberg, 1987, Holland, 1975). Others rely solely on mutation (Cobb, 1990). In most cases the rate of mutation is held at a constant mutation rate. This mutation rate determines for each gene on the chromosome the chance for it being mutated. By putting the mutation rate at 1/L where L is the length of the chromosome you have a very good chance that there is on average one mutation on each chromosome. The mutation rate is often referred to as being a local search operator as opposed to crossover being a global search operator. A mutation changes the individual only very slightly. After the mutation the new individual will usually remain in the vicinity of the old individual. Crossover on the other hand is more likely to result in bigger changes, moving the individuals around over larger areas of the representation space.

2.6 Historical Background of EAs

Now that we have seen some of the basic principles and operators of EAs we will take a look at its history and see what the archetypical features of the different classes are. Over the years many different forms of EAs have been developed. Usually a distinction is made based on the representation used and on their reliance on mutation and crossover. Some EAs have very distinct representations, such as Finite State Machines (Evolutionary Programming by Fogel 1964) or executable programs (Genetic Programming by Koza 1989). Such algorithms are usually seen as independent classes within EAs. Others are obvious derivatives from the more standard versions, like for instance the hypermutation genetic algorithm which is a GA adapted for dynamically changing optimization problems. Along the way the lines between the different classes

have blurred, where originally GAs only used binary string representation nowadays it is not uncommon to use real values.

2.6.1 Evolution Strategies

Already in the 1950's several researchers with an interest in evolution made (computer-) models where they tested out some of the basic principles (Fraser, 1957; Friedman, 1956). The earliest research that ended up as a part of the EA family is the Evolution Strategy (ES), a name taken from the German 'Evolutionsstrategie'. In 1964 Rechenberg and Schwefel of the Technischen Universität Berlin, both with a background in aerodynamics, developed ES while searching for the optimal shapes of bodies in a flow. Over the years they extended their algorithm to solve optimization problems on computers and strengthened its theoretical basis (Rechenberg 1973; Schwefel 1977).

Let's look at the archetypical features of ES. ES rely heavily on mutation. From this the most noticeable feature is the fact that the size of the mutation itself is subjected to evolutionary change. The genetic representation consists of real valued numbers, so individual $\vec{v}$ is a vector of elements $x_i \in \mathbb{R}$. Every element $\vec{v}(x_i)$ has a normal distribution associated to it that controls how big the mutation of that element will be. By controlling the size of the normal distributions you can control the speed with which each element evolves. Elements that perform well need only a small normal distribution whereas elements that perform not so well can make bigger changes. The size of the normal distributions is not controlled by the designer but it too is changed by random mutations, albeit using some heuristics for deciding when to change the size. Schwefel (1995) gave some implementations of this.

2.6.2 Genetic Algorithms

At the University of Michigan John Holland worked on what we now know as Genetic Algorithms (GAs). In 1975 he published a book on adaptation in natural and artificial systems. Leaning heavily on a biological basis he described an algorithm with genetic operators as selection, crossover, mutation and inversion. Though the last one, inversion, is rarely used as it is unclear what its benefits are (Hill, Newell, O'Riordan 2004). The standard version of a GA has three archetypical features:

- Binary representation
- Proportional selection
- Crossover emphasized for inducing variation

Over the years many researchers altered some of the features of the algorithm, particularly the selection method. As we have seen above this form of selection has its shortcomings and nowadays ranking and tournament selection seem more popular to use.

Holland showed the potential of Genetic Algorithms through examples in game theory, control and optimization, economics and artificial intelligence; areas where GAs are still used today. Interestingly enough most of these research areas have problems that are more of a dynamic nature than static. Furthermore he proposed the schemata theorem; a system with which you can observe the dynamics within a population over several generations.

A decade past and still little research was being done on EAs. It was not until the late eighties when the number of EA researchers rapidly increased. This was also the time the first real textbook on GAs appeared, written by David E. Goldberg. Building upon the schemata theorem Goldberg developed the Building Block theorem. This theorem gave a more mathematical insight into the processes of GAs and thus opened up a road to more theoretical research. The general idea is that in the beginning of a search some individuals have a combination of genes that have a high fitness. By recombining individuals you can bring building blocks together, creating building blocks with even higher fitness. The building blocks can be analyzed using the schemata theory and some functions Goldberg proposed which consider aspects of the schemata such as length and fitness.

2.7 Basic Elements of Evolution in EAs

As one can tell from earlier sections, the basic elements of evolution which we discussed in the first chapter are in many cases explicitly defined or otherwise quite obvious. Let's look at the three elements one by one.

1. Reproduction
The replicable, information carrying structure in EAs are usually strings of numbers. Some of these strings are copied to form the next generation.
2. Variation
Variation is valued very highly within the field of EAs. Much of the research has focused on finding ways to avoid the population to converge to a single solution prematurely. A low amount of variation in a population causes the algorithm to learn slower or prevents it from finding the best solution.
3. Selection
As seen above, selection is an explicit operation in EAs. Several different kinds of selection have been developed mainly with the aim of supplying strong selection pressure while maintaining sufficient variation.

2.8 To use EAs or not to use EAs, that's the question.

So far we have seen how EAs function, but this does not tell us what they can be used for. EAs can be used for a very wide range of problems. In fact, as long as you can formulate the problem mathematically and there are parameters to be tuned, EAs can be applied. But as so many things in the world, EAs have their advantages and disadvantages. For instance; when the problem is mathematically simple, such as a linear

or a convex problem, a classical optimization technique like Dynamic Programming will generally outperform most EAs. But when the task is more difficult, such as discontinuous or non-differentiable problems or tasks which have multiple criteria to be optimized, EAs are much better at finding solutions within reasonable time.

Another interesting aspect of EAs is that no knowledge of the problem is incorporated into the algorithm. This means that no full knowledge of the underlying mechanisms of the problem is needed to solve the problem. For example when Rechenberg and Schwefel were trying to find a shape that would minimize the total drag of three-dimensional bodies in a turbulent flow, they used a physical model to evaluate the solutions they created using their ES. They ended up with a shape very different from their expectations and for which, at the time, no theory existed to explain it. Similar unexpected results have been shown by Adrian Thompson (Thompson, 1997) who used a GA to design circuits on a FPGA[3]. Here, some parts of the FPGA were not connected to the functioning circuit yet they still influenced the output of the circuit probably through electromagnetic coupling.

[3] FPGA stands for Field Programmable Gate Array. These are chips with an easily modifiable circuitry.

3. GAs for Dynamic Optimization

From here on we will consider only the class of Genetic Algorithms and leave all other classes be. We will take a look at how GAs have been adapted to function for dynamic optimization and look at some specific algorithms. At the end of this chapter we will introduce our own Algorithm: the Serial Population Algorithm or SPA for short.

Dynamic problems change over time as opposed to static problems which stay the same. Normally, Genetic Algorithms are designed to find the optimal solution for a static problem. The fact that static problems do not change over time makes them very GA-friendly. Genetic Algorithms need some time to find the optimal solution and many tend to loose a lot of diversity in the population along the way. If the problem changes, the loss of diversity can cause the algorithm to never be able to find the new optimal solution. So, in order to apply Genetic Algorithms to dynamic environments we need to make some adjustments to the standard Genetic Algorithms. One can adopt several kinds of strategies, each having its own advantages and disadvantages. Jürgen Branke (2003) gave a nice overview which we will repeat in short here accompanied by a more in-depth look into two algorithms; diploid GA and triggered hypermutation GA.

3.1 Strategies for Dynamic Environments

In the literature four broad strategies can be distilled:

- Increase diversity after change
- Maintaining diversity throughout the run
- Memory
- Multiple subpopulations

3.1.1 INCREASE DIVERSITY AFTER CHANGE

Triggered Hypermutation (Cobb, 1990) and Variable Local Search (VLS) (Vavak, Jukes & Fogarty, 1998) are two well-known examples of this strategy. Once a change in environment has been detected the mutation-rate will be increased. This can happen either in one dramatic burst followed by a period of decay as with Hypermutation or by gradually increasing the rate as with VLS.

The (Triggered) Hypermutation GA (HMGA) is an elegantly simple algorithm that works well on dynamic environments. For most of the time the algorithm works like a regular GA using mainly crossover and selection to search for a good solution. When a change in the environment is detected the algorithm increases the amount of variation in the population by raising the mutation rate to a very high level. Cobb changed the rate from

0.001, which is very low, up to rates of 0.5. The increase in mutation rate is followed by a period of decay where the rate decreases back to its base rate. Very high mutation bursts like 0.5 are similar to reinitializing the population. Lower levels of mutation bursts retain some parts of the old solution and so are better capable of adapting to smaller changes. Furthermore Morrison and K. de Jong (2000) showed that larger hypermutation bursts track the optimum better when the environmental changes are frequent while lower hypermutation levels perform better when the changes are less frequent.

Cobb tried detecting the change by monitoring the fitness of the best performer in the population. When this value declines over several generations, a burst of hypermutation is triggered. Not all changes are detectable this way. Adding peaks in the multiple peaks problem or raising the maximum allowed weight in a knapsack problem may go undetected leaving the algorithm stuck on a suboptimal solution.

Over the years this algorithm has shown itself not only to work well on continuously changing environments but also on discontinuous environments which show large changes to the optima (Lewis, et al., 1998; Morrison & K. de Jong, 2000; Simões & Costa, 2003b). Problems may occur when the algorithm fails to detect the change in the environment or when the change is too large (Grefenstette, 1992).

3.1.2 MAINTAINING DIVERSITY THROUGHOUT THE RUN

Diversity was already important in GAs used for static environments in order to avoid getting stuck in suboptimal solutions. In dynamic environments this importance is amplified. If a change occurs, your once optimal solution is destined to become suboptimal at best. Diversity maintenance mechanisms such as Fitness Sharing, Random Immigrants (Grefenstette, 1992) and Crowding are common examples of this strategy. Ensuring the population holds no multiple instances of the same solution is another example.

Crowding and fitness sharing are both niching methods. Crowding is a selection mechanism which selects individuals for reproduction partly on their similarities. Fitness sharing adjusts an individual's fitness to the number of similar individuals. The more individuals are similar the lower their fitness will be. In both cases the effects are such that they cause the population to spread out more thus increasing the population's diversity. Without specifying explicit subpopulations the individuals tend to specialize, form a niche in certain parts of the fitness space.

The Random Immigrants algorithm by Grefenstette was oddly enough inspired on Cobb's triggered hypermutation algorithm, which uses the diversity *introducing* strategy. In this case the diversity is introduced not after a change but with every generation, making it a diversity *maintaining* algorithm. Another difference with the hypermutation algorithm is the way the diversity is maintained. Instead of using high levels of mutation, Random Immigrants replaces some of the worst individuals with new randomly generated individuals.

3.1.3 MEMORY

Applying memory serves two functions: First; it provides diversity by retaining former good solutions which otherwise would have been lost in the selection process and reintroducing (parts of) these solutions on a later occasion. Second; reintroducing former solutions in repetitive environments can enable the algorithm to quickly retrieve the previously encountered optimum.

Two types of memory can be distinguished:

- Explicit memory
- Implicit memory

GAs incorporating *explicit* memory usually have strategies for storing solutions and reintroducing them on later occasions during the run (Louis & Xu, 1996; Ramsey & Grefenstette, 1993; Bendsten & Krink, 2002; Eggermont & Lenaerts, 2002). GAs incorporating *implicit* memory usually incorporate some form of redundancy in their genetic representation. The most common example is using a diploid genetic structure. (Branke, 2001; Calabretta, Calbiati, Nolfi & Parisi, 1996; Lewis et al., 1998; Ng & Wong, 1995)

The Random Immigrants algorithm can be seen as a steppingstone toward many of the explicit memory algorithms. Instead of introducing random individuals to the population previously fit individuals, which are stored, are reintroduced. In the case of Eggermont's and Lenaerts' Case Based Memory GA (2002) such individuals are introduced when a change is detected. The memory population does not necessarily have to consist of individuals from previous generations. Bendsten and Krink (2002) for instance generated the memory population randomly at the start and update it during the evolutionary run.

A diploid GA is different from a regular GA by the fact it has a set of two chromosomes instead of the common single haploid chromosome. The consequence of this is that two genes compete for the same phenotypic trait in the same individual. In order to solve this dilemma a dominance mapping is devised, labeling some genes as dominant and others as recessive. If a dominant gene is paired with a recessive gene, only the former is expressed in the phenotype leaving the recessive gene unexpressed. Dominant genes are thus able to protect less fit recessive genes from being discarded by selection. Formerly fit genes can piggyback ride the fitter dominant genes they are paired with, hopefully coming into expression again when the environment is more favorable. It is this mechanism that is thought to give the GA a form of *implicit* memory.

Apart from this it is also possible for two dominant or two recessive genes to be paired. What happens in this case differs between the dominance mappings used by different researchers. Although over the years many researchers (Callabretta et al., 1996; Hollstein, 1971; Ng & Wong, 1995; Ryan, 1997) have devised their own dominance mappings there is one mapping that is commonly referred to; the triallelic dominance mapping.

The triallelic dominance mapping was first developed by Hollstein (1971) for static environments and made popular by Goldberg and Smith (1987) who first used it for a dynamic environment. The genetic strings use a trinary [0,1,2] representation instead of the regular binary [0,1].

Table 1. *A schematic view of the triallelic dominance mapping where the first row and column denote the genetic values.*

	0	***1***	***2***
0	0	0	1
1	0	1	1
2	1	1	1

The first row and column in table 1 show the genetic values (alleles) and the rest of the table shows the resulting phenotypic expression. In this mapping there is a clear bias for expressing 1's. Alternative mappings (Ng & Wong, 1995, Ryan, 1997) have been proposed to eliminate this bias. Their representations have four alleles where the probabilities of generating 0's and 1's are equal. Lewis, Hart and Ritchie (1998) showed that a diploid structure alone is not enough for a diploid GA to adapt to changing environments. Frequently switching the values from dominant to recessive and vice versa was needed to give acceptable results.

3.1.4 MULTIPLE SUBPOPULATIONS

When using multiple populations, researchers usually use one population to track the best solution and the rest to track suboptimal peaks in the fitness landscape. One has to make sure that two subpopulations do not cover the same area but that each subpopulation tracks a different peak. Branke developed a strategy called *exclusion* to avoid this problem. The best individuals of each subpopulation are compared to each other spatially. If the distance between them is smaller than a predefined amount *r* the worst scoring population is marked for re-initialization (Branke, Klaussler, Schmidt & Schmeck, 2000; Ursem, 2000).

More often than not you will find that algorithms incorporate combinations of these strategies, making clear-cut distinctions difficult. Saying that many forms of multiple subpopulations could be classified as a form of niching or diversity maintenance wouldn't be all wrong.

3.2 Serial Population Algorithm

Now that we have seen some of the strategies and algorithms used for dynamic optimization problems we will introduce our own algorithm: The Serial Population Algorithm or SPA. The strategy we use in SPA is a combination of multiple subpopulations and memory. In our case we do not use the subpopulations in parallel as with other researchers but we use them in series, thus creating a form of memory. In this section we describe how SPA functions and which mechanism we have used. In chapter 5 we give some specific details on the parameters we've used and in chapter 6, where we show and discuss the results form the experiments, we will draw our conclusions on how well these choices worked out.

The pseudo code of SPA is shown in figure 3.1. At the start of an evolutionary run a population is created and initialized randomly. The population is divided into a predefined number of subpopulations. All subpopulations are evaluated and the best[4] subpopulation is selected. Until a change in environment is detected only this subpopulation will be used. When a change has been detected all subpopulations will be evaluated on the new environment and again the best subpopulation will be selected. It is through this serial use of the subpopulations that we hope to create a form of memory. This memory-function will perform optimally when the number of subpopulations is equal to the number of optima.

For the detection of environmental changes we use a system similar to what Eggermont and Lenaerts (2002) used for their algorithm. We temporarily store the best individual and its fitness value at the end of each generation. We then evaluate this individual again at the beginning of the next generation. If its fitness value has changed we know a change in environment has occurred and in our case it triggers the algorithm to reevaluate all the subpopulations. We then select the population which contains the individual with the highest fitness.

When SPA has decided on which subpopulation to use, a child population will be created. To generate the child population we repeatedly select two parents from the subpopulation through tournament selection. These parents are recombined using two point crossover with chance *P*c followed by mutation. The resulting two children are placed in the child population.

From the child population the next generation of this subpopulation is selected, once again using tournament selection. By using elitist selection on both the parent and child population we ensure both the best parent and the best child are added to the new subpopulation. This new subpopulation is not allowed to contain any double instances thus ensuring the needed diversity in the population.

[4] The best subpopulation is either the subpopulation containing the individual with the highest fitness or the subpopulation with the highest *average* fitness. Either qualification seems to work fine.

```
until maximum number of generations:
        if change detected
                evaluate total population;
                choose best subpopulation;
        end
        else continue with same subpopulation;

        until child population is full:
                select two parents with tournament selection;
                perform:
                        crossover with chance Pc;
                        mutation;
                add kids to child population;
        end
        add the best parent and the best child to new subpopulation;
        until new subpopulation is full:
                select child with tournament selection;
                add child to new subpopulation;
                remove any double instances;
        end
        replace the old subpopulation with the new;
end
```

Figure 3.1 *Pseudo code for SPA*

4. Dynamic Environments

Dynamic environments come in many different flavors that can have dramatic effects on the functionality of the algorithms used. Therefore it is important one first gets an idea of what the problem looks like and how it behaves before deciding on what algorithm and genetic operators to use. Here we will discuss two important distinctions and two further characteristics of dynamic environments followed by two examples of commonly used dynamic environments.

4.1 Characteristics of Dynamic Environments

We now give two distinctions by which you can characterize the dynamics and we will discuss their consequences for what type EA to use:

Recurrent	vs.	Non-recurrent
Continuous	vs.	Discontinuous

Recurrent environments, like all dynamic environments, change their settings during the evolutionary run resulting in having different environments. What makes a recurrent environment different from a non-recurrent environment is that it has a limited number of such settings and these settings are revisited during the evolutionary run. This can happen either periodically/cyclic or a-periodically. If the environment changes periodically the states are visited in a specific, repeating order. If the environment changes a-periodically there is no specific order and any possible repetitions are accidental. In general you could say that a recurrent environment, be it periodic or a-periodic, is well suited for GAs that incorporate some form of memory.

Non-recurrent environments have no states that are revisited or at most merely by accident. Here, applying memory will serve little more function than adding some diversity to the population. GAs that either maintain or introduce diversity seem to have better chances of succeeding.

Continuous environments, in a strict sense, change every timestep by a small margin. They require only small genetic changes to be made to the previous found optimum in order to find the next. Such environments are state dependent functions where the next state is dependent on the previous state. Maintaining diversity throughout the run seems to be a good strategy to handle this problem. If the environment is both *continuous* and *recurrent* the amount of related yet distinct states may be too large for a memory system to be a feasible option (Cobb, 1990).

Discontinuous environments switch from one state to the next in relatively large steps. This may cause problems for some diversity maintenance GAs and diversity introducing GAs when the adjustments are too big (Grefenstette, 1992). Combined with a recurrent environment the amount of states to be found is likely to be small thus a paradise for memory incorporating GAs including SPA.

Additional to these two distinctions there are two more characteristics to consider:

- Frequency of changes
- Detectable changes

The more frequent the optimum changes the more difficult it will be for the EA to track the optimum. All EAs need several generations to find the optimum. If your goal is not to find the optimum but merely track a good solution EAs are still an option. A diploid GA (Goldberg 1989) for instance, performs better on an environment that switches between two optima every other generation than a standard GA. Although the optima will not be found it will give a smaller average error.

Some algorithms need to detect a change in environment to function. It may be the signal to increase the diversity as with the triggered hypermutation or search for a better subpopulation as with SPA.

Detecting a change is not always as straightforward as it may seem. Commonly, monitoring a possible change in fitness value is used to detect the change but this does not necessarily always work. If, for instance, you compare the fitness value of an individual over two generations you will detect a change if the maximum allowed weight for a knapsack problem is reduced. But, on the other hand, if the allowed weight is increased it may very well go undetected. In addition, a negative change does not necessarily mean a change of environment. It could also mean a temporary loss of fitness during the EA's search. As with most machine learning algorithms, GAs have to find a tradeoff between exploration and exploitation. Staying too long in the exploration phase slows down your speed of learning, doing too much exploitation may cause the algorithm to get stuck in a local optimum. So, to find the global optimum of a problem and not get stuck in a local optimum it is important to search the environment well. This may mean that sometimes you have to take a temporary loss in fitness for granted while moving away from a local optimum to a global optimum. By using elitist selection you can ensure the best individual remains in the population but this may also cause the algorithm to remain on the local optimum. Some researchers therefore use a 'repeated loss of fitness' rule to trigger the algorithm (Cobb 1990). They track the fitness of the current best individual in the population. If this fitness lowers over several generations, say five generations, they conclude a change in the environment has occurred. However, this of course slows down the reaction of the EA thus lowering the average fitness over the entire run and making the algorithm less suitable for frequent changes in environment.

4.2 Dynamic 0/1 Knapsack Problem

The Dynamic 0/1 Knapsack Problem is an oscillatory version of the standard 0/1 Knapsack Problem. The task is to fill a 'knapsack' with a subset of items. Each item has both a *weight* and a *value*. The aim is to maximize the value of the content of the knapsack without exceeding the maximum allowed weight W.
Mathematically, the standard knapsack problem can be described as:

Eq. 4.1 $$\max \sum_{i=1}^{n} v_i x_i$$

Constrained by:

Eq. 4.2 $$\sum_{i=1}^{n} w_i x_i < W$$

Where $\vec{v}$ and $\vec{w}$ are the value and weight vectors respectively, each of size n and $\vec{x}$ is the genetic representation of '0' and '1' where '1' means 'put the item in the knapsack' and '0' means 'leave the item out'.

The weight constraint is enforced by a penalty function identical to the one used in Smith and Goldberg (1987):

Eq. 4.3 $$Pen = C(\Delta W)^2$$

Where:
C = 20.
ΔW = the overweight of the individual:

Eq. 4.4 $$\Delta W = \sum_{i=1}^{n} w_i x_i - W \geq 0$$

So by combining Eq. 4.1 and Eq. 4.3 we end up with the fitness function:

Eq. 4.5 $$\sum_{i=1}^{n} v_i x_i - Pen \geq 0$$

Negative scores on the fitness function are rated as zero.

The equations 4.1 to 4.5 define the *standard* 0/1 knapsack problem. There are several ways to turn this into a *dynamic* 0/1 knapsack problem. The most common way is to alter the maximum allowed weight W (Goldberg & Smith, 1987; Smith & Goldberg, 1992; Lewis, et al., 1998; Simões & Costa, 2003b). Another way is to use different sets of items and alternate between them during the evolutionary run (Zwanepol Klinkmeijer, de Jong & Wiering, 2006).

4.3 Moving Peaks

In 1999 a dynamic environment was proposed independently by two groups of researchers (Branke, 1999; Morrison, K. de Jong 1999) which Branke refers to as: Moving Peak Benchmark and Morrison as DF-1. It is a multidimensional environment consisting of several peaks. These peaks can be changed in various ways such as their location, height and slope. Furthermore one can choose to add or remove peaks in the environment. The environments have been built with the objective to make them easily expandable so any form of dynamic characteristic can be applied.

5. Experimental Setup

For our experiments we compared three different algorithms: a diploid GA using the triallelic dominance mapping, a Hypermutation GA and our Serial Population Algorithm. We tested these algorithms on several different environments. The environment we used for the bulk of our experiments was a dynamic 0/1 knapsack problem that can be classified as a recurrent, discontinuous optimization problem. Additionally we used a dynamic 0/1 knapsack problem where we swapped two items for a slightly more continuous and non-recurrent environment.

5.1 Dynamic 0/1 Knapsack Problem

In order to investigate SPA's basic characteristics we perform several different experiments using a dynamic knapsack problem. Each experiment consists of ten evolutionary runs lasting for 2000 generations each. For the bulk of our experiments we use three different set *sizes* containing 17, 50 or 150 items. With each set size we altered two conditions:

- The *amount* of sets used; using 2 and 5 sets of items each containing weights and values.
- The *duration* of the *stationary period*; P = 10, P = 25 or P = 50 generations.

Additionally we perform experiments using 15 sets with size 50 to see how well the algorithm performs with many optima. These experiments last for 3000 generations and have stationary periods of again; P = 10, P = 25 or P = 50 generations. These tasks can be described as recurrent and discontinuous. Furthermore, we perform experiments using only one set with size 50 where we generated a change by swapping two items in the sets. This creates a genotypic difference with a hamming distance[5] of size 2 while leaving the optimal knapsack value unchanged. Our aim is to create a somewhat more continuous environment by having state dependent changes although it is not continuous in the strict sense for it doesn't change every timestep. Again these experiments consist of ten evolutionary runs lasting for 2000 generations with stationary periods of P = 10, P = 25 or P = 50 generations. Finally we perform experiments with 5 sets of size 50 but every time we revisit the set we swap two items so the optimum is slightly different each time. These experiments again last for 2000 generations with again stationary periods of P = 10, P = 25 or P = 50 generations.

[5] Hamming distance = number of differences between two binary strings.

5.2 The Algorithms

For each of the three algorithms we use the same genetic operators as much as possible in order to keep things as equal as possible. The only differences are: a triallelic encoding for the diploid algorithm, a hypermutation phase for the Hypermutation algorithm and the use of serial subpopulations for SPA. The parts that are equal are: child- and parent population sizes, tournament sizes, crossover probability, mutation rate, and not allowing multiple instances in the parent population. In the case of five sets with size 50 we also ran an additional 'basis GA' that used only these operators.

5.2.1 General Settings

We set the following parameters for all algorithms:

- (sub) population size = 50
- child population size = 150
- tournament size:
  - mating selection = 4
  - replacement selection = 8
- crossover probability $P_c = 0.9$
- mutation rate = 1/setsize

These settings were found to give good results on preliminary tests. We do not claim that these settings are optimal for any of the different environments we test the algorithms on. Using these settings it means that if SPA uses five subpopulations, its total population size will be 250. It appears to give SPA a major advantage over the other algorithms but in reality this is limited. As we mentioned in section 2.1 the evaluations usually take the longest time of the entire process, particularly in real world applications. The following three equations give the average number of evaluations per generation for each algorithm:

Eq. 5.1: $\frac{(S-1)*I}{P}+1+I+C=E$ SPA

Eq. 5.2: $1+I+C=E$ HMGA

Eq. 5.3: $I+C=E$ Diploid GA

Where:
S = number of subpopulations used in SPA.
I = number of individuals in each (sub-) population.
C = number of individuals in each child population.
P = duration of stationary period, measured in generations.
E = average number of evaluations in each generation.

After each detected change SPA's entire population will be evaluated. On average this results in Eq. 5.4 evaluations per generation. The detection system itself accounts for one extra evaluation per generation and evaluating the parent- and child populations are equal for all GA's (Eq. 5.3). The Hypermutation Algorithm does not have the subpopulations but it does have the detection system, resulting in Eq. 5.4 fewer evaluations per generation than SPA.

Eq. 5.4: $\frac{(S-1)*I}{P}$

The Diploid GA has no detection system nor does it use the subpopulations resulting in Eq. 5.5 fewer evaluations per generation than SPA.

Eq. 5.5: $\frac{(S-1)*I}{P}+1$

This means that if SPA uses five subpopulations and the stationary period is fifty generations, it will have four more evaluations than HMGA and five more than the Diploid GA. The number of evaluations per generation will go up either by increasing the number of subpopulations or by increasing the frequency of environmental change. To counterbalance this advantage of SPA we increased the populations of the Hypermutation GA and the Diploid GA by the appropriate numbers. So for this example HMGA would have 54 individuals and the Diploid GA 55.

5.2.2 The Hypermutation Algorithm

We altered the Hypermutation algorithm slightly compared to what is common. Normally, between mutation bursts, HMGA uses a very low base mutation rate and depends mostly on crossover and a large population size to find solutions. In our experiments we used the base rate of 1/L where L is the length of the chromosome. Because our population is smaller than normally used in hypermutation experiments the extra mutation is needed to compensate the loss of variation due to the population size. The mutation burst is set to be roughly 35%. This is comparable to the burst size used in Lewis et al. (1998) and to the theory that high frequencies require high mutation rates (Morrison, K. de Jong, 2000). The burst is followed by a period of linear decay. Two generations after the initial burst the mutation rate is back on the base rate. Also the detection system is different than the one used by Cobb. Our detection system has a 100% chance of detecting a change in environment. This is in part caused by the way we change our environment. If we would have altered the maximal allowed weight there would have been a chance that the change would go unnoticed. For these experiments we used the same detection system as SPA that we described earlier.

5.2.3 THE DIPLOID GENETIC ALGORITHM

The diploid genetic algorithm uses Hollstein's triallelic dominance scheme as described earlier. Apart from this and the fact it doesn't use any subpopulations the algorithm is the same as SPA.

5.3 The Performance Measures

We use two criteria to measure the performances of the algorithms; Accuracy (Acc) and Adaptability (Ada) as described by Simões & Costa (2003a & 2003b) but with a slight alteration. Accuracy measures the difference between the optimal value of that period and the best individual in the last generation before the change. We altered this slightly by taking this difference as a percentage of the optimum. This is especially useful when comparing results of tests with large differences in set size what can result in large differences in optimal values.

Acc: $\frac{1}{R}\sum_{i=1}^{R}\left[\frac{1}{K}\sum_{c=1}^{K}E_c^{'}\right]$

Where: $E_c^{'} = \frac{optimum_{c*p} - best_{c*p}}{optimum_{c*p}}$

p = the number of generations between each change.
K = the number of changes in each evolutionary run.
R = the number of evolutionary runs per experiment.

Adaptability is similar to what is commonly known as the mean fitness error. We measure the difference between the best individual of each generation with the optimum value of that period. It gives us an indication of the speed of recovery of the algorithm. These two measurements should be as close to zero as possible. The values that are shown in the tables are the averages over 10 evolutionary runs per experiment.

Ada: $\frac{1}{R}\sum_{j=1}^{R}\left[\frac{1}{g}\sum_{i=1}^{g}E_i\right]$

Where: $E_i = \frac{optimum_i - best_i}{optimum_i}$

g = the number of generations in each evolutionary run

6. Results

Until now we have seen what the algorithms we use look like, what type of environments we use and the performance measures. In this chapter we discuss the results of our experiments and draw our conclusions from them for our Serial Population Algorithm on how it functions on such tasks. The tables which we refer to are given in the appendix.

6.1 Recurrent, discontinuous tasks with set size 17

The results for these experiments are shown in tables 2, 3, 11, 12.
The optimization problem with set size 17 is apparently rather easy to find for both HMGA and SPA. In all cases the accuracy is either zero or near zero. In table 3 we see that SPA has a small error with P=50 where HMGA does not. Here SPA was not able to find one of the optima on one run. An error of 0.009% is hardly significant.
The Diploid algorithm on the other hand even has significant problems finding this simple solution. It has an incredibly low adaptability, resulting in a low accuracy. Probably the main reason for this lies in the redundancy in its genetic code. If we look in figure 6.1 at the lines for the best individual and the average of the population we notice that even though all individuals in the population are genetically unique, they still are able to have the same phenotype. The redundancy in the triallelic mapping allows for many different genotypes to have the same phenotype. This means that all individuals are located on the same location in the search space, thereby loosing the power to search the space in parallel. This may be resolved by using a much larger population but that would result in doing many more evaluations. It does not mean that triallelic diploid algorithms are useless. They may have potential where evaluation time is hardly an issue and where population sizes can be large and where genetic diversity combined with phenotypic singularity is needed. But in our case, where the number of evaluations and the speed of adaptation do matter, they seem to be misplaced.

Probably the most noticeable result of the experiments with set size 17 can be seen in table 12 where SPA scored worse on the longest period than on the two shorter periods. Because this is, of all the experiments we've done, the only occasion this occurs we feel that we can say that it was an exception on the rule.

6.2 Recurrent, discontinuous tasks with set size 50

The results for these experiments are shown in tables 4, 5, 13, 14.
For the optimization problem with set size 50 we performed experiments with one extra algorithm. This 'basis GA' is not to be confused with Holland's GA, using proportionate selection etc. (see chapter 2) but this algorithm forms the basis for all other algorithms we used. So it basically is SPA without the subpopulations or the HMGA without a hypermutation burst or the Diploid GA with a haploid representation. By comparing the

results of this ‘basis GA’ with the others we get some more insight in the effects of the add-ons of the other GAs. Both SPA and HMGA improve over the ‘basis GA’ but the diploid encoding makes it perform worse.

For this problem SPA outperforms the HMGA significantly on almost all fronts. It is only that the HMGA gets a near perfect score on the P = 50 that we see a small difference. It is interesting to notice that the HMGA’s score hardly varies whether we use two optima or five. Any change for the HMGA is equally disrupting and neither the adaptability score nor the accuracy is really affected by the number of optima. SPA on the other hand does show a strong difference. The more optima are to be found and the more subpopulations there are, the longer it takes for SPA to stabilize. More often one subpopulation will be used for different optima before settling for one single optimum as illustrated in figure 6.2.

6.3 Recurrent, discontinuous tasks with set size 150

The results for these experiments are shown in tables 6, 7, 15, 16
The HMGA makes a remarkable jump in adaptability when we compare the P=50 with the other two periods. A possible answer is that the algorithm starts to reach the optimum somewhere after 25 generations thus lowering the adaptability considerably. Once the optimum is found there is nothing to adapt to anymore. Again we can see that SPA’s performance improves when the problem has only two optima unlike HMGA adaptability which is not affected by the number of optima.

6.4 Non-recurrent, small changes

The results for these experiments are shown in tables 8, 17.
The HMGA performs much better on this task than on the tasks with large jumps. SPA still performs well although it is worse than when we compare it to the results from our discontinuous changing environment. As we can see from figure 6.3 SPA uses in this case only one subpopulation. Apparently the changes are not big enough to force the algorithm to use multiple subpopulations. This means that SPA performs equal to the Basis GA on this task. Because no multiple subpopulations are used, which is SPA’s identifying feature, SPA becomes useless for this case. Any additional subpopulation only increases the number of evaluations that need to be performed without adding any functionality.

A small caution is in place here when considering the performance differences between the algorithms. The performance of all algorithms can be improved upon when the different settings of the algorithms are better tuned to the problem. All algorithms used the same settings in the small changes task as with the discontinuous tasks. Adding the hypermutation phase here apparently worsened the performance as we know that in this case the Basis GA performs exactly the same as SPA. Adjusting the duration of the hypermutation phase and the height of the mutation rate may cause the algorithm to

improve beyond the basis GA performance. But we can still safely say that SPA performs much better on the discontinuous task than either of the other two algorithms we tested.

6.5 Recurrent, discontinuous tasks with set size 50 with small changes

The results for these experiments are shown in tables 9, 18.
SPA is still able to stabilize and attach its subpopulations to the different optima although in some cases this linkage is sometime disturbed after a stable period. (see figures 6.4 and 6.5) This is something that is expected when we consider the findings of SPA's behaviour on our regular discontinuous tasks and the small changes task. The fact that SPA is capable of finding the old optimum even when the old optimum is changed slightly is probably more caused by the fact that all individuals are unique. This is a powerful method which keeps the diversity high while keeping the number of individuals, thus also the number of evaluations, low.

6.6 Recurrent, discontinuous tasks with set size 50 using 15 sets

The results for these experiments are shown in tables 10, 19.
Even with fifteen different sets the SPA algorithm does remarkably well. This is mainly due to the fact that some of the subpopulations do converge to a single optimum. This already gives a big improvement on the performance because their accuracy is (near) perfect.

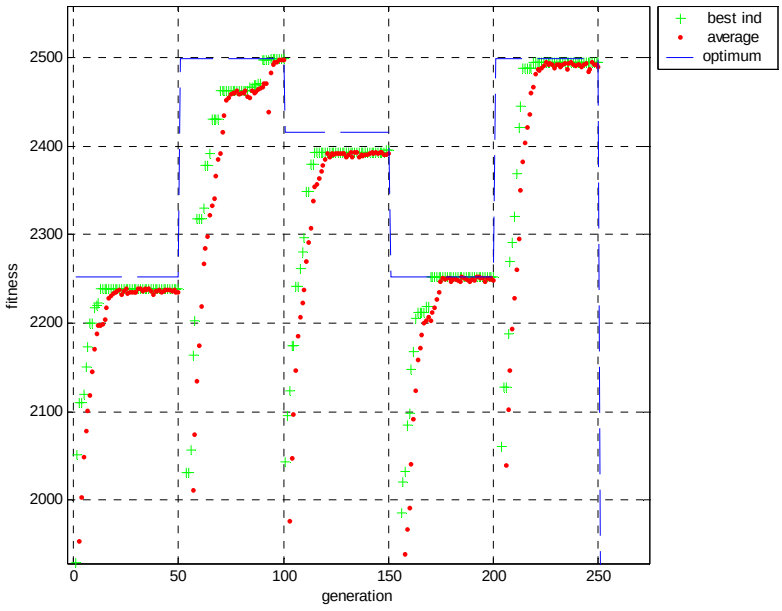

Figure 6.1 *A close-up of an evolutionary run with the Diploid GA. The average of the population (depicted by the red dots) often has the same fitness as the best individual (green crosses)*

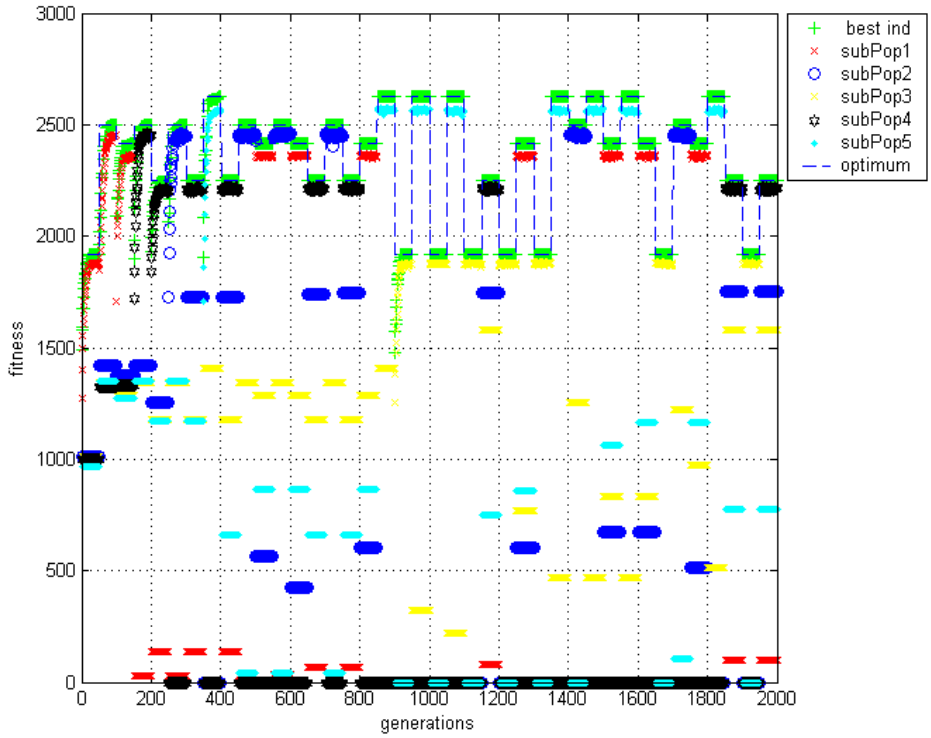

Figure 6.2 *SPA finding the different optima. The green crosses depict the current best individual. The dashed blue line shows where the optimum lies. The remaining circle, stars etc. show the subpopulations' average fitness.*

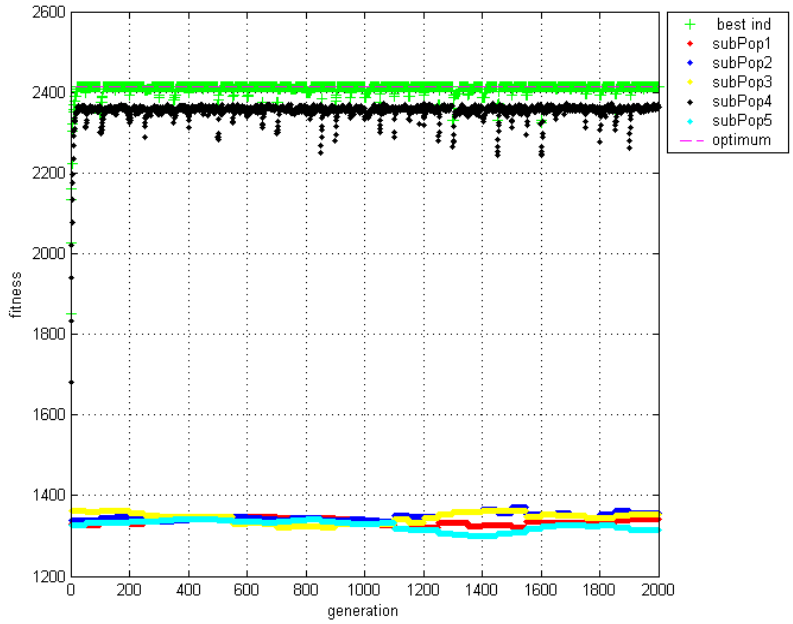

Figure 6.3 *When the optimum changes by a only small margin SPA always uses the same subpopulation.*

The figures 6.1 – 6.5 are best viewed in color

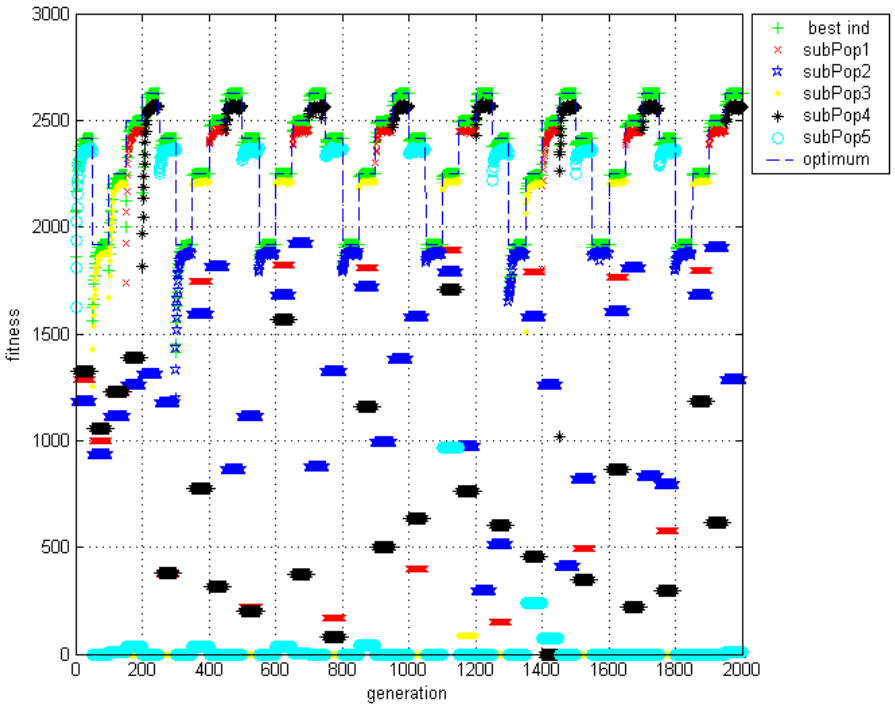

Figure 6.4 *The optima are slightly different each time they are revisited. This causes extra adaptations.*

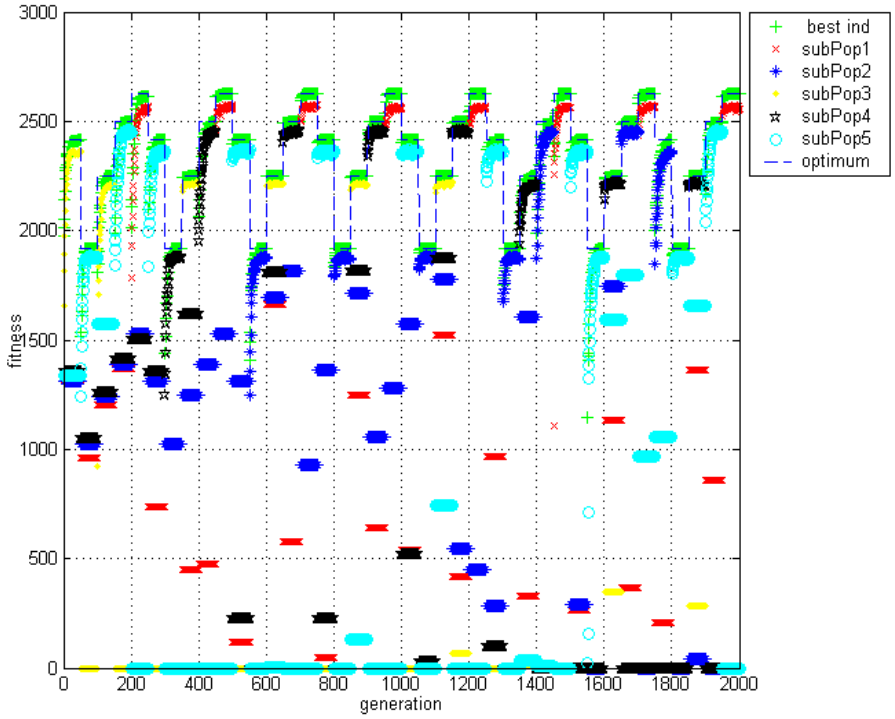

Figure 6.5 *The small changes sometimes can cause SPA to become unstable.*

7. Discussion

So what can we conclude based upon the results discussed above for the functioning of SPA?

7.1 Stabilization Period

The experiments show that the use of serial subpopulations has the effect of each subpopulation tending to converge toward a single optimum. This enables SPA to quickly regain the former solution in recurrent problems. This linking of a subpopulation with an optimum is not explicitly encoded in the algorithm but results from SPA's inner workings. By using only one subpopulation during a stationary period we cause the subpopulation to converge to that optimum. This convergence increases the likelihood of the subpopulation to contain an individual which is closer to the optimum than any other individual in the entire population. At the same time it reduces the chance that the subpopulation contains an individual which is closer to one of the other optima. Because we select the subpopulation based on the presence of the individual that is nearest to the current optimum, the longer a subpopulation has been used for one optimum the more likely it will be selected next time that optimum comes around. The time between the start of the algorithm and the time each subpopulation is linked with one optimum we call the 'Stabilization Period'. In figure 7.1 you can see subpopulation 1 first being used to search for two different optima before settling for the third and final optima.

The duration of this stabilization period is mainly influenced by the number of sets and subpopulations. As we can see in figure 7.1, when there are many different optima, it is likely some of the subpopulations will be used multiple times for different optima before settling for one optimum. In figure 7.2 there are only two optima and the stabilization period is very short. An additional influence is the amount of time a subpopulation gets to converge to an optimum. If the stationary periods are short, it may not have enough time to learn the specifics of that optimum and it turns out to be the best for a different optimum as well.

7.2 Exploitation Period

After the stabilization period both the adaptability and the accuracy will almost always remain at zero, given a simple recurrent, discontinuous task. All subpopulations are linked with their own unique optimum and the best individual is located at that optimum. It is from this point on that SPA really takes off and the longer an evolutionary run lasts the better its overall performance will be. The continuing exploitation is not guaranteed as the factors which influence the duration of the stabilization period may cause some instability later on.

7.3 Subpopulations and Optima

When the number of subpopulations is equal to the number of optima the algorithm will end up assigning a single subpopulation to each optimum. When there are more subpopulations than optima these extra subpopulations will not be used. This surplus will not be used for any of the optima as can be seen in figure 7.3. Although it doesn't affect the learning speed of the algorithm it does cause extra evaluations every time the entire population is evaluated. When applying genetic algorithms to real world problems you want to keep the number of evaluations as low as possible because generally the evaluations take up most of the algorithm's time.

When the number of subpopulations is smaller than the number of optima the algorithm becomes less stable. Some of the subpopulations may still be associated to a single optimum but others will switch from one optimum to the next. Figure 7.4 shows you a case where we have four subpopulations and five optima.

7.4 The Detection System Revisited

In chapter 3 we briefly touched upon SPA's detection system. Because of the way we change our optimization task we can see whether a change has occurred simply by checking whether the best individual's fitness (any other individual would do too) has changed. This lets us respond without delay whenever a change occurs. But the success of this detection system is dependent on the way the task is changed. If we would change the maximum allowed weight, like most other researchers have done (Goldberg & Smith, 1987, Simões & Costa, 2003) the lowering of the weight would be detected without many problems. This is because the total weight of the items will very likely be more than the new allowed maximum, thus dropping the fitness. Raising the allowed weight on the other hand won't result in a different fitness because the penalty function does not come into play. It simply allows more room in the knapsack meaning that a different, larger combination of items can be put into the knapsack.

When this detection system is used on the Moving Peaks Benchmark it will detect a change when the peak has relocated or changed its height. But when the old best peak stays the same while another peak grows and becomes the highest point in the fitness landscape, this change will not be detected.

If the fitness function or environment is very noisy, the detection system will no longer function properly. This is because it needs an individual to always have the same fitness in the same environment. If this is not the case, as with noisy environments, it will falsely detect a change and have SPA evaluate its entire population.

7.5 Expandable Core Value

The real core value of SPA is the serial use of the subpopulations. They serve as the memory of the algorithm. By trying to use them for one optimum only we ensure good memory and later exploitation. Another good thing about using serial populations is that it can easily be expanded with other methods. It can be used in combination with diversity maintenance methods such as crowding and fitness sharing or in combination with a diploid encoding. Where you should be hesitant is using it in combination with diversity introducing methods, especially hypermutation. The reason is that diversity introducing methods disturb the memory function of the algorithm.

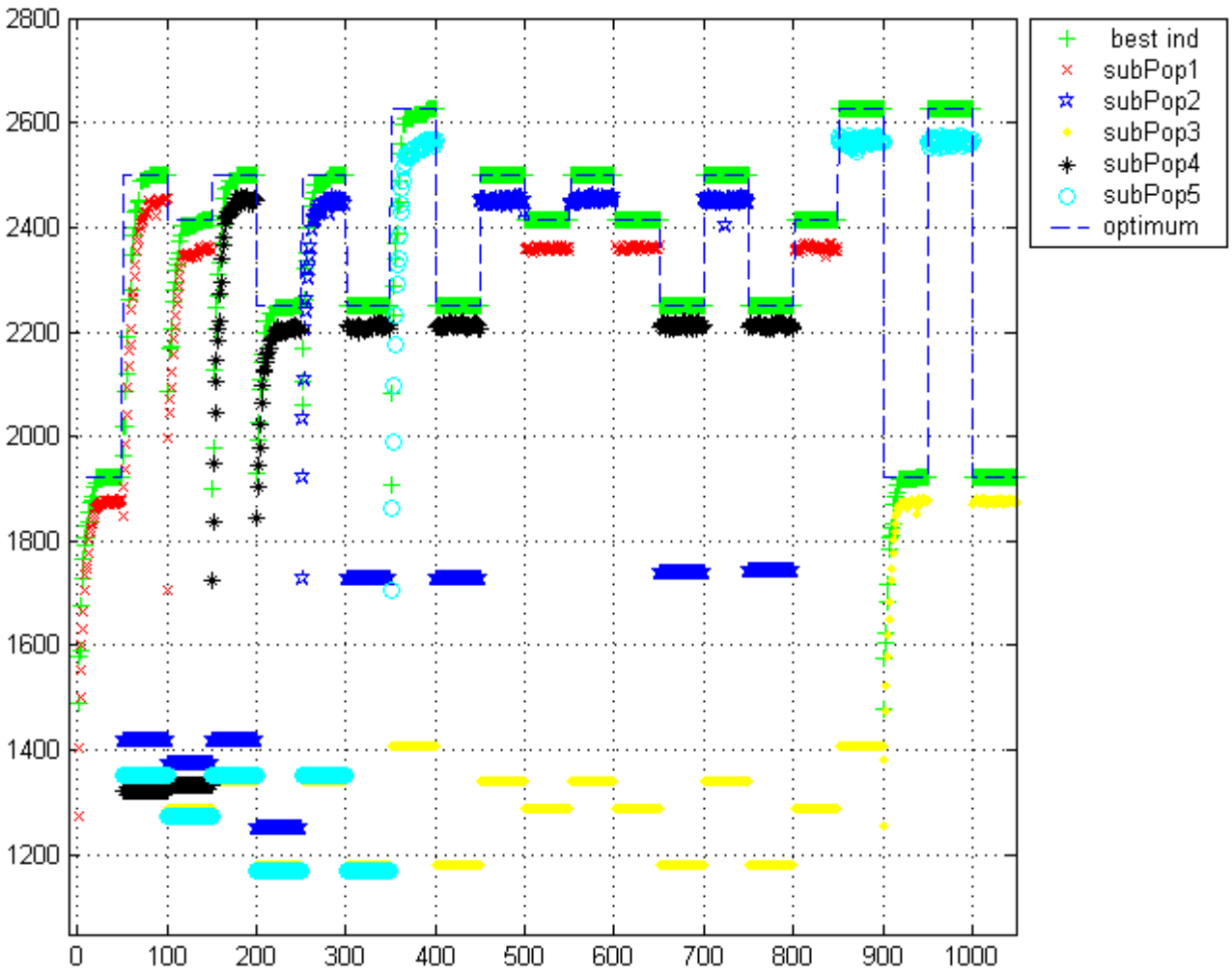

Figure 7.1 *close-up of SPA's stabilization period.*

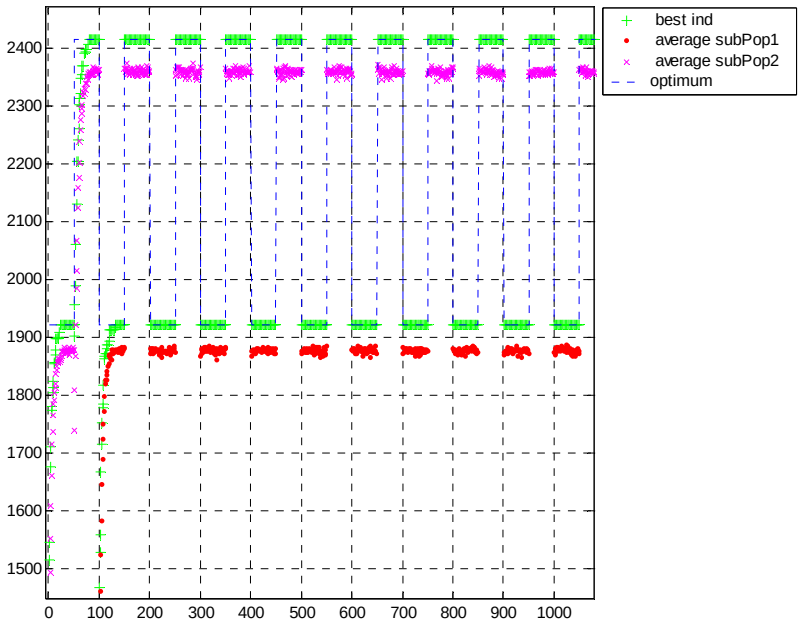

Figure 7.2 *a close-up of an evolutionary run using SPA with two subpopulations and two optima*

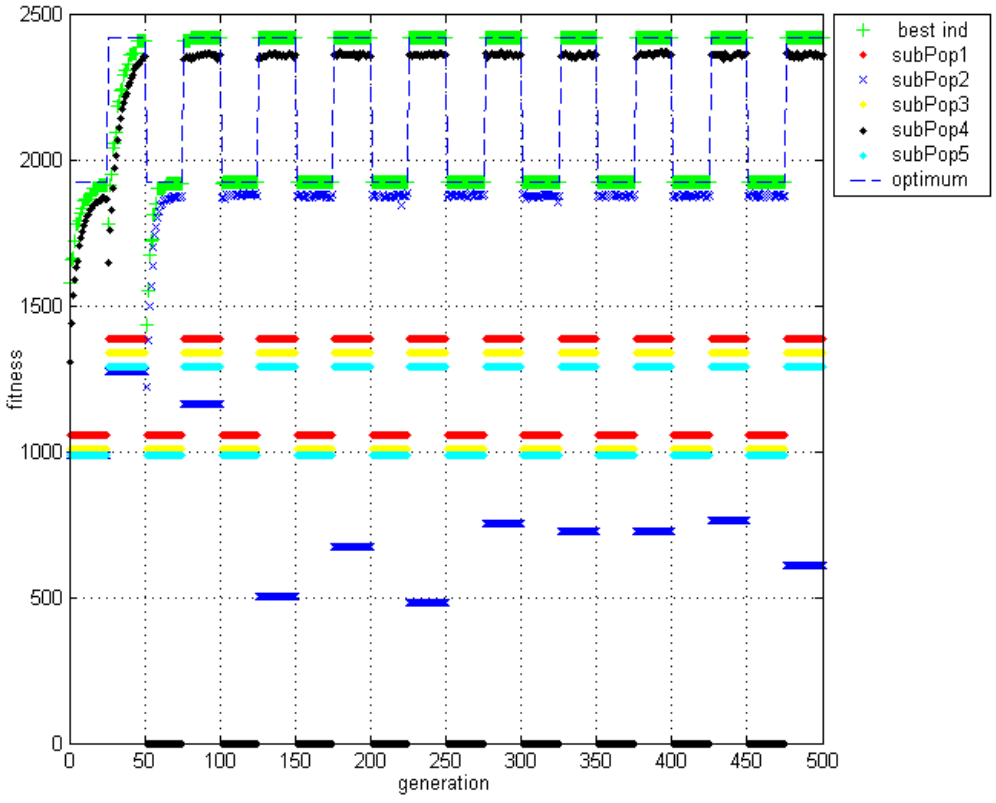

Figure 7.3 *SPA with two optima and five subpopulations*

The figures 7.1 – 7.4 are best viewed in color

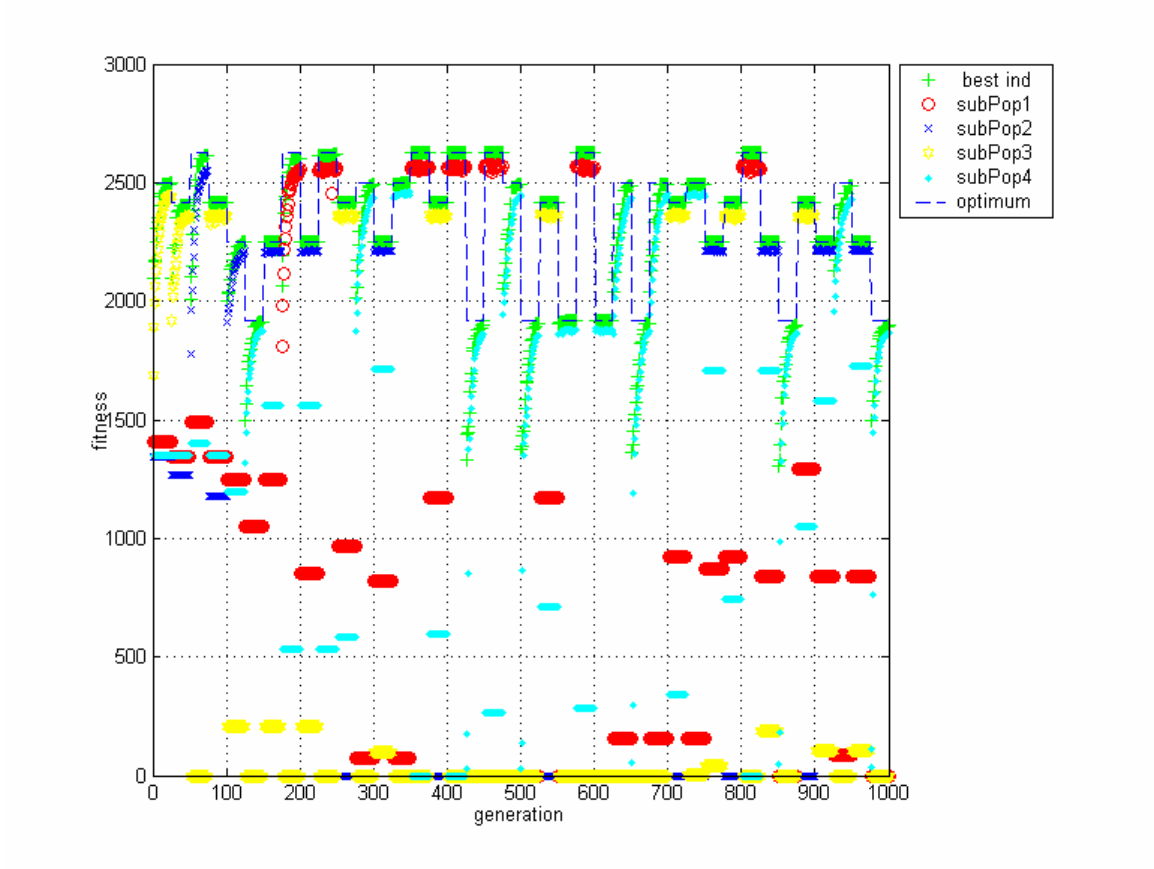

Figure 7.4 *SPA with five optima and four subpopulations.*

8. Conclusions

8.1 Summary

For this thesis we set out to develop a new Genetic Algorithm which can handle dynamic optimization problems, specifically problems that change in a recurrent, discontinuous fashion. We tested our new algorithm, dubbed Serial Population Algorithm (SPA), on a dynamic knapsack problem with several different settings. The settings differed among others in speed of change and size of the search space. SPA's performance was compared to two other genetic algorithms: a hypermutation algorithm and a diploid algorithm which used a triallelic dominance mapping. SPA showed very good results on these tests and outperformed the other two algorithms on nearly all tasks. We will end this thesis with a list of strengths and weaknesses of SPA, look at how its weaknesses can be overcome and suggest some future work.

8.2 Strengths and Weaknesses of SPA

Strengths:

- The dedication of a single subpopulation to an optimum has very positive effects on both the accuracy and the adaptability measures. Because little information gets lost, there is less adaptation needed as with other algorithms, something that rubs off on the accuracy of the algorithm.
- The way SPA uses multiple subpopulations is not limited to GA only, most other classes of EAs can benefit from this system too in cases of discontinuous, periodic environments.
- The way SPA uses multiple subpopulations is expandable with other evolutionary methods like fitness sharing.

Weaknesses:

- Higher frequency of change causes an increase in the average number of evaluations per generation.
- Larger numbers of subpopulations cause an increase in the average number of evaluations per generation.
- Fitness landscapes which are very similar may make it more difficult to link one subpopulation to one optimum.
- The efficiency of SPA is dependent on whether the number of optima and subpopulations match.
- The detection mechanism has limitations on two types of environments it can functionally be used on.

8.3 Countermeasures to SPA's Weaknesses

To reduce the number of evaluations after a change has been detected you could sample the subpopulations instead of evaluating them entirely. The probable effects are that there is a higher chance you don't pick the best subpopulation. But by sorting the subpopulation at the end of each generation on their fitness you could take the top X %, ensuring the previous best are selected. If you take their average you still may improve the chances of picking the best subpopulation.

Fitness landscapes which are difficult to distinguish can cause the stabilization period to last longer. A countermeasure may be to have the population spread out less over the representations space. This could be done by allowing multiple instances of individuals in the population or by simply using smaller populations. This will make it less likely that the subpopulations contain individuals that are good solutions for other optima but at the cost of making it harder to find a good solution at all due to the lack of diversity.

The biggest problem SPA faces is the fact you need to know in advance the number of optima you will encounter during the evolutionary run. This is in many cases not possible. To counter this we propose an expansion of SPA in the future work section.

8.4 Future Work

If one doesn't know the number of optima in advance so one can match the number of subpopulations, SPA will become less effective. As future work we will give a description for an expansion of SPA with an explicit memory function.

At the beginning we create only one (sub) population and a memory vector.

At the start of each generation, except for the first generation, we test for changes in the environment using our detection system.

If no change is detected we continue to use the same subpopulation, just as with SPA.

At the end of the generation we store the best individual and its fitness as a tuple in the memory vector, *replacing* the former best individual of that subpopulation.

If a change *is* detected we evaluate all the individuals in the memory vector on the changed environment. If none of the individuals stored in the memory vector come close to their old fitness value, say score less than 80% of their former fitness, a new subpopulation is created together with an extra space in the memory vector. This new subpopulation will be used until a change is detected.

At the end of each generation the best individual and its fitness are stored in the memory vector.

So to be clear: the memory vector stores the last best individual of each subpopulation. So if there are S subpopulations the memory vector will contain S tuples.

This algorithm is as yet untested but we can make some predictions based on our experiments with SPA. The serial uses of the populations will very likely stay intact. So those advantages will remain. But now, because after each change we only evaluate the memory vector instead of the entire population, we greatly reduce the number of evaluations. Where SPA has on average $\frac{(S-1)*I}{P}$ more evaluations per generation than the hypermutation GA, the memory enhanced SPA has only $\frac{(S-1)}{P}$ which is an enormous reduction. This allows the memory enhanced SPA to handle many more optima, much shorter stationary periods and larger subpopulation sizes.

Acknowledgements

I would like to thank Marco Wiering and Edwin de Jong for their many good comments. They steered this research into a direction that took it to the level it is now. Also I would like to thank Albert Visser for taking the time to be the third supervisor. Finally I would like to thank Walter de Back and Tijn van der Zant for it was from our many discussions, our cooperation and our friendship that I learned the most.

References

Bäck T., Fogel D.B., Michalewicz Z. (eds) (1997). Handbook of Evolutionary Computation. Oxford University Press, NewYork.

Bendtsen, C. N. and Krink, T. (2002). Dynamic memory model for non-stationary optimization. In *Congress on Evolutionary Computation*, pages 145-150. IEEE.

Branke, J. (2003). Evolutionary approaches to dynamic optimization problems - introduction and recent trends. In J. Branke, editor, *GECCO Workshop on Evolutionary Algorithms for Dynamic Optimization Problems*, pages 2-4.

Branke, J., Kaußler, T., Schmidt, C. & Schmeck, H. (2000). A multi-population approach to dynamic optimization problems. In *Adaptive Computing in Design and Manufacturing 2000*. Springer.

Calabretta, R., Galbiati, R., Nolfi, S. & Parisi, D. (1996) Two is better than one: A diploid genotype for neural networks, in *Neural Processing Letters, Volume 4, Issue 3*, Pages 149 – 155.

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 AIC-90-001, Naval Research Laboratory, Washington, USA.

Eggermont, J. & Lenaerts, T. (2002) Dynamic Optimization using Evolutionary Algorithms with a Case-based Memory. In *Proceedings of the 14th Belgium Netherlands Artificial Intelligence Conference (BNAIC'02)*

Fraser, A. S. (1957). Simulation of Genetic Systems by Automatic Digital Computers. I. Introduction. In *Australian Journal of Biological Sciences.* Vol 10 pages 484-491

Friedman, G. J. (1956). Selective Feedback Computers for Engineering Synthesis and Nervous System Analogy. *Master's Thesis UCLA*

Giraldez, R., Aguilar-Ruiz, J. & Riquelme, J. (2003). Natural coding: A more efficient representation for evolutionary learning, In *GECCO 2003*: Springer-Verlag Berlin Heidelberg, pages. 979–990.

Goldberg, D. E. & Smith, R. E. (1987). Nonstationary function optimization using genetic algorithms with dominance and diploidy. In J. J. Grefenstette, editor, *International Conference on Genetic Algorithms*, pages 59-68. Lawrence Erlbaum Associates.

Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Westey, Reading Massachusetts.

Grefenstette, J. J. (1992). Genetic algorithms for changing environments. In R. Maenner and B. Manderick, editors, *Parallel Problem Solving from Nature 2*, pages 137-144. North Holland.

Hill, S., Newell, J. & O'Riordan C. (2004). Analyzing the effects of Combining Fitness Scaling and Inversion in Genetic Algorithms. In ICCTAI'04 pages 380-387

Holland, John H. (1975 reprint 1992). Adaptation in natural and artificial systems.

Jones, J. (2005) Court ruling on Intelligent Design: Kitzmiller v Dover Area School District http://www.pamd.uscourts.gov/kitzmiller/kitzmiller_342.pdf

Lewis, J., Hart, E. & Ritchie. G. (1998). A comparison of dominance mechanisms and simple mutation on non-stationary problems. In A. E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature*, number 1498 in LNCS, pages 139-148. Springer.

Morrison, R. W. and DeJong, K. A. (1999). A test problem generator for non-stationary environments. In *Congress on Evolutionary Computation*, volume 3, pages 2047-2053. IEEE.

http://www.cs.uwyo.edu/~wspears/morrison/DF1-MatlabCode.txt

Morrison, R. W. & De Jong, K. A. (2000). Triggered hypermutation revisited. In *Congress on Evolutionary Computation*, pages 1025-1032.

Ng, K. P. & Wong, K. C. (1995). A new diploid scheme and dominance change mechanism for non-stationary function optimization. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 159-166. Morgan Kaufmann.

Simões, A. & Costa, E. (2003a). An immune system-based genetic algorithm to deal with dynamic environments: Diversity and memory. In D. W. Pearson, N. C. Steele, and R. Albrecht, editors, *Proceedings of the Sixth international conference on neural networks and genetic algorithms (ICANNGA03)*, pages 168-174. Springer.

Simões, A. & Costa, E. (2003b). A comparative study using genetic algorithms to deal with dynamic environments. In D. W. Pearson, N. C. Steele, and R. Albrecht, editors, *Proceedings of the sixth international conference on neural networks and genetic algorithms (ICANNGA03)*, pages 203-209. Springer.

Smith, R. E. & Goldberg, D. E. (1992). Diploidy and Dominance in Artificial Genetic Search. In *Complex Systems*, Vol. 6, pages. 251-285.

Thompson, A. (1997). An evolved circuit, intrinsic in silicon, entwined with physics. In Higuchi, T and Iwata, M. and Weixin, L. editors *Proceedings of the first International Conference on Evolvable Systems ICES '96.* pages 390-405, Springer-verlag.

Ursem, R. K. (2000). Multinational GA optimization techniques in dynamic environments. In D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, editors, *Genetic and Evolutionary Computation Conference*, pages 19-26. Morgan Kaufmann.

Vavak, F., Jukes, K. A. & Fogarty, T. C. (1998). Performance of a genetic algorithm with variable local search range relative to frequency for the environmental changes. In Koza et al., editor, *International Conference on Genetic Programming*. Morgan Kaufmann.

Zwanepol Klinkmeijer, L., de Jong, E. D. & Wiering, M. A. (2006) A Serial Population Population Genetic Algorithm for Dynamic Optimization Problems, *In proceedings of the fifteenth annual machine learning conference of Belgium and The Netherlands.* pages 41-48.

Appendix

Accuracy:

Table 2. Accuracy (%) on 5 sets of size 17

	P = 10	P = 25	P = 50
Diploid	0.8515	0.5484	0.3646
HMGA	0.1087	0.0010	0.0000
SPA	0.0222	0.0011	0.0355

Table 3. Accuracy (%) on 2 sets of size 17

	P = 10	P = 25	P = 50
Diploid	1.1878	0.3108	0.1261
HMGA	0.0128	0	0
SPA	0.0021	0	0.009

Table 4. Accuracy (%) on 5 sets of size 50

	P = 10	P = 25	P = 50
Diploid	7.988	1.8547	0.7381
HMGA	3.65	0.2609	0.0151
SPA	0.305	0.045	0.0059
Basis GA	5.701	0.40087	0.0354

Table 5. Accuracy (%) on 2 sets of size 50

	P = 10	P = 25	P = 50
Diploid	6.4238	2.0139	0.8077
HMGA	4.0406	0.2884	0.0151
SPA	0.0565	0.0122	0.0015

Table 6. Accuracy (%) on 5 sets of size 150

	P = 10	P = 25	P = 50
Diploid	21.414	8.333	3.440
HMGA	9.985	3.985	2.020
SPA	1.789	1.194	0.688

Table 7. Accuracy (%) on 2 sets of size 150

2 optima	P = 10	P = 25	P = 50
Diploid	15.1685	4.1251	2.1364
HMGA	9.6863	3.5759	1.8728
SPA	0.6822	0.4651	0.3254

Table 8. Accuracy (%) for ***small changes*** on size 50

	P = 10	P = 25	P = 50
HMGA	0.6907	0.0347	0.0048
SPA	0.3108	0.0246	0.0021

Adaptability:

Table 11. Adaptability (%) on 5 sets of size 17

	P = 10	P = 25	P = 50
Diploid	11.6612	5.8761	3.4162
HMGA	4.6372	1.8022	1.0003
SPA	0.3821	0.2250	0.2233

Table 12. Adaptability (%) on 2 sets of size 17

	P = 10	P = 25	P = 50
Diploid	9.3527	5.2123	2.9160
HMGA	3.7169	1.5001	0.7430
SPA	0.0586	0.0523	0.0785

Table 13. Adaptability (%) on 5 sets of size 50

	P = 10	P = 25	P = 50
Diploid	19.16	15.56	9.37
HMGA	13.56	6.73	3.45
SPA	0.73	0.60	0.48
basis GA	15.26	5.29	5.28

Table 14. Adaptability (%) on 2 sets of size 50

	P = 10	P = 25	P = 50
Diploid	19.33	16.91	10.96
HMGA	15.28	7.30	3.71
SPA	0.15	0.16	0.17

Table 15. Adaptability (%) on 5 sets of size 150

	P = 10	P = 25	P = 50
Diploid	23.3179	24.039	21.8521
HMGA	10.8078	10.2988	7.2905
SPA	1.9312	1.9982	1.7655

Table 16. Adaptability (%) on 2 sets of size 150

	P = 10	P = 25	P = 50
Diploid	15.7225	15.7520	17.1179
HMGA	10.5597	9.7956	6.9827
SPA	0.7167	0.7323	0.6807

Table 17 Adaptability (%) ***small changes*** on size 50

	P = 10	P = 25	P = 50
HMGA	1.3635	0.5520	0.3275
SPA	0.7235	0.3234	0.1813

Table 9 Accuracy (%) Size 50, 5 sets + small changes

acc	P = 10	P = 25	P = 50
HMGA	3.8704	0.3219	0.0184
SPA	0.7365	0.0908	0.0103

Table 10 Accuracy (%) size 50, using 15 sets

	P=10	P=25	P=50
HMGA	4.3123	0.4179	0.0751
SPA	0.8921	0.1790	0.0420

Table 18 Adaptability (%) Size 50, 5 sets + small changes

	P = 10	P = 25	P = 50
HMGA	9.6976	4.7099	2.2612
SPA	1.5224	0.9935	0.9635

Table 19 Adaptability (%) size 50, using 15 sets

	P=10	P=25	P=50
HMGA	10.9991	5.5424	2.9359
SPA	1.6580	1.2717	1.1623