

Genetic Algorithms for Optimisation

Background and Applications

Xavier Hüe

Edinburgh Parallel Computing Centre

The University of Edinburgh

*Version 1.0
February 1997*

available from: <http://www.epcc.ed.ac.uk/epcc-tec/documents/>



Table of Contents

1	Introduction	1
1.1	From Biology	1
1.2	... To Genetic Algorithms	2
1.3	Genetic Algorithms World	4
2	How do Genetic Algorithms work	7
2.1	Genetic representation	7
2.2	Fitness Function	8
2.3	Population	8
2.4	Selection	9
2.5	Recombination	10
2.6	Mutation	11
2.7	Replacement	11
2.8	Convergence	12
3	Comparison with other optimisation techniques	15
3.1	Random Search	15
3.2	Stochastic Hill Climbing	16
3.3	Simulated Annealing	16
3.4	The No Free Lunch Theorem	18
4	Why do Genetic Algorithms work	19
5	Parallel Genetic Algorithms	21
5.1	Global Parallelisation	21
5.2	Coarse Grained PGAs - The Island Model	22
5.3	Fine Grained PGAs	24
6	Applications	27
6.1	Numerical Function Optimisation	27
6.2	Scheduling	28
6.3	Order based problems and the TSP problem	31
6.3.1	The Permutation Representation	32
6.3.2	Other Representations	33
6.4	Configuration of Neural Networks	34
6.5	Rules Discovering	35
6.6	Image analysis with Genetic Programming	35
7	Getting more information	37
7.1	Books	37

7.2	News Group and FAQ.....	37
7.3	From the Web	37
8	References	39

1 Introduction

In *The Origin of Species* [9], Charles Darwin stated the theory of natural evolution. Over many generations, biological organisms evolve according to the principles of natural selection like “survival of the fittest” to reach some remarkable forms of accomplishment. The perfect shape of the albatross wing, the efficiency and the similarity between sharks and dolphins and so on, are good examples of what random evolution with absence of intelligence can achieve. So, if it works so well in nature, it should be interesting to simulate natural evolution and try to obtain a method which may solve concrete search and optimisation problems.

In nature, individuals in a population have to compete with each other for vital resources such as food or shelter. Also in the same species, individuals often have to compete to attract mates for reproduction. Because of such selection, poorly performing individuals have less chance to survive, and the most adapted, or “fit” individuals produce a relatively larger number of offsprings. We also can suppose that during reproduction, a recombination of the good characteristics of each ancestor can produce “superfit” offspring, whose fitness is greater than that of either parent. After a few generations, species evolve spontaneously to become more and more adapted to their environment.

In 1975, Holland developed this idea in *Adaptation in natural and artificial systems* [17]. By describing how to apply the principles of natural evolution to optimisation problems, he laid down the first Genetic Algorithm. Holland’s theory has been further developed and now Genetic Algorithms (GAs) stand up as a powerful adaptive method to solve search and optimisation problems. Today, GAs are used to resolve complicated optimisation problems, like for example, timetabling, job-shop scheduling, games playing.

1.1 From Biology ...

So Genetic Algorithms are directly derived from Natural Evolution. For a better understanding of this theory, it is important first to introduce the biological terminology used in evolutionary computation.

The *genes* code the characteristics of an individual. A gene can take different values or *alleles*. For example, we consider that there is a gene for eye colour, and all the different possible alleles are black, brown, blue and green (since nobody has red or yellow eyes!).

The *gene pool* is the set of all possible alleles present in a particular population. This gene pool can determine all the different possible variations for future generations. The diversity of the individuals in the population is directly determined by the size of the gene pool.

A *genome* is the set of all the genes of a specific species. The genome represents the vehicle of the hereditary determinant. The position of the gene in the genome is referred to as the *locus*. In fact, most living organisms store their genome on several *chromosomes*, but in the Genetic Algorithms, to simplify the representation, all the

genes are usually stored on the same chromosome. So chromosomes and genomes are often used as synonymous for people in GAs.

For a particular individual, we use the word *genotype* to describe the set of its genes. The *phenotype* describes the physical aspect of an individual, the make-up. The *morphogenesis* is the process of decoding a genotype to produce the phenotype. One interesting point of evolution is that selection is always done on the phenotype whereas the reproduction recombines genotypes. Consequently morphogenesis should be considered as something quite important as a link between selection and reproduction.

In higher lifeforms, chromosomes contain two sets of genes. This is known as *diploidy*. In the case of conflict between two values of the same pair of genes, the dominant one will determine the phenotype, whereas the other one, called recessive, will still be present and can be passed on to the offspring. Diploidy allows a wider diversity of alleles to be kept in the population. It can provide a useful memory mechanism in a changing or noisy environment. However, most Genetic Algorithms concentrate on *haploid* chromosomes mainly because they are much more simple to construct. In haploid representation only one set of each gene is stored, thus the not-so-easy question of determining which allele should be dominant and which one should be recessive is avoided.

Recombination or sexual reproduction, is a key operator for natural evolution. Technically, it takes two genotypes and it produces a new genotype by mixing the genes found in the originals. In biology, the most common form of recombination is crossover. In a crossover, two chromosomes are cut at one point and the halves are spliced to create new chromosomes. The effect of recombination is very important because it allows characteristics from two different parents to be assorted. If the father and the mother possess different good qualities, we would expect that all the good qualities will be passed onto the child. Thus the offspring, just by combining all the good features from its parents, may surpass its ancestors. Many people believe that this mixing of genetic material via sexual reproduction is one of the most powerful features of Genetic Algorithms. As a quick parenthesis about sexual reproduction, Genetic Algorithms representations usually does not differentiate male and female individuals (without any perversity). As in many living species (eg snails) any individual can be either a male or a female. In fact, for almost all recombination operators, mother and father are interchangeable.

Mutation is the other way to get new genomes. Mutation consists in changing the value of genes. In natural evolution, mutation mostly engenders non-viable genomes. Actually mutation is not a very frequent operator in natural evolution. Nevertheless, in optimisation, a few random changes can be a good way of exploring the search space quickly.

Through those low-level notions of genetic, we have seen how living beings store their characteristic information and how this information can be passed onto their offspring. It very basic but it is more than enough to understand the Genetic Algorithm Theory.

1.2 ... To Genetic Algorithms

Basically, an optimisation problem looks really simple. You know (or you are able to know) the form of all possible solutions corresponding to a specific question. The set of all the solutions that meet this form constitute what we call the search space. The problem consists in finding out the solution that fits the best, i.e. the one with the most payoffs, from all the possible solutions. If you are able to quickly enumerate all the solutions, the problem does not raise much difficulty. But, when the search space becomes large, enumeration is soon no longer feasible simply because it would take far too much time. In this case you need to use a specific technique to find the optimal solution. Genetic Algorithms provides one of these methods. Practically they all

work in a similar way, adapting the simple genetics of the previous paragraph to algorithmic mechanisms.

GA handles a population of possible solutions. Each solution is represented through a chromosome, which is just an abstract representation. Coding all the possible solutions into a chromosome is the first part, but certainly not the most straightforward one of a Genetic Algorithm. A set of reproduction operators has to be determined, too. Reproduction operators are applied directly on the chromosomes, and are used to perform mutations and recombinations over solutions of the problem. Appropriate representation and reproduction operators are really something determinant, as the behaviour of the GA is extremely dependant on it. Frequently, it can be extremely difficult to find a representation which respects the structure of the search space and reproduction operators which are coherent and relevant according to the properties of the problems.

Selection is supposed to be able to compare each individual in the population. Selection is done by using a fitness function. Each chromosome has an associated value corresponding to the fitness of the solution it represents. The fitness should correspond to an evaluation of how good the candidate solution is. The optimal solution is the one which maximizes the fitness function. Genetic Algorithms deal with problems that maximize a fitness function. But, if the problem consists in minimising a cost function, the adaptation is quite easy. Either the cost function can be transformed into a fitness function, for example by inverting it; or the selection operators can be adapted in such way that they consider individuals with low evaluation functions as better.

Once the reproduction and the fitness function have been properly defined, a Genetic Algorithm is evolved according to the same basic structure. It starts by generating an initial population of chromosomes. This first population must offer a wide diversity of genetic materials. The gene pool should be as large as possible so that any solution of the search space can be engendered. Generally, the initial population is generated randomly.

Then, the genetic algorithm loops over an iteration process to make the population evolve. Each iteration consists of the following steps:

- **SELECTION:** The first step consists in selecting individuals for reproduction. This selection is done randomly with a probability depending on the relative fitness of the individuals so that best ones are more often chosen for reproduction than poor ones.
- **REPRODUCTION:** In the second step, offspring are bred by the selected individuals. For generating new chromosomes, the algorithm can use both recombination and mutation.
- **EVALUATION:** Then the fitness of the new chromosomes is evaluated.
- **REPLACEMENT:** During the last step, individuals from the old population are killed and replaced by the new ones.

The algorithm is stopped when the population converges toward the optimal solution.

```
BEGIN /* genetic algorithm*/

    Generate initial population ;
    Compute fitness of each individual ;

    WHILE NOT finished DO LOOP
        BEGIN

            Select individuals from old generations
                for mating ;
            Create offspring by applying
                recombination and/or mutation
                to the selected individuals ;
            Compute fitness of the new individuals ;
            Kill old individuals to make room for
                new chromosomes and insert
                offspring in the new generation ;

            IF Population has converged
                THEN finishes := TRUE ;

        END
    END
```

1.3 Genetic Algorithms World

Genetic Algorithm raises again a couple of important features. First it is a *stochastic* algorithm; randomness has an essential role in genetic algorithms. Both selection and reproduction needs random procedures. A second very important point is that genetic algorithms always consider a population of solutions. Keeping in memory more than a single solution at each iteration offers a lot of advantages. The algorithm can recombine different solutions to get better ones and so, it can use the benefits of assortment. A population base algorithm is also very amenable for parallelisation. The *robustness* of the algorithm should also be mentioned as something essential for the algorithm success. Robustness refers to the ability to perform consistently well on a broad range of problem types. There is no particular requirement on the problem before using GAs, so it can be applied to resolve any problem. All those features makes GA a really powerful optimisation tool.

With the success of Genetic Algorithms, other algorithms making use of on the same principle of natural evolution have also emerged. Evolution strategy, Genetic programming are some of those similar algorithms. The classification is not always clear between those different algorithms, thus to avoid any confusion, they are all gathered in what is called *Evolutionary Algorithms*.

The analogy with nature gives to those algorithms something exciting and enjoyable. Their ability to deal successfully with a wide range of problem areas, including those which are difficult for other methods to solve make them quite powerful. But today, GAs are maybe suffering from too much trendiness. GAs are a new field, and parts of the theory have still to be properly established. You can find almost as many opinions on GAs as there are researchers in this field. In this document, you will generally find the most current point of view. But things evolve quickly in genetic algorithms too, and some comments might not be very accurate in few years.

It is also important to mention in this introduction GA limits. Like most stochastic methods, GAs are not guaranteed to find the global optimum solution to a problem, they are satisfied with finding “acceptably good” solutions to the problem. GAs are an

extremely general tool, and so specific techniques for solving particular problems are likely to out-perform GAs in both speed and accuracy of the final result. GAs are something worth trying when everything else has failed or when you know absolutely nothing of the search space. Nevertheless, even when such specialised techniques exist, it is often interesting to hybridise them with a GA in order to possibly gain some improvements. It is important always to keep an objective point of view; do not consider that GAs are a panacea for resolving all optimisation problems. This warning is for those who might have the temptation to resolve anything with GA. The proverb says: “If you have a hammer, all the problems looks like a nails”. GAs do work and give excellent results if they are applied properly on appropriate problems.

2 How do Genetic Algorithms work

This chapter gives an overview of all the technical aspects of genetic algorithms. All the different steps of the algorithm are studied separately. The essential point and some difficulties are here particularly analysed.

2.1 Genetic representation

To apply GA to a specific problem, we must firstly define an appropriate genetic representation for the solution. We must find a way to encode any solution of the problem into a chromosome of a certain structure. This structure shared by all the chromosomes is called the genetic representation.

Originally, solutions are encoded into a string of bits of a given length. Bit string chromosomes are quite handy because they can represent everything. Historically, GAs have always tried to be a universal solver, able to deal with the wider range of problems. So binary codage was seen as a standard representation that can fit almost all kinds of search space. Indeed, a string of bits can encode integers, real numbers, sets or whatever is appropriate for the problem. Moreover, binary chromosomes are very simple to manipulate. Mutation and recombination of bit strings can be done with very simple and universal operators. However, bit strings are often not really appropriate for particular problems. A problem-specific representation, integer to represent integer, real to represent real, character strings to represent sets, and so on, can be customized in a way that gives more sense and coherence to the algorithm. But, as it will be seen later in this paper, a different representation also requires a set of operators specifically adapted.

The morphogenesis function associates each genotype with its phenotype. It simply means that each chromosome must define one unique solution, but it does not mean that each solution is encoded by exactly one chromosome. Indeed, the morphogenesis function is not necessary bijective, and it is even sometimes impossible (especially with binary representation). Nevertheless, the morphogenesis function should at least be surjective. Indeed, all the candidate solutions of the problem must correspond to at least one possible chromosome, to be sure that the whole search space can be explored. When the morphogenesis function that associates each chromosome to one solution is not injective, i.e. different chromosomes can encode the same solution, the representation is said to be *degenerated*. A slight degeneracy is not so worrying, even if the space where the algorithm is looking for the optimal solution is inevitably enlarged. But a too important degeneracy could be a more serious problem. It can badly affect the behaviour of the GA, mostly because if several chromosomes can represent the same phenotype, the meaning of each gene will obviously not correspond to a specific characteristic of the solution. It may add some kind of confusion in the search.

There are some important points that a genetic representation should try to respect. Each gene should correspond to a particular characteristic. Each of these characteris-

tics should be as much as possible independent to avoid interaction between the genes. Ideally, there should be no correlation between gene values. In this hypothetical case, the optimal solution can be found by independently selecting the best allele for each gene. Correlation between gene values is called *epistasis*.

2.2 Fitness Function

Once the genetic representation has been defined, the next step is to associate to each solution (chromosome) a value corresponding to the fitness function. There is generally no problem in determining the fitness function. In fact, most of the time, it is implicitly defined by the problem as we know what we want to optimize. But, as simple as it may look, particular attention should be taken due to the fact the selection is done according to the fitness of individuals. The fitness function should not only indicate how good the solution is, but also it should correspond to how close the chromosome is to the optimal one.

In the case of multicriterion optimisation, the fitness function is definitely more difficult to determine. In multicriterion optimisation problems, there is often a dilemma as how to determine if one solution is better than another. What should be done if a solution is better for one criterion but worse for another? But here, the trouble comes more from the definition of a 'better' solution rather than from how to implement a GA to resolve it. If sometimes a fitness function obtained by a simple combination of the different criteria can give good results, it supposes that criterions can be combined in a consistent way. But, for more advanced problems, it may be useful to consider something like Pareto optimality or other ideas from multicriteria optimisation theory.

One of the most annoying problem of Genetic Algorithms is their inability to tackle constraints. Ideally, the representation should be such that only feasible solutions can be generated. If all the possible chromosomes correspond to feasible solutions, then there is no need to worry about the constraints. But it supposes that the function decoding the chromosomes into candidate solutions of the problem is able to take into account the constraint violations. Unfortunately, it is not always simple to 'repair' bad chromosomes, and sometimes, a codage that avoids infeasible solutions cannot be found. A classical approach in this case is to use a penalty function. The Penalty method allows constraints to be violated, depending on the magnitude of the violation. However, a penalty that is proportional to the degree of infeasibility is incurred which degrades the objective function. If the penalty is large enough, highly infeasible individuals will rarely be selected for reproduction, and the GA will concentrate on feasible or nearly-feasible solutions.

2.3 Population

Typically, there are two questions people ask about the population used in genetic algorithm. How is the initial population generated? And, what is the appropriate population size?

In fact, these questions should be asked in a slightly different way: how will the genetic diversity in the population affect the behaviour of the algorithm? Ideally, the first population should have a gene pool as large as possible in order to be able to explore the whole search space. All the different possible alleles of each gene should be present in the population. To achieve this, the initial population is, in most of the cases, chosen randomly. Nevertheless, sometimes a kind of heuristic can be used to seed the initial population. Thus, the mean fitness of the population is already high and it may help the genetic algorithm to find good solutions faster. But by doing this you have to be sure that the gene pool is still large enough. Otherwise, if the population badly lacks diversity, the algorithm will just explore a small part of the search space and never find global optimal solutions.

The size of the population raises few problems too. The larger the population is, the easier it is to explore the search space. But it has been established [14] that the time required by a GA to converge is $O(n \log n)$ function evaluations, where n is the population size. We say that the population has converged when all the individuals are very much alike and further improvement may only be possible by mutation. Goldberg has also shown that GA efficiency to reach global optimum instead of local ones is largely determined by the size of the population. To sum up, a large population is quite useful, but it requires much more computational cost in both memory and time. Practically, a population size of around 100 individuals is quite frequent, but anyway this size can be changed according to the time and the memory you dispose on your machine compared to the quality of the result you want to reach.

2.4 Selection

There are many ways to select individuals. What we want for selection is a method that randomly picks chromosomes out of the population according to their evaluation function. The higher the fitness function, the more chance an individual has to be selected. The selection pressure is defined as the degree to which the better individuals are favoured. The higher the selection pressured, the more the better individuals are favoured. This selection pressure drives the GA to improve the population fitness over the successive generations.

The convergence rate of a GA is largely determined by the magnitude of the selection pressure, with higher selection pressures resulting in higher convergence rates. Genetic Algorithms should be able to identify optimal or nearly-optimal solutions under a wide range of selection scheme pressure [21]. However, if the selection pressure is too low, the convergence rate will be slow, and the GA will take unnecessarily longer to find the optimal solution. If the selection pressure is too high, there is an increased chance of the GA prematurely converging to an incorrect (sub-optimal) solution. In addition to providing selection pressure, selection schemes should also preserve population diversity, as this helps to avoid premature convergence.

Typically we can distinguish two types of selection scheme, proportionate selection and ordinal-based selection. Proportionate-based selection picks out individuals based upon their fitness values relative to the fitness of the other individuals in the population. Ordinal based selection schemes selects individuals not upon their raw fitness, but upon their rank within the population. This requires that the selection pressure is independent of the fitness distribution of the population, and is solely based upon the relative ordering (ranking) of the population.

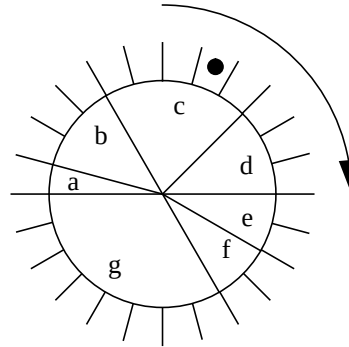
It is also possible to use a scaling function to redistribute the fitness range of the population in order to adapt the selection pressure. For example, if all the solutions have their fitness in the range [999, 1000], the probability of selecting a better individual than any other using a proportionate-based method will not be important. If the fitness every individual is bring to the range [0, 1] equitably, the probability of selecting good individual instead of bad one will be increase.

Roulette wheel selection is the classic and more popular fitness-proportionate selection. It simply assigns to each solution a sector of a roulette wheel whose size (the angle it subtends) is proportional to the appropriate fitness measure (usually a scaled

fitness of some sort), and then chooses a random position on the wheel (and the solution to which that position was assigned, as if spinning the wheel).

fitness of each individual :

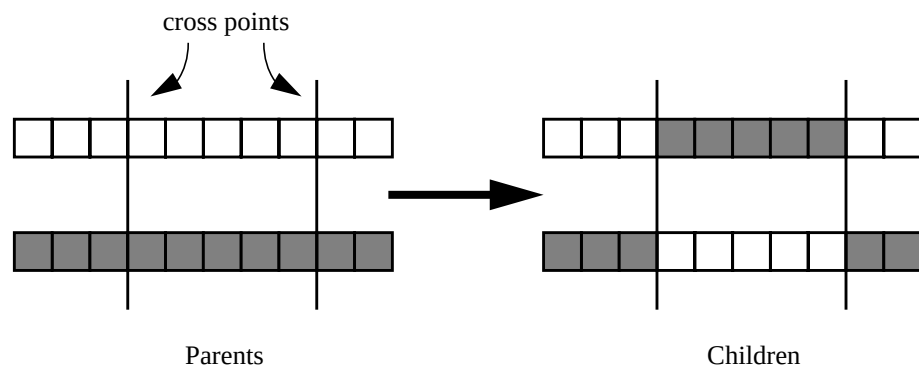
a : 1
b : 3
c : 5
d : 3
e : 2
f : 2
g : 8



Tournament selection is an alternative scheme, probably one of the more reasonable to use. It is robust and relatively simple. Many variations exist, but the basic mechanism is to pick k members of the population at random and then select one of them in some very simple manner that depends on fitness. Choosing the best members of the tournament produces a relatively strong selection pressure. So, generally the best is chosen with the probability p , if it fails to be chosen, the second best is chosen with the probability p , and so on until the final solution is chosen. Typically the tournament size k is 2, and the probability of choosing the best in the tournament is around 0.7. The selection pressure can be adjusted by changing k and p . Clearly tournament selection as described is unaffected by the absolute fitness values, and in effect depends only on the rank of any solution in the population.

2.5 Recombination

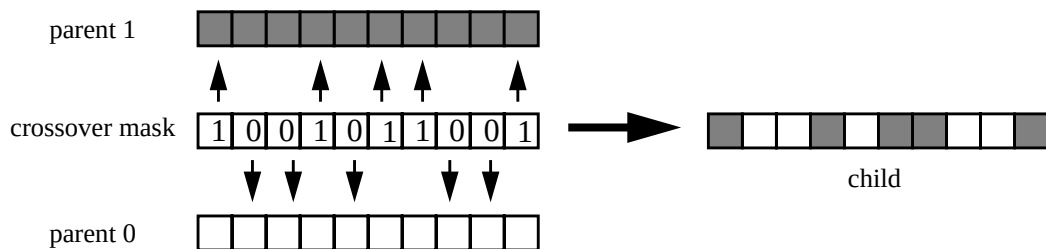
Recombination is the process of taking two parent solutions and producing from them a child. The most common reproducing operator is N -point crossover. N -point crossover takes two chromosomes, aligns them and cut them in $N+1$ segments using N cross points. A child is produced by choosing alternatively a segment from one or the other parent. Two different children can be produced depending if the first segment is taken from the mother or from the father. An example of 2-point crossover is shown below.



Originally, GAs were using one-point crossover which cuts two chromosomes in one point and splices the two halves to create new ones. But with this one-point crossover, the head and the tail of one chromosome can not be passed together to the offspring. If both the head and the tail of a chromosome contain good genetic information, none of the offsprings obtained directly with one-point crossover will share the two good features. Using a 2-point crossover avoids this drawback, and then, is generally considered better than 1-point crossover. In fact this problem can be generalised to each gene position in a chromosome. Genes that are close on a chromosome have more chance to be passed together to the offspring obtained through a N -points crossover. It leads to an unwanted correlation between genes next to each other. Consequently, the

efficiency of a N -point crossover will depend on the position of the genes within the chromosome. In a genetic representation, genes that encode dependant characteristics of the solution should be close together. To avoid all the problem of genes locus, a good thing is to use a uniform crossover as recombination operator.

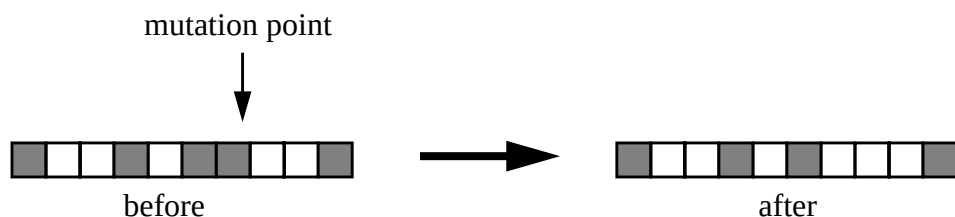
Uniform crossover is quite different from the N -point crossover. Each gene in the offspring is created by copying the corresponding gene from one or the other parent, chosen according to a random generated binary crossover mask of the same length as the chromosomes. Where there is a 1 in the crossover mask, the gene is copied from the first parent, and where there is a 0 in the mask the gene is copied from the second parent. A new crossover mask is randomly generated for each pair of parents. Offsprings, therefore contain a mixture of genes from each parent. The number of effective crossing point is not fixed, but will average $L/2$ (where L is the chromosome length).



2.6 Mutation

Mutation is a simple operator consisting of random changes in the value of genes in a chromosome. Mutation has traditionally been considered as a simple search operator. If recombination is supposed to exploit the current solutions to find better ones, mutation is supposed to help for the exploration of the whole search space. Mutation is often seen as a background operator to maintain genetic diversity in the population. It helps to kept the gene pool well stocked, and thus it ensures *ergodicity* (A search space is said to be ergodic if there is a non-zero probability of generating any solution from any population state).

There are many different forms of mutation for the different kinds of representation. For binary representation, a simple mutation can consist in inverting the value of each gene with a small probability. The probability is usually taken about $1/L$, where L is the chromosome length.



It is also possible to implement a kind of hill-climbing mutation operators that do mutation only if it improves the quality of the solution. Such an operator can accelerate the search. But you have to be careful because it might also reduce the diversity in the population and makes the algorithm converge toward some local optima.

2.7 Replacement

Once offsprings are produced, a method must determine which of the current members of the population, if any, should be replaced by the new solutions. Basically, there

are two kinds of methods for maintaining the population; generational updates and steady state updates.

The basic generational update scheme consists in producing N children from a population of size N to form the population at the next time step (generation), and this new population of children completely replaces the parent selection. Clearly this kind of update implies that an individual can only reproduce with individuals from the same generation. Derived forms of generational update are also used like $(\lambda+\mu)$ -update and (λ,μ) -update. This time from a parent population of size μ , a litter of children is produced of size $\lambda \geq \mu$. Then the μ best individuals from either the offspring population or the combined parent and offspring populations (for (λ,μ) - and $(\lambda+\mu)$ -update respectively), form the next generation.

In a steady state update, new individuals are inserted in the population as soon as they are created, as opposed to the generational update where an entire new generation is produced at each time step. The insertion of a new individual usually necessitates the replacement of another population member. The individual to be deleted can be chosen as the worst member of the population (it leads to a very strong selection pressure), or as the oldest member of the population, but those methods are quite radical. Generally steady state updates use an ordinal based method for both the selection and the replacement, usually a tournament method. Tournament replacement is exactly analogous to tournament selection except the less good solutions are picked more often than the good ones. A subtle alternative is to replace the most similar member in the existing population.

An elitism policy can be a good choice. Elitism consists of never replacing the best individuals in the population with inferior solution, so the best solution is always available for reproduction. But elitism may encourage premature convergence. It potentially makes it harder for the population to escape from a local optimum.

Avoiding duplication is also something we could look for. To keep more than one copy of an individual in the current population seems wasteful. Duplicates necessarily reduce the diversity in the population, but also may distort the way individuals are selected. Although elitism and no duplication are quite common in steady state update, it is perfectly possible to impose them in a generational update scheme too.

2.8 Convergence

Unfortunately, evolution often looks like a never-ending process. GA cannot be expected to stop spontaneously on the global optimum solution as if by magic. GAs are not even guaranteed to find the global optimum solution. So evolution has to be stopped at some point according to a pre-determined criterion. In fact there are many ways to decide to stop the algorithm. The simplest one is to stop evolution after a fixed number of iterations. A better solution consists of continuing the iteration for as long as any improvements are noticeable. Finally we can also wait until most or all of the members of the population are similar or identical, i.e. when the population has converged.

Convergence is simply a loss of diversity in the population, due to the selection pressure. In absence of genetic operators, selection will tend to reproduce the better members of the population. Recombination used alone, rearranging genetic material, tends to oppose convergence. Nevertheless recombination usually does not create new genetic material in the population, only mutation operators are really able to maintain a good genetic diversity.

Evolutionary algorithms are supposed to converge naturally around optima as they are discovered, but unfortunately these optima are not guaranteed to be the global optimal solution of the problem. A common failure mode of genetic algorithms is usually called *premature convergence*, by which is normally meant that the population converged to an unacceptably poor solution (anything but a global solution). To tackle

this, various techniques could be developed, such as increasing the mutation rate, or even better, using an adaptive mutation rate. Typically, mutation rate should be higher at the end of a run to avoid loss of diversity by (re-)introducing (lost) genetic material [10]. A larger population size or a lower selection pressure can also help avoid premature convergence. Curing premature convergence can also bring the algorithm to behave like a random search by using too low a selection pressure or too high a mutation rate.

On the other hand, when the algorithm take too much time to converge toward the optimum, we speak of *slow finishing*. Slow finishing can be resolved by increasing the selection pressure. In fact, only a good compromise between selection pressure, population size, mutation rate and so on, can lead to an adequate algorithm that finds good solutions in a relatively short among of time.

3 Comparison with other optimisation techniques

Numerous general techniques are usually proposed to resolve search and optimisation problems. Those presented here, are in direct concurrence with GA. They all deal with large space search, large means really too large to enumerate all the solutions, and they try to find solutions which maximize a given fitness function.

All the methods presented are, like Genetic Algorithms, *weak* methods, i.e. no information is required concerning the structure of the search space or properties of the evaluation function. There are often specific techniques that are quite efficient to tackle problems that structure is simple, or problems where the fitness function is known to be continuous or derivable. But the advantage of all those weak methods is that they work even when nothing is known about the problem or when all the specific and efficient known methods are not applicable.

Those methods are generally stochastic, they need the use of random number generators. Most weak methods process by making suppositions on where optimal solutions have the highest probability of being found in the search space. An implicit reduction of the search space is done without any proof that the discarded areas do not contain optimal solutions. So most of the time, such methods can unfortunately not guarantee to find the global optimal solution of the considered problem. They are just able to provide fairly good solutions in a reasonable amount of time, without being sure they are the best. That is truly the worst drawback of all those methods.

3.1 Random Search

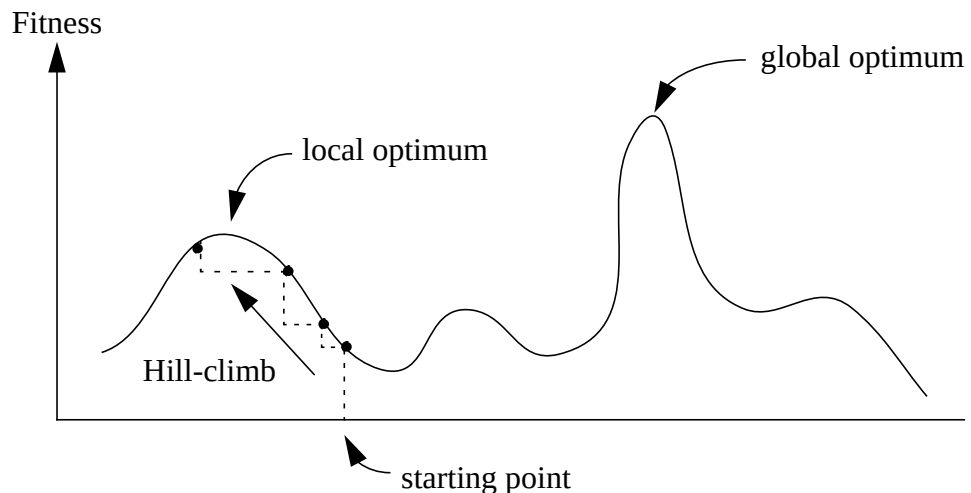
Random search is an extremely basic method. It only explores the search space by randomly selecting solutions and evaluates their fitness. This is quite an unintelligent strategy, and is rarely used by itself. Nevertheless, this method sometimes worth being tested. It doesn't take much effort to implement it, and an important number of evaluations can be done fairly quickly. For new unresolved problems, it can be useful to compare the results of a more advanced algorithm to those obtained just with a random search for the same number of evaluations. Nasty surprises might well appear when comparing for example, genetic algorithms to random search. It's good to remember that the efficiency of GA is extremely dependant on consistant coding and relevant reproduction operators. Building a genetic algorithm which performs no more than a random search happens more often than we can expect. If the reproduction operators are just producing new random solutions without any concrete links to the ones selected from the last generation, the genetic algorithm is just doing nothing else that a random search.

Random search does have a few interesting qualities. However good the obtained solution may be,, if it's not the optimal one, it can be always improved by continuing the run of the random search algorithm for long enough. A random search never gets stuck in any point such as a local optimum. Futhermore, theoretically, if the search space is finite, random search is guaranteed to reach the optimal solution. Unfortu-

nately, this result is completely useless. For most of problems we are interested in, exploring the whole search space takes far too long an amount of time.

3.2 Stochastic Hill Climbing

Efficient methods exist for problems with well-behaved continuous fitness functions. These methods use a kind of gradient to guide the direction of search. *Stochastic Hill Climbing* is the simplest method of these kinds. Each iteration consists in choosing randomly a solution in the neighbourhood of the current solution and retains this new solution only if it improves the fitness function. Stochastic Hill Climbing converges towards the optimal solution if the fitness function of the problem is continuous and has only one peak (*unimodal* function).



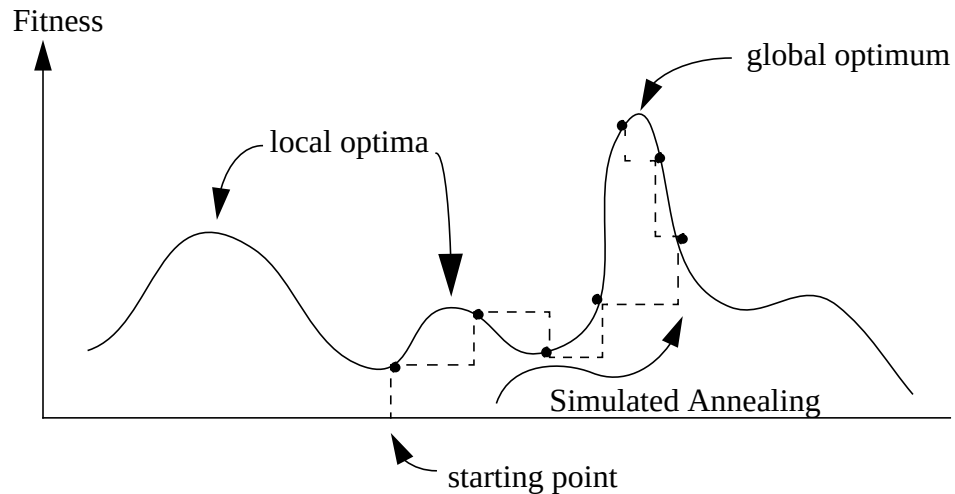
On functions with many peaks (*multimodal* functions), the algorithm is likely to stop on the first peak it finds even if it is not the highest one. Once a peak is reached, hill climbing can not progress anymore, and that is problematic when this point is a local optimum. Stochastic hill climbing usually starts from a random select point. A simple idea to avoid getting stuck on the first local optimal, consists in repeating several hill climbs each time starting from a different randomly choosen points. This method is sometimes known as iterated hill climbing. By discovering different local optimal points, it gives more chance to reach the global optimum. It works well if there is not too many local optima in the search space. But if the fitness function is very 'noisy' with many small peaks, stochastic hill climbing is difinitly not a good method to use. Nevertheless such methods have the great advantage to be really easy to implement and to give fairly good solutions very quickly.

3.3 Simulated Annealing

Simulated Annealing was originally inspired by formation of crystal in solids during cooling. As discovered a long time ago by iron age blacksmiths, the slower the cooling, the more perfect is the crystal formed. By cooling, complex physical systems naturally converge towards a state of minimal energy. The system moves randomly, but the probability to stay in a particular configuration depends directly on the energy of the system and on its temperature. This probability is given by Gibbs law formally:

$$p = e^{-\frac{E}{kT}}$$

where E stands for the energy, k is the Boltzman constant and T is the temperature. In the mid 70's, Kirkpatrick by analogy of this physical phenomena, laid out the first description of simulated annealing [18].



As in the stochastic hill climbing, the iteration of the simulated annealing consists of randomly choosing a new solution in the neighbourhood of the actual solution. If the fitness function of the new solution is better than the fitness function of the current one, the new solution is accepted as the new current solution. If the fitness function is not improved, the new solution is retained with a probability:

$$p = e^{\frac{-(f(y)-f(x))}{kT}}$$

where $f(y)-f(x)$ is the difference of the fitness function between the new and the old solution.

The simulated annealing behaves like a hill climbing method but with the possibility of going downhill to avoid being trapped at local optima. When the temperature is high, the probability to deteriorate the solution is quite important, and then a lot of large moves are possible to explore the search space. The more the temperature decreases, the more difficult it is to go downhill, the algorithm tries to climb up from the current solution to reach a maximum. When temperature is lower, there is an exploitation of the current solution. If the temperature is too low, no deterioration is accepted, and the algorithm behaves just like a stochastic hill climbing method. Usually, the simulated annealing starts from a high temperature which decreases exponentially. The slower the cooling, the better it is for finding good solutions. It even has been demonstrated that with an infinitely slow cooling, the algorithm is almost certain to find the global optimum. The only point is that infinitely slow cooling often requires near infinite time. The main difficulty of simulated annealing, consists in finding the appropriate temperature decrease rate to obtain a good behavior of the algorithm.

Simulated Annealing by mixing exploration features such as the random search and exploitation features like hill climbing usually gives quite good results. Simulated Annealing is a serious competitor to Genetic Algorithms. It is worth trying to compare the results obtained by each. Both are derived from analogy with natural system evolution and both deal with the same kind of optimisation problem. GAs differ by two main features which should make them more efficient. First GAs use a population based selection whereas SA only deals with one individual at each iteration. Hence GAs are expected to cover a much larger landscape of the search space at each iteration, but on the other hand SA iterations are much more simple, and so, often much faster. The great advantage of GA is its exceptional ability to be parallelised, whereas SA does not gain much by this. It is mainly due to the population scheme use by GA. Secondly, GAs use recombination operators, able to mix good characteristics from dif-

ferent solutions. The exploitation made by recombination operators are supposedly considered helpful to find optimal solutions of the problem.

On the other hand, simulated annealings are still very simple to implement and they give good results. They have proved their efficiency over a large spectrum of difficult problems, like the optimal layout of printed circuit board, or the famous travelling salesman problem.

3.4 The No Free Lunch Theorem

It has been shown that, for any pair of search algorithms, there are as many problems for which the first algorithm outperforms the second as for which the reverse is true. One consequence of this is that if we don't put any domain knowledge into our algorithm, it is as likely to perform worse than random search as it is likely to perform better. This is true for all algorithms including genetic algorithms.

The No Free Lunch work of Wolpert and Macready [30] is a framework that addresses the core aspects of search, focusing on the connection between fitness functions and effective search algorithms. The central importance of this connection is demonstrated by the No Free Lunch theorem which states that averaged over all problems, all search algorithms perform equally. This result implies that if we are comparing a genetic algorithm to some other algorithm (e.g., simulated annealing, or even random search) and the genetic algorithm performs better on some class of problems, then the other algorithm necessarily performs better on problems outside the class. Thus it is essential to incorporate knowledge of the problem into the search algorithm. The No Free Lunch framework also does the following: it provides a geometric interpretation of what it means for an algorithm to be well matched to a problem; it brings insights provided by information theory into the search procedure; it investigates time-varying fitness functions; it proves that independent of the fitness function, one cannot (without prior domain knowledge) successfully choose between two algorithms based on their previous behaviour; it provides a number of formal measures of how well an algorithm performs; and it addresses the difficulty of optimisation problems from a viewpoint outside of traditional computational complexity.

The conclusion is that, to resolve any problem, you need an adapted algorithm, an algorithm that is able to take into consideration the structural specificities of the problem. A magic algorithm that behaves like a black box, able to resolve any problem, does not exist.

4 Why do Genetic Algorithms work

Until now, everything we have looked at concerning the way genetic algorithms work, has been empirical. In fact, GA have been almost always developed with much regard for the result and with little regard to elegance, proof, or other mathematical niceties. Nevertheless, several hypotheses have been put forward to explain results obtained by GAs.

Holland [17] had laid down an explanation based on the *schema theorem*. According to Holland, genetic algorithms work by building blocks in the chromosome, blocks that should be present in the optimal solution. With selection and recombination, such blocks naturally emerge and are passed to the next generation. It is called the building block hypothesis. This demonstration has been granted by many to explain the power of GA. But it may lack of consistancy, mainly because too many suppositions are done. Lee Altenbergh [1] has shown some limitation of the schema theorem, and has even claimed that the schema theorem has no implications for how well a genetic algorithm is performing. According to him, GA performance is due to the correlation between parent and offspring fitness as he states in his version of Price's Theorem. Unfortunately, the recent No Free Lunch theorem of Wolpert and Macready [30] has shaken most of those attempts to explain the power of genetic algorithm. The point is that Genetic algorithm can not work properly if it does not include any domain knowledge.

5 Parallel Genetic Algorithms

One of the major aspects of GA is their ability to be parallelised. Indeed, because natural evolution deals with an entire population and not only with particular individuals, it is a remarkably highly parallel process. Except in the selection phase, during which there is competition between individuals, the only interactions between members of the population occur during the reproduction phase, and usually, no more than two individuals are necessary to engender a new child. Otherwise, any other operations of the evolution, in particular the evaluation of each member of the population, can be done separately. So, nearly all the operations in a genetic algorithm are implicitly parallel.

Parallel Genetic Algorithms (PGA) simply consist in distributing the task of a basic genetic algorithm on different processors. As those tasks are implicitly parallel, little time will be spent on communication and thus, the algorithm is expected to run much faster or to find more accurate results.

It has been established that GA efficiency to find optimal solution is largely determined by the population size. With a larger population size, the genetic diversity increases, and so the algorithm is more likely to find a global optimum! A large population requires more memory to be stored, it has also been proved that it takes a longer time to converge. If n is the population size, the convergence is expected after $n \log(n)$ function evaluations.

The use of today's new parallel computers not only provides more storage space but also allows the use of several processors to produce and evaluate more solutions in a smaller amount of time. By parallelising the algorithm, it is possible to increase population size, reduce the computational cost, and so to improve the performance of the Genetic Algorithm.

Probably the first attempt to map genetic algorithms to existing parallel computer architectures was made in 1981 by John Grefenstette [12]. But obviously today, with the emergence of new high performance computing (HPC), PGA is really a flourishing area. Researchers try to improve performance of GAs. The stake is to show that GAs are one of the best optimisation methods to be used with high performance computation.

5.1 Global Parallelisation

The first attempt to parallelise genetic algorithms simply consists of global parallelisation. This approach tries to explicitly parallelise the implicit parallel tasks of the 'sequential' genetic algorithm. The nature of the problems remains unchanged. The algorithm still manipulates a single population where each individual can mate with any others, but the breeding of new children and /or their evaluation are now made in parallel. The basic idea is that different processors can create new individuals and compute their fitness in parallel almost without any communication between each other.

To start with, doing the evaluation of the population in parallel is something really simple to implement. Each processor is assigned a subset of individuals to be evalu-

ated. For example, on a shared memory computer, individuals could be stored in shared memory, so that each processor can read the chromosomes assigned and can write back the result of the fitness computation. This method only supposes that the GA works with a generational update of the population. Of course, some synchronisations are needed between generations.

Generally, most of the computational time in a genetic algorithm is spent calling the evaluation function. Time spent manipulating the chromosomes during the selection or recombination phase is usually negligible. By assigning to each processor a subset of individuals to evaluate, a speed-up proportional to the number of processors can be expected if there is a good load balancing between them. However, load balancing should not be a problem as generally the time spent for the evolution of an individual does not really depend on the individual. A simple dynamic scheduling algorithm is usually enough to share the population between each processor equally.

On a distributed memory computer, we can store the population in one ‘master’ processor responsible for sending the individuals to the other processors, ‘slaves’. The master processor is also responsible for collecting the result of the evaluation. A drawback of this distributed memory implementation is that a bottleneck may occur when slaves idle while only the master is working. But a simple and good use of the master processor can improve the load balancing by distributing individuals dynamically to the slave processors when they finish their jobs.

A further step could consist in applying the genetic operators in parallel. In fact, the interaction inside the population only occurs during selection. The breeding, involving only two individuals to generate the offspring could easily be done simultaneously over $n/2$ pairs of individuals. But it is not that clear if it worth doing so. Crossover is usually very simple and not so time consuming; the point is not that too much time will be lost during the communication, but that the time gain in the algorithm will be almost nothing compared to the effort produced to change the code.

This kind of global parallelisation simply shows how easy it can be to transpose any genetic algorithm onto a parallel machine and how a speed-up sublinear to the number of processors may be expected. As opposed to the methods introduced in the next two paragraphs, this parallelisation is done without changing anything of the nature of the algorithm.

5.2 Coarse Grained PGAs - The Island Model

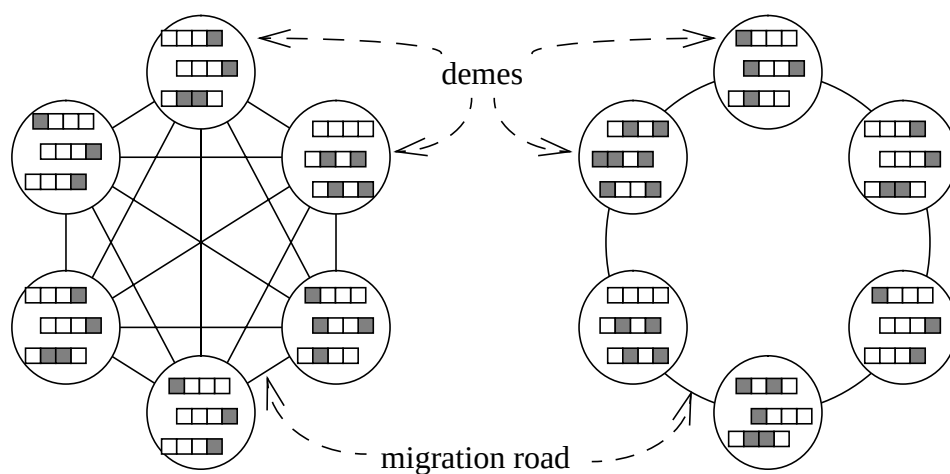
The second class of parallel GA is once again inspired by nature. The population is now divided into a few subpopulations or demes, and each of these relatively large demes evolves separately on different processors. Exchange between subpopulations is possible via a migration operator. The term *islands model* is easily understandable; the GA behave as if the world was constituted of islands where populations evolve isolated from each other. On each island the population is free to converge toward different optima. The migration operator allow ‘metissage’ of the different subpopulations and is supposed to mix good features that emerge locally in the different demes.

We can notice that this time the nature of the algorithm changes. An individuals can no longer breed with any other from the entire population, but only with individuals of the same island. Amazingly, even if this algorithm has been developed to be used on several processors, it is worth simulating it sequentially on one processor. It has been shown on a few problems that better results can be achieved using this model. This algorithm is able to give different sub-optimal solutions, and in many problems, it’s an advantage if we need to determine a kind of landscape in the search space to know where the good solutions are located. Another great advantage of the island model is that the population on each island can evolve with different rules. That can be used for multicriterion optimisation. On each island, selection can be made according to different fitness functions, representing different criterions. For example it can

be useful to have as many islands as criteria, plus another central island where selection is done with a multicriterion fitness function. The migration operator allows individuals to move between islands, and therefore, to mix criteria.

In the literature, this model is sometimes also referred as the coarse grained parallel GA. (In parallelism grain size refers to the ratio of time spent in computation and time spent in communication; when the ratio is high the processing is called coarse grained). Sometimes, we can also find the term 'distributed' GA, since they are usually implemented on distributed memory machines (MIMD Computers).

Technically there are three important features in the coarse grained PGA: the topology that defines connections between subpopulations, migration rate that controls how many individuals migrate, migration intervals that affect how often the migration occurs. Even if a lot of work has been done to find optimal topology and migration parameters, here, intuition is still used more often than analysis with quite good results.



island model

stepping stones model

Many topologies can be defined to connect the demes, but the most common models are the island model and the stepping stones model. In the basic island model, migration can occur between any subpopulations, whereas in the stepping stone demes are disposed on a ring and migration is restricted to neighbouring demes. Works have shown that the topology of the space is not so important as long as it has high connectivity and small diameter to ensure adequate mixing as time proceeds.

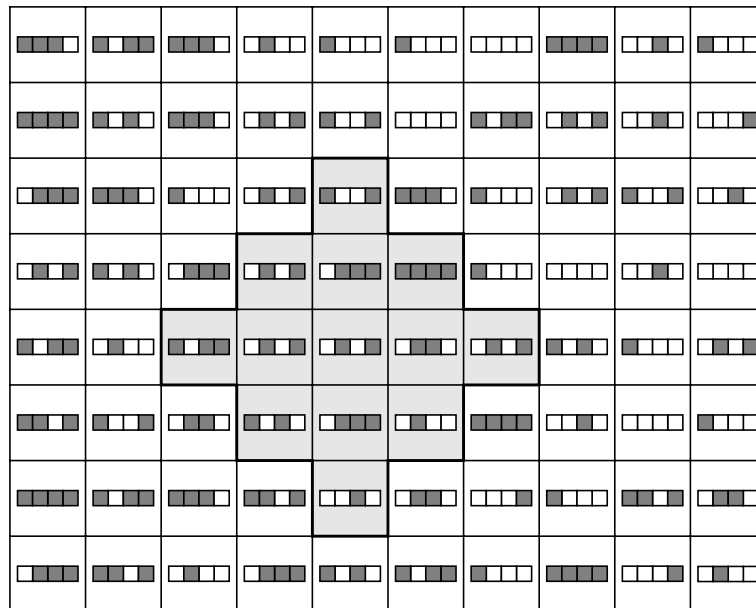
Choosing the right time for migration and which individuals should migrate appears to be more complicated. Quite a lot of work is done on this subject, and problems come from the following dilemmas. We can observe that species are converging quickly in small isolated populations. Nevertheless, migrations should occur after a time long enough for allowing the development of goods characteristics in each subpopulation. It appears also that, immigration is a trigger for evolutionary changes. If migration occurs after each new generation, the algorithm is more or less equivalent to a sequential GA with a larger population. In practice, migration occurs either after a fixed number of iterations in each deme or at uniform periods of time. Migrants are usually selected randomly from the best individuals in the population and they replace the worst in the receiving deme. In fact, intuition is still mainly used to fix migration rate and migration intervals; there are absolutely nothing rigid, each personal cooking recipe may give good results.

5.3 Fine Grained PGAs

The development of massively parallel computers triggers a new approach of parallel genetic algorithms. To take advantage of new architectures with even a greater number of processors and less communication costs, fine grained PGAs have been developed. The population is now partitioned into a large number of very small sub-populations. The limit (and maybe ideal) case is to have just one individual for every processing element available.

Basically, the population is mapped onto a connected processor graph, usually, one individual on each processor (But it works also with more than one individual on each processor. In this case, it's preferable to choose a multiple of the number of processors for the population size). Mating is only possible between neighbouring individual, i.e. individuals stored on neighbouring processors. The selection is also done in a neighbourhood of each individual and so depends only on local informations. A motivation behind local selection is biological. In nature there is no global selection, instead natural selection is a local phenomenon, taking place in an individual's local environment.

If we want to compare this model to the island model, each neighbourhood can be considered as a different deme. But here, the demes overlap providing a way to disseminate good solutions across the entire population. Thus, the topology does not need to explicitly define migration roads and migration rate.



It is common to place the population on a 2-D or 3-D torus grid because in many massively parallel computers the processing elements are connected using this topology. Consequently each individual has four neighbours. Experimentally, it seems that good results can be obtained using a topology with a medium diameter and neighbourhoods not too large. Like the coarse grained models, it worth trying to simulate this model even on a single processor to improve the results. Indeed, when the population is stored in a grid like this, after few generations, different optima could appear in different places on the grid.

To sum up, with parallelisation of GA, all the different models proposed and all the new models you can imagine by mixing those ones, can demonstrate how well GA are adapted to parallel computation. In fact, the too many implementations reported in the literature may even be confusing. We really need to understand what truly affects the performance of parallel GAs. For many researchers, the real great advantage of

GA versus other optimization technique like Simulated annealing is their ability to be parallelised.

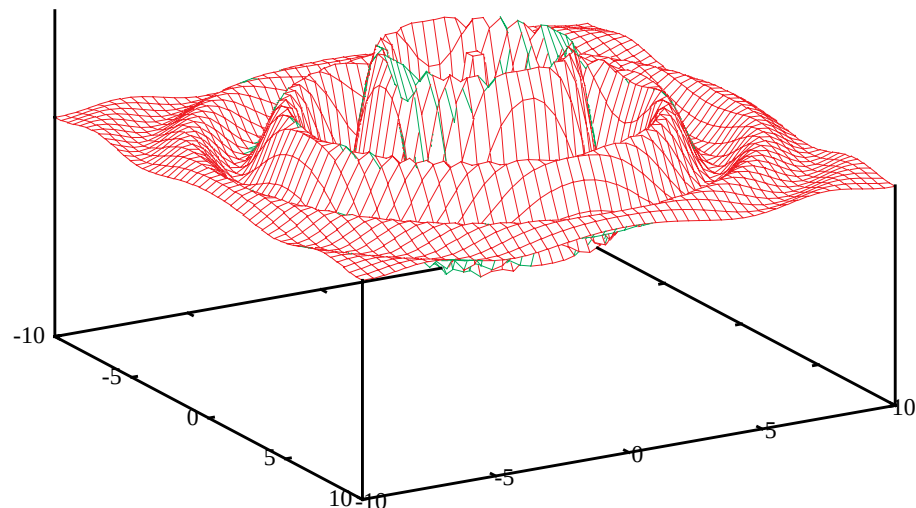
6 Applications

As a general optimisation method, GAs can deal with a large number of possible applications. But on the other hand, being a general optimisation method means that you are likely to be outperformed by problem specific algorithms. Indeed specialized algorithms integrate the specificity of each problem and so take advantage of it. In fact the favourite area of GA is where other algorithms fail. If absolutely nothing is known about the fitness function over the search space, GA can always be applied.

To compile a list of applications of GA is not so easy. If we consider all the subjects where GA can be applied, the list would be quite long and boring to enumerate. Instead, the list of applications below give problems where GA are really applied because they give good results that can not be easily reached by other methods.

6.1 Numerical Function Optimisation

In order to test the efficiency of the first generation of genetic algorithms, researchers apply them to a benchmark of numerical functions optimisation. The classical benchmark included principally functions like the ones defined by Kenneth DeJong [11], that were supposed to represent difficult optimisation problems. The F6 function used by Schaffer [27] is known to be very difficult for any hill climbing methods with circular local maxima.



$$F6(x, y) = 0.5 + \frac{\sin^2 \sqrt{x^2 + y^2} - 0.5}{1.0 + 0.001(x^2 + y^2)^2}$$

The GA performed pretty well on those examples. But today, the choice of this benchmark could be arguable. The good results obtained using Genetic Algorithms seem to be due more to some symmetrical properties of those functions (where the global maximum is situated at the origin), than really to the GAs' intrinsic abilities. Those

examples can certainly not be seen as good applications of GA, but from a didactic point of view they are still interesting.

Originally, some people considered that binary representation was the only good one to use with GAs. So they encode real and integer parameters into a binary string. The task was not very difficult. N bits can encode up to 2^N integers just by using the classical binary interpretation ($0.000=0$; $0.001=1$; $0.010=2$; ... ; $1.11=2^N-1$). For real parameters, one just has to discretised the interval into a power of 2 and use a simple binary representation once again. With this representation, the algorithm did give good results, but nobody can really say why. Indeed the reproduction operators like mutation or crossover do not produce new individuals in a very sensible way. For instance, it should be easy to go to 1000 from 0111 (the integer value is just increasing of 1), but you need to change all the bits, this phenomena is known as the Hamming Cliff. An idea is to introduce a grey coding in which you can go to any adjacent value by modifying only one bit. But even with grey coding, by randomly changing one bit of a binary string, the value of this number can change in an very incoherent way. And as for the crossover, the result is even more unpredictable. Consequently, it is difficult to interpret how the investigations done by the algorithm reach the optimum. The direction of the search does not seem to follow any intelligent or coherent way. Today, most of people working on GA, think that the use of binary representation for numerical function optimisation is a complete non sense.

A far better solution for numerical function optimisation is to use a genuine integer or real representation. It is much more natural to manipulate genes with integer value to represent integer parameters, genes with real values to represent real parameters. With integer or real representation, the coding reflects in a very sensible way the structure of the problem. Of course, new sets of operators must be defined to deal with the new representations. Those new operators are inspired from the one described for the binary representation, but they take into consideration the specificity of integer or real properties.

For the integer representation, there is not much change in the recombination operators. N -points and uniform crossover remain the same. Two mutation operators are generally introduced. The creep mutation adds ± 1 to the gene value with a certain probability. The random mutation sets the gene value to a randomly chosen value in the allowed interval of the parameter. Obviously these two mutation operators do not explore the search space the same way, the second one is much more disruptive. They can be used together with different rates.

For the real representation, we generally consider two kinds of mutation operators, like in the integer representation. The random mutation changes randomly each allele in the allowed interval with a certain probability whereas the creep mutation adds a Gaussian noise around to each gene value. There is a large choice of possible crossovers. A uniform crossover is like in the binary representation. For each gene, it selects the gene value probabilistically from the mother or from the father allele. A blend crossover chooses a value randomly in the interval defined by the two parents' alleles. This interval may be enlarged to avoid the averaging effect created by always choosing the children's gene value in between the one of its two parents.

Using such representation, Genetics Algorithms have proved their efficiency to deal with any kind of functions, even if they are non-unimodal, noisy, discontinuous or non derivable. For numerical function optimisation (and all other applications in general), the important point to underline is the role of a proper representation and proper operators in the good performances of the algorithm.

6.2 Scheduling

A general description of scheduling problems can be simply expressed as a question of how to allocate efficiently a set of resources (machines, people, rooms, facilities) to

carry out a set of predefined tasks. The suitable solutions must respect a number of hard constraints. The more obvious constraints are that resources are limited and that each resource cannot be used to perform more than one operation at the same time. Other constraints can be that the different tasks may have to be accomplished in a specific order. The suitable solution must also minimise the cost. The cost can be defined as the earliest overall completion time or amount of 'idle time' for each resource. Generally, scheduling engenders large and difficult combinatorial problems. Most of them are known to be non-deterministic polynomial, i.e. they are not currently solvable using a deterministic polynomial time algorithm. Recently, a lot of work has been done to apply Evolutionary Computation to this kind of difficult problem. It really seems that Genetic Algorithms are able to produce successfully and quickly solutions of hard scheduling problems, often real problems.

Frequently, scheduling problems are solved using a simplified model. Talbi and Muntean [28] studied a mapping problem in parallel computation, the optimal static placement of communicating processes on the processors of a distributed memory parallel machine. A parallel program was modelled by a graph $G_p=(V_p, E_p)$ where the vertices represent processes and the weight associated with the vertices represent known or estimated computation costs for these processes. The edges represent communication exchanges between processes and their weight estimate the communication costs. A parallel architecture was also modelled by an undirected connected graph $G_t=(V_t, E_t)$ where vertices represent processors and edges represent communication links between processors. The problem of assigning each process a processor was done using a simple mapping function $\Pi:V_p \rightarrow V_t$. Two contradictory mapping criteria have been considered: to minimise the sum of the total communication cost between processors, and to minimise the load imbalance across the system. The cost function was a weighted sum of the total communication cost and the loads variance of the different processors. They used a simple genetic representation corresponding to their simple model. The genome was composed by as many genes as processes, each gene representing the processor assigned for the process. They implemented a massively parallel genetic algorithm using a fine-grained model, and they compared it with two other algorithms, a hill climbing and a simulated annealing. The results show that the hill-climbing gives worse quality solutions compared to simulated annealing but they are faster. Genetic Algorithms give comparable solutions as simulated annealing, but with a search time which is of the same order of magnitude than the hill-climbing. They concluded that one of the main advantage of genetic algorithms was that they are intrinsically parallel.

Timetabling is a specific scheduling problem involving the need to assign time slots to a set of events, subject to constraints on these assignments. The most common and important such constraint in timetabling problems is an *edge constraint* between two events, so called due to the similarity of the simple timetabling problems with vertex colouring problems in graph theory. An edge constraint simply states that a given pair of events e_1 and e_2 must not overlap in time. It may be because, for example, two classes are given by the same teacher, or because one or more students must attempt both of them. A related kind of constraint is what is called an *event-spread constraint*. This is an extra requirement on the relative times assigned to pairs of edge constraint event, like consecutive classes. Timetabling may also involve the assignment of rooms or places as well as times; this brings constraints arising from room capacities and /or travel-time between location. Solving such a problem involves at least finding an acceptable timetable -one in which all hard constraints are satisfied, and for which we are prepared to suffer its violations of soft constraints. At best, solving such a timetable problem involves finding or coming close to an optimal minimal number of constraint number violations. Often, real timetabling problems seem not very hard if we only take into account the edge constraints. This fact explains why a lot of timetables are done by hand. The trouble is the inclusion of event-spread constraints, along with all sort of soft, although strongly desired constraints (such as preferred ordering for events, teachers' preferred classes, morning off, and so on) result in a very difficult

multi objective optimisation problem which requires considerable computational effort for its solution.

A simple approach to resolve timetabling problem is to use a *direct* representation. If e is the number of events, the direct representation consists of a chromosome of length e , where each gene typically corresponds to the possible start times (or time slot) of an event. The i th allele thus represents the time at which event i begins. In a problem involving room and teachers, for example, a simple direct interpretation of “(abc)(def)...” might be event $e1$ starts at time a in place b involving teacher c , and event $e2$ starts at time d in place e , involving teacher f . The fitness function commonly used in this case is a weight sum of constraint violations; a candidate solution is tested against a long list of constraints, and the penalties are assigned for each violation according to its importance. This way, evolution results in lower and lower weighted penalty scores until at least timetables suffering no hard constraint violations appear and (soon after) timetables appear which suffer few or no violations of soft constraints. Corne, Ross and Fang [7] used this direct representation and successfully solved some real examination timetabling problems arising from MSc examinations at the University of Edinburgh Department of Artificial Intelligence, and from Kingston University in London. The results obtained were clearly better than the timetable produce by human expert. They also show similar results for a collection of real lecture timetabling in [25].

A second approach to resolve timetabling problem consists in using an *implicit* representation. With an implicit representation the genes does not directly refers to the time slot of a given event, but to the way this event should be include in the time table. An implicit representation relies on the use of a heuristic to build the timetable (a schedule builder). The schedule builder takes events in a specific order and successively assigns them to a time-slot that satisfied all the hard constraints (if any), in the partially build timetable. The schedule builder is thus supposed to only generate valid timetables, or at least timetables that violate only few hard constraints. So, the optimisation is done in a reduced search space, without bothering with invalid solution. The task of the genetic algorithm simply is to find the best order in which the event should be taken into consideration for the heuristic. The chromosomes are interpreted as permutations (for a description of the permutation representation see travelling salerep problem in the next chapter). The schedule builder, unlike the genetic algorithm, completely understands the details of the scheduling task. The question of how smart the schedule builder should be, depends on how much faith we place in the genetic algorithm portion of the system. If the schedule builder is not very clever, it will seldom construct very good schedules; but, a too clever heuristic able to build a good timetable from a not so good order may be dangerous too, simply because GA crucial informations only comes from the particular ranking of the events. In the chapter called *Schedule Optimisation Using Genetic Algorithm* in Davis’ book [10], Syswerda describes this method, to schedule the use of share resources in a laboratory where different F-14 flight simulators were piloted together. The application consists of 90 tasks and a set of up to 30 resources.

Unfortunately it not so easy to compare the two different approaches for applying genetic algorithm to timetabling. The direct representation seems have difficulties in dealing with real large size problems. But because of its simplicity it is easy to built a specific mutation operator that tries to reduce constraint violations and then improve the quality of the result. On the other hand, even if the implicit representation seems better at solving large and difficult problems because it searches in a far fitter subspace, too little work has been done on this representation to be totally convinced of its real efficiency.

On the different kind of scheduling problems, some ideas introduced by Chu and Beasley [6] are quite interesting. They try to resolve the difficult airline crew scheduling problem as an application of the set partitioning problem (SPP). The schedule must assign a crew to each flight. All the difficulty comes from the problem being highly

constrained. Each crew should be assigned to a feasible round-trip rotation that begins and ends at an individual base location and that conforms to all applicable work rules. To each rotation is associated a cost. Considering the set of all the feasible round trip rotations for the crews, the problem is to find out a subset that covers all the flights and that minimises the total cost for the air company. A simple genetic algorithm was designed to search that optimal subset. A subset was represented as a simple binary string, 0-1 depending if the round trip was in the subset or not. But, since finding a feasible solution to the SPP is difficult, it is difficult for the GA to generate a valid solution. Their approach was to divide the evaluation function in two, the fitness corresponding to the cost minimisation, and the unfitness corresponding to the magnitude of the constraint violations. Thus the solutions are evaluated on a two-dimensional plane, and so the selection and the replacement is easily done to direct the search toward feasible solutions. They also used a heuristic feasible operator specific for the SPP, a mutation operator that made the solution less infeasible. Of the 55 real world problems tested, the GA was able to find at least one feasible solution for all but 4 of the problems, and for 43 of them the GA was able to find the optimal solution.

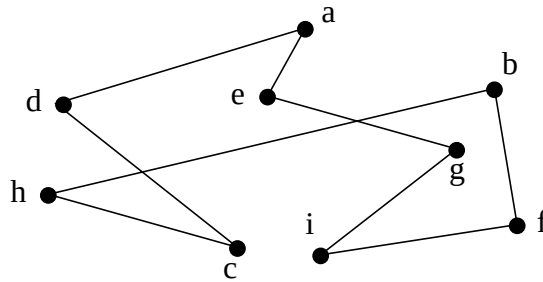
All these examples of scheduling problem applications can give a good overview of how genetic algorithms can face constraint satisfaction problems. The most widely used methods consist of a penalty function subtract from the fitness function. The separation of this penalty function from the fitness is quite good for a better evaluation of the solutions. Finally the use of a kind of morphogenesis function that produces only valid solutions is very interesting for further works. Highly constrained problems seem anyway difficult for GA, of course, they are as bad as every thing else at a needle in a haystack problem.

6.3 Order based problems and the TSP problem

The travelling sales-rep (TSP) is definitely the most classical and most famous example of a difficult optimisation problem. A traveller has to visit a certain number of cities, and he wonders what is the path visiting every city exactly once that minimises his journey. As simple as the problem seems, the only known method to find the shortest route requires an enumeration of all of them. For a problem with n cities, the number of possible tours is $n!/2n$ (the starting city and the direction of travel are not relevant), so quickly (for $n > 20$) the enumeration becomes impossible and other techniques should be used to approach the optimal journey. The important point is that the TSP problem is NP-complete. NP-complete problems form a subset of non-deterministic polynomial problems that are all equivalent; if we know how to resolve one of them then we are able to resolve all of them.

Genetic Algorithms, being not an exact method, can not claim to be able to find the shortest route of any TSP problem. Nevertheless, they appear to be really efficient to find pretty good solutions close to the optimal route. The principle of recombining different routes like it is done with GA seems to be quite powerful for this problem. The importance of the representation for the performance of the algorithm should be particularly pointed out here.

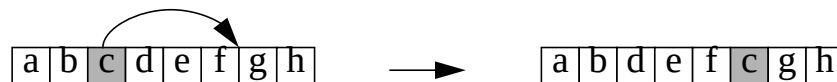
6.3.1 The Permutation Representation.



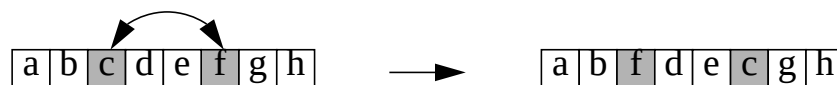
A convenient way to represent a tour is by listing the sequence of city labels in the order in which they are visited. The tour above might be represented by 'adchbfige'. This representation is straight forward. A chromosome is a permutation of as many elements as cities to be visited (n). This permutation representation can be used for any problem dealing with ordering elements. For the TSP, we try to order the cities, for a scheduling problem, we can try to order the tasks to be added in the timetable. As natural as permutations seems to represent a tour, the redundancy is quite important. Depending on the direction of the travel and of the first city visited, there are $2n$ different representations of a same solution. The main difficulty of this kind of representations is that the gene values cannot be set independently, the representation is said to be *non-orthogonal*. A simple mutation that changes one gene value into another will always produces illegal chromosomes (chromosomes that are not permutation). Cut-and-splice crossovers like the ones seen so far, will generate incorrect chromosomes as well. We definitely need new reproduction operators that respect permutation, and also that are relevant for transmitting any kind of informations present in a permutation.

Such mutation operators are easily built.

- A position-based mutation simply consists in removing the allele at a randomly chosen location in the chromosome and re-insert it at another randomly chosen location. This mutation retains most of the ordering information.



- An order-based mutation swaps alleles at two randomly chosen location. It retains most of the position information.

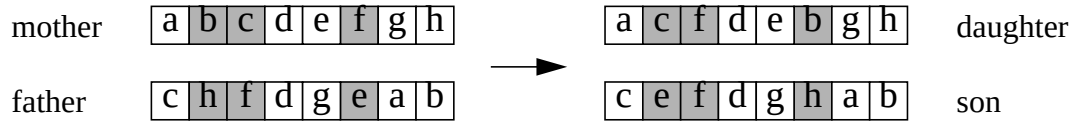


- A scramble mutation selects a sub-section the chromosome and it re-orders alleles in it randomly. It retains most of the proximity information.

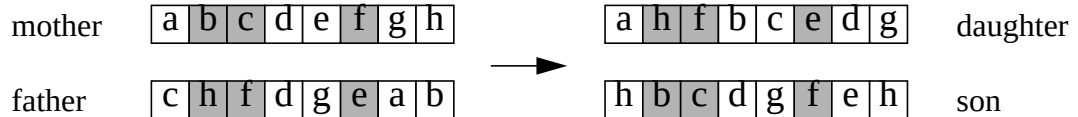


Recombination operators are more tricky to build.

- **Order-based Crossover:** A set of positions is randomly selected, and the order of alleles in the selected positions in one parent is imposed on the corresponding alleles in the other parent.



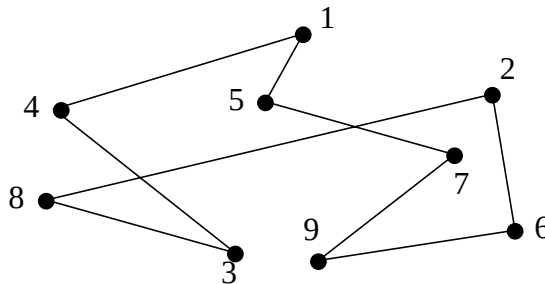
- **Position based Crossover:** Again a set of positions is randomly selected, but this time the positions of cities selected in one parent are imposed on the corresponding cities in the other parent.



This list of operators is far from being exhaustive. Many others can be build using more or less the same ideas. Operators should be used according to what in the representation is important for the solution, what kind of information should be passed on to the next population.

6.3.2 Other Representations

What information is really important for the TSP problem? The fact is that the order in which the cities are visited, if one city comes before another city, does not directly mean something in terms of quality of the solution. The length of a route depends on the edges present in a given path. This reflection brings Radcliff and Surry [23] to consider other representations directly involving edges.



The **undirected edge representation** consists in representing a tour by the set of undirected edges it contains. For example, the tour above, '143826975' is represented by {14, 15, 26, 28, 34, 38, 57, 69, 79}. Obviously the chromosomes is no longer a kind of vector of characteristics, but simply a set of alleles. The representation is said to be *allelic* (as opposed to a *genetic* representation). One of the key features of allelic representations is that the notion of genes is not formally well-defined. Consequently ordinary gene-based notions of recombination are not applicable. A *Random Assorting Recombination* (RAR) is supposed to be a generalization uniform crossover for allelic representation. All the edges from the two parents are put into a conceptual bag, with some weight factor (depending if the allele is present in one or two parents). Edges are then drawn randomly from the bag (on the basis of the weight) for possible insertion in the child. Those which cannot be inserted are discarded. This process is repeated until either the child is complete, or the bag is empty. In the latter case, some edges have to be added to complete the child into a valid tour. It is also possible to apply a Generalised N-point Crossover (GNX) to the undirect edge representation by constructing an associated *pseudo-genetic representation*.

A **directed edge representation** considered which city is visited after the i th city. The tour above '143826975' is encoded as 468319527 meaning city 4 is visited directly after the 1st city, city 6 is visited directly after the 2nd city, city 8 is visited directly after the 3rd city, and so on. In contrast to the undirect edge representation this one can be still considered as genetic. Depending on the direction there are two way to represented the a same solution.

The **corner representation** is in some ways the most complex. It is based on undirect edges, but forms a (formal) genetic representation. In this case, there is a gene corresponding to each city, each of which takes compound alleles consisting of the unordered pair of edges centred an that city. For example, the '143826975' tour is represented as [(1,(4,5)), (2,(6,8)), (3,(4,8)), (4,(1,3)), (5,(1,7)), (6,(2,9)) ...], meaning city 1 is visited directly in between cities 4 and 5. This representation is highly *redundant*, that is to say that not all the information in the genome is needed to represent a solution. This representation offer the avantage that operator are easily implementable. Futhermore, it gives excellent results.

The very point of all those different representations is simply to emphasize that the quality of genetic algorithms is extremely dependant on what is relevant for the problem. The search is directly oriented by the way the solution is encoded, and the way new solutions are generated through reproduction operators. To be efficient, a genetic algorithm must not be considered as a black box that takes as input any representation of a problem, and that gives as output good answers simply by turning the crank. The use of such relevant representations and relevant operators associated to that representation leads to algorithms able to pass on most of the essential information from one population to its successors.

6.4 Configuration of Neural Networks

Genetic Algorithm and Neural Networks (NN) are often put together in the same bag, but they are two different tools that must not be confused. Like genetic algorithms, the principle of neural network theory directly comes from biology. Neural networks are supposed to reproduce the same mechanisms as in the brain. They are able to solve very complicated problems in real time. They are widely used with excellent results for pattern recognition problems. Neural networks are also capable to learn the underlying mechanics of time series, they are often used for market dynamics in trading applications. Precisely Neural networks are not an optimisation tool like GA, but a kind of interpolation tool. A neural network is made of nodes, called, by analogy with the brain, neurones, connected by weighted edges, called synapses. Typically, the nodes are divided into different layers; there is an input layer and an output layer, and hidden layers in between. That forms the topology of the neural network. During a training phase, weight over the synapses are setup in order to pass knowledge through the network.

The point is that good neural networks are difficult to design. Many of the basic principles governing information processing are hard to understand, and there are a lot of complex interactions among network units. Hence, a lot of work has been done to apply GA in neural network design. GA can be use in different ways: for topology optimisation, for training process, or for optimising some control parameters.

In topology optimisation, GA are used to build the more appropriate architecture of the network, how many hidden layers and how many hidden nodes should be used, how the nodes should be interconnected. In this area works by Kitano [19] and Gruau [16] are very interesting and seems to hold much promise.

Some work has also been done for training neural networks with genetic algorithms. The problem is formulated as a weights optimisation problem, usually using the inverse mean squared error as a fitness measure. But so far, methods of these kind

have been outperformed by specific methods originally used to train neural networks like the classical back propagation.

Nolfi and al. [22] proposed to learn and evolve neural networks at different levels. The algorithm makes neural networks evolve at a population level, and learn at the individual level. They have shown by separating the evolutionary and learning task, that evolution and learning influence each other in very complex ways.

6.5 Rules Discovering

Genetic Algorithms can also be used to learn strategy. A very simple example can be found in Michalewicz's book [20] about the prisoners' dilemma. Two prisoners are held in separate cell, unable to communicate. Each prisoner is asked, independently, to defect and betray the other prisoner. If only one prisoner defects, he is rewarded and the other is punished. If both defect, both remain imprisoned and are tortured. If neither defects, both receive moderate rewards. This can be played as a game between two players, where at each turn, each player either defects or cooperates with the other prisoner. The players then score according to the pay-off listed below.

Player 1	Player 2	P1	P2	Comments
Defect	Defect	1	1	Punishment for mutual defection
Defect	Cooperate	5	0	Temptation to defect and sucker's payoff.
Cooperate	Defect	0	5	Sucker's payoff and temptation to defect.
Cooperate	Cooperate	3	3	reward for mutual cooperation

A strategy consists in choosing the current move depending on the outcomes of the previous moves (history). If we consider only the three last moves, there are $4 \times 4 \times 4 = 64$ different histories possible and a strategy can be simply represented by a string of 64 bits indicating what moves should be done after each of the 64 possible histories. The genetic algorithm simply maintains a population of players each with different strategy. Games are played between players, best players are selected for the next generation, reproduction is done according to a simple crossover. Such a simple algorithm give remarkable results. From a random start, the genetic algorithm evolved populations whose medium member was just as successful as the best known heuristic algorithm.

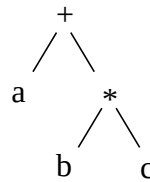
Another example of strategy acquisition with genetic algorithm done by Grefenstette can be found in Davis Handbook of Genetic Algorithm [10]. He tries to discover rules by which a slower aircraft can evade a faster but less agile missile until the missile runs out of fuel. Rules are of a fairly uniform sort, such as to turn right by 90 degrees if the missile's range, heading a certain intervals; a chromosome is essentially an ordered, variable-length set of such rules.

More recently, Andre and al. [2] used genetic programming for discovering cellular automata rules. The rules they obtained are better than those obtained by all other known human written rules, and all other known rules produced by automated methods.

6.6 Image analysis with Genetic Programming

Genetic Programming (GP) is the extension of the genetic algorithm model into the space of programs. That is, the objects that constitute the population are not simple fixed length character strings that encode possible solutions to the problem, they are programs that when they are executed, "are" the candidate solution to the problem. These programs are expressed in genetic programming as parse trees, rather than

lines of code. Thus, for example, the simple program “a+b*c” would be represented as:



The population is thus made of trees, trees considered as chromosomes for a classic genetic algorithm. Those trees are composed of elements from a function set and a terminal set, which are typically fixed sets of symbols selected to be appropriate to the solution of the problems in the domain’s interest. In genetic programming, the crossover operation is implemented by taking randomly selected sub-trees in the individuals and exchanging them. It should be pointed out that Genetic Programming usually does not use any mutation as a genetic operator.

Riccardo Poli [26] applies genetic programming for image analysis, more specifically for image segmentation. Image segmentation consists of labelling the pixel in the image with a small number of labels so that pixels belonging to regions with (more or less) homogeneous grey level have the same label and pixels belonging to regions with significantly different grey level have different labels. Genetic programming is used to discover efficient optimal filters for the segmentation. The algorithm evolves a population of filters that are easily representable with trees. Leaves of the trees are taken from the terminal set which contains the grey level of the pixel at coordinates (x,y), and means of the grey level of the pixels in different neighbourhoods of different size around (x,y). Nodes from the function set include classical operations (add, sub, mult, div, min, max) and macros for shifting in any direction around the coordinate (x,y) of the current pixel. Applied to brain images, the algorithm gives impressive results in comparison to the ones obtained on the same images by a simple neural network.

The research on genetic programming for image analysis is hampered by the tremendous demand of computational resources involved in fitness evaluation. But performance shown here seems to suggest that much more powerful tools can be discovered that way.

7 Getting more information

Since genetic algorithms are today a very trendy subject, you will easily find plenty of information. Indeed, because of the abundance of informations, a selection is necessary to avoid being quickly submerged.

7.1 Books

Three books are usually considered as a reference for a good introduction to genetic algorithms. All the three are well written and easy to read.

- *Handbook of Genetic Algorithms* [10] by Laurence Davis,
- *Genetic Algorithms + Data Structures = Evolutionary Programs* [20] by Zbigniew Michalewics,
- *Genetic Algorithms in search, optimisation and machine learning* [13] by David E. Goldberg.

7.2 News Group and FAQ

There is a newsgroup called **comp.ai.genetic** which deals with all subjects involving evolutionary computations. A 'Frequently Asked Question list' (FAQ) for comp.ai.genetic has been compiled by David Beasley, entitled "The Hitch Hicker's Guide to Evolutionary Computation". It contains a vast amount of information, primarily aimed at new comers. An HTML version of the guide is available via the World Wide Web at the following URL :

- <http://www.lib.ox.ac.uk/internet/news/faq/comp.ai.genetic.html>

7.3 From the Web

Pages collecting genetic algorithm related materials are really flourishing on the web. It is impossible and useless to mention all of them. Here are some good starting points.

The Hitch-Hiker's Guide To Evolutionary Computation has an electronic appendix called ENCORE. This web site is real gold mine to find any information about Genetic Algorithms and other related fields. You can find it at the following URL:

- <ftp://alife.santafe.edu/pub/USER-AREA/EC/Welcome.html>

Nova Genetica web site maintained by Jaime Fernandez, offers another nice collection of links on GAs:

- <http://www.aracnet.com/~wwir/NovaGenetica/>

IlligAL, the Illinois Genetic Algorithms Laboratory of David E Goldberg, have also an home page with good information:

- <http://www-illigal.ge.uiuc.edu/>

8 References

- [1] L. Altenberg. The Schema Theorem and Price's Theorem. In Foundations of Genetic Algorithms 3, San Francisco, California, USA: Morgan Kaufmann Publishers, pp23- 49, 1995
- [2] D. Andre, F.H. Bennett III, and J.R. Koza. Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. In Genetic Programming 1996, Stanford University. Cambridge, MA: MIT Press. Pages 3–11.1996
- [3] D. Beasley, D.R. Bull, and R.R. Martin. An overview of genetic algorithms: Part 1, Fundamentals. University Computing, 15(2) 58-69, 1993
- [4] D. Beasley, D.R. Bull, and R.R. Martin. An overview of genetic algorithms: Part 2, Research Topics. University Computing, 15(4) 170-181, 1993
- [5] E. CantuPaz. A Summary of Research on Parallel Genetic Algorithms. IlliGAL Report No. 95007. July1995
- [6] P.C. Chu and J.E. Beasley. A Genetic Algorithm for the Set Partitioning Problem. The management School, imperial College, London, April 1995
- [7] D. Corne, P. Ross and H.L. Fang. Fast Practical Evolutionary Timetabling. In Proceeding of the AISB Workshop on Evolutionary Computation, Springer Verlag, 1994.
- [8] D. Corne, P. Ross and H.L. Fang. Evolutionary Timetabling: Practice, Prospects and Work in Progress. Departement of Artificial Intelligence, University of Edinburg , 1994.
- [9] C. Darwin. On the Origin of Species. 1859.
- [10] L. Davis. Handbook of Genetic Algorithms. Van Nostrand Reinhold (New York), 1991.
- [11] K.A. De Jong. An analysis of the Behaviour of a class of Genetic Adaptive Systems. PhD Thesis, University of Michigan. 1975.
- [12] J. Grefenstette. Parallel adaptive algorithms for function optimization. Technical Report CS-81-19, Vanderbilt University, Nashville, T.N., 1981
- [13] D.E. Goldberg. Genetic Algorithms in search, optimisation and machine learning. Addison-Wesley (Reading, Mass), 1989
- [14] D.E. Goldberg. A comparative analysis of selection schemes used in genetic algorithms. In Gregory Rawlins, editor, Foundations of Genetic Algorithms, pages 69-93, San Mateo, CA: Morgan Kaufmann Publishers. 1991
- [15] D.E. Goldberg. The existential Pleasures of Genetic Algorithms. IlliGAL Report No 94010. 1994
- [16] F. Gruau. Cellular encoding of genetic neural networks. Technical Report, rapport No 92-21, Ecole Normale Supérieure de Lyon. 1992.

- [17] J.H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press (Ann Arbor), 1975.
- [18] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimisation by simulated Annealing. *Science*, 220(4598):671-80, 1983
- [19] H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4, pp461-476. 1990.
- [20] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolutionary Programs*. Springer-Verlag, 1992
- [21] B.L. Miller and D.E. Goldberg. Genetic Algorithms, Selection Schemes and the Varying Effect of Noise. *IlligAL report No. 95009*. 1995.
- [22] S. Nolfi, J.L. Elman and D.Parisi. Learning and evolution in neural networks. Institute of Psychology, C.N.R.-Rome. Technical Report 94-08. 1994.
- [23] N.J. Radcliffe and P.D.Surry. Fitness Variance of Formae and Performance Prediction. *Foundations of Genetic Algorithms III*, Morgan Kaufmann (San Mateo), pp51-72, 1994.
- [24] P. Ross and D. Corne. Applications of genetic algorithms. University of Edinburgh, Department of Artificial Intelligence, Genetic Algorithms Research Group, paper No 94-007. 1994
- [25] P. Ross, D. Corne and H.L. Fang. Successful Lecture Timetabling with Evolutionary Algorithms. Technical Report AIGA-005-94 (submitted to ECAI workshop on Applications of Evolutionary Algorithm), Departement of Artificial Intelligence, University of Edinburg , 1994
- [26] R. Poli. Genetic Programming for image Analysis. *Proceeding of Genetic Programming '96*, Stanford. 1996.
- [27] J.D. Schaffer, R.A. Caruana, L.J. Eshelman, and R. Das. A study of control parameters affecting online performance of genetic algortihms for function optimization. In J.D. Schaffer (ed.), *Proceedings of the Third International Conference on Genetic Algorithms*. San Mateo, Calif.: Morgan Kaufman Publishers. 1989.
- [28] E.G. Talbi and T. Muntean. General Heuristics for the mapping Problem. Institut IMAG, 1993.
- [29] G. V. Wilson. An Introduction to Genetic Algorithms using RPL2. EPCC course, 1995
- [30] D. H. Wolpert and W.G. Macready. No free lunch Theorems for Search. Technical Report SFI-TR-95-02-010, Santa Fe Institute. 1995