Genetic Algorithms with Gender for Multi-function Optimisation

Robin Allenson

EPCC–SS92–01

Genetic Algorithms with Gender for Multi-function Optimisation

Robin Allenson

EPCC–SS92–01

September 1992

Abstract

This report details the implementation of genetic algorithms as multiobjective optimisers using a specific example: the planning of a route composed of a number of straight pipeline segments from one point to another, whilst minimising first length and then environmental destruction caused by the construction of the pipeline. Using two genders, the algorithm is extended to take the two functions into account, both economic and environmental, in one reproductive plan. The use of sexual attractors between the two genders is detailed. Finally, possible extensions to the project are examined.

1 Introduction

1.1 Project Definition

The aim of this project was to produce a program that would take a large array of "soil values" - biodiversity values for specific locations - and start and end points for a pipeline, and construct the pipeline which would consist of straight segments between the two points. The program should minimise both the length of the pipeline (and hence its cost) and also the damage its construction would cause to the environment.

This aim was to be carried out with a genetic algorithm, and to augment a "normal" reproductive plan by using gender and, later, sexual attraction between genders. The intuitive and therefore rather informal motivation for this is that nature uses gender and sexual attraction between genders. Although the reason why this is so common in biological systems is not known, perhaps it does not *need* to be — by simulation in a genetic algorithm, the outcome can be compared with previous methods.

2 Genetic Algorithms: an introduction

2.0.1 Genetic algorithms as optimisers

Classically, search methods were calculus-based. They include gradient-based techniques, as well as linear and quadratic programming. These use auxiliary information such as derivatives, and use a variety of assumptions (such as continuity) about the search space before they actually start the search. They use one point, which attempts to find a local extremum (let us say it is a maximum). This one point moves about the search space in the direction of steepest ascent, found using derivatives. This is named *hill-climbing*. However, hill-climbing can easily get stuck in local extrema. It is also only useful for certain search spaces — those which fit our assumptions, as we need strong domain-specific knowledge. Where these methods apply, they can work well, but they are narrowly applicable.

Another alternative are the *random* methods. Although they make few, if any, assumptions about their search space, which means they are able to adapt to a variety of different environments, they are so inefficient as to make them un-robust. When looking at search methods, there is tradeoff between efficiency and specificity. Calculus-based methods can be very efficient, but they are frequently too specific, making too many assumptions. Random search is very general, but also very inefficient. Enumeration is often just too slow, except for occasional search spaces which are very particularly "nasty" - we have very little useful knowledge about them.

Genetic algorithms are search procedures based on the mechanics of evolution. Instead of using the original search space, they search using a coding of it. The operators that move the algorithm through the search space are defined with respect to the coding of the search space and not the space itself. Instead of using one point to do the searching (as is the norm in classical methods), they use an entire *population* of points. Just as nature uses sexual reproduction, genetic algorithms use *crossover* between two members of the population to produce a child. They also mutate the occasional individual in a facsimile of the mutation found in nature. These *operators* will be explained later.

Genetic algorithms have a significant advantage over other search methods, they are *robust* — allowing them to work efficiently in a number of different environments. This has come through their attempt to emulate natural, biological systems. Most biological systems are adaptive and self-healing; they have features which repair and guide the individual; they can reproduce. On the other hand, such features were unheard of in artificial systems, until genetic algorithms and other parallel, more distributed methods of search were found. Even now, ideas such as real-time adaptive behaviour in artificial systems is rare.

Genetic algorithms differ from traditional, gradient methods in four ways. Firstly, traditionally they place much emphasis on the use of an coding of the search space. Just as our chromosomes encode how we are built, so the *chromosomes* or *genomes* in genetic algorithms encode how each individual in the population is made up. The physical properties of each individual are known as *phenotypic*, whilst the properties

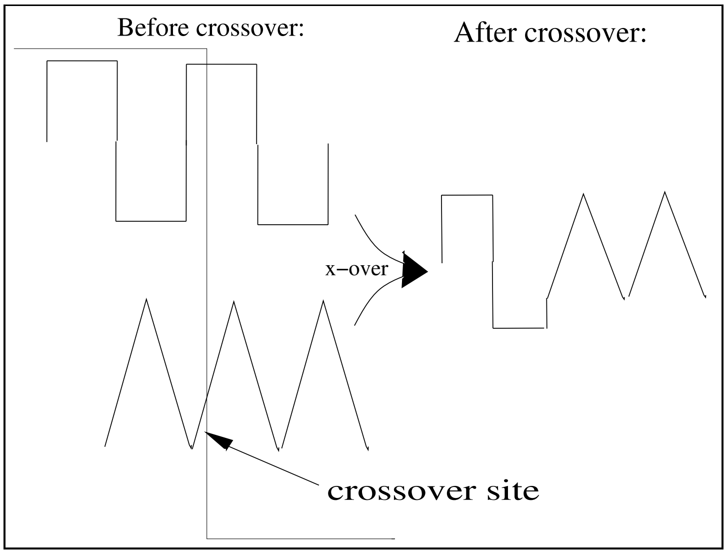

Figure 1: Standard crossover

of the encoding, of the chromosomes are called *genotypic*. How the genomes encode the phenotypic information is a critical choice for the designer of the algorithm.

The second difference, as has already been said, is that genetic algorithms use a population of points to search the encoding of the parameter space. Calculus-based search methods only use one "searcher". By using a large number of points, and by using an operator on the coding of the search space, crossover, adaptive search methods can use non-local moves through the search space and hence do not fall into local extrema so easily.

Genetic algorithms also use stochastic rather than deterministic rules, allowing a probabilistic examination of fruitful areas of the search space.

The final difference is that genetic algorithms use no derived information. This means that there is no need to assume the existence of derivatives, no need to assume the search space is continuous, and no need to assume that the function that is being optimised is unimodal. By not using such assumptions, genetic algorithms are applicable to a much larger class of problems than traditional gradient-based search methods.

2.1 The fitness function

The fitness function measures the performance of a solution relative to the objective function being minimised or maximised. Each individual is judged solely on the basis of the fitness function, the value of which is used to decide whether or not it shall survive until the next generation. Fitter solutions are allowed more reproductive opportunities than less fit solutions, in an analogue of "survival of the fittest". A standard reproductive plan would be to select parents (the selection is biased towards fitter members of the population), create children, and from these form the generation. The process is then repeated. Each new generation is composed of the children of the fittest individuals of the previous generation. To see how new individuals are created from old ones, the functions which adapt the population must be examined.

2.2 Idealised Genetic Operators

To evolve a population, the individuals in it need to be altered. Functions that change the individuals in the population are termed operators. Only two operators are used in genetic algorithms: *crossover* (also

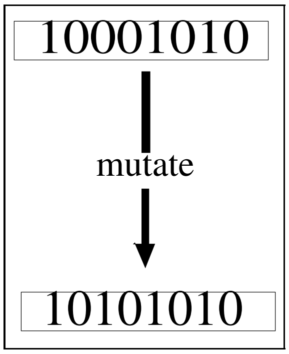

Figure 2: Standard mutation on a bit string

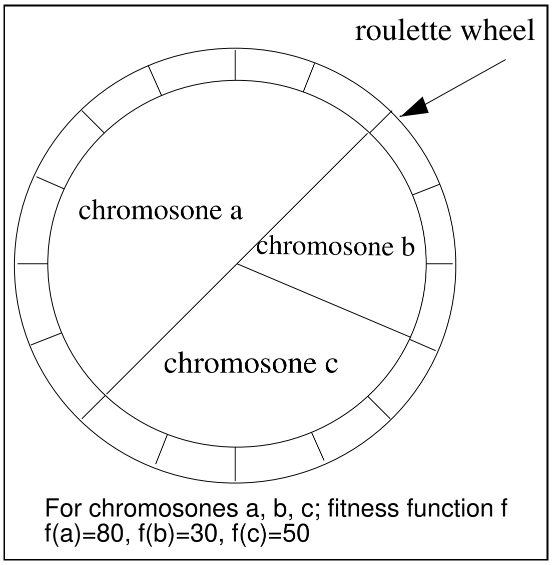

Figure 3: Roulette wheel selection: fitter chromosones are allocated larger portions of the wheel and these are more liely to be selected.

known as *reproduction* and *recombination*) and *mutation*. Crossover is the artificial analogue of sexual reproduction. It takes two individuals and from them produces a child. In the vast majority of cases (over 95% in most algorithms), the child takes on characteristics of both parents. However, occasionally one parent is cloned. To look at how crossover works, examine Figure 1. Note that genomes are not generally waves, this representation is just being used to show more explicitly how *one-point crossover* works. Other varieties of crossover that can be found in genetic algorithms include *two-point crossover*, in which there are two crossover sites, and in which the first and last segments are swapped. One- and two- point crossover can be generalised up to *n-point crossover*.

Mutation takes a single individual and alters it slightly, just as natural radiation in nature causes slight changes in the process of reproducing the odd individual in a population. In nature, the mutation caused by radiation often produces such differences in the genotype as to produce an individual that is unable to live. In genetic algorithms, each chromosome corresponds to a unique phenotype, which is always able to "live". However, it can still be unfit (as defined by the fitness function and the fitness levels of the rest of the population). This means that it will not survive until the next generation. If are genomes are bit strings (a commonly used representation in genetic algorithms), then mutation is as shown in Figure 2.

It is the combination of *selection* and reproduction that make genetic algorithms so good at searching for fitter areas of search space. Selection usually works by being more likely to choose fitter chromosomes that unfit ones. For instance, a standard selection mechanism is *roulette wheel* selection (see Figure 3).

This chooses a new chromosome from the population by spinning a roulette wheel, biased towards the fitter members of the population.

2.3 Some theory: formae

Holland wrote the seminal work on genetic algorithms [Holland, 1975]. In it he gave the *schema theorem*, which attempted to explain why genetic algorithms work so well. An n-bit base k chromosome is a particular type of schema (plural schemata). The schema theorem describes how well schemata will survive to the next generation. However, not all problems are amenable to k-ary representation. Examples include the Travelling Salesman Problem (TSP), graph optimisation, and pipeline routing (of which, more later).

Radcliffe [Radcliffe, 1991a] extended the notion of schemata to formae, where *any* chromosomes, including non-string representations, can be grouped into formae. The notion of intrinsic parallelism, gained from the schema theorem, whereby each chromosome is an *instance* of a number of schemata which the genetic algorithm implicitly processes was extended to that of formae.

2.4 Seven golden rules

Radcliffe's analysis of why genetic algorithms work produced seven design principles, which suggest useful properties of the chromosomal representations, of the formae and of the operators that manipulate these.

The first three principles describe some features of the genome representation that is desirable for a properly functioning genetic algorithm. The reason that genetic algorithms work, says Radcliffe, is that they explore equivalence classes over the chromosome space, according to each of the classes' observed fitness values. This provides further fuel for the argument that binary representations may not be the best way forward. It is not clear that the equivalence classes implicitly induced by a genetic algorithm using binary bit-strings will contain genomes of related performance. This is one of Radcliffe's first three design principles, and is very desirable if the algorithm is to function well.

The design principles are of use in the analysis of why a particular genetic algorithm works, but they should also be useful in *designing* the genome representation and the operators. It was hoped that these design principles could be applied to the project.

The most natural chromosomal representation was, for this project, obvious. A variable length pipeline composed of straight segments was given as the phenotype. The genotype needed to be a linked list of coordinate pairs. This representation has not been analysed to see if it follows the first three design principles: no viable alternatives were discussed. This left the second half of the design principles — the operators. There were three principles that had to be upheld in the design:

- respect
- proper assortment
- ergodocity

Respect states that "crossing two instances of any forma should produce another instance of that forma" [Radcliffe, 1991b]. Garbed in this rather technical terminology, the rule is at first sight rather hard to comprehend. By taking the example of that most fertile of creatures, the rabbit, it becomes a little more understandable. Respect states that if both a father and mother rabbit have white fur, then any of their children produced by crossover alone should also have white coats. The rule should now be much more intuitively sensible.

The second design principle that had to be taken into account when designing our operators was *proper assortment*. This states that "given instances of two compatible formae, it should be possible to produce a child which is an instance of both formae". Two *compatible* formae are two formae which have at

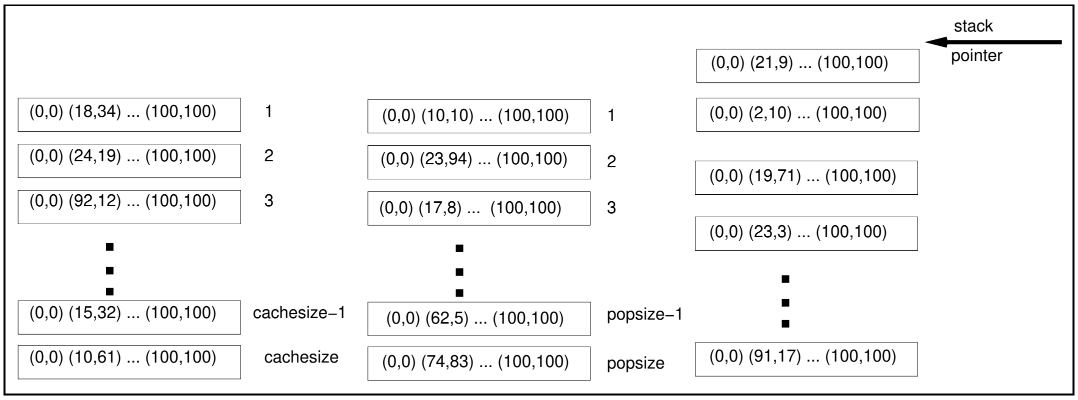

Figure 4: RPL data structures: the cache, population and stack

least one chromosome that is an instance of both of them. So, returning to our example, "brown fur" and "brown eyes" are two compatible formae, as a brown-eyed, brown-furred bunny is easily imaginable. However, "brown fur" and "white fur" are incompatible. Now proper assortment states that given a father rabbit with brown eyes, and a mother rabbit with white fur, it should be *possible* to produce a baby rabbit with both white fur and brown eyes using crossover alone. Note that proper assortment merely states this should be possible: it does not state that every cross should result in a properly assorted rabbit.

The final principle that had to be upheld was that of *ergodicity*, which states that "it should be possible, through a finite sequence of applications of the genetic operators [crossover and mutation], to access any point in the search space given any population." This translates to the notion that even if there is an entire population of brown-eyed bunnies, it should still be possible to produce a red-eyed rabbit. This task usually falls to the mutation operator.

2.5 The Reproductive Plan Language

There are an unlimited number of genetic algorithms. The actual algorithms themselves are specified by a *reproductive plan* which describes how to get the first generation, and thenceforth how to get the new generation from the old. Typically, with work in progress, this changes from week to week, depending on what specific angle of a project is being examined. For instance, in this project there were five main reproductive plans used, with a huge variety of intermediate plans used to "testbed" the instructions used in each plan. Each algorithm may use a different chromosomal representation and there exist a great number of crossover and mutation operators (even if we restrict ourselves to operators for Holland's "standard" n-bit k-ary representation). To be useful, a framework for genetic algorithms, on which researchers can hang ideas and quickly implement them, must show a balance. The framework has to be general enough to cope with arbitrary genome representations, but at the same time it needs to be simple to implement a new plan.

This balance is shown by a general framework for genetic algorithms developed at the EPCC last summer, by a then Summer Scholarship Programme Student, Claudio V. Russo [Russo, 1991] and extended by an MSc student Graham Jones [Jones, 1992]. The *Reproductive Plan Language* or *RPL* is a simple interpreted programming language on which it is possible to implement any genetic algorithm. It uses a mixture of representation-independent and representation-dependent instructions to build the reproductive plan. These instructions include selection schemes, monitoring (to gather statistics), operators, initialisation instructions, as well as inbuilt repetition constructs.

2.5.1 RPL data structures

Most genetic algorithms require some sort of "creche" in which to store children whilst recombining parents. All need somewhere to store their current population of chromosomes. Most also use some temporary storage whilst manipulating chromosomes, be it for mutation, crossover, or the gathering of statistics. These three needs are supplied by the three RPL data structures: the cache, population, and stack respectively (see Figure 4). Each element of these structures contains a reference to a genome (chromosomal structure is irrelevant to the operation of the data structures), a fitness value, and a *scaled* fitness value. The latter value is the fitness of the individual scaled against the fitness of the entire population. How this scaling operates is up to the user - it can be set by one of the predefined schemes, or a new one may be invented.

The population and cache are both arrays of "members". A member is RPL's way of describing an individual. The sizes of both data structures are set before the genetic algorithm starts its reproductive cycle. They may not be changed during a run, but as RPL is interpreted, it is easy to alter some constants in the reproductive plan, and re-run it. There exist instructions to swap the population and cache, which means that it is very easy to build up a set of children, and then ship them in as the new generation. There is also code available to sort either array according to any comparison function. When storing references in the cache or population, the top member of the stack is popped and stored in the appropriate position. When deleting references to members from either array, the reference is pushed onto the stack.

2.5.2 Parallelism

RPL was designed with an implementation on a medium-grained, distributed-memory MIMD computer in mind [Russo, 1991]. Recently, Graham Jones put this design into effect on the Edinburgh Computing Surface [Jones, 1992]. The lexical analyser and parser are written in `lex` and `yacc`, and all communications are written using CHIMP, EPCC's Common High-Level Interface for Message Passing.

A master processor and a number of slave processors all use the same reproductive plan. However, the master ignores the "working" part of the plan, and acts as statistics gatherer, analysing the results that come back from the slaves.

It has been a noted problem of genetic algorithms that *premature convergence* can occur, whereby the population settles to a set of non-optimal chromosomes. One way round this is to add heuristics which encourage *speciation*. A *species* is a class of chromosomes which have common characteristics. Speciation is simply interbreeding. Each species equates to a local optimisation. With the occasional crossbreed, there is a novel exchange of genetic material, as each genome is likely to be significantly different.

An analogy may help. Imagine a set of islands, with a small population of similar animals on each island (similar enough to produce living offspring when two animals from different islands mate), a species. Mostly, the animals are landlubbers, and stay away from the water, recombining with their own species for a number of generations. Occasionally, however, individuals wander onto a passing ship, and get off at a neighbouring island, resulting in a crossbreeding. This is termed *migration* when the individual leaves a species, and *immigration* when the individual is an incoming foreigner, ready to be assimilated.

In Jones' implementation of parallel RPL [Jones, 1992], processors are islands. Migration and immigration are controlled explicitly through the use of two RPL instructions "migrate" and "immigrate". Using these instructions, it can be arranged that the top member of the stack migrates with a certain probability, corresponding to the idea of an individual wandering onto a ship.

2.6 Multiobjective functions

There has been very little work done on applying genetic algorithms to multiobjective functions. This is surprising — a method of optimisation that claims to be robust should surely be able to cope with a number of functions to optimise simultaneously. Most real-life problems are of this type, involving a number of soft constraints to be optimised simultaneously.

There are also applications of algorithms that are able to optimise a number of non-commensurable constraints, such as cost, safety and performance, or cost and environmental concern, as is being used in this project.

Typically, the objective of multiobjective optimisation programs is a *set* of solutions, and not just a single best. The program should aim to present a set of good solutions, and show the user some of the kinds of tradeoffs that should be expected if a single solution is wanted. The ideal set that could be produced by such a program would be the Pareto-optimal set.

Pareto was a 19th century French mathematician who proposed some useful concepts dealing with vector quantities. For two vectors of the same size, the equality, less-than and greater-than relations require that these relations hold element by element. A fourth binary relation, "partially-less-than", is defined thus: given **x** = $(x_1, x_2, \ldots, x_n)$, **y**=$(y_1, y_2, \ldots, y_n)$, then **x** is partially-less-than **y** ⇔ $\forall i : x_i \leq y_i \exists i : x_i < y_i$ If the function is being minimised, then if **x** is partially-less-than **y**, it is said that **y** is dominated by **x**, or that **y** is inferior to **x**. The Pareto-minimal set, then, consists of non-inferior vectors: those members of the population that are not dominated by any other. So, one member of the set will dominate each vector outside the set. However, in the set, no vector is dominated.

Using the pipeline genetic algorithm as an example, some members of the Pareto-optimal set will be of quite a high cost (economic fitness function), but of a low biodiversity destruction (environmental damage function). Other members will be of the reverse persuasion: bad environmentally, but good economically. Hopefully, members that take both functions into account would also be found.

This remainder of this section describes three of the few past ideas that have attempted to expand the capabilities of genetic algorithms so that they are able to take more than one fitness function into account.

2.6.1 Weighted sum

A typical approach to optimising more than one function is to reduce the formulation of the problem from a vector to a scalar one. This can appear tempting for two reasons:

- We may think that the newly formulated problem has only one solution, rather than a set.
- The programmer can choose from a plethora of standard approved approaches that have been used before in the domain of single criterion optimisation.

However, this is trying to do the impossible: create a scalar of two non-commensurable functions. It also reduces the decision space prematurely — before enough information is available. The weighted sum is not a viable solution to multi-function optimisation.

2.6.2 VEGA

J. D. Schaffer created *Vector Evaluated Genetic Algorithms* or *VEGA* [Schaffer, 1985]. This was a genetic algorithm that attempted to optimise n optimisation functions $f_1, f_2, \ldots f_n$ (see Figure 5 for VEGA applied to this specific project). VEGA splits the population into n equal parts (2 halves for this project) and selects parents for each part according to a different fitness function, that is, for any genome in part n of the population VEGA uses f_n to calculate its fitness value. The population is then shuffled, and the genetic operators (crossover and mutation) are applied as per normal.

Schaffer views VEGA as optimising one function which is comprised of a number of dimensions, which is in contrast to the views of the authors of this project who look upon a multiobjective genetic algorithm as optimising a number of non-commensurable functions simultaneously. That is, Schaffer views VEGA as optimising $g(x) \equiv f_1(x)e_1 + f_2(x)e_2 + \ldots + f_n e_n$, where e_i is the ith vector in the standard base. On the other hand, we would view it as optimising $f_1(x), f_2(x), \ldots, f_n(x)$ simultaneously. The difference is, of course, purely interpretational.

Schaffer was worried that "middling" individuals, as he called them, would be lost by this method. So, he added two heuristics to alter this bias. Firstly, he added a preference when parents were selecting their "mate" (there was no notion of gender in Schaffer's work), so that they were more likely to select non-dominated individuals. This was achieved through numeric adjustments to the fitness before ranking. Because this particular genetic algorithm required the sum of the fitness measurements to result in zero,

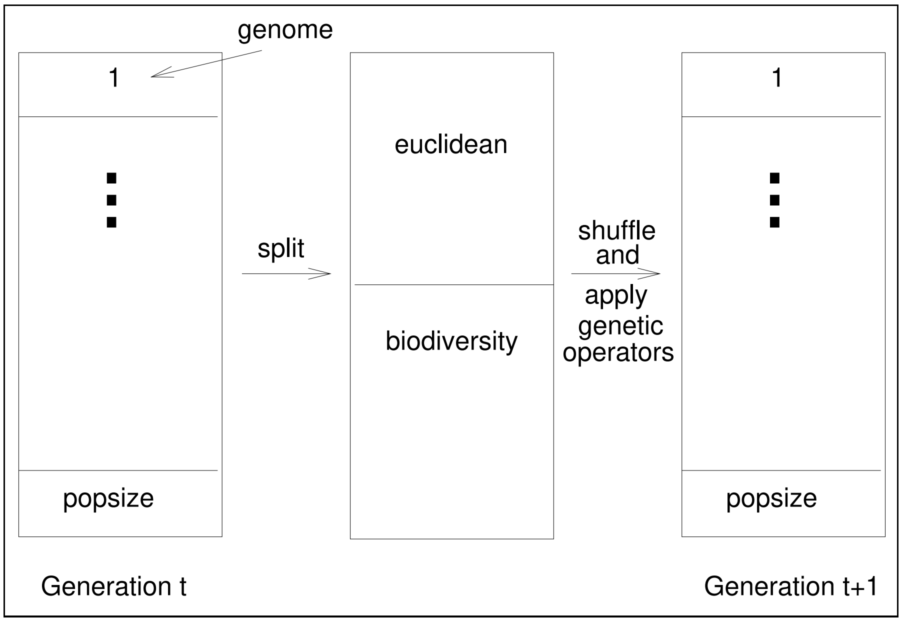

Figure 5: Schematic of VEGA

small penalties were also deducted from each non-inferior individual. A second heuristic encouraged crossbreeding, which meant mating between two individuals whose best performance was on the different fitness functions.

The two heuristics were implemented by selecting a parent at random, and then selecting a mate whose performance was most different from the first parent. This was achieved through using a "performance space", and then using the mate that was the maximum distance from the first parent. Two different metrics were used.

VEGA replicated the search of a traditional (single objective) genetic algorithm using only one fitness function. This would suggest that VEGA is generalisation from a scalar genetic algorithm to a vector. When tested on functions with greater number of dimensions, a "dangerous property" of the first heuristic (which was more likely to select non-dominated individuals) was discovered. It was likely to produce premature convergence in the population.

This was caused by individuals in early generations being non-dominated. This meant that the sum of dominated penalties was much larger than would have been hoped. It gave an overwhelming advantage to a few: the local optima that the population would eventually settle on. Generations later on did not have enough variety in the gene pool to get the system out. The second heuristic (which made members more likely to choose mates that were dominant in other dimensions) fared no better.

VEGA proved more robust than random search. Surprisingly, it was slower, but more likely to escape from local extrema. Schaffer suggested that comparisons between random search and VEGA "contain no small amount of 'apples versus oranges.'" The methods differ significantly in both the way the searches are halted and the number of solutions presented to the user. However, basic similarities exist: they "both contain stochastic elements, both conduct multidimensional search" [Schaffer, 1985], both may be started with random information and both may make use of prior knowledge of the search space.

Both heuristics were dropped when it was found that they appeared to cause premature convergence. Although the populations produced better results without them, maybe if some other technique (such as speciation) had been employed, results could have been better. This provided some of the motivation for using gender (Schaffer's reasonable results using genetic algorithms on multidimensional functions), and for using speciation when using the parallel Reproductive Plan Language.

2.6.3 Kurwase

F. Kurwase also attempted to solve the problem of extending genetic algorithms to cope with a number of objectives, that is, fitness functions [Kursawe, 1984]. He, in a similar way to Schaffer's VEGA, composed a vector of probabilities. This was used for each member to be rated. It gave the probability of using a certain fitness function (or of using one dimension of the fitness function, to use Schaffer's view). If a uniform vector probability distribution is assumed and the random number generator used random, then this should have the same effect as VEGA.

However, Kurwase allowed the vector to change over time, so that it could take into account one fitness function more than another. He found that sometimes the population had a very slow "reaction time" to respond to the sudden variation in the probability vector. To counteract this, the individual's genes were extended by using recessive as well as dominant information in the genotype. Kurwase omits an explanation of how recessive genes help to cope with changes in the probability vector.

Kurwase's genetic algorithm did achieve a final Pareto set of individuals, with a good spread of values. He states a good probability distribution for the probability vector, and a probability for exchanging dominant and recessive genes.

2.7 Similar representations

One of the difficulties inherent in this project was that an order dependent and variable length crossover was being used. For this, the standard operators are inappropriate. Imagine attempting crossover on two bit strings whose length differs. How long should the resulting child be? Will it have a meaningful phenotype? The second question is the most important. Certainly, the standard crossover operator, albeit slightly modified, may be used. But, will any meaningful children be produced? If the representation is order dependent, then recombination can often produce meaningless offspring. Obviously, we are using an order dependent representation: $\{(0,0),(50,50),(60,60),(100,100)\}$ (an ideal solution for the Euclidean metric) is very much better than $\{(0,0),(60,60),(50,50),(100,100)\}$.

Y. Davidor created a modified crossover, called *analogous crossover* [Davidor, 1989] to deal with the problems of order dependent, variable length genotypes. In contrast to the standard "homologous" crossover operator which determines corresponding cross points according to their respective position in the genome, analogous crossover uses the *phenotype* as the correspondence cross point criterion.

It is not the purpose of this paper to discuss fully the modified operator as it was for a different (although similar) representation altogether, but suffice to say that the resulting genetic algorithm compared very favourably with random search (the search space was a highly complex one) and hill-climbing (the search space was also multi-modal). Davidor gives no corresponding results for a similar genetic algorithm that used analogous crossover, which would have been useful in comparing the effectiveness of the new operator.

3 Stepping stones to a multi-function algorithm

Using RPL allowed us to gradually build up the genetic algorithm gradually, "stepping" from one algorithm to another by a new modules, each of which could be testbedded before implementation into the program. It was at this juncture in the software engineering process that RPL's framework structure was vital.

There were six steps in theory. Firstly, the algorithm should minimise length, or cost of the pipeline. Next, the other objective function could be tested in a separate genetic algorithm to ensure that the biodiversity damage function was also working. Adding gender to this would enable the two fitness functions to be "merged". The fourth step was to change the selection to $\mu + \lambda$, a German Evolutionstrategie. If there was time, we would adding sexual attractors based on the fitness of the individuals. This would crudely

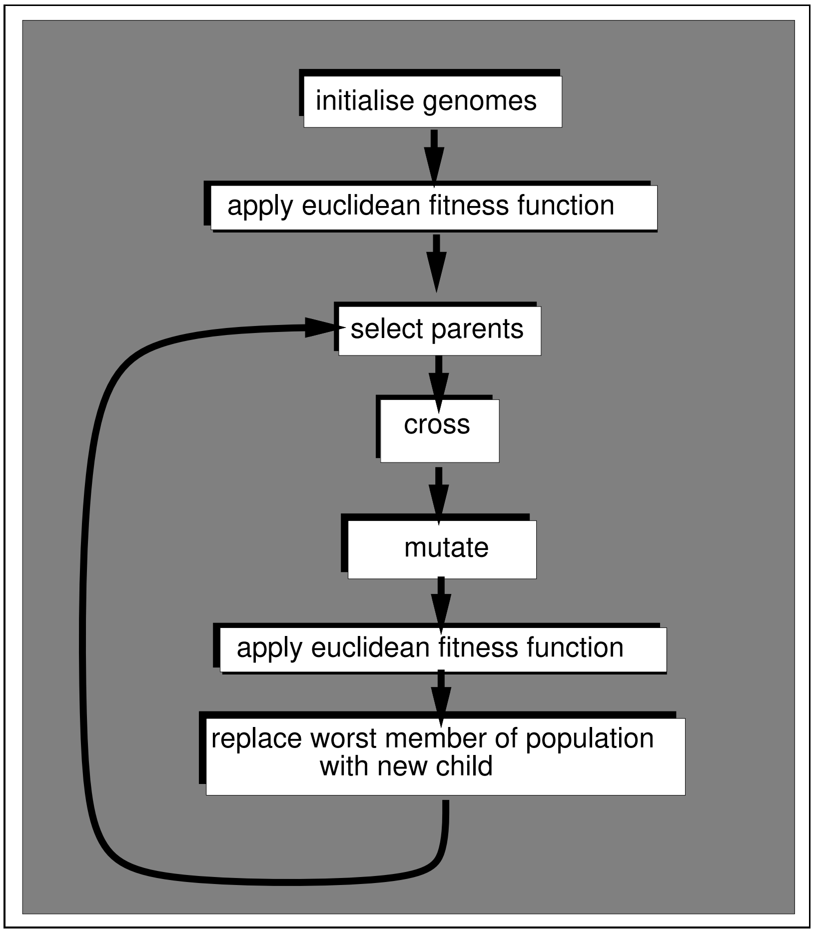

Figure 6: Flow diagram of the euclidean genetic algorithm

imitate the way that some individuals in a population are judged to be more sexual attractive than others. The final step would add a little more colour to this process, by enabling attractors to based genotypic or, indeed, phenotypic information. This might correspond one animal being attracted to another animal because its prospective mate's 2100th gene in its third chromosone was of a certain type for the former case. In the latter, we find the intuitively more plausible idea of being attracted to a certain characteristic of external make-up of an individual (e.g. "blond hair").

3.1 Euclidean

The early stages in building a multiobjective genetic algorithm were to test that the fitness functions were fully working and that genetic algorithms incorporating them did what was expected of them. Without this, there would be little point in trying to extend them into a more complex algorithm.

As has been mentioned, it was here that the power of RPL really came into its own. Because RPL is interpreted, the reproductive plans that it uses can be easily altered, extended or simplified in any way, without a long and tedious development cycle.

The first stage was to construct the economic function. These was based on the length of the pipeline, measured using the Euclidean metric. This first objective was called the Euclidean fitness function. This was implemented in a simple genetic algorithm (see Figure 6).

In it two common ideas from genetic algorithms, binary tournament selection and `replace-worst` were used. Binary tournament selection is a method of choosing parents to recombine. To choose one parent,

Figure 7: A variable length crossover

two members of the population are randomly chosen, and the better of the two is selected with a certain probability, otherwise choosing the worst. In RPL, two members of the population are copied onto the stack. The better is left there with the probability that is given, otherwise the worst is left. Whichever parent is not chosen is popped. This is repeated for the other parent. Binary tournament selection is implemented in RPL as `bin-tourn-select`.

The probability of picking the better potential parent enables us to control the aggressiveness of the algorithm in choosing its parents. If the probability is kept very high then the population is likely to converge prematurely. Certain members of the population will be picked repeatedly. The gene pool will get so small that premature convergence will soon result (this is very similar to what happened to VEGA when heuristics were added).

`replace-worst` simply replaces the worst member of the population with the current child. By repeating this a number of times for each generation, and using better than average parents (which can be chosen by using `bin-tourn-select` with a parameter that is greater than 0.5), it would be hoped that the population would improve (there are no guarantees here, this is a stochastic process). This is dependent on having good operators, and a good representation.

Here is the RPL program that was used:

```
%% produce popsize random genomes
%% to initialise the population
doi &counter 1;
    random-genome &length;
    euclid;
    assign-pop &counter;
untili &counter &popsize 1;

%% for the main reproductive cycle, binary tournament select parents
%% and replace the worst of population with the new-born child.
cycle
    doi &counter 1;
        bin-tourn-select 0.6;
        bin-tourn-select 0.6;
        one-pt-cross &crossRate;
        move-node-mut &mutateRate;
        euclid;
        replace-worst;
    untili &counter &loopMax 1;
endcycle
```

Note that this is a program of reduced size, with the non-essential elements taken out. All identifiers preceded by an ampersand are RPL variables. `doi` ... `untili` is a repeat construct that operates much like a `for` loop.

`one-pt-cross` is recombination operator (see Figure 7) that works in a similar way to one point crossover, although it has been modified to take account of the variable length of the chromosomes that it uses. It was decided that genotypic, rather than phenotypic, crossover (homologous crossover) would be used as this was considerably easier to implement. By using this, in some ways the work of Davidor of was being rejected. However, once this crossover was working, adding analogous crossover could be one of the many extensions for the project. The recombination used ran along the linked list of vectors for a short distance and then "jumped" to the other chromosome, and carried on to its end. If the second chromosome was shorter than the first then only the first part of the first chromosome would be used. If the start and end points were not the ones that our program needed these would be replaced by the correct ones.

It is interesting to note that `one-pt-cross` neither respects nor properly assorts any obvious set of formae, but still works in spite of it. These conditions (Radcliffe's "design principles") are obviously not essential, but useful, characteristics of genetic algorithms. Maybe there exists an analogous crossover that properly assorts and respects some suitable set of formae. This would be the ideal operator for us, but very hard to program (and design!).

`move-mode-mut` took the uppermost member of the stack and, with a certain probability, mutated the chromosome by moving a random node by a random offset. This upheld the principle ergodocity of Radcliffe's forma analysis. A second mutation operator `swap-node-mut` was designed and implemented, but later rejected. This takes the uppermost member of the stack and, with a certain probability, mutated the chromosome by swapping two randomly chosen nodes. However, this does *not* uphold the principle of ergodicity: pipelines will never be able to use nodes that are not in the initial random population.

This program worked well. Although with a small number nodes it came to a solution quickly (with five nodes, it can evolve a nearly perfect solution in under 100 generations), larger numbers of nodes proved much more difficult. Solutions came, but could take a number of minutes or (when numbers of nodes were greater than around 20) hours. Due to a difficulty in using both graphical output and the computing surface, the parallel program was not used.

3.2 Biodiversity

The biodiversity genetic algorithm was very similar to the Euclidean algorithm (see Figure 8) — binary tournament selection and replacing the worst member of the population with each new child were both still used. However, instead of using the Euclidean fitness function, a function was now needed that took into account the biodiversity destruction caused by the construction of the pipeline.

For this, a map extracted from a *Geographical Information System* or *GIS* was used. A GIS contains one or more sets, or maps, of spatially referenced data. Each set is comparable to a "slice" of a human-readable map. One slice may represent roads, another railways, another the number of people in each grid reference square. Only one slice was used in this project, it represented the biodiversity of the soil in each grid reference. Although the size of two dimensional array of values to be used could be varied (all memory was dynamically allocated), typical values were large: around 0.5Mb.

Next, an algorithm was needed to sum the grid squares that a pipeline crossed. As a pipeline consists of a number of straight line segments, a standard algorithm was used to sum each of the squares that one straight line passes. This "standard" algorithm (Bressenham's algorithm) was originally from the world of graphics. It is used to draw straight lines: from the start and end points of a line, Bresenham's algorithm is used to set pixels in between. The setting of pixels was changed to the summing of biodiversity values. By summing the total values in each pipeline segment, the total biodiversity destruction caused from constructing a pipeline could be reached.

In order to test the algorithm fully, without loading large soil arrays, it was decided that an extra RPL function would be added to fill the biodiversity array with test values. For instance, this would enable us to build a small maze of high biodiversity values, and allow the algorithm to work its way through. This function `fill-rect` filled a rectangular part of the array with a certain value. By using a number of calls to this instruction in succession, a small map of different values could be constructed.

Here is the RPL program that was used:

```
%% generate a very simple maze of soil values
```

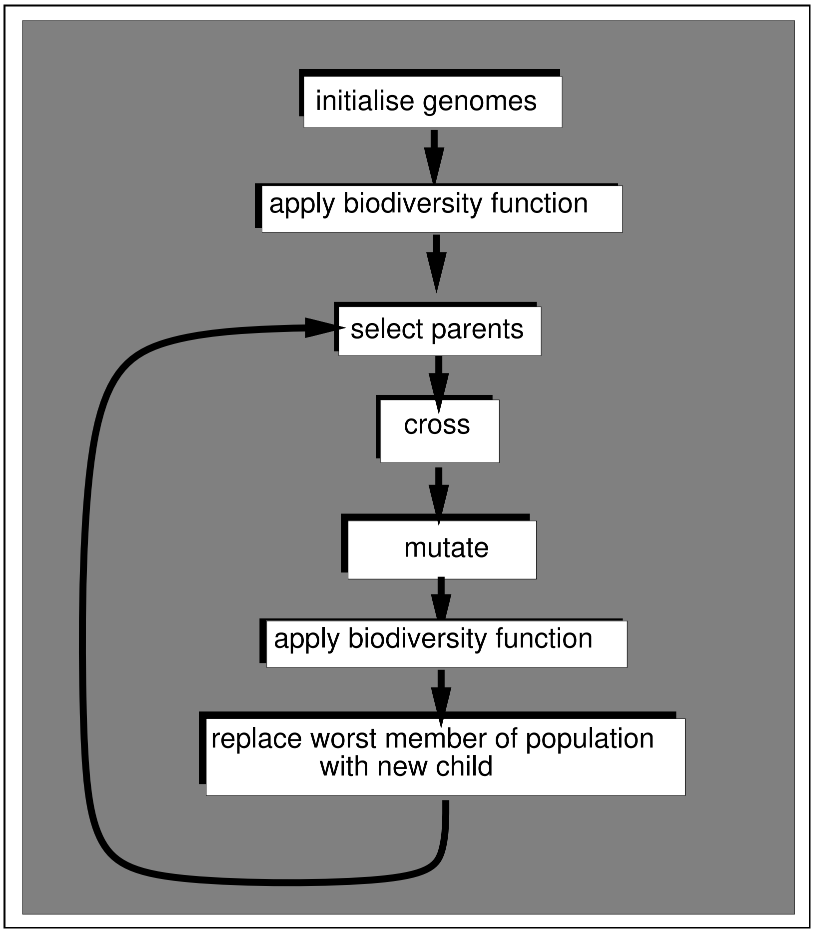

Figure 8: Flow diagram of the biodiversity genetic algorithm

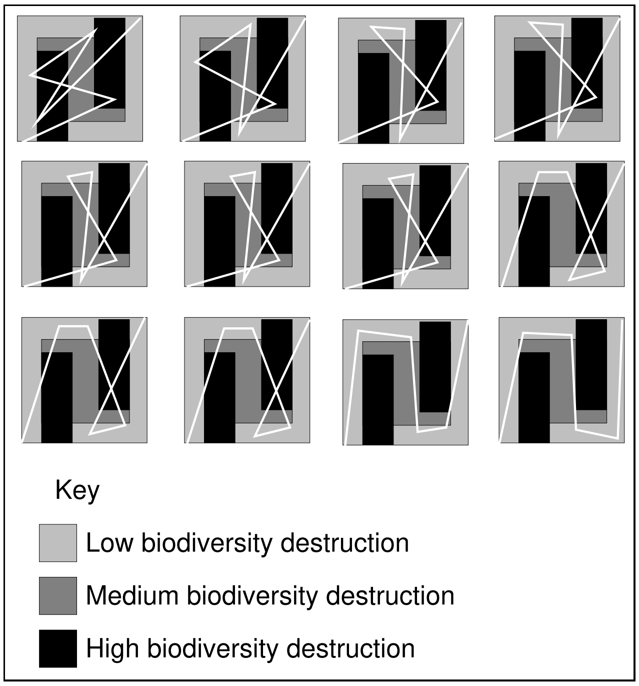

Figure 9: Successive generations using the biodiversity algorithm

```
init-soil;
fill-rect 2  2  98  98  10;
fill-rect 10 0  40  70  63;
fill-rect 80 0  40 100  63;
fill-rect 50 0  90  20  63;
fill-rect 50 30 90 100  63;

%% produce popsize random genomes
%% to initialise the population
doi &counter 1;
    random-genome &length;
    biodiversity;
    assign-pop &counter;
untili &counter &popsize 1;

%% for the main reproductive cycle, binary tournament select parents
%% and replace the worst of population with the new-born child.
cycle
    doi &counter 1;
        bin-tourn-select 0.6;
        bin-tourn-select 0.6;
        one-pt-cross &crossRate;
        move-node-mut &mutateRate;
        biodiversity;
        replace-worst;
    untili &counter &loopMax 1;
endcycle
```

The program worked well enough to find its way through a simple maze that was constructed (see Figure 9). Obviously, in a real-life situation, values would be spread throughout a large range, and not just the toy values of “low”, “medium” and “high” biodiversity destruction.

3.3 From one function to two functions

At this stage, two fitness functions that were working well on their own were had been programmed. The next part of the project involved constructing a genetic algorithm to minimise the two component functions; to construct a multiobjective genetic algorithm.

It was decided that a good way forward would be to use *gender* to augment the algorithm. By allocating each function a gender: male for environmental and female for economic (say), allowing only male-female pairings for recombination, and randomly allocating the gender of the child at birth, it was hoped that the two functions could be minimised at once. Each sex would only ever be judged on its own fitness function. The basic design of the algorithm that had been used previously would be kept, although some changes would be needed to alter the underlying code that described what each RPL instruction would do.

`random-genome` constructs the genotype of an individual. This was extended so that the instruction took an extra argument describing the gender of the new child. `bin-tourn-select`, or binary tournament selection, would remain the same; it is representation independent. Also, the operators that had been used up until now, `one-pt-cross` and `move-node-mut` would stay as they were. However, a new fitness function had to be constructed. This was called, unimaginatively, `fitness`. It examined the gender of each member of the stack, and allocated a value to the fitness field of the members' references.

To keep the number of each gender constant, there were three choices:

- randomly allocate gender at birth
- examine total number of males and females. Allocate gender so the numbers stay equal.
- initialise the population with an equal number of males and females. Ensure that gender of child is same as that of member being replaced.

It was both the first and the last that were eventually chosen.

There was no problem of normalisation. Some might think this would be needed because the worst member of the population is being replaced, which would involve searching for the worst member. This, in turn, involves comparing which members have greater or lesser values than other individuals. Hence, normalisation might be thought to have been needed between the Euclidean and the biodiversity fitness values. This is not so. The sex of the *child* was randomly allocated, and then the worst member of *that* gender was chosen.

Here is the RPL program that was used:

```
%% initialise the population
  doi &counter 1;
      random-genome &length &female;
      fitness;
      assign-pop &counter;
  untili &counter &halfpopsize 1;

  doi &counter 1;
      random-genome &length &male;
      fitness;
      assign-pop &counter;
  untili &counter &halfpopsize 1;

  cycle
      doi &counter 1;
          bin-tourn-select 0.6;
          bin-tourn-select 0.6;
          one-pt-cross &crossRate &length;
```

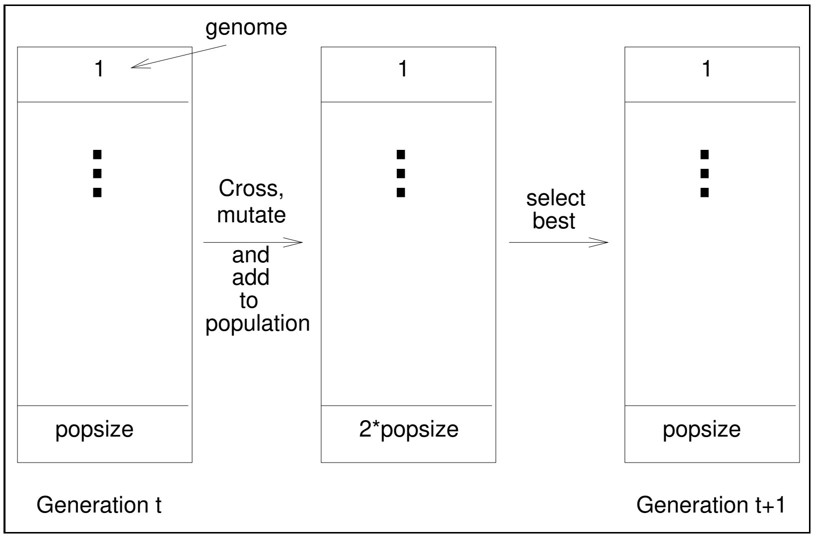

Figure 10: Flow diagram of the $(\mu+\lambda)$ reproductive plan

```
        move-node-mut &mutateRate &length;
        fitness;
        choose-sex;
        replace-worst;
    untili &counter &loopMax 1;
endcycle
```

It should be obvious that allowing only male-female pairings is a stringent version of VEGA's second heuristic. This produced premature convergence in VEGA, but no such problems were found with this project. Much higher population sizes than VEGA were being used: Schaffer used 30 members, we used 500. This meant that our gene pool would have been more likely to survive repeated use of a small number of individuals. It was also hoped that using a number of different processors, that is, implementing speciation, would reduce the likelihood of this problem.

This algorithm does appear to take both economic and environmental factors into account. Once again, simple mazes were used to test its ability.

3.4 $(\mu+\lambda)$

Evolutionstrategien are the German school of genetic algorithms. One of their algorithms, called $(\mu+\lambda)$, is a reproductive plan that gives every member of the population a reproductive opportunity. That is, everyone is given a chance to mate. Remember, the standard genetic algorithm would select some of the population to be parents. The selection is usually based on the fitness of the prospective parents.

$(\mu+\lambda)$ lets every member of the population be a parent. It moves through each member, and then chooses a random parent to be its mate. Then it adds all the parents to the amassed children for its new population. However, this leaves twice the number of members in population. Obviously, it is useless to carry on this process *ad infinitum*, and also, there will be many unfit children and parents in our population. So, the less fit half is killed off, and the fitter half is selected as the new population (see Figure 10).

$(\mu+\lambda)$ allows us the possibility of making fit children from unfit parents — after all, this is entirely possible — which a "standard" genetic algorithm would discourage (by rarely selecting poor parents). At the same time, the unfit members of the population are still removed.

Here is the RPL program that was used:

```
%% initialise the males
doi &counter 1;
    random-genome &length &male;
    copy;
    fitness;
    assign-pop &counter;
    assign-cache &counter;
untili &counter &halfpopsize 1;

%% initialise the females
doi &counter &halfpopsize;
    random-genome &length &female;
    copy;
    fitness;
    assign-pop &counter;
    assign-cache &counter;
untili &counter &popsize 1;

cycle
    doi &counter 1;
        bin-tourn-select 0.6;
        bin-tourn-select 0.6;
        one-pt-cross &crossRate &length;
        move-node-mut &mutateRate &length;
        choose-sex;
        fitness;
        assign-cache &counter;
    untili &counter &popsize 1;

    sort-cache-by-sex;
    take-best-of-cache;
endcycle
```

There is another good reason for introducing $(\mu+\lambda)$. Looking ahead, it would be interesting to add *sexual attractors*, which would give parents (that is, any member of the population, if we use $(\mu+\lambda)$) the chance to choose their mate, rather than randomly selecting it. $(\mu+\lambda)$ is ideally suited to this sort of selection scheme, as even if the crossover is unsuccessful, resulting in an unfit child, the algorithm will remove the child when it selects the fittest of the grouped children and adults for the next generation.

3.5 Sexual Attractors

In nature, sexual attractors are found everywhere. For whatever reason this is, it is possible that this reason will work with genetic algorithm. This is a very informal argument for saying “if it works in nature, it is worth trying it”.

So, pressed into using attractors as an interesting experiment, how should they be implemented? There are hundreds of possible methods, but eventually it was decided to find something simple to implement and then, depending on how well this worked, go from there.

The standard genetic algorithm does not feature explicit sexual attraction. However, the end result is similar to what would happen if attractors were implemented that always had a propensity to mate with the best in the population. The main difference is that the standard genetic algorithm uses stochastic rules, rather than deterministic ones. There would be a considerable detrimental effect on the population (namely premature convergence), if every member of the population attempted to mate with the current best. This is exactly what would happen if we used $(\mu+\lambda)$, and each solution picked the best of the

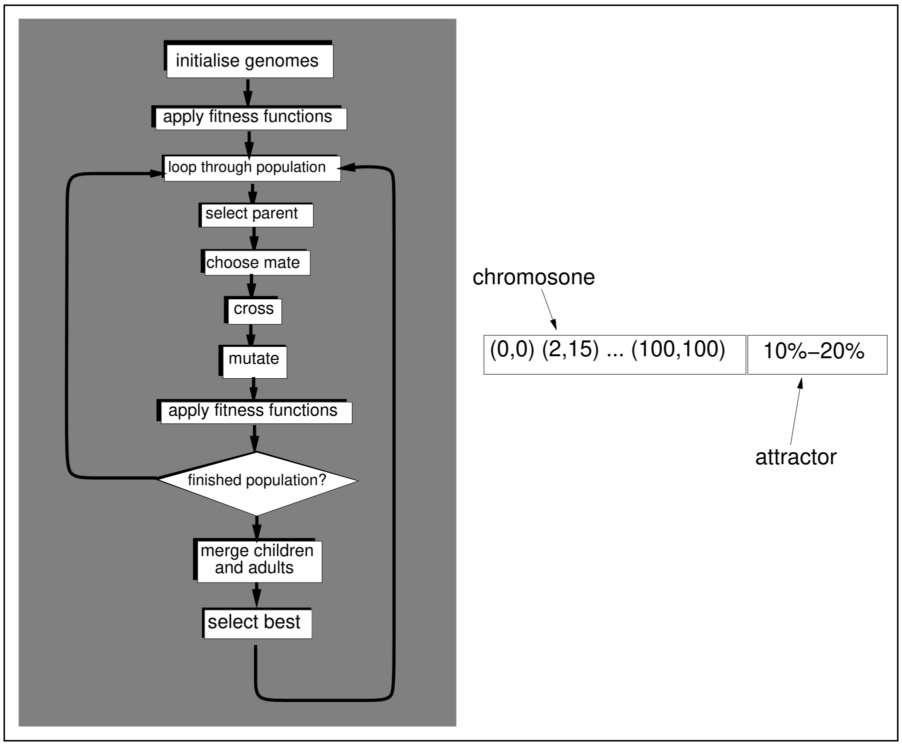

Figure 11: Flow diagram of $(\mu+\lambda)$ with sexual attractors

opposite gender as its mate. So, another method is needed.

Intuitively, even the worst members of the population aspire to mate with the best members of the population. This is not, however, what actually happens. Each parent mates with the best member that he or she is able to find. Also, each individual has a different idea of what is the best member of the population. In a standard genetic algorithm, the fitness function rules all. Whatever one individual may "think", the absolute fitness function is correct. Thus, an initial experiment was planned to see whether attraction to fitter individuals could be evolved. When implementing the preferences of each individual (their attractor), the fitness function was used as a base. That is, the rank of each member of the population was used as the mark for a genotypic attractor.

The simple implementation that was decided upon gave each individual a preference: which decile they would like their mate to come from (see Figure 11). Intuitively, it might be thought that always using the top decile would be best, but this may (as has been discussed) cause premature convergence, no matter what size population is used. The number of divisions is arbitrarily large, so although the word "decile" has been used, it could equally be "percentile".

What would happen when a child was produced from recombination of two parents? How would the new attractor be made? A number of possibilities were examined:

- randomly compose the attractor
- clone the attractor from a random parent
- clone the attractor from a selected parent, selection being probabilistically biased towards selecting a fit parent
- use a crossover operator to form a new attractor

It was this fourth alternative that was eventually settled upon. Just as the first three possibilities would be inappropriate for a genetic algorithm, they are also inappropriate here because we have the choice to utilise the power of genetic algorithm. The crossover operator for the decile attractor was Radcliffe's *random, respectful recombination operator* or R^3, an operator that respects and properly assorts, if the representation will allow it [Radcliffe, 1991b]. It is applicable to real numbers, and it is this form of R^3 that we use.

4 Conclusions

4.1 Work completed

Although the first three plans are working, there is a bug in the fourth plan (namely, $(\mu+\lambda)$) which means that it does not finish composing a new population after a certain number of generations. It is believed that this is to do with the dynamic allocation of memory for chromosomes, and the consequent freeing of these structures. However, once this bug is fixed, both $(\mu+\lambda)$ and the decile sexual attractors will be working.

Ideally, the graphics should work whether the transputers are being utilised or whether a serial machine is being used. At the time of writing, using the VOGL graphics library, this is not the case. A simple fix is to `#ifdef` out the sections of the code that use graphics. This stops graphical output with the Computing Surface, but at least the genetic algorithm will work. It is possible to output the chromosomes by printing their contents in text form, but this is tedious, and does not allow the user to see what it is going on. So, the libraries need to be changed so that the graphics and the transputers are compatible.

4.2 Further additions

This section details a number of useful and interesting extensions to the project:

- an RPL instruction to remove loops
- phenotypic operators
- phenotypic attractors

Loops are never of any use, as they increase the length of a pipeline but have no beneficial effect (negative biodiversity destruction is not allowed, although it is a novel idea). This could be implemented by checking each pipeline for loops when the algorithm has finished the random initialisation of genomes, and then ensuring that no operators can insert loops. To check if a loop exists, and to remove it, an intersection function, which calculates if and then where two vectors intersect has been coded. The top level of the loop checking procedure is simple to implement.

At present the operators have remained the same all the way throughout the project in that they all use the genotype of the the individuals in the population, rather than the phenotype. This makes the programming considerably easier. Instructions that work with the genotype often do the things at which computers are good, whilst instructions that work with the phenotype often manipulate the individuals in way at which humans are good. This is obvious when it is realised that the genotype is the internal representation of the physical image: the phenotype. No wonder it is easier to describe genotypic functions in a programming language: the genotype is the "machine view" of the phenotype.

However, phenotypic operators often work better [Davidor, 1989], so the extra effort could be rewarded with a more powerful genetic operator, and hence a more powerful genetic algorithm. Some phenotypic crossover operators were conceived.

A recombination operator, `cross-swap-at-every-interx`, creates a child whose pipeline is copied from one parent up to the first intersection, then from the other parent up to the next intersection, and so on, until the parent that the child is using is finished. This operator had one problem: when the pipelines did not intersect the operator would clone one of the parents. This is perfectly allowable, except that interchange of genetic material is supposed to be going on in crossover, and none is taking place in this operator. Coding for this function has already been started (the intersection function has already been discussed).

Just as operators may work better when their designs manipulate the phenotype, so attractors may also work better if they are based on the physical properties of an individual. (All instructions, at machine level, obviously actually manipulate the genotype. However, some are better *described* by how they manipulate the phenotype). This would seem to be more comprehensible at an intuitive level. After all, few people lust about a particular gene in their opposite sex's third chromosome. Examples abound, however, for someone to lust about some part of their opposite sex's physical makeup, their phenotype.

One example of an phenotypic attractor is a metric for similarity between two individuals. An individual might be more attracted to another pipeline that is similar (a simple analogy might be English men being attracted to English women, and not, say, Russian women). This similarity metric could be based on a number of functions, such as total distance between the ith node, sum of the squares of differences between the ith angle and so on.

Another possibility that was conceived was to have a list of phenotypic properties, including, for example, maximum and/or minimum and/or mean length of segment, angle of bend, number of loops and so on. Attractors could then be based on these.

There is enough work for another project in designing and implementing attractors, as well as comparing and analysing the results.

References

[Davidor, 1989] Y. Davidor. Analogous crossover. In *International Conference on Genetic Algorithms*, pages 98–103, 1989.

[Holland, 1975] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[Jones, 1992] Graham Jones. Parallel reproduction plan language. Technical report, Edinburgh Parallel Computing Centre, The University of Edinburgh, 1992.

[Kursawe, 1984] Frank Kursawe. *A Variant of Evolution Strategies for Vector Optimisation*. PhD thesis, Vanderbilt University, Nashville, Tennessee, 1984.

[Radcliffe, 1991a] Nicholas J. Radcliffe. Equivalence class analysis of genetic algorithms. *Complex Systems*, 5:183–205, 1991.

[Radcliffe, 1991b] Nicholas J. Radcliffe. Forma analysis and random respectful recombination. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 222–229. Morgan Kaufmann (San Mateo), 1991.

[Russo, 1991] Claudio V. Russo. A general framework for implementing genetic algorithms. Technical Report EPCC–SS91–17, Edinburgh Parallel Computing Centre, University of Edinburgh, 1991.

[Schaffer, 1985] J. D. Schaffer. Multiple objective optimization with vector evaluated genetic algorithms. 1985.