MUTANTS: A generic genetic algorithm toolkit for Ada 95

Matthew Caryl

University of Glasgow

28.4.97

Abstract

There are many computationally hard problems to which no algorithm exist that can find an optimal solution in a reasonable time. Genetic algorithms offer a shortcut, able to produce good but not perfect results much faster. They are based on the principles of natural evolution and use selective breeding on a population of potential solutions to gradually derive more successful solutions. Designing the best algorithm for the job is difficult because of the wide variety of possible variations. This paper presents a genetic algorithm toolkit which provides a library of components which can be connected together to form many basic algorithms and then customised for any particular problem.

Acknowledgements

Many thanks to Bill Findlay for suggesting the topic in the first place and providing guidance throughout the project. Thanks to Cordelia Hall from whom I borrowed the violin music notation example and who provided me with test data. And finally to my father for accidentally getting me interested in the whole concept of artificial life and genetic algorithms.

Contents

Appendices

1 Introduction

Millions of telephone calls are made every day and every call requires a connection, a route through countless cables and exchanges. Routing calls is known to be a computationally hard problem and no search algorithm can find the optimal solution in a reasonable period of time. Even so we expect connections to be made in seconds rather than hours. This and many related problems can be tackled by fast probabilistic techniques which produce good but not perfect results.

Genetic algorithms have proved competent for this challenge and have been applied to a variety of problems from job scheduling to image registration. They are based on the principles of natural evolution and use selective breeding on a population of potential solutions to gradually derive more successful solutions. A genetic representation must be chosen for the problem so that random changes and combinations can be made, the biological equivalent of mutation and crossover. Crossover combines features from successful individuals whilst mutation ensures variation and prevents the stagnation that often leads to poor solutions.

Potential solutions to the routing problem might be encoded as a list of exchanges which define the path from source to destination for each connection:

A B C	B D A	G D A	C E H

Mutation could select an alternate exchange for the list and crossover combine two previous solutions:

A B C	B D A	G **C** A	C E H

A B C	B D A	G **M** A	C **N** H

Each generation these solutions are evaluated and parents are selected to produce the new population. At every repetition bad solutions are weeded out and desirable solutions are encouraged. After many generations the population will, with luck, converge on some near-optimal solution.

Different problems pose different requirements for the algorithm, maximum computation time, real-world deadlines, or solution quality. Genetic algorithms can vary greatly in these and other respects depending on the choice of components. For example, an alternate mutation operator for the routing problem could have been biased towards overworked exchanges swapping them with those under less pressure. Extra time would be spent examining the exchanges but fewer generation would be required to distribute the workload. Changing just one component of an algorithm may reveal markedly different behaviour, making the design of a suitable algorithm a time consuming art form.

Previous work on genetic algorithms has followed two main paths - *classic* and *hybrid.* Classic genetic algorithms use a binary vector representation and a few standard reproduction operators. Identical algorithms can solve separate problems by their particular interpretation of the vector. Unfortunately such universal algorithms are slow to find good solutions and slower to find optimal ones. Hybrid genetic algorithms have no standard representations or operators. Instead, characteristics of the problem at hand are encoded into the algorithm to produce faster and better results but at the expense of loss of generality and additional time in writing the algorithm. For typical real world problems hybrid genetic algorithms are proving to be more effective and the additional experimentation required is worthwhile.

Despite their variety of forms, separate genetic algorithms share many common components. A unified model which organised components into independent classes would simplify algorithm design and a toolkit supporting such a model would expedite experimentation with new components and encourage software reuse. Currently toolkits and other techniques which combine many components in a systematic manner are still in their infancy. This project develops MUTANTS, a fast-prototyping object-oriented toolkit defining a range of generic classes containing several predefined components. New problems, outside the normal range of the toolkit, can be solved by extending existing classes.

The main goals of this project are to create a unifying model for genetic algorithms, design a toolkit based on this model, and demonstrating its flexibility with three example algorithms.

When reading this report it is helpful to have some understanding of computationally hard problems which necessitate the use of probabilistic techniques. The basic principles of genetic algorithms are introduced by the report but more extensive knowledge is useful when considering some aspects of toolkit design. Object-oriented design techniques are used but not discussed within the report. These techniques allow both functionality and state information to be encapsulate together inside genetic algorithm components.

The remainder of this project introduces genetic algorithms, discusses the design issues behind the toolkit, and demonstrates its flexibility with some example algorithms.

- Chapter 2 - introduces genetic algorithms, the concepts and terminology
- Chapter 3 - follows the development of the toolkit's unifying model
- Chapter 4 - implements the classic Travelling Salesman Problem using toolkit components
- Chapter 5 - develops the experimental components required to solve an optimisation problem for violin music notation
- Chapter 6 - summaries the successes and failures of this project and describes further work that they suggest

2 Genetic Algorithms

2.1 Nature's Genetics

In the world around us there are millions of species each different from the next and within these species there are countless billions of individuals. All of this variety stems from the creation of life around three billion years ago. For over two billion years the most advanced organisms consisted of only a single cell. The structure and behaviour of each individual is controlled by a set of instruction called *genes* written in a four letter alphabet on long strands of *DNA*. Structures within the cell interpret these letters and then changes chemicals harvested from the environment into products for power, repair, or reproduction.

These single-celled organisms reproduce by *fission*, splitting into two components, each taking a copy of the DNA instructions. But the environment is hostile, ultraviolet light streams down from the sun and radiation seeps up from the rocks below. Occasionally letters in the instructions are damaged or the copying mechanism fails. The offspring has slightly different genes from its parents and there are new responses to the environment. Sometimes such individuals are failures, condemned by their genes. Sometimes they just cannot compete with their fellows for food. Most often they are little different from their parents but occasionally these changes, these *mutations*, make the individual better than its neighbours. Those who are better suited to the environment grow faster or survive longer and ultimately give rise to more offspring. These offspring would inherit any advantage and replace the parent type over a number of generations.

The process of species changing over time to become better at survival within the environment is called *evolution*. One advance that it produced was a new method of reproduction. Cells began to carry some of their genes in rings called *plasmids.* During meetings between individual they could exchange bundles, passing the environmental knowledge embedded within the plasmid from individual to individual. Successful plasmids spread amongst the population but unsuccessful plasmids encumbered their carrier who would die sooner and have less chance to spread the bad genes. Greater benefits came when certain genes came together that worked better in combination than in isolation. The likelihood of random mutation equipping one cell with both genes is low but with the *crossover* of genes between individuals these traits can arise separately and, once they become common in the population, join together within one individual.

From this soup of tiny creatures all the diversity of the planet has arisen, from roses to butterflies, mushrooms to mankind. Single cells have grouped together to form complex multi-cellular lifeforms. In these higher organisms genes are arranged in structures called *chromosomes*. Humans have 26 pairs of chromosomes, one of which is either an XX or an XY pair, the first pair for a female and the second for a male. This division of the species helps with the replacement for plasmid exchange used by simpler creatures. Two individuals each provide one half of the chromosomes needed to make up a complete set. Combining chromosomes is an opportunity to gain something better for their offspring. Deficiencies in one parent can be made up by the corresponding genes from the other parent. Like any other animal humans try to select a successful mate to get the best chance of survival for their child.

At the change over from single celled to multi-cellular forms there was a rapid period of development and diversification called the *Cambrian explosion*. A huge range of species appear but most have no living relatives today. Although they were well equipped for life they did not survive. Perhaps they had disadvantages compared to others but maybe they were just unlucky. Such die backs are common throughout history. The dinosaurs lived on Earth for about 150 million years before their extinction around 65 million years ago. One reason given for their extinction was the impact of a asteroid in the area which is now Central America. Devastation and destruction followed but more importantly disruption to the environment. Animals that had done well died out unable to adapt fast enough. In a geological instant half of species on the planet were wiped out. This is just one example of the many mass extinctions our planet has suffered. With each such event the diversity of life is devastated and will never return to the same extent.

Evolution is the aggregation of thousands of semi-random events and the natural pressure to reproduce or die. This has been compared to climbing a mountain. Starting at the bottom a species rises up through ever fitter forms. There is no going back and those who try will die. A species cannot even stop upon reaching one of the mountain peaks because these mountains are not constant. The ever changing physical and biological environment determines what is good and bad for each generation and with this mountains slowly rise and fall. Every species must continually search for perfection and in so doing will change what is perfect for those around it.

2.2 Computational Genetics

Problems can often be simplified to the search for an optimal solution through a range of possible solutions. Such search spaces can be huge and an exhaustive search would take an unrealistic time even with the fastest computers. Simple problems may have linear *time complexity*. A problem of size 10 will take twice as long as a problem of size 5. *Intractable* problems can have factorial time complexity. A problem of size 10 will take 10*9*8*7*6 times longer than a problem of size 5. Figure 1 shows the time taken to solve linear and factorial time complexity problem on the same computer. Such intractable problems must therefore be solved by pruning away or ignoring most possibilities.

Artificial intelligence problem solving strategies fall into two categories - *strong* and *weak* method. A weak method makes few assumptions about the problem domain and hence is portable to other areas but little pruning is done and time scales grow rapidly for large problems. A strong method avoids this by making major assumptions about the problem domain but it may require significant redesign when moved to even a closely related problem. In between these two extremes genetic algorithms can be tuned to become strong or weak, either widely applicable or problem specific. Aspects of genetic algorithms can be seen in a number of other weak search techniques.

Randomly generating many solutions to a problem is usually simple. Each solution can then be *evaluated* and given a score according to its success. Call routing solutions are just lists of connected exchanges, these could be randomly assigned. Evaluation involves adding up the workload at each exchange. Positive scores for under capacity exchanges and penalty points for overworked ones.

Hill climbing is a more directed approach similar to a specie's climb up the mountains of fitness. Starting with a single random solutions and mutate slightly, swapping a single exchange. Perhaps generate a few more variations to get an idea of the locale area. Then pick the solution with the highest evaluation and start mutations again. Evaluations will rise until a optimal solutions is found. However, this may not represent the global optimum but rather a local peak with higher values to be found elsewhere. To increase the chances of finding these continue with a new random solution and hope for a better final solution.

Problem Size	Linear Time
10	0.0001 seconds
100	0.001 seconds
1000	0.01 seconds
10000	0.1 seconds

Problem Size	Factorial Time
10	0.36 seconds
20	7714 years
30	*
40	*

* indicates times greater than age of the universe

Figure 1 - Time Complexity

Simulated annealing introduces chance into its climb to avoiding the traps of local optimums. An identical random start and mutation but the best evaluation is not always selected. The probability of choosing a new solution is a function of both old and new evaluations and an additional parameter T known as the temperature. Initially the temperature is high and so is the probability of selecting a new solution independent of its evaluation. This continues until the system reaches some sort of balance with little more progress being made. The temperature is lowered and it is allowed to settle again closer to the optimum peak. At a temperature of zero the system acts like a hill climber to find the locale and hopefully global optimum.

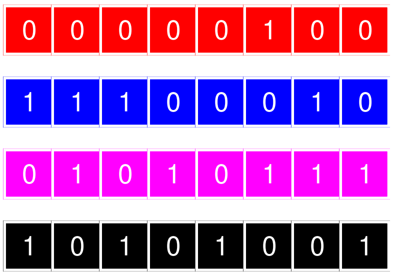
John Holland was the first to apply additional natural principles to problem searches. Within a population of random solutions crossover occurs between individuals occurs. Such combinations can bring together successful attributes making increasingly better solutions. Since their introduction genetic algorithms have been employed on a variety of search space and optimisation problems including call routing, job scheduling, and database query optimisation.

Genetic algorithms are derived from the field of biology and much of the terminology has been borrowed. A solution is know variously as an *individual*, a structure, a genotype, or a string. Sometimes chromosome is used as an alternative but this is misleading as some individuals could contain several chromosomes. Chromosomes are sequences of genes, also called features, characters, or decoders. Fixed positions within the chromosome which hold particular characteristic are *loci* and the values that might appear are *alleles* or feature values. Finally data within an individual is its *genotype* but the meaning of this representation as defined by the user is its *phenotype.*

2.3 Basic Algorithm

Genetic algorithms have six basic steps in common. The Blues is an artificial problem manufactured to illustrate these. Possible solutions to the problem are the integers from 0 to 255. Each integer represents one of four colours: red, blue, purple, and black. Success for the algorithm is measured in the number of blue solutions found in comparison to other colours.

First, a set of potential solutions must be initialised to form the starting population. To represent numbers in the range 0 to 255 a random 8-bit binary vector is used. Figure 2 shows a selection of individuals from the population. Every number has a



0 0 0 0 0 1 0 0

1 1 1 0 0 0 1 0

0 1 0 1 0 1 1 1

1 0 1 0 1 0 0 1

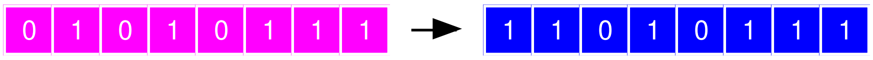
Figure 2 - Random population

corresponding colour picked by a secret function (which examines the first two bits of the binary representation).

Second, each solution is evaluated according to its fitness, in this case its colour. Blue is given priority and scores 10. Lower scores are for solutions further down the mountain of fitness – 5, 3, and 2 are given to red, purple and black respectively.

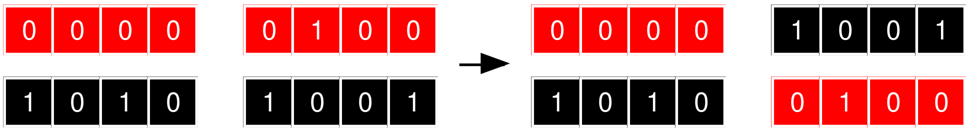
Third, new solutions are created using mutation and crossover on the current population, typically with more crossover than mutation, say a 3:1 ratio. Conserving and combining features is generally more helpful than varying them. A mutated descendent will differ from its parent in only a single bit as seen in Figure 3. Crossover requires two parents and produces two offspring. Select a random point along the binary vector and split each parent at this point. Then exchange the tail sections of each vector to produce the offspring shown in Figure 4.

Selecting those individuals to breed requires some element of chance. In nature some animals are lucky and some unlucky but those with better genes reproduce more. Genetic algorithms allow more offspring from high scoring individual than from low scoring individuals. A technique called *roulette wheel selection* has a probability of



0 1 0 1 0 1 1 1 → 1 1 0 1 0 1 1 1

Figure 3 - Mutation operator



0 0 0 0 0 1 0 0 → 0 0 0 0 1 0 0 1

1 0 1 0 1 0 0 1 → 1 0 1 0 0 1 0 0

Figure 4 - Crossover operator

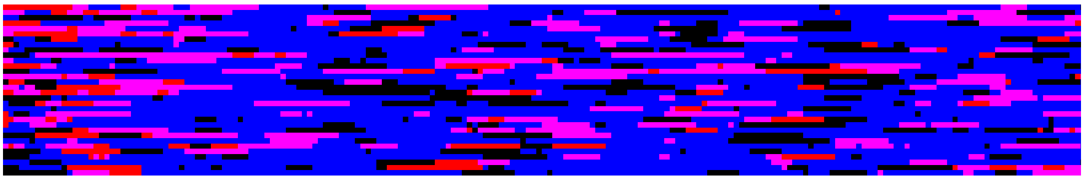


Figure 5 - The Blues

choosing an individual proportional to its evaluation. The Blues uses this to select the primary and, if necessary, the secondary parent for reproduction.

Fourth, room must be made within the current population for the new individuals. Here the entire population is removed.

Fifth, the new solutions are evaluated using the 10, 5, 3, 2 scoring system and inserted into the population.

Sixth, if there is no more time then stop, otherwise go back to step three and make some more individuals.

From a random start the roulette wheel picks out more blue individuals than other colours. If mutation makes an offspring then there is only a 2 in 8 chance that a colour change will take place. If crossover is used then there is only a 1 in 7 chance that the crossover point will be between the first and second bit, no other points can affect the offspring colour. Even if that point is chosen the other parent would have to differ in the second bit to produce a change. From this and the 3:1 crossover-mutation ratio it can be calculated that there is only a 13 in 122 chance (about 11%) of a blue individual giving rise to a non-blue offspring. With each generation more blues will be picked and most will stay blue. Other colours are also resistant to change but are picked less often. As can be seen from Figure 5, blue wins on average. Each dot is an individual and each column a population so the change from left to right represents the evolution of the population. The population is never totally blue because these probability predictions only work well for very large populations. In small populations of 32, as above, chance may result in another colour being predominant.

In summary the steps of a genetic algorithm are:

1. Initialise population with individuals
2. Evaluate each individual in the population
3. Create new individuals by mating between those in the population; apply mutation and crossover as the parents mate
4. Delete members of the population to make room for the offspring

5. Evaluate the new individuals and insert them into the population
6. Stop if time is up, otherwise go to 3

Classic genetic algorithm components are shown in The Blues. A *binary vector* used to represent integers was the first ever genetic algorithm representation. Initial solutions were generated with *random binary initialisation* which fits almost any circumstance. Evaluation technique are normally unique to an algorithm but in many instances decoding produces numbers and mathematical functions are then applied. Reproduction is with *single bit mutation* and *single point binary crossover*. Deletion of old and insertion of new individuals is often treated as a single process with one name. A *generational model* creates an entirely new population for each cycle.

2.4 Genetic Theory

Current mathematical theory for genetic algorithms is based on a binary vector representations. The notion of a schema is introduced, a template made of 1's, 0's, and *'s. A schema represents all strings which match it on all positions other than *.

Consider the schema `(11******)` which matches 64 strings including

`(11000000) (11111111) (11010101)`

It would also match all blue solutions from The Blues. Every schema matches 2^r vectors, where r is the number of * symbols in that schema. The *order* of a schema is the number of fixed positions, ie. the number of 0's and 1's. The *defining length* of a schema is the distance between the first and last fixed position. These are written o(s) and ∂(s) respectively, for example:

$$S^1 = (***001*110),\ o(S^1) = 6,\ \partial(S^1) = 10 - 4 = 6$$
$$S^2 = (****00**0*),\ o(S^2) = 3,\ \partial(S^2) = 9 - 5 = 4$$
$$S^3 = (11101**001),\ o(S^3) = 8,\ \partial(S^3) = 10 - 1 = 9$$

A schema's order proves useful in calculating the survival probabilities under mutation and the defining length can be used similarly under crossover.

A few simulations or a bit of mathematics will show that an above average schema gives rise to an exponentially increasing number of offspring in future generations.

Further, low-order schema are more likely to survive the recombination process of crossover. From this work has come two theories:

> Schema Theorem - Short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations of a genetic algorithm.
>
> Building Block Hypothesis - A genetic algorithm seeks new-optimal performance through the juxtaposition of short, low-order, higher-performance schemata, called the building blocks.

For a more detailed explanation and proof of these theories see Michalewicz (1992).

2.5 Vector Representation

Vectors, also called strings or arrays, are sequences of values all belonging to the same type. Bits, or booleans, are a typical choice but integers, real numbers, or complex structures are equally valid.

With the common binary vector representation comes a number of genetic operators for mutation and crossover. Single bit mutation and one point crossover have already been described. Multiple applications of these operators can give rise through a number of generations to any possible recombination of an individual but most would be stopped in the first generation. There are several crossover operators which provide a more diverse mixing of genes. *Two point crossover* chooses two random points and exchanges the central section of between individuals. *Multi-point crossover* extends this to a fixed arbitrary number of point pairs. *Segmented crossover* works with random number of point pairs. *Shuffle crossover* first randomly permutes the binary vector, applies another crossover operator, and then reverse permutes the vector.

Although flexible binary vectors have their limitations. To represent 100 real numbers in the range -500 to 500 accurate to the 6th decimal place would require 3000 bits. Such a long vector gives a search space of 10^{1000} possibilities. Different representation can be chosen to help the genetic algorithm, say a vector of floating point numbers. Specific mutation and crossover operators are needed for this representation. *Random mutation* replaces old numbers with new ones chosen from a specified range and *random creep* adjusts each number up or down by a small amount. Crossover can work in the same way as binary representations or can combine two numbers as an averages.

With the introduction of specified ranges for numeric values reproduction operators may drive an individual outside these ranges. Such individuals represent an impossible or illegal solution. Careful design of operators can prevent this but it is not always possible. One option is to assign a penalty to the individual during evaluation. Selection pressure could then drive out these poorly formed individuals. On the other hand these individuals may have evaluations high enough to overcome this penalty and the population would become crowded with unusable individuals. A more computationally intensive option is to repair the individual, removing its illegal features to produce the closest legal alternative. This can defeats the purpose of a reproduction operator if the final individual looks nothing like its parents.

2.6 Messy Representation

In nature individuals carry redundant or duplicate information, such as multiple copies of genes, or paired chromosomes. This is an important feature of nature’s robust designs. Messy genetic algorithms copy this, allowing redundant or even contradictory genes. A real number vector shows how this is done (0.1, 0.2, 0.4, 0.8) is encoded as ((1, 0.1), (2, 0.2), (3, 0.4), (4, 0.8)). Each value has been replaced by a (position, value) pair. Other alternatives include: reverse order ((4, 0.8), (3, 0.4), (2, 0.2), (1, 0.1)), duplicates ((1, 0.1), (2, 0.2), (3, 0.4), (4, 0.8), (4, 1.6)), and omissions ((1, 0.1), (2, 0.2), (4, 0.8)). Vague representations have to be evaluated with some default values and arbitration between conflicting values.

Only two crossover operators are required - *cut* and *splice*. The first breaks one individual into two less specified individuals. The second joins two individuals into one more specified individual.

2.7 Improved Initialisation

Nature never starts with a clean sheet, new problems are always met with modifications to previous solutions. Following this lead genetic algorithms can start with non-random populations. If an initial population of 100 individuals is required then randomly generate 500 and pick the best. Starting with these solutions is likely to speed up the genetic algorithm's search but could also hinder it in finding less obvious but higher scoring solution.

In an effort to cover the search space completely solutions can be chosen to be as diverse as possible. Generating this diverse population will take extra time. But by preventing any bias in the initial population the genetic algorithm has its best shot at finding the optimal solution.

Sometimes good results can be achieved by combining genetic algorithms with existing systems to produce hybrids exhibiting the best behaviour of both. The previous system starts by quickly generating a number of good solutions. These can be fed into the genetic algorithm as the starting population, hopefully concentrating the search to a more productive area.

2.8 Competitive Fitness

Evaluation functions try to score a representation fairly by what it can achieve, nature rarely does this. Instead examinations are competitive with little hope for the losers, no matter how good. Genetic algorithm's equivalent are fitness functions. After all individuals have been evaluated they pass to a second stage where their evaluations are compared against each other. As usual there are several different fitness techniques that can be applied, only some are mentioned here.

Windowing works on the basis of a certain minimum requirement. All individuals will have this amount removed from their initial evaluation. In this smaller range of positive values the differences between individuals are emphasised. What amount to remove must be decided, perhaps some fixed minimum score, perhaps equal to the minimum evaluation present, or some more stringent value.

In addition to emphasising differences fitness functions can reduce them. *Linear normalisation* ranks all individuals' evaluations, 1st, 2nd, 3rd, and later places are awarded. For 1st place the reward is a larger but not overwhelming share of the offspring. This prevents a single high evaluation individual from producing the lion's share of offspring in the next population. If this does happen important genes can be lost and never recovered. Those in lower places are rewarded less, and those in last place may receive nothing at all.

2.9 Breeding Techniques

Previous sections have looked at specific pieces which make up genetic algorithms but the broader picture has been ignored. These pieces must be pulled together by a breeding techniques such as the generational model used in The Blues. Before examining other models a few general parameters should be considered.

Population size, for instance, can have a great affect on the performance of the genetic algorithm in more ways than one. Choosing a small population size reduces the possible diversity of individuals and may well lead to sub-optimal solutions. Choosing a large population size will slow the algorithm down as there are more individuals to deal with and each generation will take longer.

With the selected reproduction operators there must also come the relative importance of each. Mutation explore new possibilities but can be destructive as well as creative. Crossover is safer, working with preexisting building blocks, but if an important characteristics is lost it is never recovered.

When should a genetic algorithm stop? It may be when there is no more time and an answer due. Perhaps a certain number of generations has always proved sufficient in the past and so it seems appropriate to stop then. If a solution is required with a certain minimum evaluation then the best individual from each generation can be compared against it. When there are no other outward criteria with which to make the decision it is best to stop a genetic algorithm when the majority of the population has converged towards a single answer and no more progress is being made.

The generation model sometimes suffers from the loss of the best individual through chance, particularly when the population size is small. An elitist component can be added to several models to prevent this. Before normal reproduction begins a copy of the best individual is made and transferred to the new generation to guarantee a continuation of its line.

An alternative mechanism to achieve the same effect is to replace a generational with a steady state model. Here only a few new individuals are generated at a time. Now it must be decided which individual should be replaced. One particular alternative called *preselection* compares offspring with their parents and choose the better individual. Or the choice can be made randomly, the only selection pressure will be that provided by parent selection. Or the worst individuals in the population can be chosen. Another technique which goes particularly well with the steady state model is the imposition of a duplicate individual ban. No newly created individual may be the same as any other existing individual. It is computationally expensive to perform all the comparisons but a good way to ensure diversity within the population.

This only scratches the surface of modelling techniques available. Here are quick summaries of a few more.

- Inbreeding - crossover within small populations to emphasis good traits.
- Crossbreeding - crossover between diverse individuals to combine different traits.
- Multi-criteria sub-populations - separate populations with different evaluation functions, occasional crossbreeding between populations.
- Co-evolutionary systems - several parallel genetic algorithms whose evaluation functions are interdependent.
- Mating template - mating only permitted between individuals of the same type.

3 Toolkit Design

3.1 Information Flow

Genetic algorithms have a great many variations but also share common points. All have the common features of initialisation, evaluation, populations, and reproduction. From here that the design of the generic toolkit was started by considering the information pathways of the algorithm. However, this is insufficient because certain specialisations are not described in such a model. In particular there were three areas of deficiency. A large group of genetic algorithms use fitness functions to set up a more competitive environment. Most genetic algorithm make a considered opinion on whether to finish. Finally some genetic algorithms monitor their changing populations over time for historical tracing or additional processing, these are all specifically mentioned in the statement of requirements. Taking all these aspects into account produced Figure 6 which shows information steps within a generic genetic algorithm and the information flow between steps which typically consist of potential problem solutions. Those steps dealing with the three aspects which are not common to all

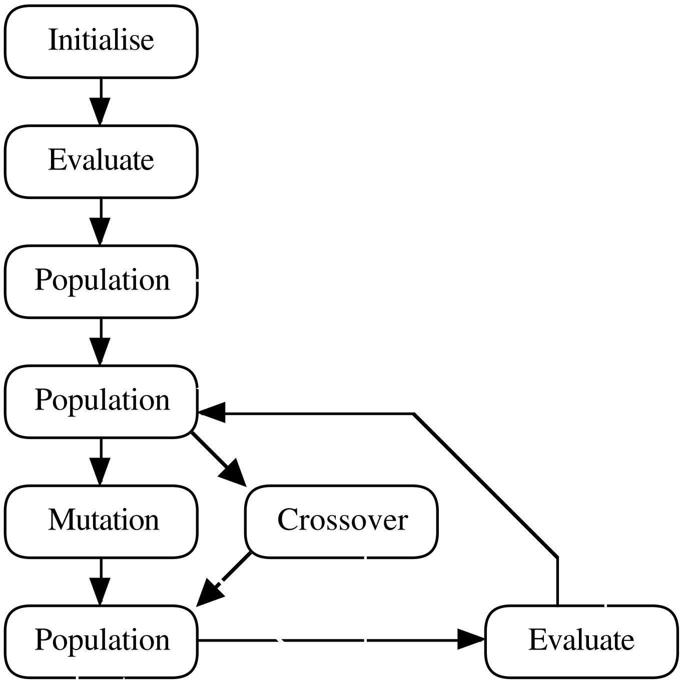


Figure 6 - Information flow

algorithms are shown in grey.

3.2 Toolkit Components

From the beginning of the project the intention had been to produce an object-oriented toolkit. Such a model is well suited to the desired extensible toolkit. Abstract classes would provide the toolkit and users inherit concrete classes for use in a particular genetic algorithm. This facility would also facilitate the construction of the toolkit enabling the range of services to be slowly expanded.

Primary in toolkit design was the notion of flexibility and the speed at which a new algorithm could be assembled. Even so this had to be balanced against implementations which would take an unacceptably long time to run. Several genetic algorithms suggested minor variations which did not fit in with the majority of other algorithms. For example the elitist breeding technique is an extension of the standard generational technique. No other similar behaviours are mentioned in the literature and so the toolkit would have to implement elitism separately and either flag if active or not for a particular algorithm. Such behaviour is contrary to the generic ideal and slows down many algorithms for benefit a few.

Fortunately a clue to the solution is already described, Lawrence Davis (1991). Two stages from the standard genetic model, the deletion of old individuals and insertion of new individuals, are often placed together. This suggested consolidating these procedures into a single object, one to do both jobs. A Model object would also take over most other stages of the genetic algorithm, its exact behaviour depending on the parameters such as reproduction operators but also on that model's particular make up.

Figure 7 shows final choice of classes for the toolkit and the dependencies between them. Each arrow represents one class's knowledge about another. Around the diagram is a circle, this is just a combination of several dependency arrows. As can be seen the Model object has direct control over almost all other classes within the toolkit. Several objects have appeared in the diagram which do not seem to have any correspondence to the information flow diagram.

Random is a pseudo-random number generator to provide for the probabilities needed by a genetic algorithm. By using a single controllable generator for the algorithm experiments can be replicated. Simply by storing the initial random number seed and the parameters used by the genetic algorithm, the same results may be obtained again

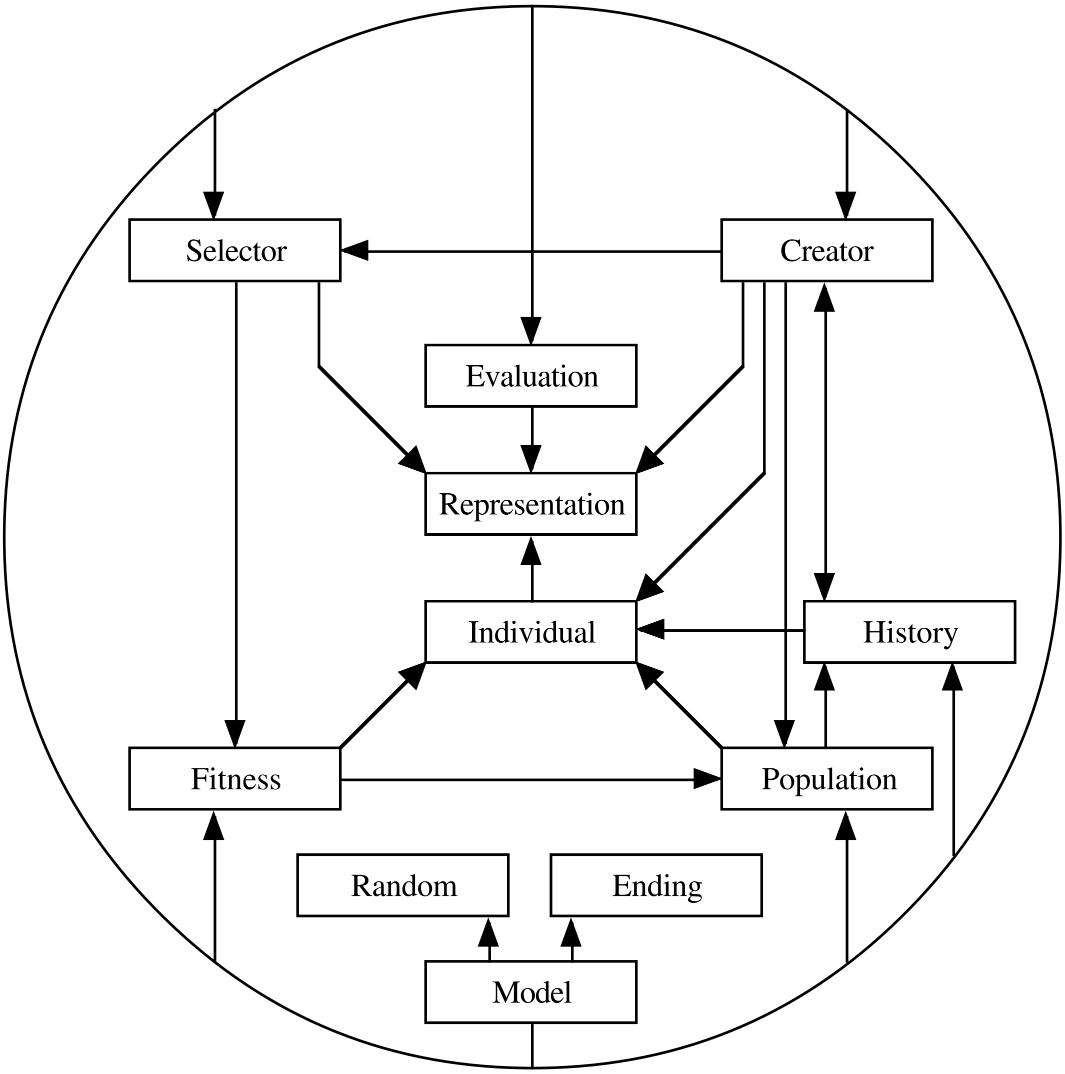


Figure 7 - Class dependencies

and again. If real or arbitrary random numbers were used every individual would have to be recorded and traced to provide the same service.

Representation is a abstract holding place for a single problem solution. Concrete examples of this class include binary vectors, ordered lists of integers, or 3D matrices. With all different representations packaged in the same way they may be dealt with equally by most of the toolkit. Only functions which depend on specific representations such as evaluation and reproduction operators need separate them.

Evaluation is apparently only a function and does not need object status. Simple evaluations can indeed be calculated directly from the representation but many need auxiliary information. An object provides storage space for such details, an example of such is shown in Chapter 5.

Individual is the encapsulation of a Representation. Each individual is uniquely identifiable not only within a single generation but within all generations in a particular algorithm. Even individuals with identical representations may be differentiated. Tracking the development of individuals over time is impossible without this. *Population* provides a simple set for storing individuals.

Ending makes the decision on whether or not another cycle of the genetic algorithm should be made. Several concrete class are available including ones based on time, generation count, and target evaluations. Complex combinations can be formed by joining simple ending objects with And, Or, or Xor operators also implemented as Ending objects.

3.3 Breeding Model

Model encompasses both breeding technique and major control of the genetic algorithm. All algorithms are to be dealt with in a uniform fashion and because of this the Model is sometimes burdened with additional work. Here is the final design used for this class formatted as - method: arguments > result:

```
ctor: Creator,population_size > Model
ctor: input_stream > Model
print: Model, output_stream
run: Model, Evaluation, Fitness, Selector, Creator,
   Ending
run: Model: Evaluation, Fitness, Selector, Creator,
   Ending, History
get_population: Model > Population
```

Two creator methods (ctor's) are used, either populating the model with newly created individuals or recovering individuals specified in an input stream. A complement to this is the print method which records the current status of the model for prosperity. The internal population may be accessed to obtain the result after the algorithm has finished. Most work is done by the `run` method which take several arguments, each specifies one aspect of the genetic algorithm. Two versions of `run` are present, one which takes a *History* object, the other does not. This is one example of where a decision has to be about generality and efficiency. Recording the births, deaths, and parentage information in a genetic algorithm will obviously take extra time. Hence two separate methods, only if this user requires the feature must the additional time be taken.

Other classes keep the work which must be done by the Model to a minimum implementing any single instance is not difficult. Here is an example of the run method from the Generational Model which is used by The Blues:

```
done := false;
while not done loop
   Attach(Fitness, Model.Population);
   Attach(Selector, Fitness);
   Create(Reproduction, population, population_size);
   Transfer(Model.Population, population);
   Done(Ending, done);
end loop;
```

3.4 Fitness Function

Fitness functions are applied after evaluation and before parent selection. An equivalent could also be used when removing individuals from the population but this is not a common feature. Evaluations of all individuals in the population must be presented to the fitness function so that comparisons can be made. One possibility is to in effect reevaluate each individual, assigning them this new value, and then carry to on as normal. Although easier to implement than other schemes this was rejected because it would destroy the initial evaluation which may be required again and could be expensive to calculate.

One solution is a wrapper around the Population class. A fitness object is passed into the genetic model and is then be attached to a given population. When attached the object calculates an associated fitness score once for each individual. To obtain fitness values a fitness iterator is created which moves through the population returning individuals and their finesses. A separate iterator is required so that multiple scans of the population can progress at the same time.

This created a small problem, The Blues does not need a fitness function but must have one because of the decision to apply the same methods to all genetic algorithms. Along with standard fitness function - Windowing, Linear Normalisation, and so on - another, the *Unity Fitness* class had to be introduced. For each individual in the population this returns the same value as found during evaluation. Null operations such as this are often not noticed within a system until a rigorous examination is made.

3.5 Parent Selector

From a fitness capped population individuals must be selected for reproduction. The obvious approach is an abstract Selector object and a number of concrete classes representing the different ways of choosing individuals. Common examples of selection techniques are best, random, and roulette wheel. The consistent approach to genetic algorithms required by the toolkit means others classes are needed. Some particular genetic algorithm prefer low values to high values. Normally one would change the evaluation function to invert values but in preference to that other selectors can be used lowering the burden on the algorithm's implementor. For example an *Anti-roulette Wheel Selector* returns individuals with a probability inversely proportional to its evaluation. If this class replaced the Roulette Wheel Selector in The Blues then black would become the preferred colour. Implementing selectors as objects also gives some speed advantages. When attached to a population any necessary calculations to make the selection can be done once and only once. For example in the instance of a best individual selector one single search for this individual is made and the result can then be recorded. Any future calls to the selector will produce the same individual instantly.

3.6 Individual Creator

Creator class combines both initialisation and reproduction. Original designs had included two separate class but these were merged in an effort to reduce the complexity of the toolkit. Both perform similar jobs, the production of new individuals, and so have similar requirements. Initialisation at the start of a genetic algorithm in the standard case does not normally require parent individuals to draw on. Some genetic algorithms start with a population of specialised individuals who have been generated in other genetic algorithms. Initialisation becomes a matter of cloning old individuals into the new population. Equally, although reproduction normally involves parents, some

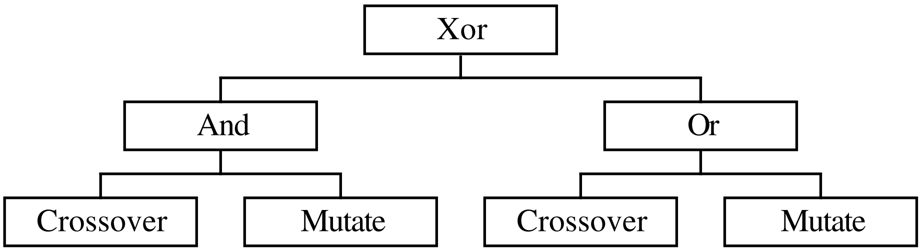


Figure 8 - Reproduction tree

algorithms may occasionally introduce completely new individuals to keep up genetic diversity.

Given that several reproduction operators are used in each genetic algorithm a mechanism must exist to choose between them. One possibility was to pass in a set of reproduction operators for each run of the genetic algorithm. Associated with each is a weight determining the probability of that operator being chosen. A more advanced technique categorises reproduction operators as to whether or not they are combined operators. That is whether they just mutate or whether they mutate then crossover individuals. Repair algorithms can also be implemented as combination operators. First the normal reproduction operator is applied and then the resultant individual is corrected of any defects, converted back into a legal representation. This technique was chosen and implemented using a tree structure. A single combined reproduction operator is passed into the Model which would execute the appropriate leaf operators. Shown in Figure 8 are special *And*, *Or*, and *Xor* operators which form internal nodes. This reproduction is one but not both of (both Crossover and Mutate) or (either Crossover or Mutate or both). Both and Xor class hold weights to determine the probability of each branch of the tree being chosen.

Some implementation issues have to be resolved. The problem of getting information from the selection operator to one or more reproduction leaves. One solution was to give each operator its own selector during initialisation, some attached to the population through the fitness class and others attached to sibling reproduction operators. Figure 9 shows selectors as arrows pointing towards their source. The population class is represented by small grey circles. Such a design would treat all creation events evenly whether they draw from the old population or another operator. Each stage would have to create complete individuals ready to pass on to another operator and this would slow the algorithm. Also the ancestry of an individual might get a little chaotic with some parents never existing in any population.

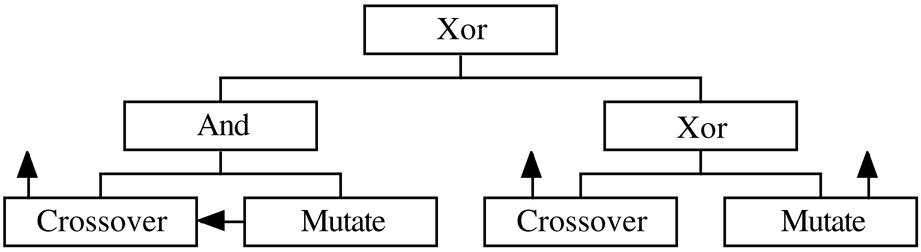


Figure 9 - Multiple selectors

The final solution was for the reproduction tree to be informed of its selector and this information propagated down the tree to all reproduction operators who required it. During a creation event control was provided through the combination operators. The And operator will have to pull individuals from the selector and pass them first to its left hand operator and receiving solution representations but not individuals, these bare representations are then passed to the right hand operator, changed and returned before begin packaged as individuals. The Xor operator can determine the relevant sub-tree and pass control down. The Or operator must decide to use either one or both sub-trees and can then act like an Xor or And operator respectively.

Several methods are required to implement this technique.

```
create: Creator, Population, size
create: Creator, History, Population, size
parents: Creator > number
breed: Creator, [Representation], Representation >
  succeed
```

When the Model object require more individuals the `create` method is called. Two variants were provided, one to keep track of ancestry, the other not. Combination operators use the `parents` method to plan their strategy. Finally the `breed` method produces new representations which could then be wrapped up as individuals after more modification.

A question never adequately answered during the design of the toolkit is how many individuals should a crossover produce. Classic genetic algorithms set the answer at two, or for some of the more complex crossover, a larger but fixed number. Creating multiple individuals did not mesh with the toolkit's generality. When asking for a single individual a crossover might be selected from the tree and that would produce two or more individuals instead. The toolkit could not know how many to expect and would not know what to do with any surplus. The chosen solution was to only ever create a single individual for any reproduction operator. More operations might have to be performed but there can be no confusion about the number of individuals produced and no time wasted on creating surplus individuals.

4 Travelling Salesman Problem

4.1 Salesman's Job

For the first demonstration of the toolkit's power the purpose was to explore the workings of the toolkit rather than develop a new genetic algorithm. Several problems are well known in the genetic algorithm field, one of these is the Travelling Salesman Problem which is also a common test for other search space techniques. Because of this large sets of data readily available for test purposes with optimal solution already found by exhaustive search. As a bonus one of standard implementations is both simple and will exercise different toolkit components than The Blues.

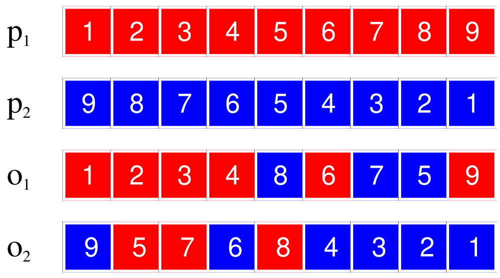
The travelling salesman problem involves planning a route from city to city for a salesman to travel so that each city is visited once and only once apart from one city which should be both start and finish point. Additionally the total cost of travelling on the journey should be minimised. Another way to describing this problem is as finding the minimum cost tour among all permutations of the n cities in the salesman's itinerary. Although expressed in terms of a salesman travel plans this problem is equivalent to many other distance minimisation problems.

> "Circuit board drilling applications with up to 17,000 cities are mentioned... , X-ray crystallography instances with up to 14,000 cities are mentioned... , and instances arising in VLSI fabrication have been reported with as many as 1.2 million cities.... Moreover, 5 hours on a multi-million dollar computer for an optimal solution may not be cost-effective if one can get within a few percent in seconds on a PC. Thus there remains a need for heuristics"
>
> Johnson (1990)

These search spaces are often extremely large and any savings that genetic algorithms may give are very valuable.

4.2 Standard Decomposition

Although several genetic representations have been used for the Travelling Salesman Problem the path representation is both common and simple. In this the tour is stored as a list of n cities in chronological order. Evaluation is a matter of totalling the distances for each leg of the journey remembering to include the final trip back to the initial or home city.



p_1	1	2	3	4	5	6	7	8	9
p_2	9	8	7	6	5	4	3	2	1
o_1	1	2	3	4	8	6	7	5	9
o_2	9	5	7	6	8	4	3	2	1

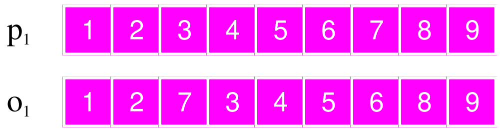
Figure 10 - Ordered crossover operator

With this ordered list representation comes the ordered crossover operator which is shown in Figure 10. Given two parents, p_1 and p_2, a subset of cities is chosen from the list. To create the first offspring p_1 is copied and the selected subset of cites is reordered to match that found in p_2. The second offspring is similar, a copy of p_2 with p_1's ordering imposed. A corresponding mutation is the order mutation operator which randomly selects two cities and inserts one before the other as shown in Figure 11.

All other components in this algorithm's implementation are identical to those used by The Blues. This includes roulette wheel selection for both parents and a generational breeding model. Any fitness function is appropriate but in this case the unity class was reused.

4.3 Toolkit Performance

It is difficult to rate the first performance of the toolkit because many of the standard components had to be implemented in parallel with the algorithm. This does shows the extensible nature of the toolkit. There is no difference between the original construction of the toolkit and the construction of additional user components. Both just require simple objects to be inherited from the abstract classes provided. For this implementation of the Travelling Salesman Problem only a new evaluation class and main program body were required, ordered lists are part of the toolkit design.



p_1	1	2	3	4	5	6	7	8	9
o_1	1	2	7	3	4	5	6	8	9

Figure 11 - Order mutation operator

Initial results were very promising. When the best individual was monitored during the algorithm it showed steadily lower costs with each generation. Unfortunately this value quickly dipped below the optimal cost solution defined for the test data. Investigation showed an error during the initialisation of individuals which allowed a salesman to stay in the same city for much of the time. After correcting this error results were still positive. Costs dropped slowly over the course of the genetic algorithm. Although the best routes were far longer than optimal route known they were shorter than the random routes which made the base population. A summary of the results can be seen in Figure 12, several of the optimum route segments have been found by the genetic algorithm. Other genetic algorithms implementations have performed much better than but here it was sufficient here to prove MUTANTS was capable of this implementation. Continued experimentation would have required more standard toolkit components to be built. Although any single component takes relatively little time to implement, making large numbers of them would have taken up

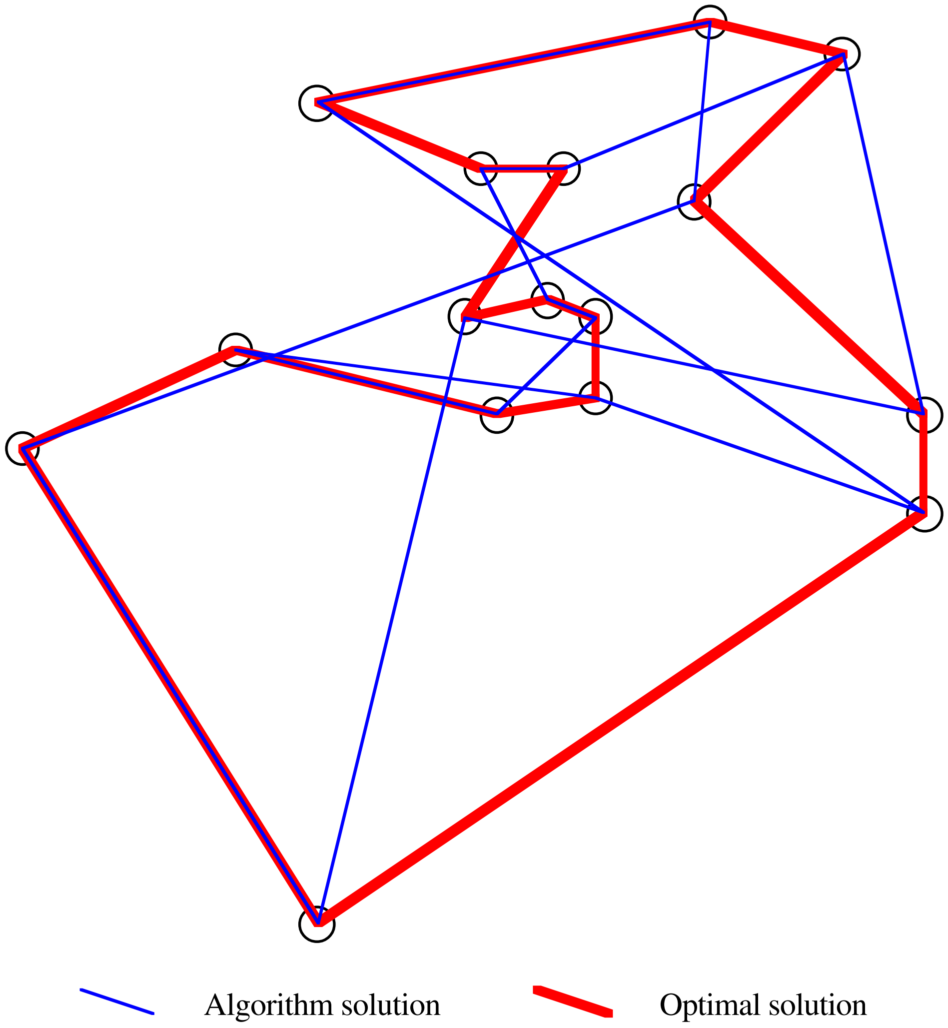


Figure 12 - Ulysses16 Travelling Salesman Problem

precious time.

5 Violin Music Notation

5.1 Violins and Bows

To play a single musical note on a violin a particular length of bow is required. When playing music the bowing direction must change frequently to prevent running off the ends of the bow. Sheet music can be marked to indicate where to change direction to achieve the best from the music. Particular passages should played on the upstroke of the bow, others on the downstroke. Some sections require a constant bow direction whereas other are better if the bow changes direction. Finding where these bowing changes should be to produce the initial annotated music is a lengthy business based on educated guess work.

Here is a complex search space currently laboriously explored by hand. Automation would at least speed up the markup process with final polishing done by the composer, perhaps it could even produce playable results within minutes instead of hours.

5.2 Problem Analysis

As this is an entirely new problem the first step is to make an analysis from the viewpoint of a genetic algorithm. A representation must be found to hold a potential solution to the problem. Music can be thought of as a sequence of notes each associated with the bow length required to play that note. An appropriate representation here seems to be a sequence of numbers equivalent to the bow lengths. In addition at each note the bow may either change direction (c) or not change direction (n).

```
1 2 3 4 5 6 7 8 9 8 7 6 5 4 3 2 1
n n n n n c n c n c n c n n c n n
```

Rules dictated by the music can also be encoded alongside. Bow direction maybe required to be upward (U), downward (D), changing (C), or not changing (N).

```
1 2 3 4 5 6 7 8 9 8 7 6 5 4 3 2 1
n n D D D c n C n c n U U n c n n
```

Each individual carries different lower-case values but identical upper-case values and lengths. Obvious mutation operators would swap letters around but this could create illegal solutions, ones which did not follow the rules. Illegal solutions could be repaired or given penalty points during evaluation but neither is ideal. Separating fixed and variable parts of the solution avoids both problems. Then only the variable part need undergo mutation and crossover.

```
1 2 3 4 5 6 7 8 9 8 7 6 5 4 3 2 1
- - D D D - - C - - - U U - - - -
n n       c n   n c n     n c n n
```

The fixed upper section is common to all individuals and so can be factored out of the individuals and placed in the evaluation function. The variable lower section consists of only two possibilities and is therefore suited a binary vector representation.

Evaluating the performance of an individual is a matter of simulating the position of the bow on the strings. Better or lower scores awarded for the fewer changes of direction. Unfortunately it now becomes apparent that illegal solutions are still possible. For example, an individual may exist in which no direction changes are specified and the bow could then overrun. Correcting this requires a more complex evaluation function which not only changes direction as direct by the individual but also when it must in order to continue playing.

This initial analysis of the problem took under an hour but never made it to the implementation stage. Subsequent thought highlighted certain detrimental effects of standard binary mutation. A single change to the first note in an individual may dramatically change the meaning the remaining notes as the evaluator is forced to make different decisions. This simulation of the music in the evaluation function is another problem. To prevent the bow from overrunning much time must be spent backing up and trying alternate direction choices. Many evaluations must be made over the duration of a genetic algorithm which would be slow as a result.

The alternative which was chosen for the final implementation modified the evaluation function to remove the need for backups. Now an infinitely long bow is simulated and the amount of that bow required by an individual is measured. Again scores could be assigned according to the number of direction changes but also on how much bow length required. Better scores are given to individuals who come close to the actual bow length.

5.3 Algorithm Success

This design is remarkably similar to that used for The Blues, only a evaluation class and main program body are different. The evaluation object is used to store the fixed sections of music and during evaluation gaps in the music are replaced with the information stored within the individual. The main program creates other objects including those controlling unity fitness, roulette wheel selection, single bit mutation, and one point crossover all of which are already found within the toolkit.

In the end such a simple algorithm was not capable of coping with the rigours of a real musical example and instead a small toy data set was used to test the algorithm. Here it is presented as above with note lengths on top and bowing rules on the bottom.

```
6 7 8 5 7 6 8 6 6 8 5 7 7 5 8
- - U U U - - - C - - - - D D
```

A target bow length of 30 was chosen for the test and the random initialisation phase managed to produced a individual which required a bow of length only 33. Because of the simple generational model constant improvement was not guaranteed and by the 43rd generation the best individual was 39 units long. However by the 71st generation other avenues had been explored and the target of 30 had been reached. Overall this seems a reasonable result given the time taken on its development.

MUTANTS proved its worth with this algorithm, almost all of the code was already in place. Design considerations took about an hour and implementation around four more. Although the algorithm has not been attempted without the use of the toolkit I would anticipate taking slightly longer to complete all the work essential to the algorithm without it. The saving in time is significant but not huge for someone familiar with coding genetic algorithms.

More important is that with the toolkit fully implemented dozens of variations could be made in almost no additional time. Simple changes to the main program body would give different fitness function, mutation and crossover operators. Some are bound to show better properties than others. In a single day this problem could be taken from scratch and numerous possible algorithms designed and implemented. Each additional specialised component written will give rise to many additional algorithms.

6 Conclusions

6.1 Success and Failure

The aim in creating MUTANTS was to allow fast prototyping of any genetic algorithm. However, the very first algorithm implemented went beyond the toolkits capabilities. In its original design The Blues used two parent selection methods. Every individual was to be selected at least once for breeding, either by crossover or mutation. The element of chance was introduced with the roulette wheel selector which picked the second parent in the case of a crossover. Although the toolkit allows the selection method to be changed this can only be done between reproductions, not during as required in this algorithm.

Ada 95 allows object-oriented implementation and generic package instantiation, both helpful in creating a flexible, extensible toolkit. During the implementation of the toolkit Ada's strong typing system hindered some previous design choices. To overcome this problem typecasting and unchecked access types (pointers) were used. Although functional the toolkit depends on sensible use by the programmer because Ada can no longer catch as many mistakes. Also full advantage was not made of default parameters available in the language. If used these would have lessened the burden of creating simple genetic algorithm by offering predetermined choice for some components to novices.

Although limited in extent MUTANTS does provide a fast-prototyping toolkit for some genetic algorithms. Classic algorithms using binary vector representation with random initialisation, mutation, and single point crossover controlled by the generational model are easy to implement. Both The Blues and Violin Music Notation fall into this category. The Travelling Salesman Problem required different components - ordered list representation and appropriate reproduction operators which are also provided. Easier and faster implementation leaves more time for experimentation and the chance of getting better results. Writing new toolkit classes is simple, each class does only a limited amount of work in the overall algorithm and once written classes can be reused in other related projects.

6.2 Further Work

Only a few from the dozens of components suggested during initial research have been implemented in the final toolkit. If all these missing components were completed the scope of the toolkit to help in the creation of good genetic algorithms would be greatly enhanced. Other languages might also be investigated to extend the toolkits application areas even further.

The interaction of creator and selector class currently lacks the flexibility needed for all algorithms. Considering this problem is a good opportunity to look at the design of the entire toolkit. Another project should consider not just a single toolkit organisation but try to find the best organisation. Many algorithm techniques were consider for the current design but no actual algorithms implementations were examined. To create the best toolkit many algorithms would have to be collected and dissected. Their requirements would then form the basis of the next toolkit design.

6.3 Summary

Genetic algorithms are based on systems seen to work in nature, systems that have produced the diverse life around us. In borrowing these ideas for searching solution spaces many separate techniques have appeared. Each combination of techniques produces a different effect and it is difficult and slow to optimise the algorithms to a particular problem.

MUTANTS the generic genetic algorithm toolkit specifies a range of connectable components. An algorithm constructed using these components will be smaller and its implementation will take less time. With this extra time other possibilities presented by the toolkit can be explored and every additional option explored gives a better chance of finding the perfect algorithm.

Bibliography

Genetic Algorithms and Robotics : A Heuristic Strategy for Optimization
Davidor, Yuval
World Scientific 1991

Genetic Algoritms in Seach, Optimization, and Machine Learning
Goldbery, David E
Addison-Wesley 1989

Genetic Algorithms and Simulated Annealing
Lawrence Davis ed
Pitman 1987

Handbook of Genetic Algorithms
Lawrence Davis ed
Van Nostrand Reinhold 1991

Local Optimization and the Traveling Salesman Problem
Johnson, D.S
Proceedings on the 17th Colloquium on Automata, Languages, and Programming
Lecture Notes in Computing Science
Vol 443, pp.446-461
Springer-Verlag 1990

Genetic Algorithms + Data Structures = Evoluation Programs
Michalewicz, Zbigniew
Springer 1992

Evaluating Genetic Algorithms For A Room Allocation Problem
Scott David Williams
Senior Honours Project
University of Glasgow 1994

Appendix A

Problem Definition

Many problems in the world that we wish to solve are intractable, that is, it will take more than polynomial time to find the optimal solution. In most situations a sub-optimal solution can be used instead and a range of techniques have been developed to provide a near optimal solution in minimal time.

Genetic algorithms are one such technique. They begin by generating a population of potential solutions to the problem, encoded on genes. Over a number of generations new solutions are bred by crossing and mutating. Solutions compete for space within the population and those who cannot compete, die out. The process of selection, similar to natural selection, will find a near optimal solution after a number of generations.

There are five characteristic components in every genetic algorithm:

- a genetic representation for solutions to the problem
- a way to create an initial population of potential solutions
- an evaluation function that plays the role of the environment, rating solutions in terms of their fitness
- genetic operators that alter the composition of children during reproduction
- values for various parameters that the genetic algorithm uses (population size, probability of applying genetic operators)

Each component can be represented in a number of different ways and used in many algorithms. MUTANTS will provide a library of these genetic algorithm components. Reuse of components speeds the creation of an algorithm giving the chance for experimentation to produce faster or better algorithms.

Appendix B

Statement of Requirements

I. Customer Goals
  A. The aim is to provide a library of components to simplify the construction of genetic algorithms.
  B. Components should be written in Ada 95.
  C. As a minimum the toolkit should be able to implement the classic genetic algorithm with:
    1. Bit string genetic representation,
    2. Random binary initialisation,
    3. Binary mutation and single point crossover genetic operator,
    4. Generation replacement model.
  D. In addition other techniques may be included, such as:
    1. Vector, ordered list, number genetic representation,
    2. Distributed and random search initialisation,
    3. Other mutation, multi-point crossover, and inversion genetic operators,
    4. Steady-state, elitist, and without-duplicate models.
  E. Other facilities which the toolkit should support:
    1. User provided fitness values for potential solutions,
    2. Family trees to relate all previous solutions,
    3. Decision support which helps the user explore the solution space,
    4. Saving of genetic algorithm state to be recreated later.
  F. The toolkit should be extensible so that components specific to a particular genetic algorithm can be added by the programmer.
  G. Components should be contained so the use of one does not require the use of another.
  H. Advanced techniques such as meta-genetic algorithms, adaptive genetics, and interpolative genetics should be implementible using the toolkit but need not be implemented.
  I. The toolkit should be proved by creating two dissimilar genetic algorithms using its components.

II. Technical Goals
  A. Functional Requirements
    1. Toolkit components should be divided into three classes
      a) core toolkit - components required for classic genetic

        algorithm with: binary string representation, random binary initialisation, simple mutation and single point crossover, generation replacement model
        b) auxiliary toolkit - components to extend functionality of genetic algorithms such as decision support
        c) extended toolkit - components for other genetic algorithm techniques
    2. The core toolkit has highest priority, then the auxiliary toolkit and finally the extended toolkit.
    3. Component organisation must:
        a) allow for the large number of possible genetic algorithms,
        b) be extensible with user defined components.
        c) not restrict the user within components
        d) be savable
    4. Two dissimilar genetic algorithms should be created to show the power and functionality of the toolkit.
    5. A manual to explain the use of the toolkit should be made.

B. Non-functional Requirements
    1. All components should be written in Ada 95.
    2. Typical population sizes in genetic algorithms do not exceed a few hundred but storing individuals permanently will increase this by several orders of magnitude.
    3. Deadlines:
        a) The core toolkit and the first demonstration genetic algorithm should be completed by the end of week 10,
        b) The second demonstration genetic algorithm and the rest of the project should be delivered by the end of week 20.

Appendix C

External Specification

The toolkit will be composed of a collection of components organised in a number of component hierarchies which describe their relationships. The taxonomic hierarchy groups components with similar tasks together, for example apples are under fruit. The functional hierarchy shows a components use of other components, for example apples are under teeth. The containment hierarchy shows a components storage of other components, for example pips are under apples. All components are also grouped as part of the core, auxiliary, or extended toolkit. Components are presented here according to the taxonomic hierarchy with other hierarchies described in the text.

To create a genetic algorithm with this toolkit the programmer has to combine a number of predefined components with a few that have been hand coded. A simple example of constructing a genetic algorithm using the toolkit follows with component names emphasised. A description of each component can be found later in the specification.

Classic Genetic Algorithm

A genetic algorithm is required to optimise a simple function.

$$f(x) = x.\sin(10\pi.x) + 1.0$$

The problem is to find x from the range [-1, 2] which maximises the function f, ie to find x_0, such that

$$f(x_0) \geq f(x) \text{ for all } x \in [-1, 2]$$

Two important decisions have to made about the genetic algorithm: how to represent potential solutions, and which model will drive the genetic algorithm. The use of classic genetic algorithm techniques dictates our choices here. The *Generational Model* will define the order of actions within our algorithm and solutions will be represented using a *Bit Vector*. To store a solution to an accuracy of 6 decimal places will require 22 bits for each vector.

To decide which solutions are better an *Evaluation* function must take the bit vector representation and return a number

$$E = -1.0 + v.3/(2^{22} - 1)$$

where v is the bit vector interpreted as a positional code integer, this can be done using *Function Binary Bit Vector Evaluation.*

The initial *Population* of solutions will be created using *Bit Vector Random Initialisation* and subsequent solutions will be bred using two *Reproduction* operators, *Bit Random Single Vector Mutation* and *One Point Vector Crossover*. Operator distribution is 22% and 25% respectively with the remainder made up using *Clone.*

The Toolkits

The *core toolkit* consists of those components required to implement the classic genetic algorithm “The Blues” as shown in the report.

> Generational Model, Count Ending, Bit Representation, Vector Representation, Individual, Population, Bit Random Initialise, Vector Random Initialise, Evaluation, Unity Fitness, Roulette Wheel Selector, Clone Reproduction, Xor Reproduction, Position-based Ordered Mutation Reproduction, Order-based Ordered Crossover Reproduction, Random

The *auxiliary toolkit* consists of those components which extend functionality for genetic algorithms such as decision support.

> History

The *extended toolkit* consists of all components suitable for other genetic algorithms.

Component Overview

The *Model*, or breeding technique, component controls the general activity of the genetic algorithm. A genetic algorithm consists of a single Model and subsidiary components controlled by it.

The *Ending* component controls the length of time spent on the genetic algorithm.

The *Representation* component encodes a possible solutions. Choosing which Representation is important to the design of the entire genetic algorithm.

The *Individual* component is a particular instance of a possible solution. Each Individual is associated with a Representation and a unique identification which separates it from all other Individuals, even those with identical Representations.

The *Population* component is a bag of Individuals or a bag of Populations. Operations include all standard bag operations.

The *Evaluation* component converts a Representation to a numerical representation of how good a solution it encodes.

The *Fitness* component converts and Individuals Evaluation into its fitness compared to the rest of the Population.

The *Selector* component chooses an Individual from a Population, perhaps using information about that Individual and the rest of the Population.

The *Reproduction* component creates N new Individuals for the destination Population from the source Population.

The *History* component can store an association of Individuals, their parents and Reproduction components or Initialisers, and the Random settings.

The *Random* component generates pseudo random numbers starting from either a given seed or the seed is taken from the time.

Model Component

The Model, or breeding technique, component controls the general activity of the genetic algorithm. A genetic algorithm consists of a single Model and subsidiary components controlled by it. Although the toolkit provides a range of Model components it is impossible to provide the variety which may be required. Instead unusual Models can be hand coded making use of subsidiary components from the toolkit.

Functional: Ending, Individual, Population, Initialise, Evaluation, Fitness, Selector, Reproduction, History, Random

Containment: Ending, Population, Initialise, Evaluation, Fitness, Selector, Reproduction, History, Random

Generational

The first Population is Initialised then a new Population of the same size is created using the Reproduction operator. The current Population is then replaced and the new Population used for further breeding.

Generational Elitism

Generational but the first Individual of the new Population is a clone of the best Individual of the previous Population.

Tournament

Generational but new Individuals are together with current Individuals. N Individuals are selected and the best is placed in the next generation. This is repeated until the next generation is full.

Steady State

The first Population is Initialised then N new Individuals are created using the Reproduction operator. These replace N individuals selected from the old population.

Steady State Without Duplicates

Steady State but new Individuals must not have identical Representations to current Individuals.

Preselection

Steady State but Individuals only replace inferior parents.

Ending Component

The Ending component controls the length of time spent on the genetic algorithm.

Functional:

Containment: Ending

And

The Model continues until both given ending conditions are true.

Or

The Model continues until either of two ending conditions are true.

Count

The Model is allowed N cycles before halting.

Time

The Model is allowed N seconds before halting.

Evaluation

The Model continues until a given Evaluation is reached.

Convergence

The Model continues until the population converges and no further change in Evaluation is seen.

Representation Component

The Representation component encodes a possible solutions. Choosing which Representation is important to the design of the entire genetic algorithm.

Functional:

Containment: Representation

Bit

A single bit, 0 or 1.

Number

A single number with value between upper and lower bounds.

Vector

A sequence of elements with identical Representations.

Matrix

A matrix of elements with identical Representations.

Ordered

A particular permutation of elements draw from a particular Representation.

Tag

A integer position paired with another Representation, can be used to make a tagged vector for use with the Inversion operator.

Individual Component

The Individual component is a particular instance of a possible solution. Each Individual is associated with a Representation and a unique identification which separates it from all other Individuals, even those with identical Representations.

Functional: Representation

Containment: Representation

Population Component

The Population component is a bag of Individuals or a bag of Populations. Operations include all standard bag operations.

Functional: Representation, Individual, Evaluation

Containment: Individual, Population

Initialise Component

The Initialise component is used to create a number of new Individuals for a given Population. The Initialiser must be tailored to particular Representations.

Functional: Representation, Population, Evaluation, Selector, Reproduction, Random

Containment: Initialise, Reproduction

Random
- Bit
  - Assigned 0 or 1 with equal probability.
- Number
  - Assigned a number between upper and lower boundary with equal probability.
- Vector
  - Initialise each element separately.
- Matrix
  - Initialise each element separately.
- Ordered
  - Assign one of the possible permutation of elements with equal probability.
- Tag Vector
  - Initialise each Tag separately and assign the correct integer position.

Search

Initialise N Individuals at a time and pick the one with the highest Evaluation. Selection between Individuals with equal fitness will be made randomly.

Distributed

Generated Individuals are as different as possible from each other. This technique must be tailored for each Representation.

Heuristic

Individuals with particular characteristics are generated using heuristics particular the problem. This technique must be tailored to each Representation and each problem.

Evaluation Component

The Evaluation component converts a Representation to a numerical representation of how good a solution it encodes. The evaluation is dependent on both the Representation and the problem. In most situations the Evaluation component will be hand coded by the programmer.

Functional: Representation, Individual

Containment: Evaluation

- Vector
  - Bit
    - Binary
      - Evaluation is equal to the positional code interpretation of the bit vector.
    - Gray
      - Evaluation is equal to the gray code interpretation of the bit vector.
- Function
  - Apply a known function to the result of another Evaluation.

Fitness Component

The Fitness component converts and Individuals Evaluation into its fitness compared to the rest of the Population.

Functional: Individual, Population

Containment:

Unity

Fitness is equal to evaluation.

Windowing

Fitness is equal to the amount an Individual's Evaluation exceeds the minimum evaluation within the Population minus some known guard value.

Linear Normalisation

Order Individuals by decreasing Evaluation. The best Individual is given a known fitness and thereafter the fitness is decreased by a constant amount.

Linear Scaling

Fitness is equal to (a * evaluation + b) where a and b are normally selected so that the average fitness is mapped to itself and the best fitness is increased by some desired multiple.

Sigma Truncation

Fitness is equal to (evaluation + average evaluation - c * s) where c is chosen from around 1 to 5 and s is the Population's standard deviation. Negative fitness values are set to zero.

Power Law Scaling

Fitness is equal to (evaluation^{k}) where k is a problem dependent value close to 1.

Selector Component

The Selector component chooses an Individual from a Population, perhaps using information about that Individual and the rest of the Population.

Functional: Individual, Population, Fitness, Random

Containment:

Best

Individual with the highest fitness within the Population is selected. Selection between Individuals with equal fitness will be made randomly.

Worst

Individual with the lowest fitness within the Population is selected. Selection between Individuals with equal fitness will be made randomly.

Random

Each Individual has an equal probability of being selected.

Roulette Wheel

Each Individual has a probability of being selected proportional to its fitness divided by the average fitness of the Population.

Anti Roulette Wheel

Each Individual has a probability of being selected inversely proportional to its fitness divided by the average fitness of the Population.

Stochastic

The first Individual is chosen using a roulette wheel and the subsequent (N - 1) Individuals taken evenly from the entire population, then the cycle repeats.

Expected Value

Associated with each Individual is a count equal to that Individual's fitness divided by the average fitness of the Population. Whenever an Individual is selected to reproduce, using another Selector component, its count is decremented by one. When a count falls below zero the Individual is no long available for further selection.

Similarity

The Individual selected is that closest to the given Individual. Selection between equally similar Individuals will be made randomly. This technique must be tailored to each Representation and each problem.

Reproduction Component

The Reproduction component creates N new Individuals for the destination Population from the source Population. Reproduction components must be tailored to a each Representation and each Problem.

Functional: Representation, Individual, Population, Selector, Random

Containment: Selector, Reproduction

Clone

Individual's Representation is an exact copy of the parent's Representation.

Xor

Individuals are created with either one Reproduction component or another but not both using the given probability distribution.

And

Individuals are created with one Reproduction followed by another. The second Reproduction component may require several Individuals to be

  - generated by the first component.
- Or
  - Individuals are created with either of two Reproduction components or both using the given probabilities. The second Reproduction component may require several Individuals to be generated by the first component.
- Inversion
  - Tag Vector
    - Biased
      - Individual’s Representation is the same as the parent’s Representation except that the order of a sequence of elements between two random points has been reversed.
    - Unbiased
      - Individual’s Representation is the same as the parent’s Representation except that the order of a sequence of element between a random point and a randomly selected end has been reversed.
- Mutation
  - Bit
    - Inverts value.
  - Number
    - Random
      - Assign a number between upper and lower boundary with equal probability.
    - Creep
      - Modify the number by either adding or subtracting a know value.
  - Vector
    - Single
      - Each element has an equal probability of being chosen and mutated.
    - Deletion
      - A randomly chosen element is deleted. Only suitable for variable length vectors.
    - Addition
      - Duplication
        - A randomly chosen element is duplicated with the duplicate placed next to the original. Only suitable for variable length vectors.
      - Initialised
        - A newly initialised element is inserted into the list. Only

suitable for variable length vectors.

- Related
  - A pair of adjacent elements is randomly chosen and a new element related to each is inserted between them. This must be tailored to each Representation and each problem. Only suitable for variable length vectors.

- Tag Vector
  - Cut
    - Parent vector is cut at a random point. Only suitable for messy genetic algorithms.
  - Single
    - Each element has an equal probability of being chosen and the non-position value mutated.
- Ordered
  - Order-based
    - Two randomly selected elements are swapped.
  - Position-based
    - One randomly selected element is placed before another.
  - Scramble Sublist
    - Select a sublist of size N and randomly permute it.
- Random Hill Climb
  - N new Representations are created using a Mutation component and the one with the best Evaluation is chosen. Selection between Individuals with equal evaluation will be made randomly.

- Crossover
  - Number
    - Average
      - Assign the average of two parent numbers.
  - Vector
    - One Point
      - A crosspoint is selected at random and the tail of each parent is swapped with the other parent.
    - Two Point
      - Two crosspoints are selected at random and the elements between are swapped with the other parent. Initial and final elements of the vector are considered adjacent.
    - Multi Point
      - N pairs of crosspoints are selected at random and the elements between are swapped with the other parent. Initial and final elements of the vector are considered adjacent.

Segmented

Multi Point Crossover in which the number of segments can vary. For each element there is a probability that the current segment will end and a new one begin.

Uniform

Each element is taken from either parent with equal probability.

Shuffle

Randomly permute both parent vectors, perform a Crossover, and reverse permute them.

Traverse

Apply given Crossover to each pair of elements in parent vectors.

Tag Vector

Splice

Parent's representations are joined end to end. Only suitable for messy genetic algorithms.

Ordered

Position-based

A set of positions is randomly selected and the positions of elements selected in one parent is imposed on the corresponding elements in the other parent.

Order-based

A set of positions is randomly selected and the order of elements in the selected positions in one parent is imposed on the corresponding elements in the other parent.

Repair

The parents representation is corrected so it no longer violates constraints made on the solution.

History Component

The History component can store an association of Individuals, their parents and Reproduction components or Initialisers, and the Random settings.

Functional: Individual, Reproduction, Initialise, Random

Containment: Individual

Random Component

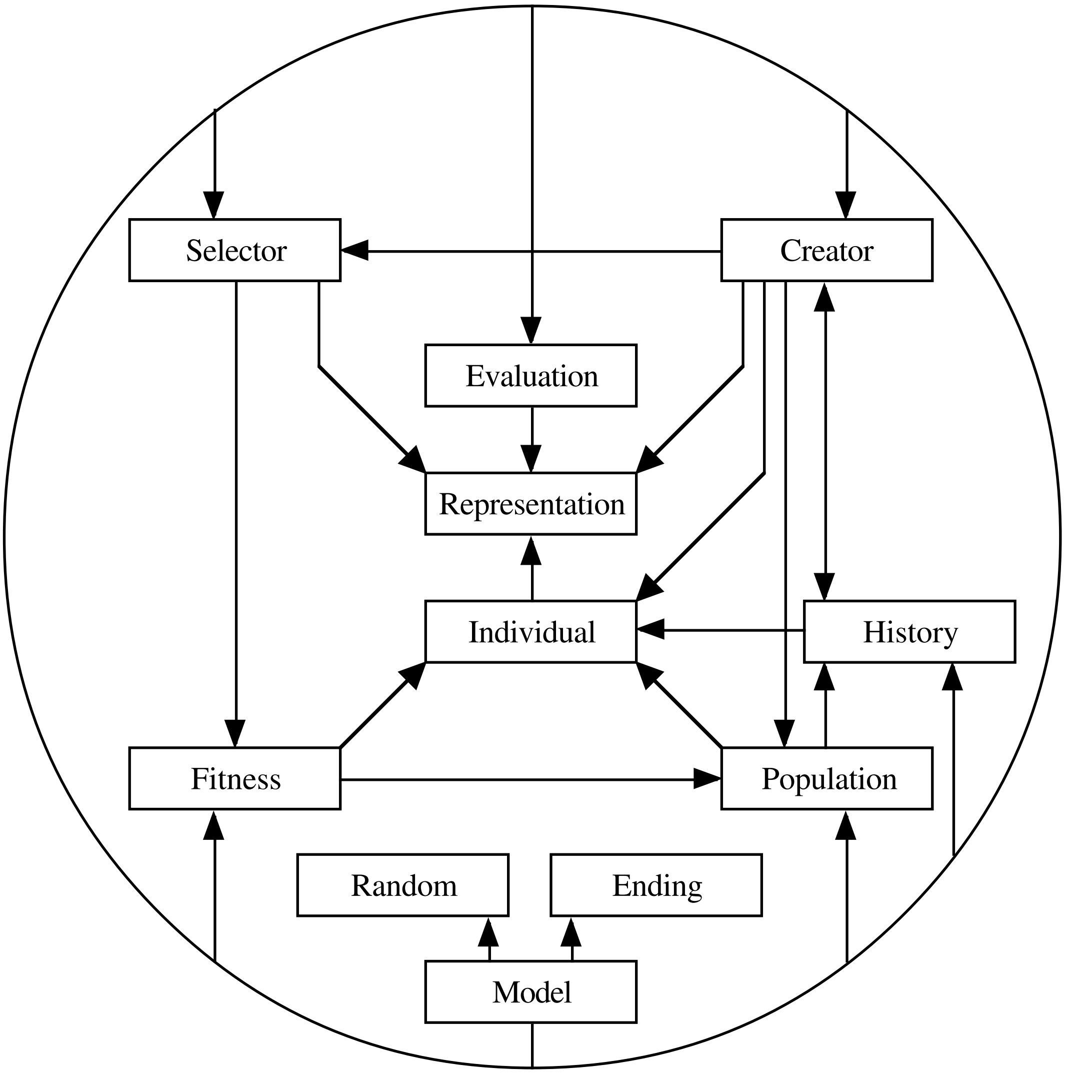
The Random component generates pseudo random numbers starting from either a given seed or the seed is taken from the time.

Functional:

Containment:

Appendix D

Class Design



Class design information is formatted as -

method: arguments > result:

Object creators are refered to as *ctor* methods. A backslash before an arguement indicates the value is passed by value rather than by reference. Arguements enclosed in squared brackets symbolise an array of elements rather than a single value. An underlined method indicates that the child classes may override this method and an underlined reference arguement means the value of the object may change.

Model Class

class Model:
ctor: Creator, \size > Model
ctor: Creator, \size, Random > Model
ctor: Istream > Model
print: Model, Ostream
run: Model, Evaluation, Fitness, Selector, Creator, Ending
run: Model, Evaluation, Fitness, Selector, Creator, Ending, History
population: Model > Population

class Generational_Model:
print: Model, Ostream
run: Generational_Model, Evaluation, Fitness, Selector, Creator, Ending
run: Generational_Model, Evaluation, Fitness, Selector, Creator, Ending, History
population: Generational_Model > Population

class GenerationalElitism_Model:
print: Model, Ostream
run: GenerationalElitism_Model, Evaluation, Fitness, Selector, Creator, Ending
run: GenerationalElitism_Model, Evaluation, Fitness, Selector, Creator, Ending, History
population: GenerationalElitism_Model > Population

class GenerationalTournament_Model:
print: Model, Ostream
run: GenerationalTournament_Model, Evaluation, Fitness, Selector, Creator, Ending, \tournament_size
run: GenerationalTournament_Model, Evaluation, Fitness, Selector, Creator, Ending, History, \tournament_size
population: GenerationalTournament_Model > Population

class SteadyState_Model:
print: Model, Ostream
run: SteadyState_Model, Evaluation, Fitness, Selector, Creator, Ending, \brood_size
run: SteadyState_Model, Evaluation, Fitness, Selector, Creator, Ending, History, \brood_size
population: SteadyState_Model > Population

class SteadyStateNoDuplicates_Model:

print: Model, Ostream
run: SteadyStateNoDuplicates_Model, Evaluation, Fitness, Selector, Creator, Ending, \brood_size
run: SteadyStateNoDuplicates_Model, Evaluation, Fitness, Selector, Creator, Ending, History, \brood_size
population: SteadyStateNoDuplicates_Model > Population

class SteadyStatePreselection_Model:
print: Model, Ostream
run: SteadyStatePreselection_Model, Evaluation, Fitness, Selector, Creator, Ending, \brood_size
run: SteadyStatePreselection_Model, Evaluation, Fitness, Selector, Creator, Ending, History, \brood_size
population: SteadyStatePreselection_Model > Population

Ending Class

class Ending:
ctor: > Ending
print: Ending, Ostream
end: Ending > \end

class And_Ending:
ctor: Ending, Ending > And_Ending
print: And_Ending, Ostream
end: And_Ending > \end

class Or_Ending:
ctor: Ending, Ending > Or_Ending
print: Or_Ending, Ostream
end: Or_Ending > \end

class Count_Ending:
ctor: \count, \decrement > Count_Ending
print: Count_Ending, Ostream
end: Count_Ending > \end

class Time_Ending:
ctor: \duration > Time_Ending
print: Time_Ending, Ostream

end: Time_Ending > \end

class Evaluation_Ending:
ctor: Model, Evaluation > Evaluation_Ending
print: Evaluation_Ending, Ostream
end: Evaluation_Ending > \end

class Convergence_Ending:
ctor: Model, Evaluation, \improvement, \period > Convergence_Ending
print: Convergence_Ending, Ostream
end: Convergence_Ending > \end

Representation Class

class Representation:
ctor: Istream > Representation
print: Representation, Ostream

class Bit_Representation:
ctor: Bit > Bit_Representation
ctor: Istream > Bit_Representation
print: Bit_Representation, Ostream
get: Bit_Representation > Bit
set: Bit_Representation, Bit

class Number_Representation:
ctor: Number > Number_Representation
ctor: Istream > Number_Representation
print: Number_Representation, Ostream
get: Number_Representation > Number
set: Number_Representation, Number

class Vector_Representation:
ctor: [Representation] > Vector_Representation
ctor: Istream > Vector_Representation
print: Vector_Representation, Ostream
get: Vector_Representation, \index > Representation
set: Vector_Representation, \index, Representation
length: Vector_Representation > \length

class Tagged_Vector_Representation:
ctor: [Representation] > Tagged_Vector_Representation
ctor: Istream > Tagged_Vector_Representation
print: Tagged_Vector_Representation, Ostream
get: Tagged_Vector_Representation, \index > Representation
set: Tagged_Vector_Representation, \index, Representation
tagged_get: Tagged_Vector_Representation, \index > Representation
tagged_set: Tagged_Vector_Representation, \index, Representation
length: Tagged_Vector_Representation > \length

class Matrix_Representation:
ctor: [[Representation]] > Matrix_Representation
ctor: Istream > Matrix_Representation
print: Matrix_Representation, Ostream
get: Matrix_Representation, \x, \y > Representation
set: Matrix_Representation, \x, \y, Representation
width: Matrix_Representation > \width
height: Matrix_Representation > \height

class Ordered_Representation:
ctor: [Representation] > Ordered_Representation
ctor: Istream > Ordered_Representation
print: Ordered_Representation, Ostream
get: Ordered_Representation, \index > Representation
set: Ordered_Representation, \index, Representation
length: Ordered_Representation > \length

Individual Class

class Individual:
ctor: Representation > Individual
ctor: Istream > Individual
print: Individual, Ostream
representation: Individual > Representation
evaluation: Individual, \evaluation
evaluation: > \evaluation

Evaluation Class

class Evaluation:
ctor: > Evaluation
ctor: Istream > Evaluation
print: Evaluation, Ostream
evaluate: Evaluation, Representation > \evaluation

class Function_Evaluation:
ctor: \function, Evaluation > Function_Evaluation
ctor: Istream > Function_Evaluation
print: Function_Evaluation, Ostream
evaluate: Function_Evaluation, Representation > \evaluation

class Binary_Bit_Vector_Evaluation:
ctor: > Binary_Bit_Vector_Evaluation
ctor: Istream, Binary_Bit_Vector_Evaluation
print: Binary_Bit_Vector_Evaluation, Ostream
evaluate: Binary_Bit_Vector_Evaluation, Bit_Vector_Representation > \evaluation

class Gray_Bit_Vector_Evaluation:
ctor: > Gray_Bit_Vector_Evaluation
ctor: Istream, Gray_Bit_Vector_Evaluation
print: Gray_Bit_Vector_Evaluation, Ostream
evaluate: Gray_Bit_Vector_Evaluation, Bit_Vector_Representation > \evaluation

Population Class

class Population:
ctor: > Population
ctor: [Individual] > Population
ctor: Istream > Population
print: Population, Ostream
add: Population, Individual
sub: Population, Individual
length: Population > \length
iterator: Population > PopulationIterator

class PopulationIterator:

ctor: Population > PopulationIterator
ctor: Istream > PopulationIterator
print: PopulationIterator, Ostream
next: PopulationIterator > \more, Individual, \fitness

Fitness Class

class Fitness:
ctor: > Fitness
ctor: Istream > Fitness
print: Fitness, Ostream
attach: Fitness, Population
fitness: Fitness, Individual > \fitness
length: Fitness > \size
iterator: Fitness > FitnessIterator

class FitnessIterator:
ctor: Fitness > FitnessIterator
ctor: Istream > FitnessIterator
print: FitnessIterator, Ostream
next: FitnessIterator > \more, Individual, \fitness

class Unity_Fitness:
ctor: > Unity_Fitness
ctor: Istream > Unity_Fitness
print: Unity_Fitness, Ostream
attach: Unity_Fitness, Population
fitness: Unity_Fitness, Individual > \fitness
length: Unity_Fitness > \size
iterator: Unity_Fitness > Unity_FitnessIterator

class Unity_FitnessIterator:
ctor: Unity_Fitness > Unity_FitnessIterator
ctor: Istream > Unity_FitnessIterator
print: Unity_FitnessIterator, Ostream
next: Unity_FitnessIterator > \more, Individual, \fitness

class Window_Fitness:
ctor: > Window_Fitness

```
ctor: Istream > Window_Fitness
print: Window_Fitness, Ostream
attach: Window_Fitness, Population, \guard
fitness: Window_Fitness, Individual > \fitness
length: Window_Fitness > \size
iterator: Window_Fitness > Window_FitnessIterator

class Window_FitnessIterator:
ctor: Window_Fitness > Window_FitnessIterator
ctor: Istream > Window_FitnessIterator
print: Window_FitnessIterator, Ostream
next: Window_FitnessIterator > \more, Individual, \fitness

class LinearNormalisation_Fitness:
ctor: > LinearNormalisation_Fitness
ctor: Istream > LinearNormalisation_Fitness
print: LinearNormalisation_Fitness, Ostream
attach: LinearNormalisation_Fitness, Population, \maximum, \decrement
fitness: LinearNormalisation_Fitness, Individual > \fitness
length: LinearNormalisation_Fitness > \size
iterator: LinearNormalisation_Fitness > LinearNormalisation_FitnessIterator

class LinearNormalisation_FitnessIterator:
ctor: LinearNormalisation_Fitness > LinearNormalisation_FitnessIterator
ctor: Istream > LinearNormalisation_FitnessIterator
print: LinearNormalisation_FitnessIterator, Ostream
next: LinearNormalisation_FitnessIterator > \more, Individual, \fitness

class LinearScaling_Fitness:
ctor: > LinearNormalisation_Fitness
ctor: Istream > LinearScaling_Fitness
print: LinearScaling_Fitness, Ostream
attach: LinearScaling_Fitness, Population, \a, \b
fitness: LinearScaling_Fitness, Individual > \fitness
length: LinearScaling_Fitness > \size
iterator: LinearScaling_Fitness > LinearScaling_FitnessIterator

class LinearScaling_FitnessIterator:
ctor: LinearScaling_Fitness > LinearScaling_FitnessIterator
ctor: Istream > LinearScaling_FitnessIterator
```

print: LinearScaling_FitnessIterator, Ostream
next: LinearScaling_FitnessIterator > \more, Individual, \fitness

class SigmaTruncation_Fitness:
ctor: > SigmaTruncation_Fitness
ctor: Istream > SigmaTruncation_Fitness
print: SigmaTruncation_Fitness, Ostream
attach: SigmaTruncation_Fitness, Population, \c
fitness: SigmaTruncation_Fitness, Individual > \fitness
length: SigmaTruncation_Fitness > \size
iterator: SigmaTruncation_Fitness > SigmaTruncation_FitnessIterator

class SigmaTruncation_FitnessIterator:
ctor: SigmaTruncation_Fitness > SigmaTruncation_FitnessIterator
ctor: Istream > SigmaTruncation_FitnessIterator
print: SigmaTruncation_FitnessIterator, Ostream
next: SigmaTruncation_FitnessIterator > \more, Individual, \fitness

class PowerLawScaling_Fitness:
ctor: > PowerLawScaling_Fitness
ctor: Istream > PowerLawScaling_Fitness
print: PowerLawScaling_Fitness, Ostream
attach: PowerLawScaling_Fitness, Population, \k
fitness: PowerLawScaling_Fitness, Individual > \fitness
length: PowerLawScaling_Fitness > \size
iterator: PowerLawScaling_Fitness > PowerLawScaling_FitnessIterator

class PowerLawScaling_FitnessIterator:
ctor: PowerLawScaling_Fitness > PowerLawScaling_FitnessIterator
ctor: Istream > PowerLawScaling_FitnessIterator
print: PowerLawScaling_FitnessIterator, Ostream
next: PowerLawScaling_FitnessIterator > \more, Individual, \fitness

Selector Class

class Selector:
ctor: > Selector
ctor: Istream > Selector
print: Selector, Ostream

attach: Selector, Fitness
select: Selector > Individual
unselect: Selector, Individual

class Best_Selector:
ctor: > Best_Selector
ctor: Istream > Best_Selector
print: Best_Selector, Ostream
attach: Best_Selector, Fitness
select: Best_Selector > Individual
unselect: Best_Selector, Individual

class Worst_Selector:
ctor: > Worst_Selector
ctor: Istream > Best_Selector
print: Worst_Selector, Ostream
attach: Worst_Selector, Fitness
select: Worst_Selector > Individual
unselect: Worst_Selector, Individual

class Random_Selector:
ctor: Random > Random_Selector
ctor: Istream > Random_Selector
print: Random_Selector, Ostream
attach: Random_Selector, Fitness
select: Random_Selector > Individual
unselect: Random_Selector, Individual

class RouletteWheel_Selector:
ctor: Random > RouletteWheel_Selector
ctor: Istream > RouletteWheel_Selector
print: RouletteWheel_Selector, Ostream
attach: RouletteWheel_Selector, Fitness
select: RouletteWheel_Selector > Individual
unselect: RouletteWheel_Selector, Individual

class AntiRouletteWheel_Selector:
ctor: Random > AntiRouletteWheel_Selector
ctor: Istream > AntiRouletteWheel_Selector
print: RouletteWheel_Selector, Ostream

attach: AntiRouletteWheel_Selector, Fitness
select: AntiRouletteWheel_Selector > Individual
unselect: AntiRouletteWheel_Selector, Individual

class Stochastic_Selector:
ctor: Random > Stochastic_Selector
ctor: Istream > Stochastic_Selector
print: Stochastic_Selector, Ostream
attach: Stochastic_Selector, Fitness, \cycle
select: Stochastic_Selector > Individual
unselect: Stochastic_Selector, Individual

class ExpectedValue_Selector:
ctor: Random > ExpectedValue_Selector
ctor: Istream > ExpectedValue_Selector
print: ExpectedValue_Selector, Ostream
attach: ExpectedValue_Selector, Fitness, \cycle
select: ExpectedValue_Selector > Individual
unselect: ExpectedValue_Selector, Individual

class Similarity_Selector:
ctor: Random > Similarity_Selector
ctor: Istream > Similarity_Selector
print: Similarity_Selector, Ostream
attach: Similarity_Selector, Fitness, Individual
select: Similarity_Selector > Individual
unselect: Similarity_Selector, Individual

Creator Class

class Creator:
ctor: > Creator
create: Creator, Population, \size
create: Creator, History, Population, \size
selector: Creator, Selector > \required
private Creator:
parents: Creator > \number
breed: Creator, [Representation], Population > \succeed

```
class And_Creator:
ctor: Creator, Creator > And_Creator
create: And_Creator, Population, \size
create: And_Creator, History, Population, \size
selector: And_Creator, Selector > \required
private And_Creator:
parents: And_Creator > \number
breed: And_Creator, [Representation], Population > \succeed

class Or_Creator:
ctor: Creator, \probability, Creator, \probability > Or_Creator
create: Or_Creator, Population, \size
create: Or_Creator, History, Population, \size
selector: Or_Creator, Selector > \required
private Or_Creator:
parents: Or_Creator > \number
breed: Or_Creator, [Representation], Population > \succeed

class Xor_Creator:
ctor: Creator, \weight, Creator, \weight > Xor_Creator
create: Xor_Creator, Population, \size
create: Xor_Creator, History, Population, \size
selector: Xor_Creator, Selector > \required
private Xor_Creator:
parents: Xor_Creator > \number
breed: Xor_Creator, [Representation], Population > \succeed

class Initialise:
ctor: > Initialise
create: Initialise, Population, \size
create: Initialise, History, Population, \size
private Initialise:
parents: Initialise > \number
breed: Initialise, [Representation], Population > \succeed

class Random_Initialise:
ctor: Random > Random_Initialise
create: Random_Initialise, Population, \size
create: Random_Initialise, History, Population, \size
```

class Bit_Random_Initialise:
ctor: Random > Bit_Random_Initialise
create: Bit_Random_Initialise, Population, \size
create: Bit_Random_Initialise, History, Population, \size

class Number_Random_Initialise:
ctor: Random, \lower, \upper > Number_Random_Initialise
create: Number_Random_Initialise Population, \size
create: Number_Random_Initialise, History, Population, \size

class Vector_Random_Initialise:
ctor: Random, Initialise > Vector_Random_Initialise
create: Vector_Random_Initialise, Population, \size
create: Vector_Random_Initialise, History, Population, \size

class Tagged_Vector_Random_Initialise:
ctor: Random, Initialise > Tagged_Vector_Random_Initialise
create: Tagged_Vector_Random_Initialise, Population, \size
create: Tagged_Vector_Random_Initialise, History, Population, \size

class Matrix_Random_Initialise:
ctor: Random, Initialise > Vector_Random_Initialise
create: Matrix_Random_Initialise, Population, \size
create: Matrix_Random_Initialise, History, Population, \size

class Ordered_Random_Initialise:
ctor: Random, Initialise > Ordered_Random_Initialise
create: Ordered_Random_Initialise, Population, \size
create: Ordered_Random_Initialise, History, Population, \size

class Search_Initialise:
ctor: Creator, \n > Search_Initialise
create: Search_Initialise, Population, \size
create: Search_Initialise, History, Population, \size
private Search_Initialise:
parents: Search_Initialise > \number
breed: Search_Initialise, [Representation], Population > \succeed

class Distributed_Initialise:
ctor: Random > Search_Initialise

create: Distributed_Initialise, Population, \size
create: Distributed_Initialise, History, Population, \size
class Heuristic_Initialise:
ctor: Random > Heuristic_Initialise
create: Heuristic_Initialise, Population, \size
create: Heuristic_Initialise, History, Population, \size

class Reproduction:
ctor: > Reproduction
create: Reproduction, Population, \size
create: Reproduction, History, Population, \size
selector: Reproduction, Selector > \required

class Clone_Reproduction:
ctor: > Clone_Reproduction
create: Clone_Reproduction, Population, \size
create: Clone_Reproduction, History, Population, \size
private Clone_Reproduction:
parents: Clone_Reproduction > \number
breed: Clone_Reproduction, [Representation], Population > \succeed

class Inversion_Reproduction:
ctor: Random, \portion > Inversion_Reproduction
private Inversion_Reproduction:
parents: Inversion_Reproduction > \number

class Biased_Inversion_Reproduction:
ctor: Random, \portion > Biased_Inversion_Reproduction
create: Biased_Inversion_Reproduction, Population, \size
create: Biased_Inversion_Reproduction, History, Population, \size
private Inversion_Reproduction:
breed: Biased_Inversion_Reproduction, [Representation], Population > \succeed

class Unbiased_Inversion_Reproduction:
ctor: Random, \portion > Unbiased_Inversion_Reproduction
create: Unbiased_Inversion_Reproduction, Population, \size
create: Unbiased_Inversion_Reproduction, History, Population, \size
private Inversion_Reproduction:
breed: Unbiased_Inversion_Reproduction, [Representation], Population > \succeed

```
class Mutation_Reproduction:
ctor: Random > Mutation_Reproduction
private Mutation_Reproduction:
parents: Mutation_Reproduction > \number

class Bit_Mutation_Reproduction:
ctor: Random > Bit_Mutation_Reproduction
create: Bit_Mutation_Reproduction, Population, \size
create: Bit_Mutation_Reproduction, History, Population, \size
private Bit_Mutation_Reproduction:
breed: Bit_Mutation_Reproduction, [Bit_Representation], Population > \succeed

class Number_Mutation_Reproduction:
ctor: Random > Number_Mutation_Reproduction

class Random_Number_Mutation_Reproduction:
ctor: Random, \lower, \upper > Random_Number_Mutation_Reproduction
create: Random_Number_Mutation_Reproduction, Population, \size
create: Random_Number_Mutation_Reproduction, History, Population, \size
private Random_Number_Mutation_Reproduction:
breed: Random_Number_Mutation_Reproduction, [Number_Representation],
    Population > \succeed

class Creep_Number_Mutation_Reproduction:
ctor: Random, \lower, \upper, \creep > Creep_Number_Mutation_Reproduction
create: Creep_Number_Mutation_Reproduction, Population, \size
create: Creep_Number_Mutation_Reproduction, History, Population, \size
private Creep_Number_Mutation_Reproduction:
breed: Creep_Number_Mutation_Reproduction, [Number_Representation], Population
    > \succeed

class Vector_Mutation_Reproduction:
ctor: Random > Vector_Mutation_Reproduction

class Single_Vector_Mutation_Reproduction:
ctor: Random, Mutation_Reproduction > Single_Vector_Mutation_Reproduction
create: Single_Vector_Mutation_Reproduction, Population, \size
create: Single_Vector_Mutation_Reproduction, History, Population, \size
private Single_Vector_Mutation_Reproduction:
breed: Single_Vector_Mutation_Reproduction, [Vector_Representation], Population >
```

```
    \succeed

class Deletion_Vector_Mutation_Reproduction:
ctor: Random > Deletion_Vector_Mutation_Reproduction
create: Deletion_Vector_Mutation_Reproduction, Population, \size
create: Deletion_Vector_Mutation_Reproduction, History, Population, \size
private Deletion_Vector_Mutation_Reproduction:
breed: Deletion_Vector_Mutation_Reproduction, [Vector_Representation], Population
    > \succeed

class Addition_Vector_Mutation_Reproduction:
ctor: Random > Addition_Vector_Mutation_Reproduction
create: Addition_Vector_Mutation_Reproduction, Population, \size
create: Addition_Vector_Mutation_Reproduction, History, Population, \size

class Duplication_Addition_Vector_Mutation_Reproduction:
ctor: Random > Duplication_Addition_Vector_Mutation_Reproduction
create: Duplication_Addition_Vector_Mutation_Reproduction, Population, \size
create: Duplication_Addition_Vector_Mutation_Reproduction, History, Population,
    \size
private Duplication_Addition_Vector_Mutation_Reproduction:
breed: Duplication_Addition_Vector_Mutation_Reproduction, [Vector_Representation],
    Population > \succeed

class Initialised_Addition_Vector_Mutation_Reproduction:
ctor: Random, Initialise > Initialised_Addition_Vector_Mutation_Reproduction
create: Initialised_Addition_Vector_Mutation_Reproduction, Population, \size
create: Initialised_Addition_Vector_Mutation_Reproduction, History, Population, \size
private Initialised_Addition_Vector_Mutation_Reproduction:
breed: Initialised_Addition_Vector_Mutation_Reproduction, [Vector_Representation],
    Population > \succeed

class Related_Addition_Vector_Mutation_Reproduction:
ctor: Random > Related_Addition_Vector_Mutation_Reproduction
create: Related_Addition_Vector_Mutation_Reproduction, Population, \size
create: Related_Addition_Vector_Mutation_Reproduction, History, Population, \size
private Related_Addition_Vector_Mutation_Reproduction:
breed: Related_Addition_Vector_Mutation_Reproduction, [Vector_Representation],
    Population > \succeed
```

class Cut_Vector_Mutation_Reproduction:
ctor: Random > Cut_Vector_Mutation_Reproduction
create: Cut_Mutation_Reproduction, Population, \size
create: Cut_Mutation_Reproduction, History, Population, \size
private Cut_Vector_Mutation_Reproduction:
breed: Cut_Vector_Mutation_Reproduction, [Vector_Representation], Population > \succeed

class Ordered_Mutation_Reproduction:
ctor: Random > Ordered_Mutation_Reproduction

class OrderBased_Ordered_Mutation_Reproduction:
ctor: Random > OrderBased_Ordered_Mutation_Reproduction
create: OrderBased_Ordered_Mutation_Reproduction, Population, \size
create: OrderBased_Ordered_Mutation_Reproduction, History, Population, \size
private OrderBased_Ordered_Mutation_Reproduction:
breed: OrderBased_Ordered_Mutation_Reproduction, [Ordered_Representation], Population > \succeed

class PositionBased_Ordered_Mutation_Reproduction:
ctor: Random > PositionBased_Ordered_Mutation_Reproduction
create: PositionBased_Ordered_Mutation_Reproduction, Population, \size
create: PositionBased_Ordered_Mutation_Reproduction, History, Population, \size
private PositionBased_Ordered_Mutation_Reproduction:
breed: PositionBased_Ordered_Mutation_Reproduction, [Ordered_Representation], Population > \succeed

class ScrambleSublist_Ordered_Mutation_Reproduction:
ctor: Random > ScrambleSublist_Ordered_Mutation_Reproduction
create: ScrambleSublist_Ordered_Mutation_Reproduction, Population, \size
create: ScrambleSublist_Ordered_Mutation_Reproduction, History, Population, \size
private ScrambleSublist_Ordered_Mutation_Reproduction:
breed: ScrambleSublist_Ordered_Mutation_Reproduction, [Ordered_Representation], Population > \succeed

class RandomHillClimb_Mutation_Reproduction:
ctor: Random, Mutation > RandomHillClimb_Mutation_Reproduction
create: RandomHillClimb_Mutation_Reproduction, Population, \size
create: RandomHillClimb_Mutation_Reproduction, History, Population, \size
private RandomHillClimb_Mutation_Reproduction:

breed: RandomHillClimb_Mutation_Reproduction, [Representation], Population > \succeed

class Crossover_Reproduction:
ctor: Random > Crossover_Reproduction
create: Crossover_Reproduction, Population, \size
create: Crossover_Reproduction, History, Population, \size
private Crossover_Reproduction:
parents: Crossover_Reproduction > \number

class Number_Crossover_Reproduction:
ctor: Random > Number_Crossover_Reproduction

class Average_Number_Crossover_Reproduction:
ctor: Random > Average_Number_Crossover_Reproduction
create: Average_Number_Crossover_Reproduction, Population, \size
create: Average_Number_Crossover_Reproduction, History, Population, \size
private Average_Number_Crossover_Reproduction:
breed: Average_Number_Crossover_Reproduction, [Number_Representation], Population > \succeed

class Vector_Crossover_Reproduction:
ctor: Random > Vector_Crossover_Reproduction

class OnePoint_Vector_Crossover_Reproduction:
ctor: Random > OnePoint_Vector_Crossover_Reproduction
create: OnePoint_Vector_Crossover_Reproduction, Population, \size
create: OnePoint_Vector_Crossover_Reproduction, History, Population, \size
private OnePoint_Vector_Crossover_Reproduction:
breed: OnePoint_Vector_Crossover_Reproduction, [Vector_Representation], Population > \succeed

class MultiPoint_Vector_Crossover_Reproduction:
ctor: Random, \n > MultiPoint_Vector_Crossover_Reproduction
create: MultiPoint_Vector_Crossover_Reproduction, Population, \size
create: MultiPoint_Vector_Crossover_Reproduction, History, Population, \size
private MultiPoint_Vector_Crossover_Reproduction:
breed: MultiPoint_Vector_Crossover_Reproduction, [Vector_Representation], Population > \succeed

class Segmented_Vector_Crossover_Reproduction:
ctor: Random, \probability > Segmented_Vector_Crossover_Reproduction
create: Segmented_Vector_Crossover_Reproduction, Population, \size
create: Segmented_Vector_Crossover_Reproduction, History, Population, \size
private Segmented_Vector_Crossover_Reproduction:
breed: Segmented_Vector_Crossover_Reproduction, [Vector_Representation], Population > \succeed

class Uniform_Vector_Crossover_Reproduction:
ctor: Random > Uniform_Vector_Crossover_Reproduction
create: Uniform_Vector_Crossover_Reproduction, Population, \size
create: Uniform_Vector_Crossover_Reproduction, History, Population, \size
private Uniform_Vector_Crossover_Reproduction:
breed: Uniform_Vector_Crossover_Reproduction, [Vector_Representation], Population > \succeed

class Shuffle_Vector_Crossover_Reproduction:
ctor: Random, Crossover > Shuffle_Vector_Crossover_Reproduction
create: Shuffle_Vector_Crossover_Reproduction, Population, \size
create: Shuffle_Vector_Crossover_Reproduction, History, Population, \size
private Shuffle_Vector_Crossover_Reproduction:
breed: Shuffle_Vector_Crossover_Reproduction, [Vector_Representation], Population > \succeed

class Traverse_Vector_Crossover_Reproduction:
ctor: Random, Crossover > Traverse_Vector_Crossover_Reproduction
create: Traverse_Vector_Crossover_Reproduction, Population, \size
create: Traverse_Vector_Crossover_Reproduction, History, Population, \size
private Traverse_Vector_Crossover_Reproduction:
breed: Traverse_Vector_Crossover_Reproduction, [Vector_Representation], Population > \succeed

class Splice_Vector_Crossover_Reproduction:
ctor: Random, Crossover > Splice_Vector_Crossover_Reproduction
create: Splice_Vector_Crossover_Reproduction, Population, \size
create: Splice_Vector_Crossover_Reproduction, History, Population, \size
private Splice_Vector_Crossover_Reproduction:
breed: Splice_Vector_Crossover_Reproduction, [Vector_Representation], Population > \succeed

class Ordered_Crossover_Reproduction:
ctor: Random > Ordered_Crossover_Reproduction

class PositionBased_Ordered_Crossover_Reproduction:
ctor: Random > PositionBased_Ordered_Crossover_Reproduction
create: PositionBased_Ordered_Crossover_Reproduction, Population, \size
create: PositionBased_Ordered_Crossover_Reproduction, History, Population, \size
private PositionBased_Ordered_Crossover_Reproduction:
breed: PositionBased_Ordered_Crossover_Reproduction, [Ordered_Representation], Population > \succeed

class OrderBased_Ordered_Crossover_Reproduction:
ctor: Random > OrderBased_Ordered_Crossover_Reproduction
create: OrderBased_Ordered_Crossover_Reproduction, Population, \size
create: OrderBased_Ordered_Crossover_Reproduction, History, Population, \size
private OrderBased_Ordered_Crossover_Reproduction:
breed: OrderBased_Ordered_Crossover_Reproduction, [Ordered_Representation], Population > \succeed

class Repair_Reproduction:
ctor: > Repair_Reproduction
create: Repair_Reproduction, Population, \size
create: Repair_Reproduction, History, Population, \size
private Crossover_Reproduction:
parents: Repair_Reproduction > \number

History Class

class History:
ctor: > History
ctor: Istream > History
print: History, Ostream
add: History, Individual, [Individual]
retrieve: History, Individual > [Individual]

Random Class

class Random:
ctor: > Random
ctor: \seed > Random
ctor: Istream > Random
print: Random, Ostream
value: Random > \boolean
value: Random, \range > \integer
value: Random, \lower, \upper > \integer
value: Random, \range, \precision > \real
value: Random, \lower, \upper, \precision > \real

Appendix E

Maintenance Document

The toolkit is a large collection of object many of which depend on each other for their function. Classes for the core toolkit were written and tested in a specific order to try to minimise the construction of test harnesses. The Random and Ending classes are entirely self contain and so they were chosen for the initial stages of implementation. Each test harness used tries all method calls individually and also in certain combinations which might otherwise have hidden certain flaws. Next the Representation, Evaluation, and Individual were written. Each of these has very minimal interfaces and required only small harnesses. The Population, Selection, and Fitness classes form a sequence of dependence and their creation and testing followed this. Because of its smaller size Model was written next although testing was held back. The most difficult implementation area was the Creation class and its children which are all highly connected. Testing of these classes was combined with the integration of all other classes. Fault finding was more difficult because of this but otherwise complex test harnesses would have been required. After the initial problems had been solved in the simplest possible toolkit organisation new components were completed and tested in place.

Appendix F

Status Report

There are three issues present in considering the status of the project - size, functionality, and quality of the toolkit. The Statement of Requirements demanded support for a variety of genetic algorithm techniques and also suggested that others would be helpful.

As a top priority a core toolkit was defined to allow the classic genetic algorithm to be implemented using the toolkit. This first target was reached, to achieve it approximately 20 components had to function in cooperation. Although the basic algorithm performs perfectly well there are some design decision, such as the *Attach* method for Fitness and Selector classes, which, in retrospect, might have been made differently. One current difficultly with the toolkit is that Individual objects are never disposed of and because of this any genetic algorithm built using the toolkit will eventually run out of memory.

There were a number of other suggested techniques such as vector, ordered list, and number representations which have also been completed to provide support for the more complex example algorithms. Work on reproduction operators, particularly for crossover, has not reached the range discussed during the design stage. Amongst numerous breeding techniques which could have been used only the simplest, the generational model, was implemented.

Another section of the Statement of Requirements gives a list of other facilities which might be useful in a genetic algorithm toolkit. These were all considered during the design stage and should be possible within the confines of the current design but none have been implemented. There are two areas in which this is a particular disappointment - family trees provided by a History class and the saving the status of an active genetic algorithm.

The demonstration algorithms proved that the toolkit was capable of speeding the creation of genetic algorithm by a combination of predefined and extensible components. However, none of these examples are good examples of a genetic algorithm. The Blues is only a toy suitable for the teaching of ideas. My implementation of the Travelling Salesman Problem produces very poor results compared to other techniques and the violin music notation algorithm is only likely to perform well with small data sets but does show scope for improvement.

Appendix G

Summary Log

August - September 1996

- review of available material
- initial plans possible components

October 1996

- problem definition
- statement of requirements
- component design

November 1996

- external specification
- component design
- first summary of genetic algorithms
- The Blues in Java and Ada
- Travelling Salesman Problem in Ada

December - January 1997

- core toolkit coding and testing

February 1997

- Travelling Salesman Problem using toolkit
- violin music notation using toolkit
- second summary of genetic algorithms

March 1997

- first draft of report

April 1997

- finalise report
- presentation

The project was allocated 12 hours each week for 21 weeks and therefore the estimated total time spent on it is 252 hours.

Appendix H

Project Code

Presented here is all toolkit code completed during the project:

- mutants.ads
- mutants-core-creator.bit-mutation-reproduction.adb
- mutants-core-creator.bit-mutation-reproduction.ads
- mutants-core-creator.onepoint-vector-crossover-reproduction.ads
- mutants-core-creator.bit_random_initialise.adb
- mutants-core-creator.bit_random_initialise.ads
- mutants-core-creator.bxor.adb
- mutants-core-creator.bxor.ads
- mutants-core-creator.clone_reproduction.adb
- mutants-core-creator.clone_reproduction.ads
- mutants-core-creator.onepoint_vector_crossover-reproduction.adb
- mutants-core-creator.onepoint_vector_crossover-reproduction.ads
- mutants-core-creator.single_bit_vector_reproduction.ads
- mutants-core-creator.single_vector_reproduction.adb
- mutants-core-creator.single_vector_reproduction.ads
- mutants-core-creator.vector_bit_random_initialise.adb
- mutants-core-creator.vector_initialise.adb
- mutants-core-creator.vector_initialise.ads
- mutants-core-creator.ads
- mutants-core-ending-count.adb
- mutants-core-ending-count.ads
- mutants-core-ending-time.adb
- mutants-core-ending-time.ads
- mutants-core-ending.ads
- mutants-core-evaluation-binary_bit_vector.adb
- mutants-core-evaluation-binary_bit_vector.ads
- mutants-core-evaluation-one.adb
- mutants-core-evaluation-one.ads
- mutants-core-evaluation.ads
- mutants-core-fitness-unity.adb
- mutants-core-fitness-unity.ads
- mutants-core-fitness.ads
- mutants-core-model-generation.adb
- mutants-core-model-generation.ads

- mutants-core-model.ads
- mutants-core-model.adb
- mutants-core-population.adb
- mutants-core-population.ads
- mutants-core-random.adb
- mutants-core-random.ads
- mutants-core-representation-bit.adb
- mutants-core-representation-bit.ads
- mutants-core-representation-bit_vector.ads
- mutants-core-representation-integer.adb
- mutants-core-representation-integer.ads
- mutants-core-representation-order.adb
- mutants-core-representation-order.ads
- mutants-core-representation-vector.adb
- mutants-core-representation-vector.ads
- mutants-core-representation.ads
- mutants-core-selector-antiroulette.adb
- mutants-core-selector-antiroulette.ads
- mutants-core-selector-cyclic.adb
- mutants-core-selector-cyclic.ads
- mutants-core-selector-high.adb
- mutants-core-selector-high.ads
- mutants-core-selector-low.adb
- mutants-core-selector-low.ads
- mutants-core-selector-roulette.adb
- mutants-core-selector-roulette.ads
- mutants-core-selector.ads
- mutants-extended-creator-integer_order_initialise.adb
- mutants-extended-creator-integer_order_initialise.ads
- mutants-extended-creator-integer_order_mutation_reproduction.ads
- mutants-extended-creator-integer_order_random_reproduction.ads
- mutants-extended-creator-order_crossover_reproduction.adb
- mutants-extended-creator-order_crossover_reproduction.ads
- mutants-extended-creator-order_initialise.adb
- mutants-extended-creator-order_initialise.ads
- mutants-extended-creator-order_mutation_reproduction.adb
- mutants-extended-creator-order_mutation_reproduction.ads
- mutants-extended-creator-order_random_reproduction.ads
- mutants-extended-creator-order_random_reproduction.ads
- mutants-extended-creator.ads

- mutants-extended-evaluation.ads
- mutants-extended-representation-bit_vector.ads
- mutants-extended-representation-integer_order.ads
- mutants-extended-representation.ads
- mutants-extended.ads

This is the code for the example genetic algorithms constructed using the toolkit:

- blues.adb
- blues_evaluation.adb
- blues_evaluation.ads
- tsp.adb
- tsp_evaluation.adb
- tsp_evaluation.ads
- music.adb
- music_evaluation.adb
- music_evaluation.ads