Grammatical Evolution in Python

Tim Painter
BSc (hons) in Mathematics and Computing

2006
Grammatical Evolution in Python
Submitted by Tim Painter

COPYRIGHT

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop).

This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed.......................................

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed.....................................
Abstract

Grammatical Evolution is a system developed by O’Neill and Ryan [2003] that seeks to create a solution to a problem by adapting observations from natural genetics to evolve a solution. It is based upon the genetic algorithm developed by Holland [1975] but extends it to utilise a grammar in Backus Naur Form to guide potential solutions. This document describes the development of a version of this system written in the programming language Python. The emphasis of the project is simplicity and clarity of the code; the aim of this system being to demonstrate a practical implementation of grammatical evolution for the use of students in the field. A secondary aim is to investigate the effect of the new operators added into the grammatical evolution system. Although system fails to self-improve effectively, it is not considered a failure, since it has potential as a learning tool. Additionally, other re-implementations of this system demonstrate the same problem, so there may be the need for further information on the implementation of the original system. The process of crossover is examined, and several variants are tested. The use of adaptive mutation is also considered, but no reliable solution is found in the field considered. Other new operators are considered, in particular a prune operator. In conclusion the system fails to achieve the same level of performance that O’Neill and Ryan claim, but presents a technically correct system that allows an inexperienced user to learn how a grammatical evolution system works.
Acknowledgements

To Dr Alwyn Barry, my project supervisor, thank you for suggesting the project idea and for the constant and unstinted guidance, encouragement and enthusiasm shown throughout the length of the project.

To my mother and sister, Susan and Louisa; thank you for your sterling work in helping to proof read the document.
# Table of Contents

1. Introduction .................................................. 1

I  Background Research .............................................. 3

2. Evolutionary Algorithms ........................................... 4
   2.1. Natural Genetics – the motivating model................................. 4
       2.1.1. Structure ........................................................................ 4
       2.1.2. The Phenotype .............................................................. 4
       2.1.3. Transcription and Translation ........................................ 5
   2.2. Evolutionary Algorithms ............................................... 5

3. Genetic Algorithms ................................................ 7
   3.1. GA Operators .................................................................... 7
       3.1.1. The Utility of the Operators ......................................... 8
       3.1.2. Crossover ..................................................................... 8
       3.1.3. Mutation ...................................................................... 10
       3.1.4. Exploitation vs. Exploration ......................................... 10
       3.1.5. Other proposed operators ........................................... 10
   3.2. Selection within a Genetic Algorithm .................................. 11
       3.2.1. Fitness Proportional Methods ....................................... 11
       3.2.2. Ranking Methods ......................................................... 13
   3.3. Additional techniques ..................................................... 14
       3.3.1. Elitism ............................................................................ 14
       3.3.2. Generational vs. Steady State Selection ......................... 14

4. Grammatical Evolution ............................................ 15
   4.1. Limitations of Genetic Algorithms ..................................... 15
       4.1.1. Lack of Genotype-Phenotype Distinction ....................... 15
       4.1.2. Fixed Length Strings ................................................... 15
   4.2. The use of Grammars in Genotype-Phenotype Mapping ........... 15
       4.2.1. Grammars in other Evolutionary Algorithms ................. 16
       4.2.2. Benefits of using Grammars ....................................... 16
       4.2.3. Backus-Naur Form ....................................................... 17
   4.3. Mapping from the Genotype to the Phenotype ....................... 18
   4.4. A Genetic Algorithm as base for Grammatical Evolution ........ 18
       4.4.1. Example ................................................................. 19
   4.5. Proposed Additional Features ......................................... 20
       4.5.1. Wrapping ................................................................. 20
       4.5.2. Degeneracy and Neutral Mutation ............................... 20
       4.5.3. Prune operator ......................................................... 21
       4.5.4. Duplication ............................................................ 22

5. From Theory to Practice ............................................ 23
# Design and Implementation

## Requirements

6.1. Functional Requirements

6.2. Language Requirements

6.3. User Requirements
   - 6.3.1. End User
   - 6.3.2. Researcher

6.4. Conclusion

## Design

7.1. Structure
   - 7.1.1. The Foundation
   - 7.1.2. The Extension

7.2. Data Import

## Implementation

8.1. Issues of Clarity

8.2. Trade-offs

8.3. BNF Grammar Import
   - 8.3.1. Text Based BNF
   - 8.3.2. XML BNF

8.4. XML Parameters and Inputs

8.5. Logging

8.6. Optimisation

8.7. Problems

8.8. Dynamically Extending the System

# Testing and Experimentation

## Testing

9.1. Full System Test – Symbolic Regression
   - 9.1.1. Experiment Tableau
   - 9.1.2. Results

9.2. Full System Test – Santa Fe Ant Trail

9.3. Conclusion

## Experimental Work and Additional Operators

10.1. Crossover and Self-Improvement
   - 10.1.1. Two Point Crossover
   - 10.1.2. One Point Crossover
   - 10.1.3. One Point Variable Length Crossover
   - 10.1.4. Homologous Crossover
   - 10.1.5. Homologous Transpose Crossover
   - 10.1.6. Variable Mutation Rate
   - 10.1.7. Combined Experiment
   - 10.1.8. Results Distribution
   - 10.1.9. Homologous Transpose
10.2. Prune..........................................................................................51
  10.2.1. Prune Experiment.................................................................51
10.3. Wrapping ...................................................................................52
10.4. Duplicate ....................................................................................53

IV Conclusions 54

11. Conclusions 55
  11.1. Reflections on the Aims of the Project......................................55
    11.1.1. The End User......................................................................56
    11.1.2. The Researcher.................................................................56
    11.1.3. Conclusions .....................................................................57
  11.2. Self-Appraisal.........................................................................57
  11.3. Improvements........................................................................58
  11.4. Closing Comments ...............................................................58

12. References 59

V Appendices 61

A. Test Plan 62

B. Program Listing 65
  B.1. Interaction of Modules ..............................................................66
  B.2. GE.py ..................................................................................67
  B.3. BNF.py ................................................................................74
  B.4. GEsttings.py .........................................................................81
  B.5. GEparameters.py .................................................................84
  B.6. GEinput_Regression.xml .........................................................87
  B.7. BNF_Regression.xml ...............................................................89
  B.8. Regression.bnf .....................................................................91
  B.9. GEinput_SantaFe.xml ...............................................................92
  B.10. SantaFe.py ........................................................................94
  B.11. BNF.xsl ..............................................................................97
  B.12. BNF.xsd .............................................................................99

C. Usage Guide 101
  C.1. Prerequisites ........................................................................101
  C.2. Installation ............................................................................101
  C.3. Set up ..................................................................................101
  C.4. Running the System..............................................................101
    C.4.1. Additional Options ...........................................................102
    C.4.2 Access to the Supporting Files ...........................................102
1. Introduction

Problem solving is a cornerstone of the vast majority of computer applications, and one theme common to most applications is that the solution is programmed directly – the computer is told what to do at each step of the way. Alternatively, the computer is provided with a set of rules by which to process the data; therefore, for each problem, either a new solution must be programmed or an existing solution must be modified to include the details of the problem. However, as Arthur Samuel remarked [Samuel 1959 cited by O’Neil and Ryan 2003]:

“Tell the computer what to do, not how to do it”

This approach is a step aside from the usual methods, as it requires automated programming – the computer is given a high-level problem description for which it should generate a programmatic solution itself [Koza et al 1999]. This approach is surprisingly effective, and even works well across a wide selection of problems. For any one problem area, its performance can be beaten by a specialised method that exploits all of the facets of the given problem; but the beauty of the genetic algorithm (and other evolutionary computation tools) is that it sees the actual problem as a ‘black box’ [Goldberg 1989], and only the outputs matter. This means that it is technically classified as a “weak” method, since it does not require much information about the problem. However, it presents an apparent anomaly in problem theory, due to its uniformly efficient solution of problems – a weak but yet powerful method. [Davies 1987] Goldberg [Goldberg 1989] provided a good pictorial representation of this, as seen in Figure 1.1. The genetic algorithm would be an example of a robust scheme, while the specific method can be seen as a spike, which is very good on one problem or one class on problems but is of no help on any other problems.

![Figure 1.1](image.png)

**Figure 1.1** Many traditional schemes work well in a narrow problem domain. Enumerative schemes and random walks work equally inefficiently across a broad spectrum. A robust method works well across a broad spectrum of problems.
However, we can go a step further, and consider problems that require a more general solution, or a symbolic solution. It is in these cases that Grammatical Evolution allows complex solutions to be built that grow to meet the bounds of the problem. Grammatical evolution takes the constraints of the problem (encoded in Backus Naur form) and uses a genetic algorithm to control the choices made at each step of generating the solution code [Ryan et al 1998a]. This grammatical approach ensures that solutions are valid for the problem, side stepping the problem in other systems, like genetic programming, where invalid structures exist. This approach also has the side effect that any grammar describing a problem can be modified when further information is discovered – enriching the problem space and lessening the ‘weakness’ of the method without compromising on the power.

A few examples where this approach have been be applied are:
- To evolve a program in a given language to perform a specified task
  - Santa Fe Ant Trail problem [O’Neill, Ryan 1999]
- To produce a symbolic result that expresses the problem
  - deducing trigonometric identities [Ryan et al 1998b]
  - symbolic integration of functions [O’Neill and Ryan 1998]

Previous implementations of Grammatical Evolution have been written in C, LISP or Java; but there are a number of potential benefits to implementing Grammatical Evolution in Python. These are further expanded upon in the Requirements chapter.
- Readability
  - Python enforces a logical indentation scheme that clearly shows the grouping of code segments.
- Simplicity
  - Python uses a straightforward syntax which bears similarity to natural language.
- Self-Execution
  - Python can parse and execute Python code passed as a string to the interpreter. A phenotype can be directly evaluated at run time.

In this document we aim to explore the theoretical background to the grammatical evolution system and use this as a basis upon which to plan the development of an implementation of grammatical evolution. The development process will be documented and conclude in a reflection on the system as implemented.

This project offers a number of challenges to the author along the way, notably there are three major challenges. The first challenge is the sheer breadth and depth of the theory. It will be seen in the following chapter that the field of genetic algorithms is vast and there is copious documentation on many aspects within the field. Limiting the scope to that which is necessary for the project will be the first challenge. The second challenge will be the process of determining the requirements and then implementing them into a viable system, while the third challenge will be to further extend this system, as well as to investigate any issues that arise during the implementation cycle.
Section I:

Background Research
2. Evolutionary Algorithms

2.1. Natural Genetics – the motivating model

In order to understand the process by which a genetic algorithm functions, it is helpful to first consider the human cell. The study of “the central dogma of Molecular Biology” [O’Neill and Ryan 2003] provides some understanding of the processes of transcription and translation of genetic material.

2.1.1. Structure

As Koza points out, “structure is a consequence of fitness” or alternatively “structure arises from fitness” [Koza 1992]. DNA is a successful (i.e. fit) structure (as evidenced by its presence in all living beings), so its structure merits closer study. Because of this fitness, many concepts that are adopted in genetic algorithms and grammatical evolution directly follow from the subdivisions of the DNA molecule, as listed below:

- The atomic values are called alleles, with a value chosen from a set of possible values. In DNA, these alleles are represented by C, T, A and G
- Alleles are grouped into codons. In DNA, there are three alleles to a codons
- The set of codons is called the chromosome.
- The set of chromosomes is called the genotype.

![Figure 2.1](image)

**Figure 2.1**

2.1.2. The Phenotype

The counterpart to the genotype is the phenotype. The phenotype is the collection of expressed characteristics of an individual, and is created from a complex combination of proteins. The connection between them is that the genotype encodes the instructions for the sequence of proteins that form the phenotype. We can see that there is a mapping from the genotype to the phenotype (a genotype will create a specific phenotype). However it is not possible to propose an inverse
mapping due to the complex interactions of proteins. Some of the physical traits are determined by a single protein and other traits are determined by a combination of proteins. This is extremely difficult to decompose, leading Holland to comment that “the greatest complexities come about because the effects of different enzymes are not additive – a phenomenon known as epistasis” [Holland 1992]. As “the amalgam of observed characteristics – the phenotype – depends strongly upon these epistatic effects” [Holland 1992], there is no simple way to deduce the genotype when given a phenotype. The essential point is that changing one allele could potentially have large effects – but not necessarily\(^1\).

2.1.3. Transcription and Translation

To perform the mapping from genotype to phenotype, there are two main processes: transcription and translation. Study of both of these processes is rewarding, as they form a basis for the implementation of grammatical evolution. For this to occur, the cell needs to make a copy of the information about the proteins and their ordering, to send to the protein manufacturing area. Thus the information on the DNA strand is transcribed onto a new strand, called messenger RNA (mRNA).

This then “carries the encoded instructions of the genes to the machinery of the cell that can synthesise proteins” [O’Neill and Ryan 1998]. This process is illustrated by the top arrow on Figure 2.2. Once the mRNA is in place, the translation can occur, shown as the bottom arrow on Figure 2.2. The mRNA strand is read codon by codon, and each codon is translated into the protein that it encodes. Each codon encodes for a particular amino acid. It is a degenerate mapping, as the 64 different types of codon map to the 20 different amino acids (proteins) [Banzhaf et al 1998].

2.2. Evolutionary Algorithms

The natural genetic process mentioned above provided inspiration for computer scientists working on the principles of machine intelligence and adaptation. One of the first implementations was developed by John Holland [Holland 1975, republished 1992], and was based on the principles of natural selection combined with the processes described in the previous section; except that there was no

\(^1\) There is also a theory regarding neutral mutations, whereby a codon changes value, but still encodes for the same protein by the degenerate relationship between codons and proteins. This has been shown to have benefits in some circumstances.
phenotype. This algorithm was refined and simplified and the **Simple Genetic Algorithm (SGA)** was described by Goldberg [Goldberg 1989]. The SGA is also referred to as the canonical genetic algorithm [Whitley 1993], as it is the basis upon which all subsequent genetic algorithms rest. Here we will only examine the details of genetic algorithms, but it must be noted that there do exist other implementations of natural genetics theory.

During a similar time period others also worked upon implementations, for example

- **Evolution Strategies** developed by Rechenberg
  - Optimisation of real valued parameters for devices such as airfoils

- **Evolutionary Programming** developed by Fogel, Owens and Walsh
  - Represented by finite state machines, with mutation as the only operator
    [Fogel, Owens and Walsh 1966 referenced by Mitchell 1996]
3. Genetic Algorithms

Genetic Algorithms (GA) were designed for the purpose of harnessing the adaptive process that drives the natural process of genetic evolution and utilising it to solve problems. According to Mitchell [Mitchell 1996], genetic algorithms were invented for the purpose of a formal study of the phenomenon of adaptation as it occurs in nature, as well as being an abstraction of biological evolution. It is useful to note the linguistic root of the word adaptation: *ad aptare* – to fit in [Holland 1992]. With this in mind, we can see the potential for use of genetic algorithms as a search technique – through iteration, the solution should adapt in such a way as to conform to the parameters set, to fit into the niche created for this purpose.

The SGA is a population based algorithm, a population (genotype) of candidate solutions (chromosomes) is maintained, each with its own fitness value; and a new population is generated from the previous population at each time step [Mitchell 1996] [Koza 1992], this is also referred to as a beam search, as a population of points is maintained, and the most promising points at each stage are kept. [Banzhaf et al 1998]. This is then a multi-dimensional search, which derives its power from its encouragement of information formation and exchange between candidate solutions. [Michalewicz 1996] A typical representation for a chromosome is as a binary string of fixed length, it has codons consisting of one allele and this genotype can be evaluated to a fitness value by some predefined function on the string.$^2$

### 3.1. GA Operators

The initial operators suggested for the genetic algorithm took the observations from natural genetics, and simply applied them to the population of strings in the algorithm.

Holland initially proposed the operators crossover, inversion and mutation [Holland 1992], combined with some selection function. However, in the description of the structure of the SGA, Goldberg [Goldberg 1989] refers to

---

$^2$ for example, a binary to decimal conversion
reproduction (selection), crossover and mutation as the three operators to consider; inversion is covered later in his book, but relies upon each allele retaining its meaning without regard to its actual position to be used effectively.

3.1.1. The Utility of the Operators

By themselves, the operators do not make the genetic algorithm, both crossover and mutation (and inversion, which is not used as frequently) are forms of random search. Crossover combines existing individuals in some way to create a hybrid individual, and mutation alters an existing individual into something new. Both are random in their approach, the main difference being that crossover searches deeper within the current search area, while mutation expands out into new areas. There is a balance to be applied, which is discussed later in this section. It is selection that pulls all of these pieces together and propels the search forward. Selection promotes fitter individuals, and thus provides the basis upon which crossover and mutation may be applied effectively, thus all three components are required for the genetic algorithm to be truly useful.

3.1.2. Crossover

Crossover provides a way in which characteristics from two individuals can be combined so as to produce a (hopefully) improved individual that has moved closer to the optimum fitness. In this way, it could be said that crossover acts as an accelerator that promotes emergent behaviour [Whitley 1993] in as much as it allows good characteristics to be passed on at an increasing rate. This is because this operator exchanges subparts of the chromosome, roughly mimicking the effect of recombination between two haploid individuals [Mitchell 1996].

There are several forms of crossover in use; the simplest of which is the single point crossover

3.1.2.1. Single Point Crossover

---

3 although examples have been made where either crossover or mutation is dropped – but never selection
In this operation, a random locus is chosen on each parent chromosome (if the chromosomes are of fixed length, then the same locus is chosen for each) and each chromosome is divided about this point, then the tails of each chromosome are exchanged, creating two new chromosomes – the offspring. This method provides a balance between the disruption of any sequence of co-adapted alleles (a sequence that provides some desirable trait, much as some physical traits are the product of multiple sequential proteins) and a deeper exploration of the existing search space, moving towards positions of higher fitness within the search space being investigated. One problem is that this method has a positional bias with respect to short low order schema [Eschel et al 1989]. That is, the short schemas are more likely to survive intact and thus more likely to be propagated whole. Any longer sequences of co-adapted alleles have a much higher chance of being destroyed by this operator.

3.1.2.2. Two Point Crossover

This operation is similar to single point crossover, but instead of swapping the tails of the parent chromosomes, two loci are selected and the section between those points is swapped to form the offspring. This method follows the actual process of DNA recombination more closely than single point crossover does; the advantage of this method is that it contains less positional bias than single point crossover, so useful ‘building blocks’ within the chromosome are more likely to be preserved and passed on. [Mitchell 1996] It is by no means perfect, as the method still exhibits some positional bias. [Eschelman et al 1989]

3.1.2.3. Uniform Crossover

In response to the concern over the positional bias exhibited by the first two crossover methods, uniform crossover was developed. Instead of choosing a random locus to swap a sequence of genes, this method swaps individual genes at the same locus between the parent chromosomes. This occurs probabilistically, much like mutation, however, the problem is that it is highly destructive to any blocks of functionality within the chromosome [Mitchell 1996], as they are likely to be broken up by the operator.
3.1.3. Mutation

The mutation operator is very simple in implementation, but its effects can be far-reaching. The operator moves along each locus on the chromosome, and probabilistically mutates that locus by reversing the allele stored there. This assumes a binary representation, which is the most common form; for other representations, a random element of the allele ‘alphabet’ should be chosen. It has sometimes been questioned as to why an intelligent search method such as GA requires such a random and destructive operator, but experimentation has shown that mutation (at a low probability) shows many positive effects. Whitley states that there is strong empirical evidence that highly disruptive crossover operators can sometimes search more effectively than less disruptive ones. [Whitley 1993]

The main benefit is that of diversity – sometimes a population may fall prey to the phenomenon of premature convergence, when the search converges to a globally sub-optimal solution. [Koza 1992] In this case crossover cannot help, as it does not introduce any new material, therefore the only way out lies with mutation, which (if it is beneficial) is then propagated by the crossover operator and the population has a chance of advancement. In this way, mutation opens up a new portion of the search space. This is shown in DNA, as mutation causes a gene to become slightly different. Sometimes, the mutated gene is actually better at performing its task, and thus continues to replicate and replaces the original gene.

3.1.4. Exploitation vs. Exploration

The relative importance of crossover versus mutation has been frequently debated, and comes down to a balance between exploitation of the existing material and exploration of the rest of the search space. This is similar to Whitley’s claim that the issue is the relative importance of diversity (mutation) and construction (crossover) [Whitley 1993]. Rawlins [Rawlins 1991] goes further, to suggest how to balance the two disparate directions in which the search proceeds. It was suggested that this issue is a trade-off and that either slow growth could be chosen (which helps prevent premature convergence) or high growth could be used, followed by the re-discovery of important blocks of codons via a high mutation rate.

3.1.5. Other proposed operators

3.1.5.1. Inversion

In his original specification, Holland [Holland 1975 (revised 1992)] suggests inversion as one of the operators to use on a GA. This operator is similar to the two point crossover in that it selects two random loci on the chromosome, but instead of switching this section with another chromosome, the selection is reversed in place. This operator is not often used in genetic algorithms, as it requires that additional positional information is stored about each gene and the meaning of each gene is dependent in some way on this positional data.
3.1.5.2. Duplication
Holland [Holland 1992] also considers the use of a duplication operator. This creates a copy of selected alleles at the end of the chromosome. Supposedly it allows spare copies of good building blocks to be preserved in case of loss of the originals. Again, this probably required positional information to be preserved as well for practical use. Most modern genetic algorithms do not use this operator; however it has been reintroduced into the Grammatical Evolution system – its effectiveness has yet to be investigated.

3.1.5.3. Crowding
De Jong experimented with a crowding operator to help promote diversity in the population [Mitchell 1996]. This punishes any individual that is similar to one that is already in the population by imposing a penal fitness value – ensuring that it is unlikely to be picked for reproduction, this can also involve restrictions on the mating operator (i.e. crossover) that prevents incestuous combinations. Goldberg and Richardson [Goldberg and Richardson 1987, cited by Mitchell 1996] also considered the idea of fitness sharing – each individual’s fitness being decreased by the presence of similar individuals. However both of these can induce a situation referred to as speciation, where the population converges to many points in the fitness landscape [Mitchell 1996], as it is discouraged from converging upon any one point.

3.2. Selection within a Genetic Algorithm
Selection is one of the most crucial parts of the process of generating a new generation, as it is this which determines which chromosomes are picked to have the operators applied to them. As previously mentioned, selection is crucial to the operation of a genetic algorithm, as it reverts to a random search in its absence. Selection chooses good individuals to apply crossover, mutation and any other operators to, to generate the next generation that is composed of the best that the current generation has to offer. For variety’s sake, less fit individuals are also allowed through, but the bulk of selection is biased towards highly fit individuals.

3.2.1. Fitness Proportional Methods
3.2.1.1. Roulette Wheel
The roulette wheel method is the simplest of the fitness proportional methods. It can be envisioned as a wheel which has been divided into slices – one slice for each individual. The angle allocated to each slice being the fitness of the individual as proportional to the total fitness of the population. A random angle is generated, and the individual at this angle is the one chosen. This is repeated until n individuals are chosen (n being the population size). This is very simple to implement, but can cause problems later in the evolution, as it is a biased selection method [Goldberg 1989] that allows outliers to skew the selection. A fairly-highly fit individual can ‘hijack’ the population early on, and stifle competition by dominating the gene pool. The population then may consist of nearly identical individuals [Griffith 2002]. This can lead to premature
convergence of the population to a ‘globally sub-optimal fitness.’ [Koza 1992] This is demonstrated occurring in practice in the genetic algorithm within Griffith’s [Griffith 2002] implementation of a grammatical evolution system.

On a computational level, this method’s complexity has been calculated as being between the orders of $O(n^2)$ and $O(n\log n)$ [Rawlins 1991]. It requires the sum of all fitnesses to operate – necessitating the calculation of the fitness of every individual in the population.

### 3.2.1.2. Stochastic universal sampling

To address some of the issues mentioned above, stochastic universal sampling reduces the chance of outlying individuals taking over the population at an early stage. It does this by taking $n$ equally spaced angles, and selecting the individuals at each point. This does distribute the selection between the individuals in a more varied way, however it is still fitness-proportional and thus fit individuals can still take over the population [Mitchell 1996], although the risk has been reduced.

The actual selection process is cheaper in terms of computation than the roulette wheel method, since only one pass around the wheel is made, not $n$; consequently the complexity of this method is $O(n)$. [Rawlins 1991] However the fitness proportionate step still requires that all fitnesses are calculated at each generation.

### 3.2.1.3. Scaling and Standardised Fitness

To address some of the problems with fitness proportional selection, there have been many methods introduced that perform some form of standardisation or scaling upon the fitnesses. This allows much better control, and also permits variation throughout the run – similar to the style of simulated annealing. This means that differences in fitness are scaled down at the start of the run, to prevent premature convergence, and scaled up as the run progresses – to permit finer differentiation between highly fit and very highly fit individuals.

These methods are a field of study on their own, and beyond the scope of the investigation at hand. The ranking methods described below side-step the problems related to fitness proportionality entirely, and thus scaling is not a necessary component of selection.
3.2.2. Ranking Methods

3.2.2.1. Rank Selection
To combat the problem of premature convergence of the population, which is inherent to fitness proportional methods, it was purposed to detach the process of selection from the actual distribution of fitness values. [Koza 1992] The simplest form is ranking, whereby the individuals are ordered by fitness and then numbered in that order. This maintains a constant differential between the best and worst individuals [Whitley 1993] and so outliers are still promoted (first ranking) but cannot dominate. This method is often more successful at finding a result, due to the diversity that it promotes [Mitchell 1996]. However, computationally, it still requires all fitnesses to be calculated at each generation. As a selection method, it is typically calculated to be at worst O(nlogn). [Rawlins 1991]

3.2.2.2. Tournament Selection
Tournament selection is a more sophisticated rank selection method. From the population two individuals (or more – depending on the tournament settings, 4 to 5 individual tournaments are common) are selected and compared. Sources differ on the precise details of this method. Rawlins states that the best individual is selected [Rawlins 1991]; however Mitchell’s view is that the victor should be selected probabilistically – with the better individual having a higher chance of being chosen, but not a certainty. [Mitchell 1996] This approach allows variety to be maintained, as individuals that have a few unique and desirable features are not automatically excluded, and may survive to spread these features.

This method has the advantage of eliminating the centralised step on every generation, that of calculating the fitnesses of all the individuals. [Koza 1992] Most methods require two passes through the population, instead of the single pass that this method requires, thus it has an appreciable advantage of speed over the rank selection method [Koza 1992] and the fitness proportional methods – only the fitnesses necessary for the tournament are calculated.

In terms of computability, it is also very efficient as the algorithm is of O(n) [Rawlins 1991]. Thus it should scale well to larger populations, if necessary. As far as selection pressure goes, Mitchell states that the pressure is similar to that of rank selection [Mitchell 1996], but if the case of a tournament between more than two individuals is considered, the selection pressure may be adjusted as desired. Banzhaf et al show that small tournaments offer lower selection pressure than larger tournaments. [Banzhaf et al 1998] So, if it is desirable that fitter individuals are more likely to be selected, and the loss of less fit individuals will not significantly impact on the variety, then tournaments between multiple individuals come into their own.
3.3. Additional techniques

3.3.1. Elitism

A part of the problem concerning premature convergence is the loss of important genetic sequences early on in the process. One (or a few) individuals may dominate the population and crossover may become ineffective due to the similarity. A partial solution is the passing (also known as cloning) of one or two individuals unchanged from one generation to the next. De Jong described this as a generation gap $R_1$ (one individual treated as elite) [De Jong 1975 cited in Whitley 1993] and noted that it significantly improved genetic algorithm performance [Mitchell 1996]. This can be biased towards the best individuals (to preserve good individuals from the random nature of probabilistic selection) or the worst individuals (to preserve variety that would likely be lost under the selection pressure). Alternatively one at each end of the spectrum could be passed for balance.

Whitley also describes another form of elitism, the singling out of a particular individual (usually the worst) and giving it special treatment, for example: replacing it with a random chromosome of alleles [Whitley 1993].

3.3.2. Generational vs. Steady State Selection

Developing the concept of elitism further, De Jong [De Jong 1975 cited by Whitley 1993] examined the effect of varying the overlap between generations.

He defined the ‘generation gap’, $G$; ranging between $1.0$ ($R_d$) and $\frac{1}{n}$ ($R_1$), which describes the number of individuals remaining unchanged between subsequent generations. $R_d$ is the standard generational model, where an entire population is generated at each step (although it must be noted that there is usually an implicit overlap, due to the selection method’s probabilistic design). From this, the notion of steady state arose, where only one or two individuals were changed per generation. This is an elitist strategy on the majority of the population and is surprisingly successful in reaching a near-optimal result. Typically, methods will select a number of individuals, perform crossover and mutation upon them, and then return the offspring directly into the population, replacing another individual (probabilistically biased towards the less fit individuals). In Tournament selection, the choice of individuals to be replaced is often limited to a member of the tournament – with the same bias.
4. Grammatical Evolution

4.1. Limitations of Genetic Algorithms
As good at problem solving as they are, there are limits to the genetic algorithm’s utility. Some of the key differences are expanded upon below.

4.1.1. Lack of Genotype-Phenotype Distinction
In a genetic algorithm, the genetic operators act directly upon the candidate solutions [Koza 1992]; the genotype and the phenotype are one and the same. This is a different process to that in the natural model from which inspiration is drawn; in the natural system there is a clear distinction between the generating genotype and the expressed phenotype. As previously stated, the expressed phenotype depends upon the epistatic effects of the different genes within the genotype – the fact that the relation between genes and traits is not additive [Holland 1992] means that even simple modifications to the genotype can have significant effects upon the phenotype. If there is a complete separation between the search and solution spaces, then the phenotype can be as complex as necessary, as all the genetic operators are applied on the genotype [O’Neill and Ryan 2003].

4.1.2. Fixed Length Strings
While operating over a fixed length chromosome is acceptable in many cases, Koza [Koza 1992] states that this representation leaves many issues unsettled. One outstanding issue is that of the output: the required output is frequently in programmatic form, and it is “difficult, unnatural and constraining” to represent a hierarchical computer program of dynamic size and shape by a fixed length string [Koza 1992].

4.2. The use of Grammars in Genotype-Phenotype Mapping
To separate the search space from the solution space, there must be some operation that can provide an ‘onto’ mapping – there must be a unique mapping from the genotype to the phenotype \(^4\) [O’Neill and Ryan 2003]. The use of a context-free grammar such as Backus-Naur Form\(^5\), fulfils this need – when combined with an appropriate method for the selection of the rule to be applied. This has the additional benefit of utilising a pre-existing structure that is a clear to read and unambiguous. O’Neill and Ryan comment in the introduction to their book that “the beauty of grammars is that they provide a single mechanism that can be used to describe all manner of complex structures” [O’Neill and Ryan 2003]. It must be noticed that the use of a grammar does not interfere with the operation of the underlying genetic algorithm; since a grammar simply states what could be done, while the search dictates what should be done. [O’Neill and Ryan 2003]

---

\(^4\) but not necessarily the reverse – consider neutral mutation
\(^5\) sometimes referred to as Backus Normal Form
4.2.1. Grammars in other Evolutionary Algorithms

This approach is not unique to the system of grammatical evolution. Both Whigham [1995] and Wong and Leung [1995] developed systems based on genetic programming (an adaptation of genetic algorithms that utilises trees rather than linear chromosomes) that used grammars to evolve code. [Ryan et al 1998a] Indeed, Koza reports on using grammars to check the validity of the solution trees generated from the genetic programming algorithm, due to the problems associated with invalid individuals being returned. The major difference is that the grammar is applied after the individual has been constructed, rather than constructing the individual via the grammar. When constructing an individual via the grammar, as in grammatical evolution, the only way that an invalid individual can be created is if the sequence does not terminate.

4.2.2. Benefits of using Grammars

4.2.2.1. Generality

The primary benefit is the generality offered. It has already been noted that one of the advantages of genetic algorithms is its weak yet powerful search (weak means that it requires very little problem-specific information to work, and powerful means that is efficient in finding a solution over a wide search space). While this weakness allows it to be applied even if the process being analysed is a complete mystery, more information can be used to refine the search even more. But, with genetic algorithms, if more is known about the problem, then there is no easy way to incorporate this insider information without adjusting the program code. However, this insider information can be used to modify the input grammar, and the grammatical evolution method immediately becomes stronger without sacrificing power. As O’Neill and Ryan [O’Neill and Ryan 2003] point out, “The more one knows about a problem, the more one can enrich the grammar with all manner of insider information about the problem.” Further, the output can be in any format that can be coded by a grammar. Ryan and O’Neill have demonstrated implementations that return symbolic information [Ryan et al 1998] [Ryan and O’Neill 1998] and implementations that return compilable C code [O’Neill and Ryan 1999a] [O’Neill and Ryan 1999b], and Griffith has demonstrated a version that outputs in postfix notation [Griffith 2002]. It is not for nothing that the major publication in this field is titled “Grammatical Evolution: Evolving Programs for an Arbitrary Language” [Ryan et al 1998a] [O’Neill and Ryan 2003]

4.2.2.2. Legality

It is also a property of these structures that a well-formed grammar will always output a valid individual (this discounts the case in which an infinite loop is entered into, this can be dealt with separately). This is because, instead of constructing the structure directly (as in genetic algorithms and genetic programming) the structure is built from the ground up through repeated application of a derivation sequence to the grammar – a sequence of choices dictating the grammar rules to use [O’Neill and Ryan 2003].
4.2.3. Backus-Naur Form
The style of context free grammar chosen by Ryan and O’Neill is Backus-Naur Form, usually referred to as BNF. This is a notation for expressing a language in the form of its production rules. The grammar contains a set of terminals (T), a set of non-terminals (N), a set of production rules (P) and a start symbol (S∈N)

4.2.3.1. Terminals
The terminal symbols are those that can appear in the final language, an example quoted by Ryan et al [Ryan et al 1998a] is:

\[ T := \{ \text{Sin, Cos, Tan, Log, +, -, /, *, (, )} \} \]

4.2.3.2. Non-terminals
The non-terminal symbols are those that can still be expanded, by applying another production rule, they cannot occur in the final language. The example above continues, to define:

\[ N := \{ \text{expr, op, pre_op} \} \]

4.2.3.3. Start Symbol
The start symbol indicates the non-terminal to start the selection from, for example:

\[ S := \langle \text{expr} \rangle \]

4.2.3.4. Production Rules
These are the rules that map the elements of N onto a mixture of elements from N∪T. The non-terminals are enclosed in angle braces \(<N>\), the first token in the line is the element of N to map from, and the remainder presents the choices for the mapping, delimited by |:

\[ <\text{non-terminal}> ::= \text{choice 1 | choice 2 | choice 3 | … | choice n} \]

The choices may be mixtures of terminals and non-terminals, the non-terminals being distinguished by angle braces.
Returning to the previous example, the production rules Ryan et al demonstrated are:

1. \( <\text{expr}> ::= <\text{expr}> <\text{op}> <\text{expr}> \quad (A) \)
   \( \quad | \quad ( <\text{expr}> <\text{op}> <\text{expr}> ) \quad (B) \)
   \( \quad | \quad <\text{pre-op}> ( <\text{expr}> ) \quad (C) \)
   \( \quad | \quad <\text{var}> ; \quad (D) \)

2. \( <\text{op}> ::= + \quad (A) \)
   \( \quad | \quad - \quad (B) \)
   \( \quad | \quad / \quad (C) \)
   \( \quad | \quad * ; \quad (D) \)
(3) \texttt{<pre-op> ::= + \quad (A)}
| \texttt{ -;} \quad (B)

(4) \texttt{<var> ::= X;} \quad (A)

The numbers on the left, indicate the rule numbering, and the letters on the right indicate the choices within each rule.

### 4.2.3.5. The BNF Grammar

The whole grammar may now be described by

\[ G := \{ N, T, P, S \} \]

[Ryan et al 1998a][O’Neill and Ryan 2003]

### 4.3. Mapping from the Genotype to the Phenotype

The principle advantage of grammatical evolution is the existence of a distinct phenotype that can be arbitrarily complex without impacting the underlying genotype. The process closely follows the natural system, as shown in the diagram to the right. At all stages, the two systems are analogous. The first step is the binary string that is operated on by the underlying genetic algorithm, which is interpreted codon by codon into the integer string. This is then applied to the grammar described above, to select rules, then terminals, which finally specify the programmatic candidate solution. Each transition into a rule can be treated as a single protein, on its own this transition cannot generate a trait, but when other proteins are present then physical traits can be generated. [Ryan et al 1998a]

#### Figure 4.1

![Diagram of the grammatical evolution system and a biological system](image)

### 4.4. A Genetic Algorithm as base for Grammatical Evolution

Here, we describe how to apply the theoretical framework from above into an actual process for mapping from genotype to phenotype. The base functionality in a grammatical evolution system is a genetic algorithm. However, the binary string chromosomes are used to map the derivation sequence for the grammar, rather than directly act as the candidate solutions. The chromosome is still a
binary string, and is to be divided into a variable number of codons, each containing 8 alleles of value 0 or 1. Then each codon may represent a decimal number, encoded in base 2 form. (It is unlikely that any rule may offer more than 256 different choices) [O’Neill and Ryan 1998] This is our derivation sequence, which we can regard as a queue of integers. This can then be used to direct the choices of rules from a BNF grammar, as illustrated below. However the integer could well be greater than the number of rules, thus the modulo operator is used to get the integer into the necessary range.

The codon value is reduced by the number of possible choices for the rule being considered.

Choice := Codon value (MOD Count of Choices)

4.4.1. Example
(Taken from [Ryan et al 1998a], and working through the BNF defined above)
We have the queue:

- Take the starting symbol, $S:=<\text{expr}>$, and locate the appropriate rule, rule (1)
  - Rule (1) has 4 choices, so we compute $220 \pmod{4} \equiv 0$,
  - Therefore we choose the first choice in rule (1) and replace $<\text{expr}>$:
    \[ <\text{var}> <\text{op}> <\text{expr}> \]

- Now we must examine the first non-terminal in the sequence.
  - This is $<\text{expr}>$ again, so re-examine rule (1)
  - Compute $203 \pmod{4} \equiv 3$, and therefore replace this term by the 4th choice:
    \[ <\text{var}> <\text{op}> <\text{expr}> \]

- Again, examine the first non-terminal in the sequence
  - In this case, $<\text{var}>$ corresponds to rule (4), but there is only one choice, therefore, it is not necessary to read a value from the queue
    \[ X <\text{op}> <\text{expr}> \]

- Repeat for $<\text{op}>$, the next non-terminal
  - Rule (2) has 4 choices, therefore compute $17 \pmod{4} \equiv 1$ and apply the 2nd rule
    \[ X - <\text{expr}> \]

- Repeat for $<\text{expr}>$, the next non-terminal
  - Compute $3 \pmod{4} \equiv 3$, and choose the 4th rule
    \[ X - <\text{var}> \]
• Repeat for <var>
  o Again, there is no need to read a value from the queue
    \[ x - x \]

This chromosome has generated the expression \( x-x \) as a candidate solution. There are also some unnecessary codons on the end of the chromosome, marked in red. These are called introns, as they have no function that can be deduced at this stage.

4.5. Proposed Additional Features

With the adoption of a genotype-phenotype system, grammatical evolution has opened up additional directions in which to explore in seeking to improve performance. Most of these are lifted directly from the biological model and applied in the hope that they will provide a similar benefit to the algorithm.

4.5.1. Wrapping

Due to the recursive nature of BNF, there is no guarantee that a particular structure will terminate in a timely manner, thus there is a problem of what to do if the end of the chromosome is reached before the individual terminates. One option, favoured both by O’Neill and Ryan [O’Neill and Ryan 2003] and precedent from the field of genetics [Lewin 1999 cited by O’Neill and Ryan 2003], is to return to the beginning of the sequence and reuse the values already generated. Since the values merely guide the selection of a particular rule, there is no loss of variety. This was discovered to have beneficial effects when tested; valid and fit individuals were generated that were much more compactly represented than they would otherwise be. The degeneracy inherent to the system permits this functionality.

4.5.2. Degeneracy and Neutral Mutation

The term degeneracy indicates the reversion to a previous or simpler state. The concept of degeneracy is clearly shown when we consider DNA codons mapping to the proteins that they create. There are 4 possible bases for an allele and 3 alleles to a codon, so there are 64 possible combinations to map to 20 proteins. In practice there are three different codons that map to each protein, a degenerate system that is open to neutral mutation. As previously mentioned, some mutations do not have any direct effect on the actual phenotype. If a mutation changes one codon into another that produces the exact same value (or protein), then there is no appreciable difference, and the mutation may just as well have not occurred – except that this can often have beneficial effects despite the reasoning of intuition. Since some codon combinations construct their proteins more efficiently than others, there could well be an efficiency gain as well as additional
variety available for exploration. By using the mod operator in choosing the rule to apply, grammatical evolution clearly possesses some of the characteristics of a degenerate system. Degeneracy has been observed and documented in the living cell, how much advantage this lends to an algorithmic implementation is yet to be fully investigated.

### 4.5.3. Prune operator

While the wrapping operator caters for individuals that are longer than the chromosome allows, it is also possible for individuals to terminate before the end of the chromosome – as occurred in the example above. The remainder of the chromosome is of no interest in this case, and is referred to as an intron, in reference to those sections of a DNA strand that do not have any apparent function. In actual fact, there is some evidence that introns are often used as a backup repository and to break up sections of useful material – exons – so that any recombination (crossover) is less likely to break up sequences of established co-adapted alleles – a so-called ‘selfish’ nature of genes [Ryan et al 1998a]. Ryan et al claim that introns hamper the crossover operator, since it is the function of crossover to swap useful attributes between chromosomes. If the swapped portions are formed of introns, then the benefit is negligible, as shown in Figure 4.2.

![Figure 4.2](image)

**Fig. 4.2. Crossover being hampered by introns**

[Ryan et al 1998a]

The only (obvious) positive effect is that there is an amount of genetic material being held in reserve, should further variation be required. However, it could also be viewed as wasted space that is contributing little to the operation. Thus the prune operator was proposed: so that any intron portion of an individual is stripped off and discarded. Ryan et al [Ryan et al 1998a] claim that the effect of applying this operator is that crossover becomes faster and better, dramatically so.
4.5.4. Duplication

Duplication was one of the operators suggested by Holland [Holland 1992] for the initial genetic algorithm, but which was dropped from the SGA and is rarely implemented. Ryan et al suggested its reintroduction into the grammatical evolution system, their claim is that it increases the robustness of the system, allowing functionality to survive mutation and permit parallel evolution of genes. [Ryan et al 1998a]
5. From Theory to Practice

As can be seen from the preceding discourse, the field of genetic algorithms and grammatical evolution is a very large one, with a vast scope for investigation at all levels. As a consequence, the scope of this project must be constrained by practical concerns. The first and foremost aim is to design and build a working grammatical evolution system in the programming language Python. This must be effected while keeping in mind the secondary aim – producing a clear and legible system. Because of to the vast quantity of theory behind the genetic algorithm, some techniques must be discarded due to the finite time allocated to the project. This includes the entire field of scaling and standardization of fitness levels, since it is not directly relevant or particularly helpful to the current line of enquiry. Other techniques will be discarded as a result of the investigations above. In particular the more common roulette wheel selection will be replaced by tournament selection, due to the many advantages discovered – not least the degree of protection against premature convergence, the bane of genetic algorithms. This phenomenon came up frequently in much of the source literature, referred to in multiple settings, and the only other implementation of grammatical evolution discovered (besides that of the original developers) concluded with a system that became locked in a state of premature convergence [Griffith 2002]. For this reason, as many protections as can be sensibly implemented will be added to the system. As already said, the selection mechanism will be tournament selection rather than roulette wheel selection for this reason. Also, an elitist strategy will be implemented to preserve important material for use in later generations. Indeed, this concept may be expanded upon and a full steady state system may be implemented for even more precautionary measures.
Section II:

Design and Implementation
6. Requirements

One of the problems with identifying the requirements for this project is that it is neither strictly Software Development nor strictly Experimental. As the primary task is implementing the grammatical evolution system in an alternate language (Python) there are the special requirements unique to software development that must be considered – not least of which is the consideration of the end user. On the other hand, once the primary portion has been completed, the requirements for experimentation come to the fore – including the need for clear repeatable experiments drawn from clear hypotheses. However, many of the requirements cannot be divided into one of the two sections, but are necessary for both. Also, many of the requirements for the development of the project become prerequisites to ensure that the experimental stage of the project is meaningful.

Given both of the above points of view, an amalgamated requirements discussion is presented below.

6.1. Functional Requirements

One of the most important criteria is that governing the final functionality of the system. This specifies the basic structure that is expected, as drawn from previous work in this field, as well as the expected behaviour of the finished product.

The requirement that was set when the project was initiated was that the system should be a Python implementation of Grammatical Evolution. This implies that the system should be a complete and fully functional grammatical evolution system that follows the descriptions and processes as outlined in Ryan and O’Neill’s publications [1998, 2003]. In particular, this means that the following basic points must be covered:

- The system must be based upon a genetic algorithm which operates on the individual’s genotype. This is required for comparability with any previous results, as it is the basis upon which the entire premise of the system stands.
- The genotype must be mapped onto a phenotype by means of a grammar. This is the centre of the grammatical evolution system proposed by Ryan and O’Neill.

As an additional requirement, Ryan et al [Ryan el al 1998a] proposed a number of additional operators and concepts to complement the system, namely wrapping, pruning and duplication. As a part of the experimental phase, these are to be investigated and the effect on the operation of the system will be recorded.
6.2. Language Requirements

As stated in the project brief, the task is to implement Grammatical Evolution in the language Python. However it remains to justify the choice of Python, as a case may be made for many other languages, the most prominent being C and Java. However, the main aim of this project is for clarity and simplicity. There have been implementations in both C and Java previous to this, and both are difficult to follow – even if the author intends for it to be readable. Thus we are sacrificing the execution speed, particularly of C, in favour of a simple and structured program in Python.

Python’s major advantage is its handling of variables. There are no types, and variables need not be declared before use. This is very useful because it removes a number of distractions to the reader, but does require self-discipline not to abuse the freedoms granted. Additionally, Python supports many data structures natively, such as lists, tuples, dictionaries and iterators without the complication of declaration and definition that are required in C. Over and above this, Python enforces a logical indentation scheme that clearly demonstrates the sequencing of code. Finally, Python provides strong support for object orientation – a necessary provision for a program of the structure required for this project. All of the above lead to code that is generally more legible than that of other languages, which fulfils one of the main criteria of the project.

It is notable that Python includes one unique feature that greatly simplifies the interpretation portion of grammatical evolution – access to Python’s own interpreter. This allows greater flexibility, as solutions to the problem posed may be generated in the form of Python code and executed immediately to determine its fitness sidestepping the problem of compilation or postfix execution.

6.3. User Requirements

As previously mentioned, the system’s requirements must be considered from both the point of view of the end user of the system and the point of view of someone experimenting with the system. Therefore, there are two separate users, who require different things. However, there are many similarities between these needs, as the principal need is for a complete and functional system from which to start.

One problem with formulating a requirement for an end user is that it is the author’s estimate of what a user will require. This system is not being built for anything other than a demonstration and perhaps learning tool, therefore there are no potential users to canvas for their views.

6.3.1. End User

The end user is not concerned with the inner workings of the system; all they are interested in is defining the setup of the system and problem space and then running it to gain the output. To this end, the system should be completely encapsulated and complete within itself. The user should not need to alter the
code for any reason (bugs aside) but instead should be able to modify input files that are then read by the system. This should cover the details of the problem space and allow definition the grammar to use for the mapping.

6.3.2. Researcher

Many of the researcher’s requirements are already covered by the end-user’s requirements, in particular the importance of the external file import for easy setting up of experiments. However, the researcher’s needs extend to another import that allows the parameters of the system to be modified. This allows much deeper control of the system without requiring interaction with the program code. One of the researcher’s needs is a visual summary of the performance of the algorithm, some form of plot that shows how the population has changed throughout the generations of the run. In addition to this, to back up the plot, the details of the population should be saved to a comma separated values (csv) file for further analysis and study.

6.4. Conclusion

It is normal in a project of this size to propose a formal requirements specification against which to check that the system is as expected. However, due to the nature of this project, it was judged that a formal specification was unnecessary, since the system is to be measured against the performance of another system – the original – and against a question of clarity and completeness of the code. Both of these points are addressed perfectly well in the requirements discourse above and there is little in the way of detailed requirements to necessitate a formal specification.

When setting the requirements for the project, it is as well to heed the lessons of those who have gone that way before. In this case, it must be noted that the only other attempt to re-implement this system [Griffith 2002] failed on performance grounds, specifically premature convergence. Even though Griffith followed all of the instructions provided by the developers of the system, her system still fell short of the original implementation, despite copious testing and debugging. This must be taken into consideration, as we take the requirements and create a design outline from them.
7. Design

For this problem, it would be easy to write a procedural version that optimises the running of the system and is, as a consequence, difficult to follow – even with the clarity of code that is inherent to Python. However, this would run against one of the objectives of this project, namely to demonstrate how grammatical evolution works in a practical way. It is for this reason that the object oriented paradigm is embraced, since it encapsulates functionality into logical groups that can be directly related to the theoretical background. In this way, the code is much cleaner, and the relationships between the different parts of the system are clearly defined for the inexperienced user.

Another benefit of object orientation is that, if there is a genetic algorithm written in object oriented style, then the grammatical evolution system may extend these objects via the standard inheritance mechanism. Then the package involves both a functional grammatical evolution system and a functional genetic algorithm, both written in the same style.

7.1. Structure

7.1.1. The Foundation

Before designing the structure of the system, it is necessary to consider the candidate genetic algorithms so as to determine the foundation upon which we may build. Firstly, there is an algorithm called GAS⁶, a combination genetic algorithm and genetic programming project. This has an object oriented style, but uses multiple classes in an overly complicated manner. This confuses the issue, and it is difficult to see the theory being put into practice. The next algorithm for consideration is named Galileo⁷ and exhibits a neat two-class system that uses aggregation to define a Population consisting of instances of Chromosomes. However the coding is not very compact and is non-trivial to follow. Furthermore, the system does not run itself, but has to be manipulated by commands from the calling method, adding extra complexity and moving some of the functionality out of its logical setting. GA542⁸ fills this gap with a three-class system, that also implements an Environment that works on the Population that is aggregated by instances of the Theory class. This is still rather complicated and the coding style is verbose enough to lose track while following the sequence of the system. All the systems mentioned contrast strongly against the chosen algorithm: pyGA⁹. pyGA has a simple but effective structure of a two-class system that integrates the Population into the Environment seamlessly while retaining a separate Individual class that is aggregated into the Environment class. The coding style is exceptionally concise and shows a direct relation between theory and practice.

---

⁶ GAS by Dirk Heise.
⁸ GA542 by Nathan Denny. http://www.ece.arizona.edu/~denny/projects/GA542/
⁹ pyGA: author unknown
7.1.2. The Extension

As noted above, imposing the object oriented paradigm upon the task at hand causes us to divide the problem into collections of functionality linked via both inheritance and aggregation. Coupling this with the structure of the chosen genetic algorithm – pyGA – we can design the structure to which the grammatical evolution system must adhere. The core of the genetic algorithm is formed of two classes: an Individual and an Environment. The Individual class encapsulates all of the methods acting upon a single individual, while the Environment class contains the collection of individuals, and deals with the tasks associated with moving from one generation to the next. In this way the Individual class is aggregated into the Environment, with the Environment maintaining a population list containing instances of Individuals. From this basis, the grammatical evolution system extends the genetic algorithm via inheritance. The GE module itself consists of the counterparts to the two classes in the genetic algorithm. Both of the GE classes extend the corresponding class in the genetic algorithm, overriding the evaluation function of the individual class and adding the additional features as defined by Ryan et al [1998a]. Figure 7.1 illustrates this inheritance and aggregation process via a UML representation. The combination of inheritance and aggregation allows the system’s functionality to be divided into separate classes, but yet be very meaningful – in fact more so than if a completely procedural version has been written.
7.2. Data Import

One requirement of the end user is for them to be able to define the problem space and specify how to compare the generated result. The researcher also requires that the parameters of the system can also be imported from an external file. In considering the differing file formats available, there were a number that were practical for use.

The first format would be a simple text based system that had successive settings on successive lines. However, there would be no indication within the file what each line would mean; so although it would be among the easiest to read and write the settings into, it would have little meaning on its own. It would be easy to make an error but not so easy to discover it. Overall, it is not the best format to use, with the exception of a BNF encoded text file for the grammar. This is because BNF is defined in textual terms – and thus this format is natural for representing BNF files. The second format would be a derivative of the text based approach, and would be the easiest to import from. This would use the Python tools pickle and unpickle that could take any Python object or code and store it in a retrievable way. However this would be even more difficult for the user to edit than the text file, so is not practical in this instance. The third format would take the form of an INI file, with sections and defined names of settings. This would be more structured than the text based format, but is still not completely optimal. There is still nothing preventing the file being edited and losing some of the necessary structure. However, it is much more descriptive than the previous formats – someone opening the file would know what each setting did. However, none of the previous formats supported validation in any way. This would have to be performed at parsing time by the system. Fortunately, there is another form that is now a leading standard for data storage – XML. Not only is it self describing and easy to read and edit (as are INI files), but the data can also be structured arbitrarily and contain metadata about how to handle the data. On top of this, it supports validation, with both DTD and Schema validation, causing this format to be much more robust. This format is the one chosen for use by this system, because its qualities overwhelm the other contenders, and because the input files are readable in any number of XML tools that can allow the non-technical user to setup the inputs without the need to understand XML syntax. Furthermore, there is a strong link between Python and XML, and XML parser support is native to Python. As the parser is integral to Python, much less code is needed to perform the same task as it would if a custom parser had to be written.

In the consideration of the text file above, a caveat was made regarding BNF files and their affinity for text format. However, XML as a format is particularly appropriate for the representation of a BNF grammar since it has a very clear structure. However, this cannot handle the ‘standard’ text BNF files, which many users will have pre-written. Because of this, it would be best to permit use of either format to accommodate the needs of all users, with the XML form as the preferred format.
8. Implementation

The first major step in producing a grammatical evolution system, producing a genetic algorithm, was accomplished by reusing a previous program, pyGA. This program fulfilled the object oriented outlook and clarity sought, and was a masterpiece of Python programming – providing much inspiration from its elegant style. As was discussed in the Design chapter, there were numerous other Python implementations of a genetic algorithm, but no other algorithm approached the level of pyGA. Unfortunately the author of this algorithm could not be traced, and thus cannot be credited. In the design phase, the inheritance model discussed appears too simple to be true, but pyGA’s structure allowed implementation in exactly the same way as described – the only alteration needed for a bare-bones system is the overriding of the evaluate function within the individual class. However, for the more extensive modifications, many more functions have been coded or over-ridden.

8.1. Issues of Clarity

As previously stated, one of the aims is to produce a grammatical evolution system that is clear and easy to follow. For this reason, many of the functions have been split up into their constituent parts for easy testing and logical separation. A prime example is that of the evaluate function; one of the more complex functions, since it has to decode the chromosome into the genotype, map the genotype onto the phenotype and evaluate the phenotype to determine the score. These three operations were better understood by separating them out.

Another issue is the approach to looping in the mapping phase of the evaluation. It is traditional for a loop of an unknown but bounded length to be implemented via a while loop, however it was more natural while contemplating the process to implement it as a for loop over a list the same length as the bound, utilising a break to indicate early termination. This construct, while not generally advised, does demonstrate the theoretical background being implemented. In writing the loop in this manner, there was the added advantage that Python is optimised for running for loops in preference to while loops. Since this is a function that is run for each of the 500 individuals at every generational step, there is a small efficiency gain.

8.2. Trade-offs

One of the advantages of tournament selection, as mentioned in the background summary, was the efficiency gain of only computing the fitnesses of the individuals chosen for the tournament. However, in practice, the use of elitist strategies and the developmental need to produce entire population logs meant that the fitness of each individual must be calculated at each generational step anyway. However, there was still sufficient incentive to utilise tournament selection (not least because the genetic algorithm came with it pre-written) as it
still allowed control of the selection pressure and worked to avoid premature convergence.

In the evaluation stage of an individual, there is a loop that takes the variable (as defined in the XML file) and loops though all the defined values for that variable. The phenotype is designed to contain this variable within it, by the grammar, and so the exec() statement of Python is called to execute a string of text as if it were a Python command. The variable is set via this command, and then the phenotype is executed via another call to exec. This allows the system to pick up the value (i.e. X=1) and evaluate the expression at that point. This works fine for one variable; however it fails to cope with the multivariate case. There is some support in the XML file and parser for inputs with multiple variables defined, but it was difficult to visualise a way in which to process an arbitrary number of variables. This is because it was not possible to encode an arbitrary number of nested for loops – one for each variable – and there was no obvious way of evaluating for all combinations of variable values. This was a minor issue since all of the examples of problems use at most one variable and it was judged that this feature was not crucial to the success of the system. Nonetheless it is an area in which further development could be effective, to widen the range of problem spaces the system can work on.

8.3. BNF Grammar Import

In continuation from the discussion of the format of the input files from the Design chapter, the system’s provision for the import of a BNF grammar merits further consideration.

8.3.1. Text Based BNF

In all the reference materials considered, the BNF grammar used was defined in a plain text format, with fixed structure – which contained minor differences depending upon the source of the grammar. Thus it was important to be able to parse and load a supplied .bnf file, even though text based systems were judged too primitive to be chosen for the main settings. However, at most the user would have to make minor adjustments to their existing grammars to comply with the encoding scheme chosen. On the other hand, to parse these files required a custom parser to be written especially for this project by the author. This parser was designed to read and error check the grammar while building an object oriented structure (via aggregation) that allowed the system to access the grammar. This parser is logical and as clear as possible, however there is much detail that the reader need not be aware of, and which obscures the essential points. In this way, a pre-written parser (like pyParser, SPARK or PyLR), could have been clearer, but it was not initially obvious how to apply these packages to the desired end. One improvement would be to return and implement one of these packages, and bring this code up to the clarity standards aimed for.
8.3.2. XML BNF

In the Design chapter, XML was judged to be the best choice for importing data, particularly structured data; due to its validation, structure, self-description and wide acceptance. Thus it was decided to allow the user the choice of XML or text format, since XML seemed ideally suited to the representation of BNF. The primary benefit is that the user may define the grammar in a clearly structured way; defining the rules to apply, then defining the available options within each rule and then specifying the sequence of terms that make up the body of an option. A secondary benefit is that it also allows the file to be validated against a schema and reject any invalid grammars before the parser begins to build the BNF grammar object. This removes any such functionality from the importing module and thereby removes much code that has very little value to the student of grammatical evolution. One problem with this approach is that no XML schema parser for Python 2.4 could be found, so this functionality is not yet implemented – but is a simple task once a validator is found. The existing tool, pyLTXML, failed to work with Python 2.4, but if this functionality is needed then the system may be run under Python 2.3 and modified to load pyLTXML.

The benefits of using an existing parser to a custom coded one is obvious by comparing the two modules that handle the different file types – the XML module is one third the size of the text module and much easier to follow. Although there is no tool for translation from the text form into the XML form, an XSLT transform was created for the XML file, which allows a user to view it as if it was a text BNF file and ensure that it is truly as desired. This fixes the one problem with BNF XML files, the difficulty of visualising the grammar compactly. Also, this prevents the user being locked into a custom file format: any user that writes their grammar in XML may extract it in text form for use with any other system. Examples of the files written for the project may be seen in the appendix.

8.4. XML Parameters and Inputs

Due to this structured nature of XML, the implementation of the importing of settings is almost trivial. The imports were divided up into separate files, according to functionality. The BNF grammar is one entity, the grammatical evolution parameters another and the input data for the comparison process is the other. In addition, there is a single XML file with a set file name that tells the system where to find the other files.

Although the groups of settings have been split into three files, there are no overlapping tags between them (by design). This makes it is possible for all three to be combined into a single file if a researcher wishes to keep all of the setup for an experiment bound together.

8.5. Logging

Due to this project being part experimental, the researcher’s needs had to be considered in addition to those of the end user. One of the prominent
requirements was to incorporate multiple levels of logging into the system. The backbone of the system was the use of Python’s inbuilt logging module, which was spread throughout the project. For this to have complete coverage, the genetic algorithm, pyGA, was modified to output every important stage to the log file. The implementation that is available in Python supports five levels of logging in increasing priority: DEBUG, INFO, WARNING, ERROR, CRITICAL. Thus, the priority attached to each logging entry allowed appropriate handling – WARNING and up indicate that there has been an error within the program (CRITICAL indicates that it is not recoverable) and are echoed to the user via the console. Everything is logged to a log file with priority DEBUG and up, however large runs (i.e. 20 iterations of the system) build large log files, so it is possible change it to record from INFO and up instead. In general, the categories were utilised to indicate the severity of the error; in particular: CRITICAL messages are sent just before the program exits, to explain the reason for this and to allow the user to correct the problem. WARNING messages indicate that there was a fault, often when the generated expression is unfit (e.g. divide by zero), where it can be fixed without exiting. INFO maintains a trace on what operations are being performed upon which individuals, while DEBUG prints extended information about those operations – for example the before and after chromosomes of a mutated individual. In addition to the use of Python’s logger, there was also provision made for the population’s data to be recorded at every step. During the report phase, two files were written, one recording the phenotype and fitness of every individual in to population at this generational step and one recording the best only. Both were written out to a Comma Separated Value (.csv) file for loading into a spreadsheet program to examine the trends and visually check the progress of the algorithm.

Further, there is a mechanism by which statistics are passed to the module that initiates the grammatical evolution system. For the benefit of the researcher, this module uses the additional packages NumPy\textsuperscript{10} and pyLab\textsuperscript{11} to organise and summarise that data, and then presents the user with a plot of the best fitness individual and a plot of the average fitness of the population at each generation. In this way the researcher can easily see any trends that occur without having to analyse the output files every time.

8.6. Optimisation

Since Python is an interpreted language, it is necessarily slower than the corresponding system written in a language such a C. However there exists a Just In Time compiler for Python, called Psyco\textsuperscript{12}. This optimises sections of the code at run time, and claims to increase performance of executed code between 2x and 100x. This is enabled in the system, since it works best on algorithmic code that is frequently run – such as the contents of the evaluate function. The cost is a high memory overhead. In practice the use of Psyco is disabled by default, since

\textsuperscript{10} http://sourceforge.net/projects/numpy/
\textsuperscript{11} http://sourceforge.net/projects/matplotlib - note that this package also requires Numeric (http://numeric.scipy.org/)
\textsuperscript{12} http://sourceforge.net/projects/psyco/
it takes too many resources. In one test running on the main development machine\textsuperscript{13}, the system slowed down considerably since 80\% of the resources were devoted to the Psyco process. This lead to the Python interpreter being starved of resources, which caused the system slow significantly, thus negating any efficiency gain. The failure of this optimisation technique is not a serious problem, since the original code runs fairly fast when run upon a ‘decently’ specified system\textsuperscript{14}. In addition, the code itself has been optimised by the addition of cut outs that prevent duplication of work, such as not re-evaluating an individual that has already been evaluated once.

8.7. Problems

When the system is run, the results show a tendency to plateau out at a fairly fit level. There is obviously something wrong, since the system is built in accordance with all the information available – yet does not yield results comparable with the claims of O’Neill and Ryan [O’Neill and Ryan 2003].

![Combined Plot of Best Score for Ten Runs on Regression](image)

**Figure 8.1** Combined Plot of Best Score for Ten Runs on Regression

Close study of the log files that were output by the system show that fairly fit individuals are hijacking the population as the system progresses. This is a classic symptom of premature convergence, one of the problems that have often plagued

\textsuperscript{13} Machine Specification: Pentium 4 3GHz Hyper threaded, 1Gb of RAM, Windows Server 2003. The system runs with the settings from the Testing chapter and takes about one minute to do so.

\textsuperscript{14} Recommended specification is similar to that of the test machine above, but the system has been run successfully and efficiently on a lower specification machine: Pentium M 1.6GHz, 512Mb of RAM, Windows XP.
researchers working within the field of genetic algorithms. As can be seen in the background research section, this is a phenomenon that crops up in many studies and has a large number of techniques devoted to mitigating the damage caused to the population. A number of the suggested techniques were implemented in this system in the hope that they would allow the population to grow and improve. Notable among the measures taken are the use of an elitist strategy and the inclusion of tournament selection, which spreads the selection over a larger proportion of the population. In fact, the elitist strategy was further upgraded to a steady state approach as suggested by De Jong [De Jong 1975, cited by Whitley 1993] and recommended by the developers of the grammatical evolution system [O’Neill and Ryan 1998]. Since so many precautions have been made against premature convergence, it may not be premature convergence itself that is the root cause of the problem. There could be an underlying issue with the improvement of the population, which could lead to the same symptoms. One possibility is that the improvement mechanism is not performing as well as it should be. Indeed, looking at the problem in a theoretical way, there appears to be a problem with the crossover operator’s interaction with the process of mapping the genotype to the phenotype; specifically the lack of intelligence as regards the grammatical nature of the phenotype. For these reasons, the crossover operator will be investigated in further detail in the Experimentation chapter. However, it may be noted that the fitness of the plateau is fairly high, even though none of the runs self-improve to meet the 0.01 level specified by O’Neill and Ryan [O’Neill and Ryan 2003]. The population jumps to a fairly high level and stays there. Therefore it is also possible that it is difficult for the population to improve itself beyond this point and that crossover is ineffective for this reason rather than there being a fundamental problem.

8.8. Dynamically Extending the System

Another problem occurred during implementation of one of the primary tests in the field of evolutionary programs: the Santa Fe Ant Trail. While attempting to encode this example, Python’s syntax indentation did not allow direct comparability with the original C implementation, as the nested if statements could not be translated directly into Python. In C, the program could enclose the line in a pair of braces, but Python required the correct number of tab characters at the start of every line. The solution to this problem showcases the most powerful argument in favour of using Python for grammatical evolution – its dynamic execution. There is a tag in the input XML file that allows the user to import any Python modules that they choose, originally intended to allow access to Python modules such as decimal, math and string. However, it was possible to write a module containing a class implementation of an ant and import this module at evaluation time. The getScore function then passes the phenotype to an instance of the ant class and may recover the score by interacting with the instantiated object. This can be done for any problem – some external Python code modules used to determine the score of the phenotype, however the user desires. Even if the user chooses not to write an external module, they can enter a Python statement as the evaluation function, and it will be executed in the same way that any other statement in the system is executed.
Section III:

Testing and Experimentation
9. Testing

One of the most important steps is the process of testing the system to ensure that it is as described. The requirement for the end user was that the system should be fully functional, and it is testing that allows us to verify this. Further, a researcher may well wish to modify the code of the system at a later time, and a clearly defined framework will allow them a structured way to ensure that any modifications fulfil the set requirements.

One significant problem was the fact that much of the system is highly interdependent, and so the only true test was to run the system and study the results. This caused some difficulty, not least because the system required a lot of time to run through ten or twenty iterations and the log files sometimes reached sizes measured in gigabytes. The results of these combined tests were not favourable, as the system did not work as expected. The results were not as was claimed for the original system, and it was very difficult to determine the cause of the problem. After some analysis of the populations change over time it appeared that a state of premature convergence was the problem, apparently caused by the ineffectiveness of the crossover operator, discussed in the next section.

However, in accordance with the simplicity and clarity aims of the project, all code was separated out into logical groups. For example, the evaluate function was long and complicated, but consisted of three distinct parts that were split out into their own functions. This allowed component testing of each individual part, to ensure that each part of the whole is working. This does not entirely relieve the problem of testing the whole, but does help to ease the strain.

For testing of this sort, Python provides a package known as unittest, a package based upon JUnit for Java. This allows the definition of a suite of tests that can be run in sequence. Each test contains a number of assertions that are evaluated, any errors being reported as failures. This allowed structured component testing of the system to occur. Each important function was tested on its own, with all of the appropriate preliminary setup performed programmatically. The test cases checked normal usage, as well as the performance on the boundary points, such as at the maximum of the range. These tests are sequential, with the BNF grammar portion being checked before the grammatical evolution portion, and then a pre-constructed BNF grammar is provided upon which evaluation is tested.

This structured test proved effective during the construction of the system, as it could be run at every stage, to ensure that no component had been inadvertently damaged. This greatly shortened the development process, as the test framework could be run immediately after an alteration, to verify that all was well.

For the test plan summarising these tests, see Appendix A.
9.1. Full System Test – Symbolic Regression

As is noted above, the main test is a ‘big-bang’ integration test that examines the system as a whole. We present here the results referred to in the above section. The system was run with the set up as detailed in the tableau. The test was run 25 times due to the probabilistic nature of the system.

9.1.1. Experiment Tableau

Experiment: Symbolic Regression  
Terminals: X, 1.0  
Fitness Case: [-1, 1] step 0.1  
Raw Fitness: Average of absolute difference between target and solution (with penalty difference 10 given when there is an error evaluating a data point and penal value 100 for completely invalid individuals)  
Crossover: 0.9  
Mutation: 0.01  
Steady State: On

9.1.2. Results

As can be seen, each individual run reached and remained at a plateau. This state is indicative of either premature convergence or a failure in the self-improvement operator. This is discussed in more detail in the next chapter. However, there does appear a slight tendency towards improvement occurring on the population as a whole during the early generations. Unfit individuals are being removed via a variation on the elitist strategy – the randomisation of invalid individuals.
9.2. Full System Test – Santa Fe Ant Trail

Although the Symbolic Regression problem was the main one considered, it would be foolish to rely on merely one problem to test the system and to make general statements regarding the performance issues. Since the Santa Fe ant trail is a well known problem in the field of evolutionary programming, it was decided to implement this in addition to the symbolic regression problem. This problem is of a fundamentally different type to the symbolic regression problem. Not only does it not check a range of variable values to determine the score, but the score is determined via passing the phenotype to an external module for processing rather than evaluated as a Python code segment in and of itself. The experiment tableau and testing strategy are the same as detailed in the preceding section, only for the Santa Fe ant trail problem.

![Graph of 25 Runs of GE on Santa Fe Ant Trail](image)

Figure 9.2 Graph of 25 Runs of GE on Santa Fe Ant Trail

The result of applying the system to this problem can be seen in Figure 9.2. It is important to note that this problem aims to maximise the score, rather than minimise it as with symbolic regression, therefore the desired trends are reversed. As can be easily seen, the pattern of results is very similar to those for the symbolic regression. The main difference is that the population as a whole (bottom) improves slightly better for this problem.

9.3. Conclusion

It can be seen that both examples show distinct signs of premature convergence that may or may not have been caused by the failure of the self-improvement operator – crossover. Other than this, the system is complete and correct, at least in a logical sense. The issues regarding self-improvement will be addressed in the following chapter.
10. Experimental Work and Additional Operators

10.1. Crossover and Self-Improvement

When the Grammatical Evolution system was run, there was a clear tendency for the results to reach and remain at a plateau. After examining the evolution of each population, it became apparent that the self-improvement effect of the crossover operator was not working as it should.

One possible reason for this is that the nature of the crossover operator appears to interact poorly with the mapping onto a grammar that grammatical evolution is based on. In a grammatical evolution system, the first element of the phenotype is chosen by mapping the first codon of the genotype onto the grammar in a repeatable way (modulo the number of choices for the <start> element). The first non-terminal element of this phenotype is used to pick the rule to apply next, and the next codon of the genotype is used to pick the option to apply from within this rule. Thus the encoding of the phenotype is directly related to the sequencing of the genotype, as well as its codon values.

From this we can observe that the effect of a particular codon $G[i]$ of the genotype $G$ is dependent upon $G[0:i]$ – the sequence of codons that determine the rule upon which $G[i]$ gets to choose between options. Hence, there is no specific meaning attached to a codon at any particular place within the body of the genotype, and a codon loses its former meaning if it is swapped into another genotype with a different sequence $G_2[0:i]$ before it. This is in contrast to the situation within a
genetic algorithm, where there is a direct positional mapping on the genotype and crossover allows the system to enter a cycle of self-improvement. This was discussed in the background research as the additive nature of genes.

10.1.1. Two Point Crossover
The default crossover operator in the original genetic algorithm was the two point crossover. For the reasons detailed above, it is suggested that this crossover fails to move the population forward. In particular, its lack of intelligence is a concern.

10.1.1.1. Hypothesis
Two point crossover is not the most effective improvement operator and does not help to provide variety in the population.

10.1.2. One Point Crossover
For the same reasons as for the two point crossover, one point crossover is judged ineffective when applied to the system. Similar to the two point crossover, it does not understand the genotype-phenotype distinction and thereby cannot combine logical blocks of code as was the original aim.

10.1.2.1. Hypothesis
One point crossover is not the most effective improvement operator and does not help to provide variety in the population.

10.1.3. One Point Variable Length Crossover
There is little to distinguish this from the previous operator except for the fact that the length of segment taken from each individual is taken from a different point on each. Since there is no stricture on length of individual, this may be more appropriate than one point crossover, but still does not address any of the fundamental concerns.

10.1.3.1. Hypothesis
One point variable length crossover is not the most effective improvement operator and does not help to provide variety in the population.

10.1.4. Homologous Crossover
The above crossover types work purely on the chromosome of the individual, without respect for the mapping process that is integral to grammatical evolution. Therefore, they are unintelligent and cannot preserve the meaning of any blocks of code. This has already been noted in the field of genetic programming, where “the benefit or otherwise of crossover in genetic programming is often disputed” [O’Neill et al 2003]. To combat this, an improvement to crossover, named homologous crossover, has been proposed [O’Neill et al 2003]. This is a new form of crossover that respects the mapping process; it uses information from the mapping process to determine a valid point from which to crossover the
individual. During mapping, a history of the rule choices taken is recorded, and the homologous crossover places the two histories of rules side by side. These histories are read sequentially from the start, while the rules chosen are identical for each history. This section at the beginning of the lists, where both individuals chose the same rule, is termed the ‘region of similarity’. The left point of crossover is set to the end of this region. The right point of crossover is selected at random from the region of dissimilarity of each individual, and a two point crossover is performed. In this way, the meaning of the initial section is not lost.

10.1.4.1. Concerns

Although this operator appears to be intelligent and able to improve the phenotype in addition to the genotype, investigation shows that it appears to just be an illusion. After close examination, this crossover method is shown to be merely another way of swapping the tails of two individuals. To see this, consider two individuals that chose the following rules to apply: (** indicates unused intron codon)

<table>
<thead>
<tr>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
<th>#6</th>
<th>#7</th>
<th>#8</th>
<th>#9</th>
<th>#10</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;expr&gt;</td>
<td>&lt;expr&gt;</td>
<td>&lt;var&gt;</td>
<td>&lt;op&gt;</td>
<td>&lt;expr&gt;</td>
<td>&lt;var&gt;</td>
<td>***</td>
<td>***</td>
<td>***</td>
<td></td>
</tr>
<tr>
<td>&lt;expr&gt;</td>
<td>&lt;expr&gt;</td>
<td>&lt;var&gt;</td>
<td>&lt;op&gt;</td>
<td>&lt;var&gt;</td>
<td>&lt;op&gt;</td>
<td>&lt;expr&gt;</td>
<td>&lt;var&gt;</td>
<td>***</td>
<td>***</td>
</tr>
</tbody>
</table>

The region of similarity ends after codon 3. Let us suppose that the right hand point chosen is after codon 7, then we get the children below:

<table>
<thead>
<tr>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
<th>#6</th>
<th>#7</th>
<th>#8</th>
<th>#9</th>
<th>#10</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;expr&gt;</td>
<td>&lt;expr&gt;</td>
<td>&lt;var&gt;</td>
<td>&lt;op&gt;</td>
<td>&lt;var&gt;</td>
<td>&lt;expr&gt;</td>
<td>&lt;expr&gt;</td>
<td>&lt;var&gt;</td>
<td>***</td>
<td>***</td>
</tr>
<tr>
<td>&lt;expr&gt;</td>
<td>&lt;expr&gt;</td>
<td>&lt;var&gt;</td>
<td>&lt;op&gt;</td>
<td>&lt;expr&gt;</td>
<td>&lt;var&gt;</td>
<td>***</td>
<td>***</td>
<td>***</td>
<td>***</td>
</tr>
</tbody>
</table>

Codons 8 and 9 of the second parent are no longer used, and codons 8 to 12 of the first parent have acquired new meanings (that require the actual genotype to determine). However, note that if a one point crossover was performed at codon 7 then we would achieve the same result. This is because of the dependant and unique nature of the mapping that this method attempts to exploit. A particular codon applied to a particular rule will always return the same choice, by construction. Therefore, performing a two point crossover between codons 4 and 7 changes nothing when we consider the first seven codons of each individual. The first child becomes the head of the second individual joined to the tail of the first individual, and visa versa for the second child. For this reason, homologous crossover cannot be ranked significantly above one or two point crossover. It is, however, slightly better than both of these in that one end of the crossover is always in the exon region\(^\text{15}\) and so the crossover can never be completely within the intron portion and fail to change either individual.

10.1.4.2. Hypothesis

Homologous crossover is not the most effective improvement operator and does not help to improve the population in grammatical evolution as well as the

\(^{15}\) Except if the individuals are identical in their exon regions, in which case homologous crossover will have no effect
alternatives below. It is however marginally more effective than one or two point crossover.

**10.1.5. Homologous Transpose Crossover**

Even though the homologous crossover is not as intelligent as was desired, it does prompt further development. When considering the theoretical reasoning behind the homologous crossover, question prompted was: why not swap complete blocks of functionality? Since we keep a list of the rules upon which the codon acts, it should be possible to select a range by considering the first and last rule in the region. Then, if another range is picked from the second individual with the same start and end points but possibly different length, these two ranges could be swapped over without losing the meaning of the portion of the phenotype encoded in the process. The rule choices will not change because of the nature of the mapping that is mentioned above.

To illustrate this, consider the following example: (<> is an unused codon)

<table>
<thead>
<tr>
<th>Parents</th>
<th>Children</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>&lt;expr&gt;</td>
<td>&lt;expr&gt;</td>
</tr>
<tr>
<td>&lt;expr&gt;</td>
<td>&lt;expr&gt;</td>
</tr>
<tr>
<td>&lt;expr&gt;</td>
<td>&lt;expr&gt;</td>
</tr>
<tr>
<td>&lt;expr&gt;</td>
<td>&lt;expr&gt;</td>
</tr>
<tr>
<td>&lt;expr&gt;</td>
<td>&lt;expr&gt;</td>
</tr>
<tr>
<td>&lt;expr&gt;</td>
<td>&lt;expr&gt;</td>
</tr>
<tr>
<td>&lt;var&gt;</td>
<td>&lt;expr&gt;</td>
</tr>
<tr>
<td>&lt;op&gt;</td>
<td>&lt;expr&gt;</td>
</tr>
<tr>
<td>&lt;expr&gt;</td>
<td>&lt;expr&gt;</td>
</tr>
<tr>
<td>&lt;expr&gt;</td>
<td>&lt;var&gt;</td>
</tr>
<tr>
<td>&lt;var&gt;</td>
<td>&lt;op&gt;</td>
</tr>
<tr>
<td>&lt;op&gt;</td>
<td>&lt;expr&gt;</td>
</tr>
<tr>
<td>&lt;expr&gt;</td>
<td>&lt;var&gt;</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>&lt;var&gt;</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>&lt;op&gt;</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>&lt;expr&gt;</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>&lt;var&gt;</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>&lt;&gt;</td>
</tr>
</tbody>
</table>
If we consider the phenotypes of the individuals, we have:

**Parents:**

A: \[-(\neg(\neg(\neg(\neg(X))))/(1.0+(-X)))\]

B: \[-(\neg(\neg(\neg(\neg(X)))+(1.0)*(-X)))*(-X))\]

**Children:**

C: \[-(\neg(\neg(\neg(X)))*(-1.0))\]

D: \[-(\neg(\neg(\neg(X))+(1.0)/(1.0+(-X)))*(-X))\]

As can be seen, this crossover has affected the phenotype, without the complication of actually operating on the phenotype. This is in the spirit of the grammatical evolution system, which was developed to retain the distinction between genotype and phenotype. As discussed before, the swapped sections still have their original interpretation. This is comparable to the two point crossover in a genetic algorithm, swapping a meaningful block of code between individuals.

**10.1.5.1. Hypothesis**

Homologous transpose crossover is a better improvement operator that the preceding crossover operators. It provides the population with variety as it moves from generation to generation.

**10.1.6. Variable Mutation Rate**

Not all improvement operators need to be via a crossover operator. Since, in many ways, crossover appears not to be effective; it is natural to turn to mutation to fill the gap. This is similar in focus to a system that was developed at about the same time that genetic algorithms were developed – Evolution Strategies [Rechenberg 1973, 1984 referenced by Mitchell 1996]. One simple technique that will be implemented is to adapt the mutation rate to suit the progression of the algorithm. In the early stages, much exploratory action is necessary, so the mutation rate is high. The longer the system runs, the lower the mutation rate drops. For this, an inverse exponential is a useful function to govern the mutation rate. The exponential will be scaled: mutation rate \( m(g) = e^{-\frac{g}{10}} \), at generation \( g \).

**10.1.6.1. Hypothesis**

Variable mutation rates will help to provide variety to the population better that the original one and two point crossovers will.

**10.1.7. Combined Experiment**

The five different improvement operators detailed above will be run as a single experiment. For each operator, the system will set up a grammatical evolution environment according to the tableau below, run it and store the fitness of the best individual and the average fitness of the population at each generation step. Due to the influence of chance in this problem, this will be repeated ten times and the average over the ten runs will be recorded. Due to the distorting affect of individuals of the form \( \frac{1}{x} \) which can have fitness values in the region of \( 1\times10^{15} \), these are removed from the data before averaging.
10.1.7.1. Experiment Tableau

Experiment: Symbolic Regression  
Terminals: X, 1.0  
Fitness Case: [-1, 1] step 0.1  
Raw Fitness: Average of absolute difference between target and solution (with penalty difference 10 given when there is an error evaluating a data point and penal value 100 for a completely invalid individuals)  
Crossover: 0.9 (0 under variable mutation)  
Mutation: 0.01 (or variable under variable mutation test)

10.1.7.2. Results

As can be seen from Figure 10.1, the performance of the operators is somewhat reassuring, as they all tend, if only slightly, towards self improvement. Overall, though, all of them still appear to be partially ineffective, or be beset by premature convergence. From the top graph, it can be seen that the homologous transpose and, surprisingly, the two point crossover are the best. However, looking at the bottom graph shows that they are all demonstrating a similar tendency, with the ordering different from that in the top graph. In both cases, the fitnesses quoted are all sub-optimal and very close to each other. There appears to be no
predominantly better method, not even the homologous transpose which is claimed to be far more intelligent than the others.

With regard to the hypotheses, there is evidence supporting the statement that the operator is not the best, with the exception of the two point crossover where there is evidence to reject the hypothesis. It is interesting to note that one point crossover is significantly better than one point variable length, and so there is evidence to reject the hypothesis of the one point crossover. It is very prominent that the findings are primarily negative; none of the measures detailed in the preceding sections have helped to improve the performance of the system. Were it not for the example of grammatical evolution that Ryan and O’Neill have exhibited, we would be tempted to conclude that there is some fundamental flaw in the process of self-improvement in the system. There may be a flaw in this Python implementation, but close scrutiny has not revealed any inconsistencies, and the results are similar to the results of others implementing this system. The major difference from a genetic algorithm is that there is a mapping involved, which complicated the process. Considering the results of the experiment, there is evidence to support the view that crossover (and variable mutation) interacts poorly with the distinction between genotype and the phenotype generated from it.

10.1.8. Results Distribution

Delving deeper into the actual effect of the crossover operators, it was decided to perform a number of crossovers between two valid individuals and compare the fittest parent against the fittest child so as to examine what is happening. To this end, the test will be repeated for 1,000 iterations with a variety of crossover operators. There are no hypotheses for these tests, since we have only the simple wish to find out more about what is occurring, not having any understanding at the moment.
10.1.8.1. One Point Crossover

If we examine Figure 10.2, we can see that there is a clearly defined diagonal, where the fitness of the parents and the fitness of the children are identical. This echoes the point made in the background research, on the concern regarding crossover in the intron segment of the chromosome. This has absolutely no effect and means that the crossover is failing to operate. In fact, this diagonal contains approximately 90% of the individual points on the plot, an extremely significant quantity. This is demonstrated by the bar chart of the number of individuals that have a specific fitness difference to their parents (Figure 10.3). This provides strong evidence for the need to prune, and is used as motivation to implement that operator. There is evidence to reject the hypotheses for the homologous transpose and variable mutation operators, since neither of them stand out as noticeably better than any other.

![Figure 10.3](chart.png)

**Figure 10.3** Chart of Change in Fitness for One Point (note columns 0 and 1 are off the scale, as indicated)

10.1.8.2. Homologous Crossover

Despite the flaw in the reasoning of this operator that causes it to be no more than the swapping of the tails, it does have one advantage. Unlike the one point crossover above, one end of the crossover is always within an exon section, with the other usually in an intron section. This allows it to open up individuals better, and completely avoids the problem of crossing over in the intron section and effectively doing nothing.
As can be seen in Figure 10.4, the picture has dramatically changed, despite the apparent similarities between one point crossover and homologous crossover. There is still a diagonal with no change in fitness, although it is less obvious on the plot. 45% of the individuals inhabit this diagonal; a significant number, but still a decrease of half from the previous operator!

![Figure 10.4 Plot of Fittest Parent vs Fittest Child for Homologous](image)

![Figure 10.5 Chart of Change in Fitness for Homologous](image)
When Figure 10.4 is examined, a certain regular shape may be remarked upon. The individuals appear to resolve into bands of the same fitness – this appears to be a side effect of degeneracy. As was stated in the introduction to grammatical evolution, one of the features was the degenerate mapping. This allows different genes to map to the same rule, since the modulo operator is used. Further to the degeneracy of the mapping, the grammar’s solutions are also degenerate in evaluation. The individuals $(X)+(X)^*(X)$ and $(((X)+((1.0)+X)+(- (1.0)))^*(1.0))^*(X)$ are algebraically the same expression, but are generated by two totally different genotypes. This leads to the situation displayed in Figure 10.4. It is heartening to note that the distribution of the individuals is improving, as they spread out (slightly) more. This gives more scope for the selection operators to improve the population.

10.1.9. Homologous Transpose

It is notable that the distribution of points is very similar to the distribution of points for the homologous crossover. To this we can ascribe the same reasons as we put forward in the previous section, as the operators work in very similar ways. It is in many ways remarkable that these operators exhibit similar results, even though they take different approaches overall. Homologous transpose crossover is an extension of the thought behind homologous crossover, yet they are not alike in function.

If all of these crossover operators are failing, it must point to some systematic problem – whether to do with the system or the implementation. There is insufficient time to explore this as deeply as it merits, particularly as we have restricted our study to a small sample of possible operators. There are whole
fields of theory available for implementation if a deep study were to be made. One possible starting point would be to consider the mappings as the tree objects that they are and examine the effect of swapping sub trees and ripple effects. These techniques are outside the scope of this investigation, but do show promise.

10.2. Prune

Since the mapping of the phenotype does not always use all of the genes in its chromosome, there is often a non-coding section at the end of the chromosome. It has been noted, particularly in symbolic regression, that frequently a crossover operation merely produces another copy of the parents. This can be seen if we consider the sequences \(<\text{start}> \rightarrow \text{<expr>} \rightarrow \text{<var}> \rightarrow \text{X}\) and \(<\text{start}> \rightarrow \text{<expr>} \rightarrow \text{<var}> \rightarrow 1.0\). Both of these use two genes out of the 50 available (250 if 5 times wrapping is used). Therefore, any crossover between these is extremely unlikely to result in a new individual.

This situation can be particularly seen in the plot of the one point crossover test. 90% of points were situated on the diagonal. It was lessened by the introduction of the homologous crossover, where 45% of individuals did not change after crossover. Even though the rate has been cut in half, it is still not a desirable trait.

Ryan et al [Ryan et al 1998a] proposed a resolution to this: an operator that removes the excess genes from the chromosome. They claim that the effect of introducing pruning is “faster, better crossover”.

10.2.1. Prune Experiment

10.2.1.1. Hypothesis

Pruning promotes the formation of compact, high fitness individuals

10.2.1.2. Test

To test this hypothesis, the system will be run without pruning and with pruning at 50%, 90% and 100% probability. The length and fitness of the top 10 individuals will be recorded and compared.

10.2.1.3. Experiment Tableau

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Symbolic Regression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminals</td>
<td>X</td>
</tr>
<tr>
<td>Fitness Case</td>
<td>[-1, 1] step 0.1</td>
</tr>
<tr>
<td>Raw Fitness</td>
<td>Average of absolute difference between target and solution (with penalty difference 10 given when there is an error evaluating a data point and penal value 100 for completely invalid individuals)</td>
</tr>
<tr>
<td>Crossover</td>
<td>Homologous Transpose crossover</td>
</tr>
<tr>
<td>Pruning</td>
<td>On</td>
</tr>
<tr>
<td>Steady State</td>
<td>On</td>
</tr>
</tbody>
</table>
10.2.1.4. Results

Judging by the plots shown in Figure 10.8, there is no clearly defined tendency. However, it can be seen that environments that have been pruned can produce much more compact individuals with similar fitnesses.

![Graph of 100 Runs with different Pruning Levels](image)

**Figure 10.8  Graph of 100 Runs with different Pruning Levels**

This provides evidence to support the hypothesis – pruning does promote compact and fit individuals.

10.3. Wrapping

Since the genes of the genotype are free from any positional bias, the original developers of grammatical evolution proposed a new operator – wrapping. If the end of the genotype has been reached, this allows the mapping process to continue by starting again at the beginning of the genotype. This, it is claimed, promotes more compact solutions. However, it has the side-effect of reducing the number of invalid individuals in a population, since it gives more genes to work with. On the other hand, it also complicates the use of crossover and mutation, since a single gene changed affects multiple positions within the individual.

One complication is the interaction between wrapping and pruning. Since pruning cuts the chromosome at the point where the evaluation ends, it only truly has meaning to do this if the individual has not wrapped around. Wrapping has been implemented by default in the system.
10.4. Duplicate

Another addition to the genetic algorithm operators reintroduced by Ryan et al [Ryan et al 1998a] is the duplicate operator. This was first suggested by Holland for use on the genetic algorithm, but is not commonly implemented as part of the simple genetic algorithm. This passes a copy of a fit individual directly from the old population into the new population, bypassing crossover. The aim is to increase the proportion of fit individuals in the population. This is probabilistic and uses the crossover probability. The crossover typically has a 90% chance of occurring, the remaining 10% is then used for duplication. Since the duplication is tied into the crossover operator, it cannot be tested separately.
Section IV:

Conclusions
11. Conclusions

11.1. Reflections on the Aims of the Project

In considering the stated aims of the project, it may be recalled that the main motivation was to provide a clear and simple grammatical evolution system that can be used to demonstrate how the system is constructed. Therefore there are two main threads to examine – is the system a grammatical evolution system and is it sufficiently clear and simple.

The first point is not entirely trivial to answer. As has been noted in the Implementation section (See 8.7) the system’s behaviour is not exactly the same as that reported by the developers of the system. The expected behaviour was for the system to begin with a random set of individuals, and then use crossover, selection and mutation to improve the population. Therefore the plot of the fitness of the best individual for each generation was expected to be of similar form to an exponential decay. This system has a tendency to plateau out and, as mentioned before, the population becomes overtaken by the individuals on this plateau – a sign of premature convergence. In the reference material for researching the background to the system, much discussion was made of the threat of premature convergence, and as many precautionary steps as are practical have been built into this system. For example, one of the advantages of using tournament selection is that it spreads the selection power over a wider range of individuals, thereby reducing this threat. It is interesting to note that another implementation of grammatical evolution that was discovered, a Java implementation by Rebecca Griffith [Griffith 2002], also exhibits similar behaviour. Further, it cannot be just coincidence that another student studying a similar topic has encountered similar difficulties in implementing a Python variant of grammatical evolution [Stallard 2006]. One possible reason for this, beside premature convergence, is that crossover is not the most appropriate operator for moving the population on. This has been shown by the crossover experiment as detailed before; however the proposed variants also fail to meet the criteria. Even an alternate approach, variable rate mutation, fails to perform as expected. There was not sufficient time to consider the tree nature of the mapping, but perhaps that would yield better results, as the swapping of sub-trees could have a positive effect. If this is the case, then many of the developments from genetic programming could be brought to bear.

However, the system follows all of the instructions that were provided by the developers: Ryan et al [Ryan et al 1998a]. There are no tricks or optimisations recorded that could help. Indeed, much time was spend examining the original C source code for the original system, however it omitted the genetic algorithm portion and was not very legible (hence the motivation for the second aim) and therefore it is difficult to establish exactly how the system works. Additionally, due to the lack of genetic algorithm, the supplied code did not compile on its own, so no direct performance comparisons could be made. Under close scrutiny by the project supervisor, an experienced member in this field, no error in the theoretical implementation could be found; therefore it would not be correct to
mark this project a failure. Despite its lack of performance, it still provides almost all of the features required of a grammatical evolution system.

Therefore we can progress to the second aim: clarity. We can say that in this sense the project was a success. As noted in the preceding paragraph, the system is logically correct, and a student or researcher can easily follow the process by following the code. Due to the object oriented model, the system can even be used to learn about the genetic algorithm first, and then follow the development by considering the grammatical evolution system as an extension of the genetic algorithm. As much code as possible has been segmented out into a self-contained function, so that it is easier to follow than trying to digest a large block of code. Finally, the indentation that Python enforces – along with the readability of the syntax – ensures that the code is easy to follow.

11.1.1. The End User
In the Requirements section, one of the stakeholders was identified as the (potential) end user. As already noted, the system is not entirely ready for distribution, but still does produce a result that is at least fairly fit. However, once the crossover problem is solved, then the other features of the system come into their own. The program is self contained and requires no packages that are not distributed directly with Python – except for the numeric and graphing packages used by the running mechanism, but which are not really a part of the system and can be bypassed if desired.

To set up the system, all the user need do is specify the location of their own BNF grammar (in text or XML form) and the location of their input XML document to instruct the system in how to evaluate the function generated. For advanced users, the parameters of the system can also be set; in this case, the user must also specify the location of the parameters XML file. As noted in the Implementation section, it is permissible to keep any combination of the three XML sets in one file for ease of keeping a set-up together.

11.1.2. The Researcher
This system also accommodates the second stakeholder identified: the researcher. As for the end user, the various settings are easy to set up. In addition, there is a large amount of information recorded about the system, which provides much insight into how the system is operating.

The fulfilment of the needs of the researcher was easier to evaluate than the needs of an inexperienced end user because these needs were exercised in a practical way while performing the experiments detailed in Chapter 10. In particular, the information from the log file was used extensively while trying to figure out why the performance of the system was not as expected.
11.1.3. Conclusions
From the discourse above, we can conclude that even though the system does not match the performance expected, it is still a success in many ways – and it still provides scope for development. The question of the appropriateness of crossover and whether there are any better alternatives still has not been definitively resolved and would merit further investigation.

As far as contributions to the research field go, there is now a learning instrument for those learning about grammatical evolution to study and follow, something not previously available. In addition, the homologous transpose crossover has been developed and described here. This is the only crossover operator known to the author that can crossover meaningful elements on the phenotype rather than just the genotype. Although it did not perform as expected in the experimentation, it has demonstrated an approach that can be taken further by other researchers in this field.

11.2. Self-Appraisal
As the project progressed and improved, so did the author. Initially, not much was known about the system, and most of the background information dealt with the underlying genetic algorithm or the parallel system of genetic programming. As the understanding of the system took shape, so too did the design of what was needed, and the author’s own appraisal of what was required shaped the project from the beginning. For a more robust requirements specification, it would have been better to have contacted others who use or might consider using a grammatical evolution system to find out their needs and wishes. The design was very much top-down from the start, since this was the logical way of looking at the system as extending an existing object oriented module. However, other design choices were not considered, and it cannot be guaranteed that the chosen mode was the best.

As expected in a developmental and experimental project, the timing was approximate. While a project timescale was produced at the start of the project, it was not known exactly what weighting to give portions of the project. This lead to much reorganisation as the project reached its conclusion. It is fortunate that buffer time was allocated, or the overruns in experimentation would have curtailed the experimentation phase. The last third of the project did indeed contain half of the work and two thirds of the bugs! [Anon]

Overall, this project was a significant learning experience, more akin to a workplace project in detail and scope than any previous project. One of the biggest lessons was of the importance of scheduling, followed by the importance of following a predefined structure rather than striking out in the hope of landing upon a solution.
11.3. Improvements

For the most part, the system is good as is. However there are a few places in which gaps have been left unfilled. One of these is the multivariate case. There is provision within the input XML file for values in multiple variables to be loaded, but there was no clear way to implement this in the evaluation code. It would require an arbitrary number of nested loops, one for each variable – which would be difficult to implement. This may merit further consideration, but many of the examples of problems are merely univariate and the facility may not be of use. The issue about the performance of the system, whether it is premature convergence or a problem with crossover, bears further study. Due to time constraints, it was not possible to investigate the tree structure of the mapping or to apply the research done in the field of genetic programming. However, operators like ripple crossover could become significant, as could the use of grammar inserted introns or sequence break points. To further improve the transparency of the system, the parsing of the text BNF file could be rewritten to implement an external parser. This would make the process clearer to the uninitiated. Further clarity could be achieved by the addition of a user interface. Currently the basic system settings are loaded from an XML file in a static position, which tells the system where the remaining settings may be found. This would be much simpler to handle if the user could use a dialogue box to edit this file. Otherwise, there is little need for user interface, as all other communication already occurs within the interpreter.

11.4. Closing Comments

Much has been made in the preceding chapters of the lack of convergence of the system, but what has never been emphasised is that the same grammar has been used throughout – the grammars described by Ryan and O’Neill [Ryan and O’Neill 2003]. However, one of the points of the system is that the input can be enriched by insider knowledge. The genetic algorithm does not adapt to different problems, but grammatical evolution reads in a BNF grammar to use in its mappings. Therefore additional problem specific information can be used to enrich the grammar and enhance its performance. Doing this has allowed ‘perfect’ solutions to be generated – even given the system’s state. One example of this uses the Symbolic Regression problem as a base, and supposes that the user wishes to find the closest polynomial that fits the data. In this case, the original grammar is unnecessary since it is far too general. A better grammar defines constructs such as a post-operator that raises a variable to an integer power. This then guides the system more exactly into the solution required.

This adaptability is one of the powers of a grammatical evolution system, and should not be overlooked amid the quest for generality. It is worth pointing out that this Python implementation allows unparalleled adaptability as was demonstrated by the Santa Fe problem. Using Python’s dynamic execution abilities, user-written modules can be imported and used to interpret the phenotype in any way desired. Therefore, any technically competent user can indefinitely extend the evaluation subsystem without need to modify (or even view) the source code of the system.
12. References


Section V:

Appendices
A. Test Plan

The following table is a summary of the tests performed in the GEtest module as discussed in Chapter 8. The criterion for success is for all of the listed tests to succeed. This is an automated test that should be run after any changes to the system.
<table>
<thead>
<tr>
<th>Test Module</th>
<th>Test Item</th>
<th>Test Case</th>
<th>Input</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNF</td>
<td>getTerms</td>
<td>Single non-terminal</td>
<td>'&lt;expr&gt;'</td>
<td>['&lt;expr&gt;']</td>
</tr>
<tr>
<td></td>
<td>Function</td>
<td>Single long non-terminal</td>
<td>'&lt;long expr&gt;'</td>
<td>['&lt;long expr&gt;']</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Single terminal</td>
<td>'1'</td>
<td>['1']</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Terminal with escaped</td>
<td>'esc&lt;'</td>
<td>['esc&lt;']</td>
</tr>
<tr>
<td></td>
<td></td>
<td>character</td>
<td>Non-terminal with escaped</td>
<td>['&lt;&lt;expr&gt;']</td>
</tr>
<tr>
<td></td>
<td></td>
<td>character</td>
<td>Compound expression</td>
<td>['&lt;expr&gt;', '+', '&lt;var&gt;']</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Incomplete non-terminal</td>
<td>'&lt;&gt;'</td>
<td>BNFError raised</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Empty non-terminal</td>
<td>'&lt;&gt;'</td>
<td>BNFError raised</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Invalid non-terminal</td>
<td>'&lt;&lt;expr&gt;'</td>
<td>BNFError raised</td>
</tr>
<tr>
<td></td>
<td>getRules</td>
<td>Legal compound expression</td>
<td>'&lt;expr&gt; ::=&lt;term1&gt;</td>
<td>&lt;term2&gt; term3 term4 &lt;term5&gt;'</td>
</tr>
<tr>
<td></td>
<td>function</td>
<td>No rules (null expression)</td>
<td>'&lt;expr&gt;::='</td>
<td>BNFError raised</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No rules (trivial expression)</td>
<td>'&lt;expr&gt;::= '</td>
<td>BNFError raised</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No rules (but some rule dividers)</td>
<td>'&lt;expr&gt;::=</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Term without rule definition</td>
<td>'&lt;expr&gt;'</td>
<td>BNFError raised</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Invalid initial non-terminal</td>
<td>'&lt;expr&gt;::=&lt;term&gt;'</td>
<td>BNFError raised</td>
</tr>
<tr>
<td>GA</td>
<td>OneMax</td>
<td>Run OneMax test environment</td>
<td>OneMax settings</td>
<td>[1]*30</td>
</tr>
<tr>
<td>GE</td>
<td>bin2dec</td>
<td>Evaluate all ones</td>
<td>[1]*8</td>
<td>255</td>
</tr>
<tr>
<td></td>
<td>Function</td>
<td>Evaluate all zeros</td>
<td>[0]*8</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Evaluate intermediate</td>
<td>list [0,0,0,1,0,1,0]</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>mutation</td>
<td>Mutate first gene</td>
<td>chromosome = [0]*30</td>
<td>chromosome[0] = 1</td>
</tr>
<tr>
<td></td>
<td>(gene)</td>
<td>Mutate last gene</td>
<td>chromosome = [0]*30</td>
<td>chromosome[29] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mutate intermediate gene</td>
<td>chromosome = [0]*30</td>
<td>chromosome[10] = 1</td>
</tr>
<tr>
<td>Test Module</td>
<td>Test Item</td>
<td>Test Case</td>
<td>Input</td>
<td>Expected Result</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
<td>-----------</td>
<td>-------</td>
<td>-----------------</td>
</tr>
<tr>
<td>GE</td>
<td>Mutation (individual)</td>
<td>Mutation probability seen over 1000 iterations</td>
<td>chromosome = [0]*30</td>
<td>Average score within ±0.1 of expected mutation count</td>
</tr>
<tr>
<td>Crossover</td>
<td>Crossover of two disjoint individuals, 100 iterations</td>
<td>[0]*80 and [1]*80</td>
<td>Each individual has a sum S: 0&lt;S&lt;79</td>
<td></td>
</tr>
<tr>
<td>getGenotype</td>
<td>Minimum chromosome</td>
<td>[0]*80 (binary chromosome)</td>
<td>[0]*10 (decimal genotype)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximum chromosome</td>
<td>[1]*80</td>
<td>[255]*10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Intermediate chromosome</td>
<td>(see testing code)</td>
<td>[220, 203, 17, 3, 109, 215, 104, 30, 0, 255]</td>
<td></td>
</tr>
<tr>
<td>getPhenotype</td>
<td>Evaluate phenotype (symbolic regression grammar)</td>
<td>genotype [220, 203, 17, 3, 109, 215, 104, 30, 0, 255] and regression grammar</td>
<td>-(((1.0 + 1.0) + 1.0) + 1.0))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Evaluate phenotype (trigonometric identities grammar)</td>
<td>genotype [220, 203, 17, 3, 109, 215, 104, 30, 0, 255] and trigonometry grammar</td>
<td>X-X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Evaluate non-terminal individual</td>
<td>genotype [144, 87, 175, 96, 131, 95, 182, 149, 131, 224] and regression grammar</td>
<td>{see testing code for phenotype} and score set to the penal value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Evaluate non-terminal individual (with wrapping disabled)</td>
<td>genotype [144, 87, 175, 96, 131, 95, 182, 149, 131, 224] and regression grammar</td>
<td>{see testing code for phenotype} and score set to the penal value</td>
<td></td>
</tr>
<tr>
<td>getScore()</td>
<td>Target function</td>
<td>'((X<em>X</em>X<em>X)+(X</em>X<em>X)'+(X</em>X)+(X)'</td>
<td>Score is 0 (to 10 decimal places – for rounding errors of order 10^{16})</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Constant function</td>
<td>'1.0'</td>
<td>Score is 1.11 (to 2 decimal places)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Function that is undefined at each point of comparison</td>
<td>[\prod_{i=-10}^{10} \frac{1}{(X - \frac{i}{10})}]</td>
<td>Punishment value as defined by user</td>
<td></td>
</tr>
</tbody>
</table>
B. Program Listing

The code for this project is collected into a folder, pyGE, which contains the file __init__.py which provides access to the rest of the system. This can then be imported in the same way as any other Python package. The source code for selected modules (underlined) is shown in the following sections. The remaining files may be found on the CD bound to this document.

Main
__init__.py  The access point for the user (allows pyGE to be a package)
GErun.py    Set up and run the system
GE.py       The bulk of the GE system
pyga.py     The original genetic algorithm (provided pre-written)

Settings Import
GEsettings.py  Co-ordinates the import of the various settings
GEbasic.py     Imports the basic settings for the system
GEinput.py     Imports the input data for the problem
GEparameters.py Imports the parameter data for the problem
GEbnf.py       Imports the BNF grammar for the problem (XML)
BNF.py         Imports the BNF grammar for the problem (text)

Testing and Experimentation
GETest.py      Runs the structured unit tests
Tests.py       Runs the tests for the Experimentation chapter

Settings Files
System Settings
GEbasic.xml    The basic settings for the system

Symbolic Regression
GEinput_Regression.xml The input settings for Symbolic Regression
GEparam_Regression.xml The parameters for Symbolic Regression
BNF_Regression.xml The grammar for Symbolic Regression as XML
Regression.bnf   The grammar for Symbolic Regression as text

Santa Fe Ant Trail
GEinput_SantaFe.xml The input settings for the Santa Fe Ant Trail
GEparam_SantaFe.xml The parameters for the Santa Fe Ant Trail
BNF_SantaFe.xml   The grammar for the Santa Fe Ant Trail as XML
SantaFe.py       Encapsulates the ant class used to run the problem

XML Tools
BNF.xsl        Displays BNF XML files as their textual equivalent
BNF.xsd        XML Schema to validate XML BNF files
GEbasic.xsd    XML Schema to validate GEbasic.xml
GEinput.xsd    XML Schema to validate input XML file
GEparameters.xsd XML Schema to validate parameters XML file
B.1. Interaction of Modules

All modules are grouped into a folder named pyGE which allows Python to import it as a package, and the __init__ file provides access to the rest of the system.
GE.py

Contains the main functionality of the grammatical evolution system.

Contents:
- Classes
  - GE class
  - GEenv class
# extend the GA class to incorporate GE functionality
import string, decimal, random, logging, math
#import BNF, BNFgrammar, BNFError
import pyga, GEsettings

# initialise the settings class
Settings = GEsettings.Settings()

if Settings.BASIC['run_psyco'] == 'yes':
    import psyco
    psyco.full()

class GE(pyga.Individual):
    "Encapsulates one individual of the population"
    score = phenotype = None
    genotype = []   rules = {}

    def __init__(self, chromosome = None, length = None, log = None):
        "Initialise a new individual, optionally with chromosome, length and log file"
        length = length or Settings.PARAM['allelesPerCodon']
        self.chromosome = chromosome or self._makechromosome()
        self.optimization = Settings.PARAM['compare']
        self.score = None
        self.log = log or logging.getLogger('GE')

    def bin2dec(self, codon):
        "Convert binary codon string into its decimal representation"
        result = 0
        for exponent, allele in enumerate(codon):
            result += ((2 ** (len(codon) - exponent - 1)) * int(allele))
        return result

    def getGenotype(self):
        "Convert the binary genotype into its decimal counterpart"
        # loop through the chromosome, using the specified frame size
        return [self.bin2dec(self.chromosome[i:i+length]) for i in range(0, len(self.chromosome), length)]

    def getPhenotype(self, genotype):
        "Map the integer genome onto the BNF grammar"
        # set the initial symbol as the start symbol

        phenotype = ['<start>']
        self.rules = []

        # build the phenotype, looping through the genotype to do so
        # (incorporate wrapping)
        for step in range(len(genotype) * Settings.PARAM['wrapNos']):
            # rules with only one choice do not read a value from the genotype, so loop until a value must be read
            readGenotype = False
            while not readGenotype:
                # find the first non-terminal in the phenotype
                pheno = -1
                for termNo, term in enumerate(phenotype):
                    if term[0] == '<':
                        pheno = termNo
                        break
                # if last term was -1, there were no nonterminals left, break out of the loop
                if pheno == -1:
                    break
                # get the term object relating to the term being examined
                currTerm = Settings.BNF.getTerm(phenotype[phpointer])
                # if there are multiple options to this rule, choose one via the genotype
                if currTerm.countOptions() > 1:
                    # if there are multiple rules, use the genotype to pick
                    # one, and then move onto the next allele
                    readGenotype = True
                    # choose which option to select (current codon value
                    # modulo count of options)
                    optionNo = genotype[step % len(genotype)] %
                    currTerm.countOptions()
                    # keep a record of the rule to which this element is
                    # applied (if wrapping is not in effect)
                    if step < len(genotype):
                        self.rules.append(phenotype[phpointer])
                        # insert the chosen option in place of the term being
                        # examined
                        phenotype[phpointer] = currTerm.getOption(optionNo)
                    else:
                        optionNo = 0
                        self.rules.append(phenotype[phpointer])
                        # insert the chosen option in place of the term being
                        # examined
                        phenotype[phpointer] = currTerm.getOption(optionNo)
                    # if the phenotype has terminated,
                    if phenotype[phenotype.index('<start>')] == phenotype[-1]:
                        break
                else:
                    # if the for loop was not broken, the individual did not
                    # terminate - assign it a penal fitness value
self.score = Settings.PARAM['penal']
self.phenotype = ''.join(phenotype)

def experiment_getPhenotype(self, genotype):
    "Map the integer genome onto the BNF grammar - experimental
to create version without breaks"
    # set the initial symbol as the start symbol
    phenotype = ['<start>']
    nonterminalPhenotype = True
    step = 0
    # build the phenotype, looping through the genotype to do so
    while step < (len(genotype) * Settings.PARAM['wrapNos']) and nonterminalPhenotype:
        # rules with only one choice do not read a value from the
        # genotype, so loop until a value must be read
        readGenotype = False
        while not readGenotype and nonterminalPhenotype:
            # find the first non-terminal in the phenotype
            phpoiner = -1
            termNo = 0
            # loop until a non-terminal is found, or there are no
            more terms in the phenotype
            while phpoiner == -1 and termNo < len(phenotype):
                if phenotype[termNo][0] == '<':
                    phpoiner = termNo
                    termNo += 1
                # if phpoiner is -1, there were no non-terminals left,
                # indicate that the phenotype terminates
                if phpoiner == -1:
                    nonterminalPhenotype = False
                    break
                else:
                    # get the term object relating to the term being
                    examined
                    currTerm = Settings.BNF.getTerm(phenotype[phpoiner])
                    if currTerm.countOptions() > 1:
                        readGenotype = True
                        # choose which option to select (current codon value
                        modulo count of options)
                        optionNo = genotype[step % len(genotype)] %
                        currTerm.countOptions()
                        else:
                            optionNo = 0
                            # insert the chosen option in place of the term being
                            examined
                            phenotype[phpoiner:phpoiner+1] =
                            currTerm.getOption(optionNo)
                            step += 1
                    # if the phenotype did not terminate, assign it a penal
                    fitness value
                    if nonterminalPhenotype == False:
                        self.score = Settings.PARAM['penal']
                        self.phenotype = ''.join(phenotype)
    def getScore(self):
        "Compare the phenotype with the user defined function"
        # the user specifies the function to compare with and how to
        # compare - both as Python code
        # (The compare is in the form of Python code where
        'phenotype' denotes the phenotype and 'userfct' denotes the user
        function)
        result = []
        # load the modules required
        for module in Settings.INPUT['import']:
            exec('import ' + module)
        for variable in Settings.INPUT['values']:
            for value in Settings.INPUT['values'][variable]:
                # setup variable
                exec(variable + '=' + value[0])
                # evaluate phenotype and user function (ignore any code
                errors at this stage)
                try:
                    # run any preliminary setup / processing
                    if 'pre-code' in Settings.INPUT:
                        exec(Settings.INPUT['pre-code'])
                    # put brackets around the variable to allow negative
                    powers to be dealt with properly (-X)^2 not -(X^2)
                    _evaluation_ph = self.phenotype.replace(variable,
                    ('' + variable + '))
                    exec('_evaluation_usr = ' + str(value[1]))
                    # run any processing and/or clean-up
                    if 'post-code' in Settings.INPUT:
                        exec(Settings.INPUT['post-code'])
                    result += [eval(Settings.INPUT['compareFunction'])]
                    except:
                        self.log.info('Evaluation:	Error - not all functions
                        may be evaluated at the point '+ variable + "="+ value[0])
                        result += [Settings.INPUT['punishment']]  
                self.score = sum(result) / len(result)
    def prune(self):
        "Prune the chromosome of any unused genes"
if len(self.rules) < len(self.genotype) and random.random() < Settings.PARAM['pruneProb']:
    self.chromosome = self.chromosome[:len(self.rules) * Settings.PARAM['allelesPerCodon']]

def evaluate(self, optimum = None):
    "Evaluate the individual's score"
    # if the evaluation has already been performed, abort
    if self.score != None:
        return
    # convert the binary chromosome into an integer genotype
    self.genotype = self.getGenotype()
    # map the genotype onto the grammar to obtain the phenotype
    self.phenotype = self.getPhenotype(self.genotype)
    # if the phenotype mapping terminated, evaluate its score
    if self.score != Settings.PARAM['penal']:
        return self.getScore()

    if Settings.PARAM['prune'] == 'yes':
        self.prune()

    # override GA mutation
    def mutate(self, gene):
        self.chromosome[gene] = not self.chromosome[gene]

    def crossover(self, other):
        "Call the chosen crossover method"
        if Settings.PARAM['crossoverType'] == "1 point":
            return self._onepoint(other)
        elif Settings.PARAM['crossoverType'] == "1 point var":
            return self._onepointvar(other)
        elif Settings.PARAM['crossoverType'] == "2 point":
            return self._twopoint(other)
        elif Settings.PARAM['crossoverType'] == "homologous":
            return self._homologous(other)
        else:
            return self._onepoint(other)

    def _onepoint(self, other):
        "Creates offspring via one-point crossover between mates."
        self.left = other.left = 0
        self.right = other.right = self._randrange(1, self.length-1)
        return self.mate(other), other.mate(self)

    def _onepointvar(self, other):
        "Creates offspring via one-point crossover (variable length) between mates."
        self.left = other.left = 0
        self.right = other.right = random.randrange(1, self.length-1)
        other.right = self._randrange(1, other.length-1)
        return self.mate(other), other.mate(self)

    def _homologous(self, other):
        "Create offspring via a two-point homologous crossover"
        # loop through the two sequences of rules until the region of similarity is ended
        while self.rules[pointer] == other.rules[pointer] and
        pointer < (min(len(self.rules), len(other.rules)) - 1):
            pointer += 1
        self.left = other.left = pointer *
        self.right = other.right = self._randrange(pointer, other.length - 1) *
        return self.mate(other), other.mate(self)

    def _homologous_transpose(self, other):
        "Creates offspring via a two-point homologous crossover variant that preserves the meaning of the transposed segment"
        # identify all possible points of crossover
        points = []
        # for each rule, note all the positions it comes in
        for term in Settings.BNF.getTerms():
            points[term] = []
        for term in enumerate(self.rules):
            points[term][0].append(index *
            Settings.PARAM['allelesPerCodon'])
            for index, term in enumerate(other.rules):
                points[term][0].append(index *
                Settings.PARAM['allelesPerCodon'])
        # remove rules that are not present in one or both of the individuals
        for term in points.copy():
            if len(points[term][0]) > len(points[term][1]) == 0:
                # pick random 'start' and 'end' rules
                start = random.choice([term for term in points])
                end = random.choice([term for term in points])
                # pick the position of the rule to use (removing all lesser
positions before choosing the end position)
    s_pivots = [random.choice(points[start][0])]
    for point in points[end][0]:
        try:
            if point <= s_pivots[0]:
                del points[end][0][points[end][0].index(point)]
            except:
                print point, s_pivots[0]
                print points[end][0]
                print points[end][0].index(point)
                raise
        s_pivots.append(random.choice(points[end][0] + len(self.rules) * Settings.PARAM['allelesPerCodon']])

    o_pivots = [random.choice(points[start][1])]
    for point in points[end][1]:
        if point <= o_pivots[0]:
            del points[end][1][points[end][1].index(point)]
        o_pivots.append(random.choice(points[end][1] + len(other.rules) * Settings.PARAM['allelesPerCodon'])))
    other.left, other.right = min(o_pivots), max(o_pivots)
    s_pivots = [random.choice(points[end][0])]
    for point in points[end][1]:
        if point <= s_pivots[0]:
            del points[end][1][points[end][1].index(point)]
        raise
        s_pivots.append(random.choice(points[end][1] + len(self.rules) * Settings.PARAM['allelesPerCodon'])))
    self.left, self.right = min(s_pivots), max(s_pivots)
    other.left, other.right = min(o_pivots), max(o_pivots)
    self.mate(other), other.mate(self)

    self.phenotype = self.phenotype
    twin = self.__class__(self.chromosome[:])
    twin.score = self.score
    twin.phenotype = self.phenotype
    twin.rules = [rule for rule in self.rules]
    twin.index = self.index
    return twin

    # redefine environment to add features
    class GEenv(pyga.Environment):
        bestIndividuals = []  # list of scores of best (and average)
        populationArchive = []
        def step(self):
            """Advance to the next generation (modified to allow adaptive mutation)"
            if Settings.PARAM['variable_mutation'] == 'yes':
                Settings.PARAM['mutation_rate'] = math.e^{-Settings.PARAM['mutation_multiplier'] * self.generation}
            # run the usual step process
            pyga.Environment.step(self)

            # override tournament to allow settings to be loaded
            def _tournament(self, size=None, choosebest=None):
                size = size or Settings.PARAM['tournamentSize']
                choosebest = choosebest or Settings.PARAM['chooseBest']
                competitors = [random.choice(self.population) for i in range(size)]
                competitors.sort()
                if random.random() < choosebest:
                    return competitors[0]
                else:
                    return random.choice(competitors[:])

            def _crossover(self):
                """According to users selection, perform either steady state or generational crossover"
                if Settings.PARAM['steadyState'] == 'yes':
                    self._steadystate()
                else:
                    # run original crossover function
                    pyga.Environment._crossover(self)

            # randomise some of the invalid individuals
            for individual in self.population:
                if individual.score == Settings.PARAM['penal'] and
```python
random.random() < Settings.PARAM['steadyStateReset']:
    self.log.info('Reset:	' + str(individual.index or '<new>') + ' reset')
    individual = self.kind(log = self.log)

def _steadystate(self):
    "Perform steady state selection rather than generational selection"
    # define number of items to change as
    # choose the individuals to replace (randomly choose positions in population)
    next_population = [self.best.copy()]
    position = set()
    # select the individuals to be replaced under steady state
    while len(position) < (Settings.PARAM['steadyStateNo']):
        # select a position to replace
        position.add(random.randrange(1, self.size))
        # reserve best element (0)
        for i in range(1, self.size):
            # if the individual is to be replaced, use crossover
            if i in position:
                mate1 = self._select()
                mate2 = self._select()
                offspring = mate1.crossover(mate2)
                self.log.info('Crossover:	' + str(mate1.index) + ' with ' + str(mate2.index))
            # otherwise, copy the individual into the new population
            else:
                individual = self.population[i].copy()
                self.log.info('Duplicate:	' + str(individual.index) + ' duplicated')
                self._mutate(individual)
                individual.evaluate(self.optimum)
                next_population.append(individual)
    self.population = next_population[:self.size]
    # define counterpart to best
    def worst(self):
        doc = "individual with worst fitness score in population."
        def fget(self):
            return self.population[-1]
        return locals() worst = property(**worst())
    # also output to csv file (if user wishes)
    def report(self):
        "Report the details of the population to the user and log files"
        if Settings.BASIC['print_report'] == 'yes' or self.generation > self.maxgenerations:
            print '='*70
            print "generation: ", self.generation
            print "best: ", self.best
            self.log.info('="*' + str(self.generation))
            self.log.info('="*' + str(self.best))
            self.log.info('="*' + str(self.worst))
            # print generation, score, phenotype for each individual in population
            total = []
            for i, individual in enumerate(self.population):
                if Settings.BASIC['report_population'] == 'yes':
                    Settings.files['complete'].write(','.join([str(self.generation), str(individual.index), str(individual.score), '', str(individual.phenotype)]) + '
')
                    total.append(individual.score)
                    self.populationArchive.append([self.generation, individual.index, individual.score, individual.phenotype])
                    self.bestIndividuals.append((self.generation, self.best.score, sum(total) * 1.0 / len(total)))
                if Settings.BASIC['report_population'] == 'yes':
                    Settings.files['complete'].flush()
                if Settings.BASIC['report_summary'] == 'yes':
                    Settings.files['summary'].write(', join([str(self.generation), str(self.best.score), str(self.worst.score), self.best.phenotype]) + '\n')
                    Settings.files['summary'].flush()
```
BNF.py

Contains the custom parser written to read in text bnf files. Also contains the object oriented structure of the grammar, as seen by the system – this structure is reused by the XML bnf parser – GEbnf.py

Contents:

- Classes
  - BNFgrammar class
  - BNFrule class
  - BNFError class

- Methods
  - getRules
  - getTerms

<table>
<thead>
<tr>
<th>BNFrule</th>
<th>BNFgrammar</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td><strong>init</strong></td>
</tr>
<tr>
<td>addOptions()</td>
<td>addFile()</td>
</tr>
<tr>
<td>addOption()</td>
<td>addRules()</td>
</tr>
<tr>
<td>getName()</td>
<td>addExternalRules()</td>
</tr>
<tr>
<td>getTerms()</td>
<td>checkConsistency()</td>
</tr>
<tr>
<td>countOptions()</td>
<td>getTerm()</td>
</tr>
<tr>
<td>getOption()</td>
<td>getTerms()</td>
</tr>
<tr>
<td><strong>repr</strong>()</td>
<td><strong>repr</strong>()</td>
</tr>
</tbody>
</table>

1..* 1
# read BNF grammar from text file and build parse tree
# BNF is defined by:
# One rule per line:
#   Exactly one non-terminal followed by ::= 
#   followed by at least one sequence of terms
# A sequence of terms consists of:
#   n non-terminals (enclosed between < and >) and m terminals
#   such that n+m>0
#
# import string

def getRules(rule):
    "Function to break up an BNF rule into its constituent options,
    which are then broken into their individual terms"
    # first, extract the rule name ( <name> ::=)
    definition = rule.find('::=')  # find the position where the
    # definition sequence resides
    if there is no ::= in rule, an error is to be raised
    if definition == -1:
        raise BNFError('No rule definition symbol', 'rule')
    else:
        # there must be precisely one term, delimited by < and > (any
        # characters outside <> are ignored
        char = rule[0]
        termbegin = False;  # indicator - set if a beginning of term
        delimiter is seen
        termend = False;    # indicator - set if an end of term
        delimiter is seen
        name = ''  # the defined 'name' of the rule

        # the loop below moves through the portion of the rule that
        # is before the ::= symbol
        # it waits until the first '<', then records every character
        # until the next '>'.
        # this is the name of the rule
        # if there are multiple '<' characters (multiple terms), or
        # no '>' (no full term) an error is raised
        for i in range(definition):
            if rule[i] == '<':
                if termbegin == False:  # if this is the first such term,
                    termbegin = True;
                    name = '<'
                else:
                    raise BNFError('Invalid rule definition', 'rule')
            elif rule[i] == '>':
                termend = True;
                name = name + '>'
            elif (termbegin and (not termend)): # i.e. there has been
                ' '< but no '>' yet
                name = name + rule[i]
        else:
            # anything else is ignored

        # if the term was not terminated, then raise error for
caller to handle
        if termend == False:
            raise BNFError('Invalid rule definition, no valid term to
            define', 'rule')

        # crop the rule to only leave the rules
        rule = rule[definition + 3:]

        # the first entry into the terms list is the name
terms = [name]  # list of lists, one list for each option:
            # consisting of the terms contained within the option

        ### Now extract the terms from the rule
        # if there are no options or terms on the right hand side,
        raise an error
        if len(rule.strip()) == 0:
            raise BNFError('Invalid rule definition, no options
defined', 'option')

        # split into options, delimited by | (first remove doubles :
        # : as empty rules are not permitted - but not errors
        rule = rule.replace('::=', '|', 0)
        # check that there is some data in the rule (to weed out
        # instances of '::=|')
        if rule.replace('::=', '').strip() == '':
            raise BNFError('Invalid rule definition, no valid term to
            define', 'rule')

        options = rule.split('|')

        # split into options, delimited by | (first remove doubles :
        # : as empty rules are not permitted - but not errors
        rule = rule.replace('||', '||')
        # check that there is some data in the rule (to weed out
        # instances of '||')
        if rule.replace('||', '').strip() == '':
            raise BNFError('Invalid rule definition, no options
defined', 'option')

        # try:
        #    terms.append(getTerms(enumerate(options))
        except BNFError, error:
            pass

    # now we have our terms, extract the individual terms
    terms = [getTerms(o) for o in options]
# this is called if there is an error while parsing the terms
raise BNFError(error.description, error.level, error.term, optionNo + 1)  # pass additionally the option number

# return the list of terms to the caller
return terms

###################################################
# Function to break up a BNF option into its constituent
terms (terminal and non-terminal)
Non-terminal terms are delimited by < and >, and all
characters outside these (including spaces)
are considered terminals

    terms = []  # list of terms in the option
termType = ''  # indicates whether the loop is searching for the
dead of a term or not (and if so, which type, N or T)
termNo = 1  # keep count of term number for error reporting

controlChar = False  # do not process escaped characters (i.e.
if '<' is a character in a terminal, put '\<' instead)

# parse non-terminals <\> and terminals from the option
for char in option:
    # firstly, if the character is an escape character, treat it specially
    if controlChar == True:
        currTerm = currTerm + char
        controlChar = False
    # otherwise check whether the next character will be escaped
    # (note that '\\' must also be escaped in the code)
    elif char == '\\':
        controlChar = True
    # only if this is not the special case of an escaped character, do standard term processing
    else:
        # if we are searching for an end delimiter (">") of a non-terminal
        if termType == 'N':
            # if we have the end delimiter of a non-terminal term
            if char == '>':
                # first check that the non-terminal has non-null value,
                raise an error if not
                if currTerm == '<':
                    raise BNFError('Empty non-terminal term', 'term', termNo)
                else:
                    # add term to list and reset variables
                    terms.append(currTerm + char)
                    termNo = termNo + 1  # increment term number (for error reporting)
                    currTerm = ''
                    termType = ''

        # if we are seeking the end of a non-terminal term,
        # another '>' is forbidden (must be written \>)
        elif char == '>':
            raise BNFError('Term error, unexpected ">"', 'term', termNo)

        else:
            # otherwise, extend current term
            currTerm = currTerm + char
            # if we have '<', start a new non-terminal
            elif char == '<':
                # first save any terminal terms to terms list
                if currTerm != '':
                    terms.append(currTerm)
                termNo = termNo + 1  # increment term number (for error reporting)

                currTerm = char
                termType = 'N'

            # check for stray instances of > (must be escaped as \\) -
            # raise error to help located missing brackets
            elif char == '\>':
                raise BNFError('Term error, unexpected "\.>"', 'term', termNo)

            # otherwise, extend current term and loop again
            else:
                currTerm = currTerm + char

    # finally, if there are terminal characters left over, append them to the end of the list
    if currTerm != '':
        terms.append(currTerm)

    # if we have run through the above and have a partial term,
    # raise an error for the caller to handle
    if termType != '':
        raise BNFError('Unterminated term', 'term')

    # return the terms extracted from the option

    currTerm = term
    termNo = termNo + 1

    controlChar = False
    termType = ''
def getTerms(self):
    """Return a set containing the names of all non-terminals referenced in this option"
    names = set()
    for option in self.options:
        for term in option:
            if term[0] == '<':
                names.add(term)
    return names

def countOptions(self):
    """Return the number of options defined"
    return len(self.options)

def getOption(self, optionNo):
    """Return a list containing the terms of the option who's number is passed"
    if len(self.options) < optionNo:
        raise BNFError('Option #'+repr(optionNo) + ' does not exist', 'option')
    else:
        return self.options[optionNo]

def __repr__(self):
    return str(self.options)
def __init__(self, fileName = None):  
    self.N = {}  # list of non-terminals (as instances of 
               # BNFrule), in dictionary format indexed by name 
    self.S = start # start name now defined within input file -- 
                 # (not used, start *must* be defined <start>) 
    if fileName != None: 
        self.addFile(fileName)

#-------------------------------

def addFile(self, fileName): 
    "Parse BNF from a given filename" 
    try: 
        f = open(fileName, 'r') # open read-only 
    except IOError:  
        # pass error on down as a BNF Error (rather than IO) 
        raise BNFError('File Error: ' + fileName, 'file') 
    else:  
        # read the lines of the file into a list 
        rules = f.readlines() 
        f.close() 
        self.addRules(rules)

#-------------------------------

def addRules(self, rules):  
    "Parse the BNF syntax passed in text form 
    Create a new instance of BNF term for each individual rule 
    (delimited by a new line)       
    This class will parse the given rule to recover the options 
    within" 
    obsolete ruleNo = 1 # count the rules defining non 
                       # terminals, for reporting purposes 
    names = set() # keep a record of the non-terminal names in 
                   # each rule (for checking consistency) 
    for ruleNo, rule in enumerate(rules): 
        # remove any outstanding newline characters 
        rule = rule.rstrip('
') 
        # create a new term instance holding the parsed information 
        # from the rule 
        try: 
            newTerm = BNFrule(rule) 
        except BNFError, error:  
            # if there was a rule or term error, handle it here 
            raise BNFError(error.description, error.level, 
                            error.term, error.option, ruleNo + 1) 
        else:  
            if newTerm.getName() not in self.N: 
                self.N[newTerm.getName()] = newTerm 
                #debug print '%-8s, %-3s, %s' % (newTerm.getName(), 
                #newTerm.countOptions(), newTerm.options) 
                # (make a list of terms referenced within this term's 
                # definition, add it to the list already held) 
                names = names | newTerm.getTerms() 
            else:  
                raise BNFError('Duplicate non-terminal definition, term: 
                                name = ' + newTerm.getName(), 'rule', 
                                rule = ruleNo + 1) 
        self.checkConsistency(names)

#-------------------------------

def addExternalRules(self, rules, names): 
    "Add a dictionary of rules from an external parser to the 
    internal dictionary of rules" 
    if isinstance(rules, dict): 
        for rule in rules: 
            if rule not in self.N: 
                self.N[rule] = BNFrule(name = rule, options = 
                                      rules[rule]) 
                #debug print '%-8s, %-3s, %s' % (rule, 
                # validate the grammar 
                self.checkConsistency(names)

#-------------------------------

def checkConsistency(self, names): 
    "Check that the all terms used in the BNF are defined, and 
    verify that there is a start symbol" 
    for rule in rules: 
        if rule not in self.N: 
            raise BNFError('Inconsistent BNF definition, start symbol 
                            "<start>" not defined', 'rule') 
        for name in names: 
            if name not in self.N: 
                raise BNFError('Inconsistent BNF definition, term "' + 
                                name + '" not defined', 'rule')
def getTerm(self, name):
    """ Return the requested term object
    (or raise an error if the name is not found within the
dictionary)
    """
    if name in self.N:
        return self.N[name]
    else:
        raise BNFError('Requested term does not exist', 'rule')

def getTerms(self):
    "Return a list of defined terms"
    return [name for name in self.N]

def __repr__(self):
    return ''.join(['%-10s ::= %s
' % (rule, str(self.N[rule]))
          for rule in self.N])

# Exception to indicate problems parsing the BNF
# All parameters are optional, but may be used to provide more
text of error
class BNFError(Exception):
    def __init__(self, description='BNF Parse Error', level='',
                 term=-1, option=-1, rule=-1):
        self.description = description
        self.level = level
        self.term = term
        self.option = option
        self.rule = rule

    def __str__(self):
        return''
GEsettings.py

Encapsulates all the settings of the system for the GE and GErun modules. Co-ordinates all of the settings import modules.

Contents:
- Classes
  - Settings class

<table>
<thead>
<tr>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>+files</td>
</tr>
<tr>
<td>+PATH</td>
</tr>
<tr>
<td>+BASIC</td>
</tr>
<tr>
<td>+INPUT</td>
</tr>
<tr>
<td>+PARAM</td>
</tr>
<tr>
<td>+BNF</td>
</tr>
<tr>
<td>+log</td>
</tr>
<tr>
<td><strong>init</strong>()</td>
</tr>
<tr>
<td>refresh()</td>
</tr>
<tr>
<td>getBasic()</td>
</tr>
<tr>
<td>getSettings()</td>
</tr>
<tr>
<td>getFiles()</td>
</tr>
<tr>
<td>setupLog()</td>
</tr>
<tr>
<td>cleanup()</td>
</tr>
</tbody>
</table>
# Load and hold the settings for the GE

```python
import logging, time, string
from BNF import BNFgrammar, BNFError
import GEbnf
import GEinput, GEparameters, GEbasic

class Settings(object):
    "Class to encapsulate the different settings, and allow them to be refreshed at run time"
    def __init__(self):
        "Get the initial settings and inputs for the GE"
        self.files = {}  # first obtain the basic settings that determine where the settings are to be loaded from
        self.log = logging.getLogger('GE')
        self.log.setLevel(logging.DEBUG)
        self.fileHandler = logging.FileHandler(self.PATH + 'GE.log', 'a')
        self.log.addHandler(self.fileHandler)
        self.log.info('Logs setup
'
        "open the files needed to output statistics to"
        self.files['complete'] = open(self.PATH + 'Logging\GE_Complete.csv', 'a')
        self.files['summary'] = open(self.PATH + 'Logging\GE_Summary.csv', 'a')
        self.log.info('Files opened
'
        "Get the basic settings before anything else"
        self.BASIC = GEbasic.getBasic(self.PATH + 'GEbasic.xml')
        self.getBasic()
        self.setupLog()
        self.getFiles()

    def getBasic(self):
        "Get the basic settings before anything else"
        self.BASIC = GEbasic.getBasic(self.PATH + 'GEbasic.xml')

    def getFiles(self):
        "Get the settings and inputs for the GE from external files"
        self.log.info('GE Run started at ' + time.strftime('%Y-%m-%dT%H-%M-%S', time.localtime()))
        self.getBasic()
        self.setupLog()
        self.getFiles()

    def refresh(self):
        "save start time, for logging"
        self.now = time.strftime('%Y-%m-%dT%H-%M-%S', time.localtime())
        self.getBasic()
        self.setupLog()
        self.getFiles()

    def setupLog(self):
        "log any initial errors (those outside a GE run) to a global log file"
        self.log.info('Logs setup
'
        "Set (or reset) the log in preparation for a GE run"
        self.log.info('Logs setup
'
        "for non-absolute paths, prepend the current path"
        if self.BASIC[filename][1] == 'no':
            self.BASIC[filename][0] = self.PATH + self.BASIC[filename][0]
        else:
            self.BNF = GEbnf.getBNF(self.BASIC['grammar'][0], self.log)

    def getSettings(self):
        "Get the settings and inputs for the GE from external files"
        self.log.info('GE Run started at ' + time.strftime('%Y-%m-%dT%H-%M-%S', time.localtime()))
        self.getBasic()
        self.setupLog()
        self.getFiles()
```

```python
self.PARAM = GEparameters.getParameters(self.BASIC['paramfile'][0], self.log)
self.log.info('Inputs:
'
        for item in self.INPUT:
            self.log.info('Inputs:
'
        self.log.info('Parameters:
'
        for item in self.PARAM:
            self.log.info('Parameters:
'
        self.log.info('Grammar:
'
        for line in self.BNF.__str__().split('
'):
            self.log.info('Grammar:
'
        self.log.info('Logging:
'
        self.log.info('Grammar:
'
        exec('self.log.setLevel(logging.%s)') % self.BASIC['log_level'][0])
        else:
            self.log.setLevel(logging.DEBUG)
        self.log.info('Logs setup
'
        self.log.info('Logging:
'
        self.log.info('Logging:
'
        self.log.info('Logging:
'
        if self.BASIC['grammar'][2] == 'bnf':  # for non-absolute paths, prepend the current path
            self.BASIC[filename][0] = self.PATH + self.BASIC[filename][0]
        else:
            self.BNF = BNFgrammar(self.BASIC['grammar'][0])
        self.INPUT = GEinput.getInput(self.BASIC['inputfile'][0], self.log)
        self.log.info('Inputs:
'
        self.log.info('Parameters:
'
        self.log.info('Grammar:
'
        self.log.info('Grammar:
'
        exec('self.log.info('Logs setup
'
```

```python
# for non-absolute paths, prepend the current path
if self.BASIC[filename][1] == 'no':
    self.BASIC[filename][0] = self.PATH + self.BASIC[filename][0]
else:
    self.BNF = BNFgrammar(self.BASIC['grammar'][0])
```

```python
self.INPUT = GEinput.getInput(self.BASIC['inputfile'][0], self.log)
```

```python
self.log.info('Inputs:
'
        exec('self.log.info('Logs setup
'
```

```python
self.log.info('Logs setup
'
```

```python
self.log.info('Logs setup
'
```
```python
self.BASIC['log_level'].upper() + '}
else:
    self.fileHandler.setLevel(logging.DEBUG)  # record DEBUG and higher messages to the file, as well as warnings
    self.fileHandler.setFormatter(format)
    self.log.addHandler(self.fileHandler)

def cleanup(self):
    for file in self.files:
        self.files[file].close()
    self.log.removeHandler(self.fileHandler)
```
GEparameters.py

Imports settings from the parameters XML file. GEbasic.py, GEinput.py and GEbnf.py have very similar structure.

Contents:
- Classes
  - GEParameterHandler class (extends xml.sax.ContentHandler)
- Methods
  - getParameters method

```
GEParameterHandler
- readContent
- content
- type
- log
+ __init__()
- startElement()
- characters()
- endElement()
```
import xml.sax, string, sys, logging, decimal

# define a dictionary containing the default values for each parameter
parameters = {}
parameters['compare'] = 'minimize'
parameters['allelesPerCodon'] = 8
parameters['codonsPerGenotype'] = 10
parameters['popSize'] = 100
parameters['maxGenerations'] = 100
parameters['crossover'] = 0.90
parameters['crossoverType'] = 2
parameters['mutation_rate'] = 0.01
parameters['mutation_multiplier'] = 0.1
parameters['variable_mutation'] = 'no'
parameters['wrapNos'] = 5
parameters['steadyState'] = 'no'
parameters['steadyStateNo'] = 1
parameters['steadyStateReset'] = 0.01
parameters['tournamentsize'] = 8
parameters['chooseBest'] = 0.9
parameters['prune'] = 'no'
parameters['pruneProb'] = 0.5

def getParameters(filepath, log):
    """ Read input from an XML file, and substitute defaults where necessary """
    try:
        file = open(filepath)
    except IOError, e:
        # this is not a fatal error, just return the default parameters
        log.warning('File IO Error: Default parameters loaded')
    else:
        # initialise an XML parser
        handler = GEParameterHandler(log)
        p = xml.sax.make_parser()
        p.setContentHandler(handler)
        # parse the XML file
        try:
            p.parse(file)
        except xml.sax._exceptions.SAXParseException:
            log.warning('XML Parse Error: invalid file, default parameters loaded')
        finally:
            file.close()
    return parameters

class GEParameterHandler(xml.sax.ContentHandler):
    """
    Reads the input for the GE
    """
    def __init__(self, log):
        self.readContent = False
        self.content = '
        self.type = ''
        self.log = log # log errors for the user's information
    
    def startElement(self, name, attrs):
        # read in the values that the variables must take
        if name in
            self.readContent = True
            # for numeric values, convert to either float or int
            if self.type == 'int':
                content = int(self.content)
            elif self.type == 'float':
                content = float(self.content)
            elif self.type == 'inf':
                # penal values may be + or - Infinity in addition to floats or ints
                if self.content == '-

    content = -decimal.Inf
    else:
        content = decimal.Inf
    else:
        content = self.content.lower()
    parameters[name] = content
    self.readContent = False
    self.content = self.tmpNum = self.type = ''
GEinput Regresssion.xml

Defines the input details for Symbolic Regression, and how to determine the score of an individual.
<GE xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="GEinput.xsd">
<input>
  <import>math</import>
  <function type="compare">abs(phenotype-userfct)</function>
  <function type="user">X**4+X**3+X**2+X**1</function>
  <punishment>10</punishment>
  <value variable="X">
    <varvalue>-1.0</varvalue>
  </value>
  <value variable="X">
    <varvalue>-0.9</varvalue>
  </value>
  <value variable="X">
    <varvalue>-0.8</varvalue>
  </value>
  <value variable="X">
    <varvalue>-0.7</varvalue>
  </value>
  <value variable="X">
    <varvalue>-0.6</varvalue>
  </value>
  <value variable="X">
    <varvalue>-0.5</varvalue>
  </value>
  <value variable="X">
    <varvalue>-0.4</varvalue>
  </value>
  <value variable="X">
    <varvalue>-0.3</varvalue>
  </value>
  <value variable="X">
    <varvalue>-0.2</varvalue>
  </value>
  <value variable="X">
    <varvalue>-0.1</varvalue>
  </value>
  <value variable="X">
    <varvalue>0.0</varvalue>
  </value>
  <value variable="X">
    <varvalue>0.1</varvalue>
  </value>
  <value variable="X">
    <varvalue>0.2</varvalue>
  </value>
  <value variable="X">
    <varvalue>0.3</varvalue>
  </value>
  <value variable="X">
    <varvalue>0.4</varvalue>
  </value>
  <value variable="X">
    <varvalue>0.5</varvalue>
  </value>
  <value variable="X">
    <varvalue>0.6</varvalue>
  </value>
  <value variable="X">
    <varvalue>0.7</varvalue>
  </value>
  <value variable="X">
    <varvalue>0.8</varvalue>
  </value>
  <value variable="X">
    <varvalue>0.9</varvalue>
  </value>
  <value variable="X">
    <varvalue>1.0</varvalue>
  </value>
  <optimum>0</optimum>
</input>
</GE>
BNF\_Regression.xml

Defines the input grammar for Symbolic Regression, and how to determine the score of an individual.
Regression.bnf

The following is the text bnf file that defines the same input grammar as the XML file.

```
<start>  ::=<expr>
<expr>   ::=(<expr>)<op>(<expr>)|<pre-op>(<expr>)|<var>
<op>     ::=+|-|/|*
<pre-op> ::=math.sin|math.cos|math.tan|math.log
<var>    ::=X|1.0
```
**GEinput_SantaFe.xml**

Defines the input details for Santa Fe Ant Trail and how to determine the score of an individual (via external file SantaFe.py shown later).
<input>     <import>
SantaFe</import>
<function type="compare">ant.getFood()</function>
<function type="user">NullVariable</function>
<post-code>ant = SantaFe.ant(2); nant.interpret(phenotype)</post-code>
<punishment>0</punishment>
<value variable="NullVariable">
  <varvalue>1.0</varvalue>
  <fctvalue>1.0</fctvalue>
</value>
<optimum>89</optimum>
</input>
</GE>
SantaFe.py

Encapsulates an ant object to allow the trail to be evaluated.

Contents:

- **Classes**
  - ant class

- **Methods**
  - makeTrail method
  - getTrail method

<table>
<thead>
<tr>
<th>ant</th>
</tr>
</thead>
<tbody>
<tr>
<td>- orientation</td>
</tr>
<tr>
<td>- trail</td>
</tr>
<tr>
<td>- path</td>
</tr>
<tr>
<td>- position</td>
</tr>
<tr>
<td>- foodCount</td>
</tr>
<tr>
<td>- timestep</td>
</tr>
<tr>
<td>+ <strong>init</strong>()</td>
</tr>
<tr>
<td>- next_position()</td>
</tr>
<tr>
<td>- move()</td>
</tr>
<tr>
<td>- left()</td>
</tr>
<tr>
<td>- right()</td>
</tr>
<tr>
<td>- food_ahead()</td>
</tr>
<tr>
<td>+ interpret()</td>
</tr>
<tr>
<td>- _interpret()</td>
</tr>
<tr>
<td>+ getPath()</td>
</tr>
<tr>
<td>+ getFood()</td>
</tr>
<tr>
<td>+ getMoves()</td>
</tr>
<tr>
<td>+ getScore()</td>
</tr>
<tr>
<td>+ <strong>repr</strong>()</td>
</tr>
</tbody>
</table>
class ant(object):
    "Class to represent an ant on the Santa Fe trail"
    def __init__(self, trail):
        "Initialise an ant in position (0,0) facing 'down'"
        self.orientation = [1, 0] # 'down'
        # either a square trail encoded as a list of lists, or an
        # index number is accepted (a zero trail indicates error)
        if isinstance(trail, list):
            if len(trail)*len(trail) == len(row) for row in trail):
                self.trail = trail
            else:
                self.trail = makeTrail(0)
        else:
            self.trail = getTrail(trail)
        self.path = [[0, 0]]
        self.position = [0, 0]
        self.foodCount = self.trail[0][0]
        self.timestep = 0
    def next_position(self):
        "Return the co-ordinates for the next position to move to"
        return [(self.position[i] + self.orientation[i]) %
                len(self.trail) for i in [0, 1]]
    def move(self):
        "Move in the direction the ant is facing (wrapping as needed)"
        self.position = self.next_position()
        self.path.append(self.position)
        self.foodCount += self.trail[self.position[0]][self.position[1]]
        self.trail[self.position[0]][self.position[1]] = 0
    def left(self):
        "Turn the ant to its left"
        self.orientation = [-self.orientation[1], self.orientation[0]]
    def right(self):
        "Turn the ant to its right"
        self.orientation = [self.orientation[1], -self.orientation[0]]

def food_ahead(self, trueCommands, falseCommands):
    "Choose between two sets of commands, depending upon whether
    or not there is food in the next space"
    next = self.next_position()
    if self.trail[next[0]][next[1]]:
        self.interpret(trueCommands)
    else:
        self.interpret(falseCommands)

def interpret(self, commands):
    "Follow the sequence of ant commands passed as a list (Major
    version)"
    self.timestep = 0
    while self.timestep < 615:
        self._interpret(commands)
    def _interpret(self, commands):
        "Follow the sequence of ant commands passed as a list (Minor
        version)"
        for command in commands:
            if len(command) > 0 and self.timestep < 615:
                self.timestep += 1
                if command[0] == 'left':
                    self.left() 
                elif command[0] == 'right':
                    self.right() 
                elif command[0] == 'move':
                    self.move() 
                elif command[0] == 'if_food_ahead':
                    self.food_ahead(command[1], command[2])
    def getPath(self):
        "Return the ant's pathway as a list of 'co-ordinate' lists"
        return [point for point in self.path]
    def getFood(self):
        "Return the amount of food encountered"
        return self.foodCount
    def getMoves(self):
        "Return the number of moves the ant has made"
        return len(self.path)
    def getScore(self, penal = 100):
        "Return a score (with bias towards ants of greater than the
        penal size)"

if self.foodCount < penal:
    return (self.foodCount / penal)
else:
    return self.getFood

def __repr__(self):
    "Return the ant's pathway and quantity of food"
    return "Ant object, food = \' + str(self.foodCount) + \', path = \' + str(self.path) + \'\"

# Define trail tools to generate Santa Fe trails
#
import random

def makeTrail(foodCount = 89):
    "Return a 32x32 trail containing a specified number of pieces of food"
    trail = [[0 for i in range(32)] for j in range(32)]
    food = 0
    while food < foodCount:
        i = random.randrange(0, 32)
        j = random.randrange(0, 32)
        if trail[i][j] == 0:
            trail[i][j] = 1
            food += 1
    return trail

def getTrail(index):
    "Return a predefined trail (or a zeroed trail if the requested trail is not available)"
    if index == 1:
        return [large list – omitted, see code on CD]
    elif index == 2:
        return [large list – omitted, see code on CD]
    else:
        return makeTrail(0)
BNF.xsl

The XSLT file that allows a BNF XML file to be opened into a web browser and display the plain text BNF equivalent.
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/bnf">
    <html>
      <head>
        <title>BNF Grammar</title>
      </head>
      <body>
        <style>
          body {
            font-family: courier, monospace;
          }
          .terminal {
            color: blue;
          }
          .rule {
            color: red;
          }
        </style>
        <xsl:for-each select="rule">
          <b>
            <xsl:value-of select="@name"/>
          </b>::=
          <xsl:for-each select="option">
            <xsl:for-each select="term">
              <xsl:element name="span">
                <xsl:attribute name="class">
                  <xsl:value-of select="@type"/>
                </xsl:attribute>
                <xsl:if test="@type = 'rule'">&lt;</xsl:if>
                <xsl:value-of select="."/>
                <xsl:if test="@type = 'rule'">&gt;</xsl:if>
              </xsl:element>
            </xsl:for-each>
            <!-- place an option delimiter after every option except the last (i.e. between each option) -->
            <xsl:if test="position() != last()">
              |</xsl:if>
          </xsl:for-each>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
**BNF.xsd**

The XML style sheet that allows the grammar to be validated before being used.
C. Usage Guide

The following guide aims to provide instructions for the user to install and run this system. Since the system is aimed for a fairly technically minded user, a reasonable level of understanding is assumed.

C.1. Prerequisites

This system requires the latest stable version of Python (2.4.2) available from http://www.python.org/download/

This system also uses the following packages:

- NumPy - http://sourceforge.net/projects/numpy/
- Numeric - http://numeric.scipy.org/
- ptylab / matplotlib - http://sourceforge.net/projects/matplotlib
- Psyco - http://sourceforge.net/projects/psyco/

Psyco is only required if the run_psyco portion of the basic XML file is enabled, it is disabled by default. The others are only required for any graphing work. When running a basic system, specifying graphs = False runs a system that does not require any additional packages.

C.2. Installation

To install, simply copy the folder pyGE into a folder on the Python path or into the Python library folder. The Python path may be determined (in Windows) by typing SET PYTHONPATH at the command line, and the library is typically found at \Libs\site-packages\ within Python’s home directory.

C.3. Set up

The file GEbasic.xml is required by the system and tells it where to find all of the other set up files. This is located in the root of the pyGE folder. Modify this file to specify alternative locations for settings files.

For each problem, at least two files are required – grammar and inputs. The grammar may be in BNF or XML form, but XML is the preferred format. In addition, the parameters of the system may be overwitten for finer control over the system.

C.4. Running the System

In the Python interpreter, type:

import pyGE
pyGE.run()

The best individual at each generation will be printed to the console at each generation and a plot of the best and average scores will be displayed at the end of the run.
C.4.1. Additional Options
To run more than one run of the system, type `pyGE.run(repeats = r)` where 
r is the number of times to run the system. The plot of results plots points from all runs, with an average line.

To run without plotting a graph, type `pyGE.run(graph = False)`. This does not require the use of any packages that are not already present in the default Python installation.

C.4.2. Access to the Supporting Files
To work with the grammatical evolution classes, type `import pyGE.GE`. The classes are available from `pyGE.GE.GE` and `pyGE.GE.GEnv`.

To work with the underlying genetic algorithm, import `pyGE.pyga` and use `pyGE.pyga.Individual` and `pyGE.pyga.Environment`.

To run the unittest testing structure, import `pyGE.GEtest` and call `pyGE.GEtest.run()`.