JAZZ by Evolution

Christopher Harte
Langwith College
University of York
June 2001
4th Year MEng Project Report

Abstract

The purpose of this project is to investigate whether Evolutionary Computing techniques can be used to compose Jazz. The project combines Jazz, Evolutionary Computing techniques and Algorithmic Composition. A thorough review of these subjects is given in the first part of the report. The design of a system for composing Jazz using Evolutionary Computing techniques and the details of how it was implemented are presented. The preliminary test results from the system are analysed and then various areas of further work are discussed. The report finishes with a short conclusion.

Contents

1 Preface

Jazz by Evolution. It should be the title of an album. Maybe in the future it could be…

In this report I have tried to explain clearly, with the aid of diagrams where possible, the way in which the Jazz by Evolution system works. This report has been written with the continuation of "Jazz by Evolution" in further projects in mind.

On the whole, I am pleased with how the project has progressed. The majority of aims and objectives that were set have been met and those areas requiring further work and investigation have been addressed in Section 7. I hope that any student who continues with part or all of the project in the future will find the report useful and accessible as a start point for their work.

2 Introduction

The aim of this project is to investigate the ways in which evolutionary computing techniques can be applied to the task of composing Jazz music.

Most approaches to computer music composition are based on statistical methods, which are hard-coded by the designer of the system. This often leads to the music sounding unnatural. In an evolutionary system, the start point is randomness. The system refines the random material over a large number of iterations; a process similar to the way we compose.

The report begins with a review of the various disciplines that the project combines. Section 3, *Background Research,* is split in to four main parts: *Jazz*, *Evolutionary Computing Techniques*, *Algorithmic Composition* and *MIDI and Standard MIDI Files.*

In Section 3.1, defining features of Jazz such as Swing and Jazz Harmony are discussed.

Section 3.2 gives an introduction to Evolutionary computing techniques and suggests why they might be well suited to use in the composition of music. There is then a more detailed look at Genetic Algorithms and Genetic Programming.

The various techniques used in Algorithmic Composition are discussed in Section 3.3. These include the application of Generative Linguistics Theory to music in order to define a model for how we perceive structure in pieces. Other commonly used techniques are investigated then a review of current evolutionary composition systems is presented. This is followed with a discussion of the most challenging part of any evolutionary composition system: *The Fitness Function.*

Section 3.4 is an introduction to the structure of Standard MIDI files.

In Section 4, *A System for Composing Jazz by Evolution,* a design for an evolutionary composition system is proposed. The section starts with an overview of how such a system will operate.

Section 4.2 introduces an Abstract model for a piece of Jazz based on the ideas researched in Section 3. This is followed, in Section 4.3, by a design for the Genetic Representation of a piece using a two-level evolutionary system incorporating both Genetic Algorithms and Genetic Programming. The way in which the Genetic Programming Language will work is then described in further detail in Section 4.4. Section 4 concludes with an analysis of a real piece of music, which is used to code a representation of that piece using the representation scheme that has just been defined.

The implementation of the composition system in software is described in Section 5. The various types and classes used in the software are explained in detail in Sections 5.1 through to 5.6.

The progress of the project and the results that have been obtained from tests are evaluated in Section 6. The most recent set of test results is analysed in detail and some preliminary conclusions are drawn from them.

Section 7 of the report presents areas of further work and how they may be approached as a set of possible starting points for future projects.

Section 8 is the Final Conclusion, followed by Acknowledgements, References and Appendices.

3 Background Research

This section outlines the various areas that have been researched for the project. It is split in to four main sub-sections: *Jazz* (3.1), *Evolutionary Techniques in Computing* (3.2), *Algorithmic Composition* (3.3) and *MIDI and Standard MIDI Files* (3.4).

Section 3.1 gives an overview of why Jazz is suited to use in an evolutionary composition system and details the aspects of Jazz theory relevant to this project. Section 3.2 gives a comprehensive background of Evolutionary Computing Techniques and explains how they work. Section 3.3 gives an overview of Algorithmic Composition and a description of some of the models and techniques commonly used in the field. The use of evolutionary techniques in algorithmic composition is then addressed in more detail with discussion of both the advantages and disadvantages of the paradigm.

Section 3.4 gives an overview of how the Standard MIDI file is arranged.

3.1 Jazz

Jazz began to emerge as a musical style in America during the mid to late part of the nineteenth century. The city generally held to be the birthplace of jazz is New Orleans, where at that time the various strands that went to form jazz were gathering together. New Orleans was a flourishing seaport, which had become home to many former slaves. West African music, with its complex rhythms, became mixed with European musical influences, particularly the melodies and strong basic pulses of marches and the harmonic base of hymns. This mixture of musical cultures first gave rise to the Negro spiritual and blues, and then developed into the style of music now known as jazz [10, 11, 32, 34].

Much of jazz is based on improvisation. The style utilises standard musical elements such as 4/4 time, songs of uniform length and form (usually 32 bars long with an A-A-B-A form), consistent and logical harmonies, stylised melodies and rhythms, and even an established order of introductions, statements of themes, sequence of soloists, and codas and endings. Many jazz bands play the "standard" tunes. Jazz standards are well-known pieces, which are used to provide a familiar framework for improvisation. Such pieces can be found in books of standards scored as a melody line and the chord progression that accompanies it (see figure 3.1). This is, in general, all the information necessary for a musician to give a rendition of the piece.

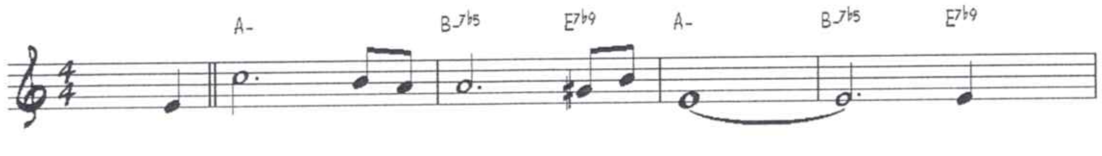


Figure 3.1: An extract from the jazz standard "Black Orpheus" by Louis Bonfi

Most musicians agree that the most important aspect of jazz is "swing" [10, 32] . When asked to explain what swing is, however, a definitive answer is hard to find. Much of jazz is approximated in notation as 4/4 or duple meter but it is really more complex. The actual timing is something nearer 12/8 as swung quavers are played as a crotchet-quaver pattern (see figure 3.2). However, the degree of swing (the ratio of the swung-pair rhythms)is variable; it is not fixed at 2:1.

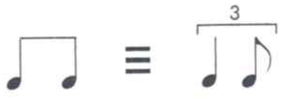


Figure 3.2 A swung pair of quavers is approximately equivalent to a triplet crotchet quaver

Jazz harmony uses many different chord types and many different extensions of those types. There are several families of chords: the *Major type chords*, the *Minor type chords*, *Dominant type chords*, *Minor 7th type chords* and *Diminished type chords* [10,11]. For each chord from each family there is a set of extensions that can be added to it within the rules of harmony (technically, any extension can be added to any chord but some will just sound horrible). The extensions commonly used in jazz are the *9th*, *11th* and *13th*. Figure 3.3 shows the different chord types and their extensions for chords with the root of C.

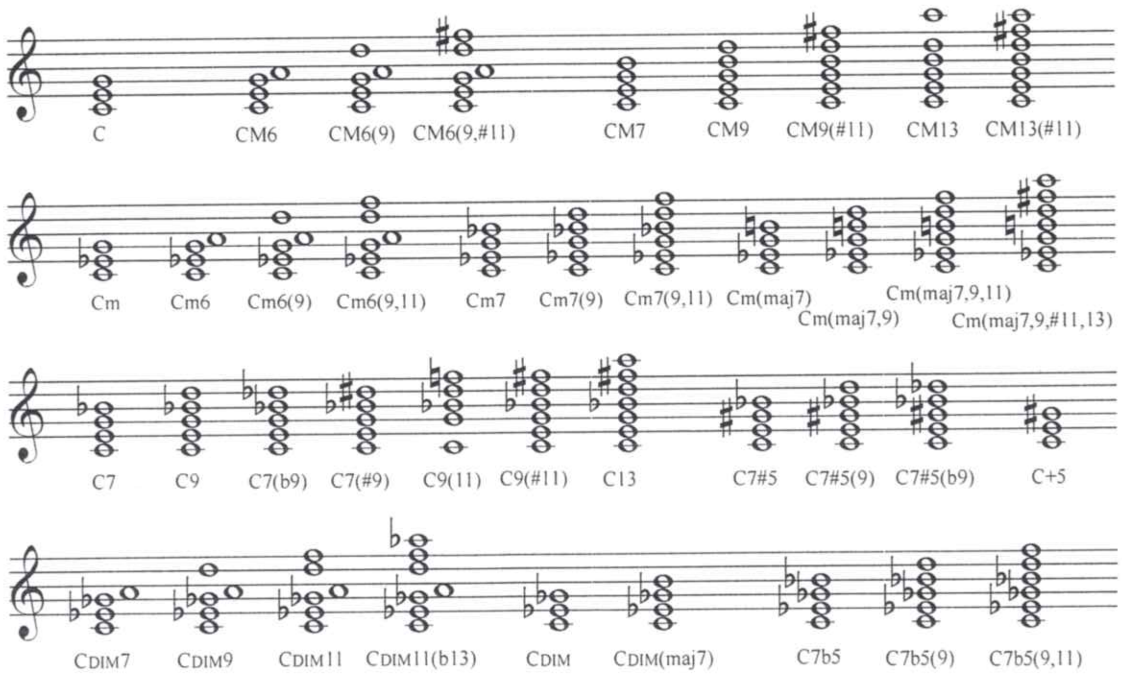


Figure 3.3: The standard jazz chord types and their extensions

The families of chords perform specific functions in the harmonic structure of a piece. This concept is known as *functional harmony*. The tonic chords offer rest, whereas the dominants (chords with the 5th as their root) and minor sevenths create tension which need to be resolved. Shown below is the most common function of each chord type. The V^7, II^{m7} and I^{M7} chords (1,2 and 3) comprise approximately 75% of all chords.

1. V^7	Dominant of I
2. II^{m7}	Functions as subdominant of I, precedes dominant, and is substitute for IV
3. I^{M7}	Tonic
4. VI^7	Precedes II
5, III^{m7}	Substitutes for I. Often follows V^7
6. VI^{m7}	Substitutes for I. Often follows I or occurs between III and II
7. I^7	Dominant of IV
8. IV^{M7}	Tonic relief. Temporary (usually key center)
9. V^{m7}	II^{m7} of IV. Usually precedes I^7 (dominant of IV)

3.2 Evolutionary techniques in computing

"I could see evolution as a creative process, the essence of making something out of nothing"

John Holland

Taken from an interview in Steven Levy's book *Artificial Life: The quest for a new creation* [26], this quote identifies the underlying idea behind the use of evolutionary computing techniques. The notion of making *something out of nothing* is an extremely attractive one; you can start with an unformed structure, replicate the machinery of nature, and let behaviour *emerge*. In the context of computer-generated music, this can provide an approximation to the human creative process (see section 3.3).

In both biological and artificial systems, the information central to an organism has to be regarded in two manners - both as genetic information to be duplicated and as instructions to be executed. The information form of the individual is known as the *genotype* and the expression of those genes in a physical manifestation of the organism is known as the *phenotype*. The most important aspect of any evolutionary system is the way in which the problem to be solved is represented.

3.2.1 Genetic Algorithms

The Genetic Algorithm (GA) was first proposed in 1975 by John Holland in his book *Adaptation in Natural and Artificial Systems* [15]. Genetic algorithms are search algorithms based on the mechanics of genetics and natural selection. They are particularly suited to searching complex non-linear search spaces; they are generally fast and very robust. These properties have led them to be used for a wide variety of practical applications including scheduling, financial modelling and optimisation.

A genetic algorithm (or any evolutionary algorithm) for a particular problem must have the following five parameters:

- a genetic representation for potential solutions to the problem,
- a way to create an initial population of potential solutions,
- an evaluation function that rates the solutions in terms of their 'fitness',
- genetic operators which alter the composition of children,
- values for various parameters used by the algorithm (population size, operator rates, etc)

A random population of individuals is generated. An individual is an encoding of a possible solution from the solution space in the form of a 'chromosome' (a string of genes). Each individual is rated using a fitness function (a test to grade how well the solution coded by the individual's genome solves the given problem). The individuals fitness value determines how likely it is to be chosen as a potential parent candidate for the next generation. Figure 3.4 shows the cycle of the genetic algorithm.

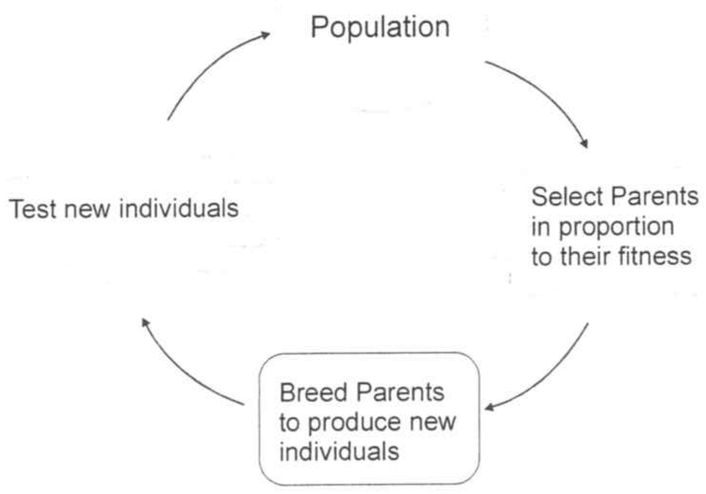

Population
Select Parents in proportion to their fitness
Breed Parents to produce new individuals
Test new individuals

Figure 3.4: The genetic algorithm cycle

A simple genetic algorithm that yields good results is composed of three operators: Reproduction, Crossover and Mutation [12].

Reproduction is the operation of selecting individuals from the current population to be carried in to the next as parents. There are several methods of selection. It is possible to pick just the top scoring candidates from the population. This approach may give a reasonable solution fairly fast but the search will tend to converge quickly, possibly missing better areas of the search space due to the effective 'inbreeding' caused by limiting the population diversity.

A better solution (and the one most widely implemented) is the *roulette wheel* model [29]. Using this approach, the probability that an individual is picked to be a parent in the next generation is equal to the individual's normalised fitness. The process can be visualised as spinning a roulette wheel with unequal segments representing each member of the population. A highly fit individual will occupy a large section of the roulette wheel and hence have a high chance of being chosen (see figure 3.5). A low scoring individual will occupy a small section so will be less likely to be chosen.

Allowing less fit individuals the chance to breed on to future generations provides the diversity the system needs for searching effectively. It is possible that an unfit individual may contain a sequence of genes that when bred with a fit individual produces a super-fit child. It is therefore important that such individuals are not lost at an early stage.

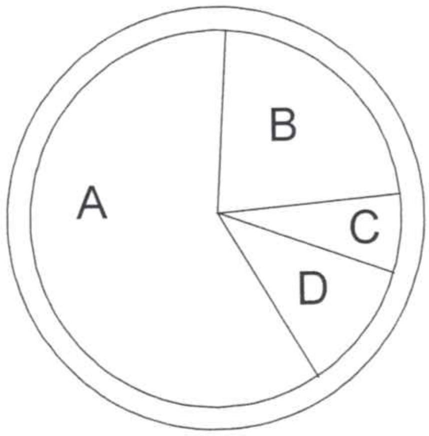


Figure 3.5: Roulette wheel selection. Individual A has the highest fitness therefore it takes up the largest area of the wheel

A third method of selection is that of tournament selection [22]. In this method of selection, candidates are picked from the population at random and pitted against each other in a fitness contest. The winner becomes a parent candidate for the next generation. This method of selection retains diversity as the individuals selected for a tournament may both be of relatively poor fitness yet one still wins the contest and becomes a parent candidate.

Once a group of individuals has been selected, they are copied and subjected to the operation of crossover. They are arranged in to arbitrary pairs. Each pair produces two new individuals that are made up of parts of both parents.

The simplest form of crossover is one-point crossover. For a representation using a string of length l for each individual, a random integer value i is selected in the range $[1, l-1]$. The two new strings are created by simply swapping the characters of the parents between $i+1$ and l inclusively (see figure 3.6 overleaf).

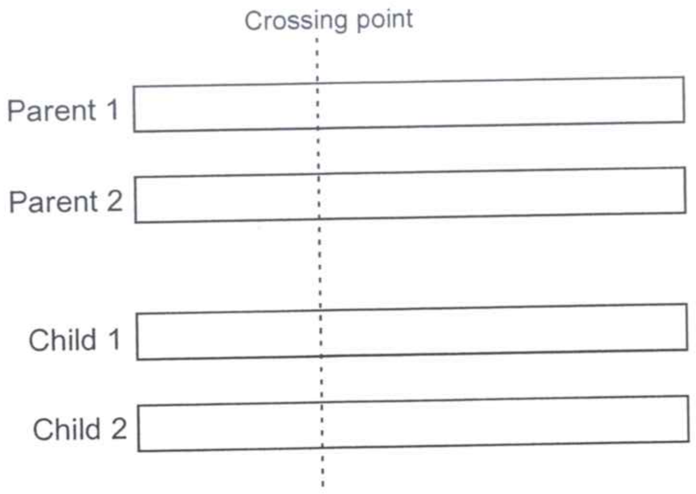


Figure 3.6: One point crossover

One-point crossover tends to give the endpoints of a string biased treatment as these are always swapped between parents. This is known as the *endpoint effect*. Another problem is *hitchhiking* where a low fitness gene is carried along with a high fitness schema (schemata are discussed later in this section) with little chance of being removed.

Using more than one crossover point helps solve these problems and reduces the importance of the gene order on the chromosome. A schema with genes at either end of the chromosome would have little chance of survival with one-point crossover but stand much more chance with two or more.

Nature is a chaotic environment. Mutation is an inevitable result of operating in such an environment; there is a probability that in copying a gene, the process will fail to copy it exactly.

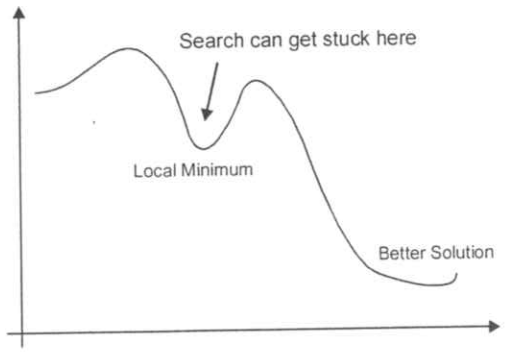


Figure 3.7: local minimum in a search

Mutation in genetic algorithms is used to retain diversity where a population may tend to converge on a certain point in the solution space. In a complicated search space there may be many local minima in which the algorithm can become stuck (see figure 3.7).

The operation of mutation ensures that a population will not stagnate in such local minima by injecting new genetic material. In the case of binary string representation, this process consists of merely flipping one or more random bits on the chromosome (see figure 3.8).

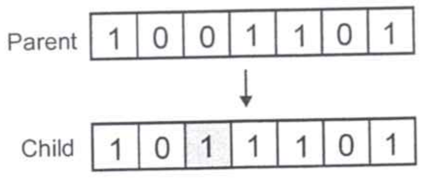


Figure 3.8: Mutation

In a representation using a higher cardinality alphabet, mutation will change a randomly selected gene to another symbol in the alphabet. In some cases, choosing any other possible value at random may be sufficient. In many practical cases, however, mutation consists of taking a new value from a statistical distribution around the current value. The idea being that, in most instances, mutation will produce a small change in an allele (a gene in a particular position on the chromosome), effectively searching a small neighbourhood in the search space [27, 29]. In this way, if the search is close to a potentially optimal solution, the mutation operation will tend to move it around in that same area, increasing the chance of finding that solution rather than moving it away in large steps.

Why does the Genetic Algorithm work?

One theory is based on the concept of *schemata* [12, 15]. A schema is a similarity template that describes a subset of strings with similarities at certain string positions. To examine the schema theory, let us consider a genome encoded with the binary alphabet {0,1}. By introducing a third "don't care" state, *, in to the alphabet it is possible to create strings (schemata) over the ternary alphabet {0,1,*}. A schema can be thought of as a device for pattern matching: a schema matches a particular string if at every location in the schema a 1 matches a 1 in the string, a 0 matches a 0, or a * matches either. For example, consider strings and schemata of length 5. The schema *0000 matches two strings: 10000 and 00000. The schema *101* matches four: 01010, 01011, 11010, and 11011.

The number of possible schemata in the example above, where $l = 5$, is $3^5 = 243$ as each of the five symbols may be a 0, a 1 or a *. In general, for an alphabet of cardinality k there are $(k+1)^l$ schemata, compared to number of possible strings k^l. This result shows that we have actually access to more information than is initially apparent.

To understand the way this information is processed, the effect of the genetic operators on schemata must be considered. Since a highly fit string stands more chance of selection, an ever-increasing number of samples is given to the best schemata that are observed; reproduction alone, however does not sample new points in the space. Crossover leaves a schema intact if it does not cut the schema, but it may disrupt the schema when it does. For example the schema 1***0 is more likely to be destroyed during crossover than **10*. As a result, schemata of short defining length are left alone by crossover and reproduced at a high sampling rate by reproduction. Mutation at normal, low rates does not disrupt a particular schema very frequently. From this, it can be seen that highly fit schemata of short defining length (known as *building blocks*), are propagated generation to generation by giving exponentially increasing samples of the observed best. This all occurs in parallel with no special bookkeeping or memory other than the population of size n. The number of schemata processed usefully in each generation is approximately n^3 [12], which compares favourably with the number of fitness evaluations (n). This processing power, apparently unique to genetic algorithms, is known as *implicit parallelism*.

Goldberg [12] offers a more mathematical treatment of schema theory. The order of a schema, $o(H)$, is the number of fixed positions on the schema (e.g. 1*1** has order 2 whereas 1*010 has order 4). The defining length of a schema, $\delta(H)$, is the distance between the first and last specific string position (e.g. 10*1* has defining length of 3).

At a given time step, t, if there are m examples of a particular schema H, in a population $A(t)$, we can write $m = m(H,t)$. During reproduction, strings are selected according to their fitness. The effect of reproduction on the expected number of schema in the next generation can be expressed thus:

$$m(H,t+1) = m(H,t)\frac{f(H)}{f'},$$

where $m(H, t + 1)$ is the expected number of examples of schema H at time $t + 1$, $f(H)$ is the average fitness of strings representing the schema H at time t and f' is the average fitness of the population.

The probability, p_s ,that a particular schema survives simple crossover can be expressed as:

$$p_s \geq 1 - p_c \cdot \frac{\delta(H)}{l-1},$$

since the schema is likely to be disrupted when the crossover point lies within the defining length $(l - 1)$.

In order for a schema to survive mutation, all the specified positions must themselves survive. Mutation is the random alteration of a single position with probability p_m. As a single allele survives with probability $(1 - p_m)$, and since each mutation is statistically independent, a schema survives when each of the $o(H)$ fixed positions within the schema survives. Multiplying the survival probability $(1 - p_m)$ by itself $o(H)$ times gives the probability of surviving mutation, $(1 - p_m)^{o(H)}$. For small values of p_m ($p_m << 1$), this may be approximated by the expression $1 - o(H)\cdot p_m$.

Therefore, the expected number of copies a particular schema H receives in the next generation is given as the number of representatives after reproduction multiplied by the probability of surviving both crossover and mutation. This is given by the following equation (ignoring small cross-product terms):

$$m(H,t+1) = m(H,t)\frac{f(H)}{f'}\left[1 - p_c \cdot \frac{\delta(H)}{l-1} - o(H)p_m\right].$$

From this result, it can be seen that fit, short, low-order schema receive exponentially increasing trials in subsequent generations. These small 'building blocks' contain code patterns beneficial to the search. Solutions eventually become made up entirely of these building blocks. The crossover and mutation operators favour the fittest blocks until the strings are composed entirely of fit blocks; the less fit blocks having succumbed to disruption. The schema theorem is the most widely quoted explanation for the effectiveness of genetic algorithms.

3.2.2 Genetic Programming

The representation used in an evolutionary system can limit the window by which the system observes its world. A limitation of the Genetic Algorithm is that a fixed-length string is generally used to encode a representation of the problem to be solved. Such a representation scheme cannot vary dynamically to suit the requirements of the problem at hand.

Genetic Programming is a technique, developed by John Koza [21], to breed computer programs genetically [22, 28]. Koza's system uses the computer language LISP as the medium for the programs it creates and applies the same techniques of genetic algorithms to create those programs. The process of solving problems is reformulated as a search for a highly fit individual computer program in a space of possible computer programs.

Instead of identifying the units of crossover as single characters, genetic programming uses symbolic expressions (S-expressions) written in the LISP syntax. S-expressions are made up of mathematical functions and inputs appropriate to the problem; they are essentially sub-routines, which are commonly viewed as tree structures.

The set of possible structures in genetic programming is all the possible programs which can be composed from the set of N_F functions $F = \{f_1 f_2, \ldots f_{NF}\}$ and the set of N_T terminals from $T = \{a_1, a_2, \ldots a_{NT}\}$. Each particular function in the function set takes a specified number of arguments $z(f_i)$.

The functions in the function set may include:

- arithmetic operations (+, -, ×, / ,etc),
- mathematical functions (sine, cosine, exponential, etc),
- Boolean logic functions (`AND`, `OR`, `NOT` etc),
- conditional operators (`IF-THEN_ELSE` etc),
- functions causing iteration (`DO-UNTIL` etc),
- functions causing recursion,
- any other problem-specific functions that may be defined.

Terminals are either variable atoms (possibly representing inputs or other variables of a system) or constant atoms (such as number constants or Boolean constants).

Consider the function set:

F = {AND, OR, NOT}

and the terminal set:

T = {D0, D1}.

As an example, consider the XOR function (see truth table in Table 3.1):

input a	input b	output
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.1: Truth table for the XOR function

This function returns TRUE if either input a or input b is true but not both. This function can be expressed by the following LISP S-expression:

```
(OR (AND (NOT D0) D1)(AND D0 (NOT D1))).
```

Figure 3.9 shows the corresponding tree structure for this expression.

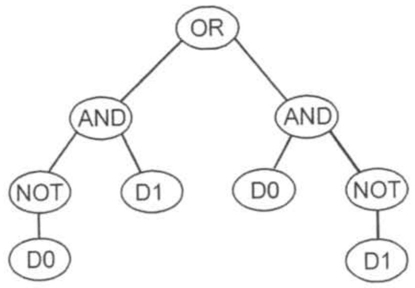


Figure 3.9: XOR function

In genetic programming, the terminal set and function set should be selected as to satisfy the requirements of *closure* and *sufficiency*. Closure is the requirement that each of the functions in the function set be able to accept, as its arguments, any value and data type that may possibly be returned by any function or assumed by any terminal. This ensures that the programs created by the system will compile properly as any combination of functions and terminals is allowed. The Sufficiency property requires that the set of terminals and the set of functions be capable of expressing a solution to the problem. The user must know or at least believe that some composition of the functions and terminals available can yield a solution to the problem. If this requirement is not fulfilled, the space searched may not contain a solution to the given problem.

To perform crossover in genetic programming, two parent programs are selected from the population (using the usual fitness-based selection methods). A random point on each parent is chosen as the crossover point. The crossover fragment of a parent program is the entire subtree below the crossover point. These crossover fragments are swapped between the parents to result in two new programs which are the offspring (see figure 3.10).

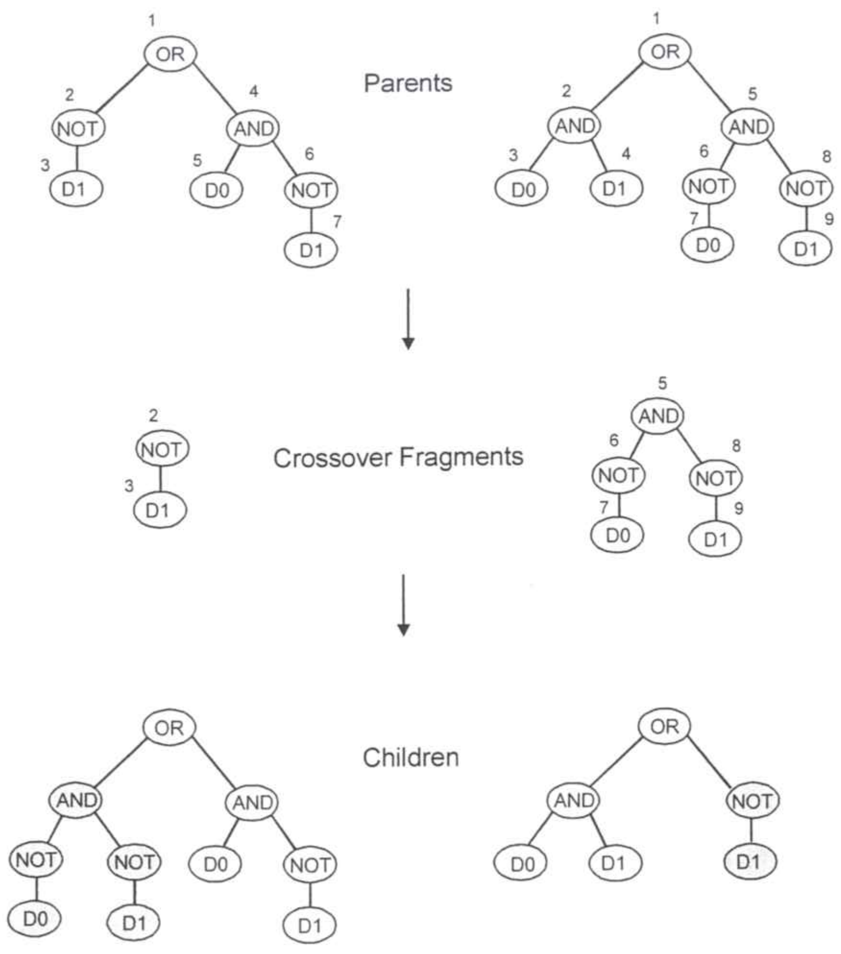


Figure 3.10: Crossover in tree structures of genetic programs

Mutation in genetic programming is regarded as a secondary operation. The mutation operation selects a point in the program (a function or a terminal) and replaces it with a new, randomly generated subtree.

The genetic programming paradigm is a never-ending process. However, in practice a run of the paradigm terminates when the *termination criterion* is satisfied. This criterion will either be that a specific maximum number of generations have been run or a problem-specific *success predicate* has been satisfied. The success predicate could be that an individual with a

100% fitness has been found. For problems where an exact solution is not expected to be found, some sort of appropriate lower criterion is applied.

Genetic programming is a very powerful paradigm, capable of handling a wide variety of problems. The system does have limitations however; because programs rely on incremental improvements in fitness, if there is no way of measuring partial fitness in solving the given problem, genetic programming cannot be applied.

3.3 Algorithmic Composition

There are two distinct types of creativity [18]: the proverbial bolt out of the blue (inspiration or genius) and the process of incremental revisions (hard work). As we do not understand genius, it is impossible to create a model for it. The goal of algorithmic composition, therefore, is to model the creative process of the composer when applying the 'hard work' approach to composition. This type of creativity often involves trying many different combinations against each other and choosing one over the others.

> "A sequence (set) of rules (instructions, operations) for solving (accomplishing) a [particular] problem (task) [in a finite number of steps] of combining musical parts (things, elements) into a whole [composition]"

The above is a [re]definition of Algorithmic composition as given by Cope [9]. Algorithmic composition is the application of rigid, well-defined rules to the process of composing music. It is often frowned upon by 'traditional' composers as it can be used as a means of expanding one's musical pallete. However, throughout history, when composing music, many composers have followed rigid rules; many compositions in classical music could be considered algorithmic compositions. Indeed, today's A-level Music syllabus includes a whole section on the mathematical rules for harmonisation of chorales in the style of J. S. Bach. Johann Kirnberger is regarded as having been the first author of an algorithmic composition system after publishing his paper "Der allezeit fertige Polonoison und Menuettencomponist" (The ever-ready Polonaise and Minuet composer) in 1757 [7]. The system comprises a fixed number of small musical pieces. These are arranged to form a longer composition by using a die to determine which one will be used as which section.

With the invention of computers in the twentieth century, the notion of automated composition became possible. Probably the name best known in the use of computers for composition is Iannis Xenakis [40]. Xenakis used computers to aid in the composition of scores and for live ensembles, using statistical and stochastic methods. His system, 'stochastic music program', would 'deduce' a score from a list of note densities and probabilistic weights supplied by the user, leaving specific decisions to a random-number generator. His 1962 work for four instruments, *Morsima-Amorsima,* was composed in this manner.

3.3.1 Models and Techniques used for Algorithmic Composition

In their book *A Generative Theory of Tonal Music* [24], Lerdahl and Jackendoff apply linguistics theory to music. Their theories are the based on the ideas of generative-transformational grammars for linguistics. Generative Linguistic Theory is an attempt to characterise what a person knows when they know how to speak a language; it is most widely known through the work of Noam Chomsky [8]. Chomsky argues that linguistic description on the syntactic level is formulated in terms of constituent analysis (parsing). Lerdahl and Jackendoff apply these theories to music showing how it can be split in to a set of hierarchical levels.

The lowest level in the hierarchy is a single note. This note has two fundamental attributes: a pitch and duration. A small group of notes form a motive (a short musical idea). Motives are the building blocks of melody. A motive is heard as part of a phrase, a phrase as part of a phrase-group (or section) and a section as part of a piece (see figure 3.11).

These different levels in the music operate in different temporal scopes.

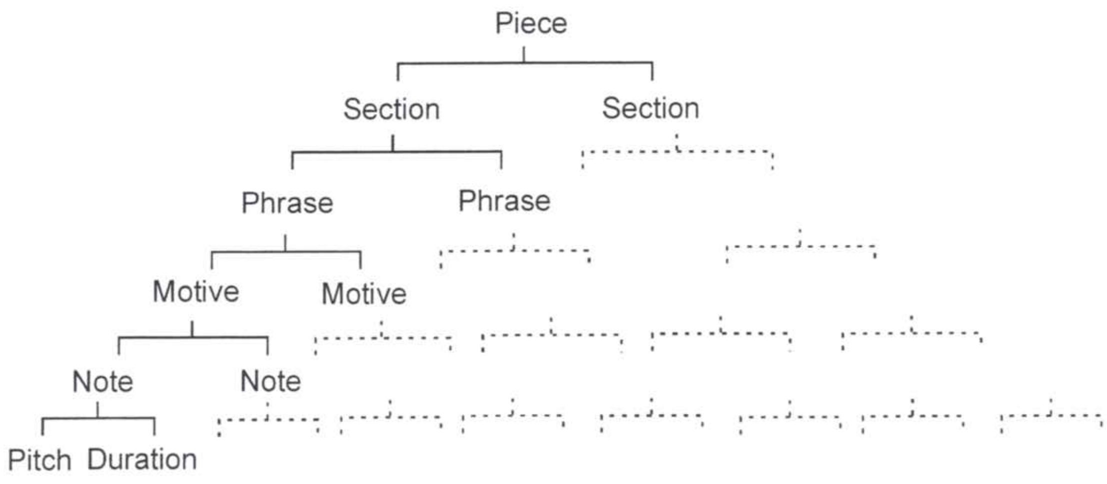


Figure 3.11: A piece of music modelled as a parse tree

Schema theory has also been applied to music. Lerdahl [25] presents a series of analyses showing the presence of underlying musical schemata in both phrase structures and harmony.

Stochastic algorithms are the most commonly used types of composition algorithm. They depend on the laws of probability, which makes it impossible to predict the precise outcome of the process at any point in the future [28]. The outcome of a stochastic process depends on certain underlying *probabilistic distributions*. This is a term used to describe how the probabilities are divided between the possible outcomes.

The simplest probability distribution is the *uniform distribution,* where the probabilities of all outcomes are equal (see figure 3.12 (a) overleaf). This can be implemented using a simple random-number generator.

For many cases in the composition of music, the uniform distribution is inadequate. Better models are provided by using alternative distributions such as *Linear distributions*, *Exponential distributions*, *Bell-shaped distributions* and *U-shaped distributions* (see figure 3.12 (b) to (e)).

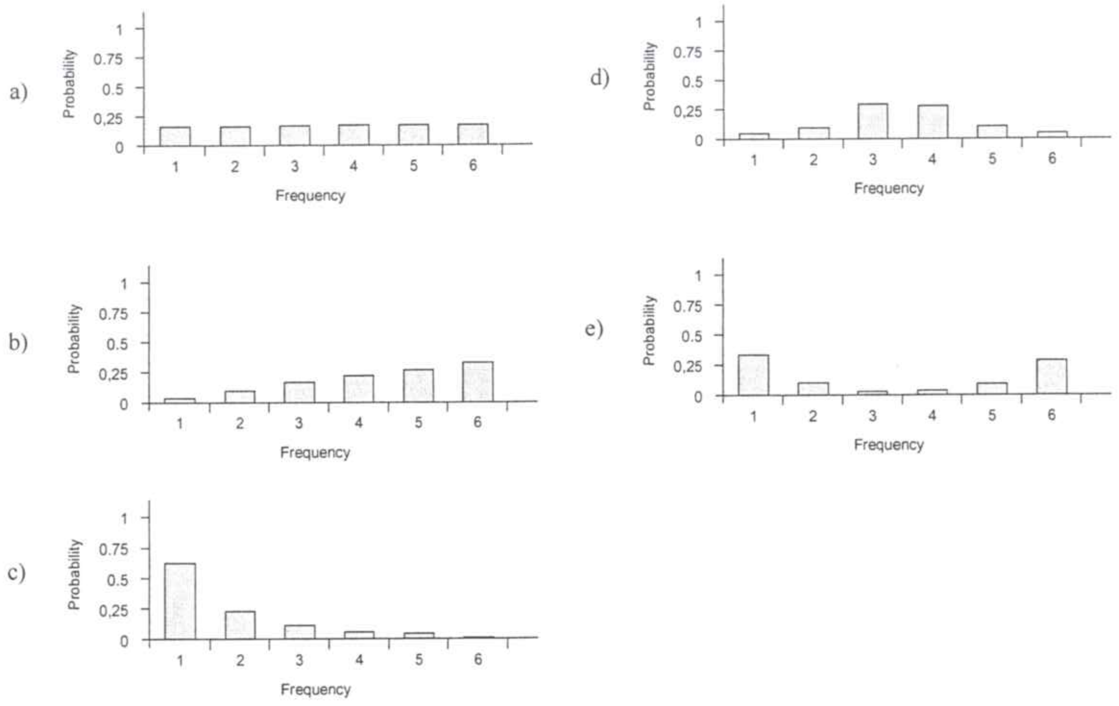


Figure 3.12: Different probability distributions: a) uniform b) line c) exponential d) bell-shaped e) U-shaped

Another stochastic process used in algorithmic composition is the *Markov Chain*. A Markov chain (or model) is a discrete probability system in which the probability of future events depends on one or more past events [28]. The number of past events that are considered at each stage is known as the *order* of the chain. An *Nth*-order Markov chain can, in general, be represented as a state-transition matrix – an $N + 1$ dimensional probability table. A state transition matrix for a possible first-order Markov chain is shown in Table 3.2. The state transition diagram for the same model is shown as a labelled, directed graph in figure 3.12 overleaf.

	A	B	C
A	0.8	0.2	0
B	0.1	0.5	0.4
C	0	0.5	0.5

Table 3.2: State transition matrix for a first-order Markov chain.

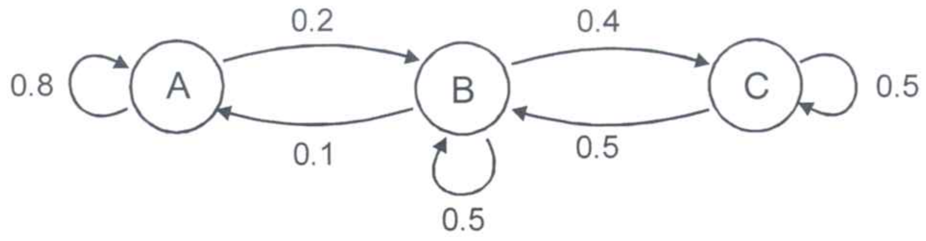


Figure 3.13: State Transition diagram for Markov model of Table 3.2

3.3.2 Use of Evolutionary Techniques in Algorithmic Composition

Evolutionary systems for algorithmic composition either evolve music generating "musician" algorithms [3,6,33,38] or evolve the piece itself [17].

John Biles' system, GenJam [3], is a GA-based model of a novice jazz musician learning to improvise. GenJam creates jazz-like solo improvisations from information relating to the chord structure of an existing piece of music. The performance of the hypothetical novice musician is judged by a human mentor and in real time. While judging a solo, if a particular section is judged to be good, the mentor can encourage the system by giving 'good' responses. When the output is less pleasing, this is registered using 'bad' responses. The fitness of a particular section in the music is the number of good responses given in that period minus the number of bad responses.

Biles emphasises the importance of having a good representation for a GA to function successfully. GenJam uses a cooperating, two-level (i.e. two temporal-levels see section 3.3.1), position-based binary representation scheme. It has two simultaneously evolving populations of individuals: one of phrases, one of measures. GenJam improvises a solo by building a sequence of MIDI notes from the decoded representations of measures[1] and phrases. A phrase individual consists of the indices of four measures. A measure encodes a series of eight events, one for each quaver in the measure. An event can be a new-note, a rest or a hold. There are 14 new-note events, which are mapped to MIDI pitches through scales suggested by the chord progression being played. This means that GenJam will not play a technically 'wrong' note in relation to the current chord. However, its choice of note may be bad in the context in which it is played. During a performance, all the members of both populations can be used. The phrases to be played are chosen by tournament selection considering both phrase fitness, and constituent measure fitnesses.

Spector and Alpern's system [33] evolves 'constructed artists' using Genetic Programming techniques. The genetic programs generated by this system take 'call' measure as the input and produce a 'response' measure as the output. Melodies are represented as vectors of (articulation, note) pairs, 48 per measure, where each articulation is above a threshold if a new note starts and below the threshold otherwise. Each note is a scaled MIDI note or a 0 for a rest. This allows for all standard durations down to 16th notes[2] and 32nd note-triplets[3]. The

[1] Measure is the American musical term for bar.
[2] 16th is the American musical term for semiquaver.
[3] 32nd is the American musical term for demisemiquaver.

genetic programs are built from eight different functions. These functions are operations that genetic programs produced by the system can perform on the 'call' measure to produce a 'response' measure.

To test the fitness of the artist programs, Spector and Alpern use a three-layer neural network with 192 inputs; one for each articulation and for each note over a two measure (call + response) fragment. The networks were trained using extracts from Charlie Parker solos.

Bruce Jacob's system [17] composes a piece by using GAs as data filters, which filter acceptable material output by a stochastic music generator (see section 3.3.1). It operates three GAs at different temporal levels; generating motives, building phrases from these motives and finally arranging sequences of these motives to form the structure of the piece. The three GAs are the *composer*, *ear*, and *arranger* modules; these modules are the genetic agents. The composer produces music, the ear filters out unsatisfactory material and the arranger imposes an order on what is left. The composer and ear agents are evolved until they cooperate to produce 'good' music (when they reach this point is determined by a human judge). The full system is then allowed to generate musical material, which can be judgedby a human operator.

The system designed by Johanson and Poli [19] allows users to evolve short musical sequences using interactive genetic programming. The system works by using a genetic programming algorithm, a small set of functions for creating musical sequences, and a user interface which allows the user to rate individual sequences.

Waschka's system, GenDash [39], is an evolutionary composition system in which, to avoid the 'fitness bottleneck' (see section 3.3.3), a random selection is applied to a population consisting of twenty-six bars of music. Waschka emphasises the importance of the initial population for this system though, regarding the initial input material as themes for the system to develop in an evolutionary manner. His work "Empty Frames" (1996) uses an initial population drawn from extracts of Beethoven's Symphony Number 7: Movement 2.

Thywissen's GeNotator [36] system takes a different approach. He treats a composition as a whole made up of interrelated parts that all play a part in the overall structure. A genetic algorithm is used to manipulate a variety of compositional structures within a hierarchical and generative grammar-based model of music composition. The grammar developed is claimed to be powerful enough to describe *deep structure* and *transformational rules.* The language includes musically useful constructs such as scales, keys, rhythms, phrases and larger structures relating to form. The evolutionary start point for the system is not random. The user creates a compositional structure, which is used as the genotype, then can interactively evolve it in to any number of phenotypes.

3.3.3 Fitness Functions in Music

The biggest problem in using evolutionary approaches for composition is the fitness test. The assumption is that the fitness test for most evolutionary algorithms is itself algorithmic. An algorithm that is capable of judging the quality of music would be the solution but since the appreciation of music is subjective, this is not an easy algorithm to produce. Many evolutionary composition systems use a human judge as the fitness test but, as evolutionary techniques usually function over thousands of generations, this is not the ideal approach. Biles [5] identifies this problem as the "fitness-bottleneck"; as human evaluation of all the pieces generated by these systems is extremely time consuming.

Some systems use deterministic fitness functions. Repetition of material is recognised as an important factor in the determination of musical structure. In many forms of music, including jazz, there are well-defined rules and forms that can be coded to use a rule-based fitness function.

One possibility is the encoding of preference rules [24, 35]: a set of rules which state certain underlying preferences which includes the *Strong-beat rule*, the *Harmonic-variance rule,* the *Compatability rule,* the *Pitch variance rule* and the *Length rule* amongst others.

The 'Strong-beat' rule states that we prefer chord spans to begin on the strong beats (beats 1 and 3 in a 4 beat bar). The 'Harmonic-variance' rule states that we prefer roots such that roots of nearby chord spans are close together on the line of fifths (see below).

The line of fifths: ...Bb F C G D A E B F# C#...

The 'Compatibility' rule states that we prefer roots that result in certain pitch-root relationships (i.e. the tune fits with the chords). The 'Pitch variance' rule states that we prefer spellings for pitch events so that nearby ones are close together in the line of fifths. The 'length' rule states that we prefer structure that aligns strong beats with the onsets of longer events.

Other approaches have been investigated in making automated fitness functions for music. Several approaches have used neural networks [5, 19,33]. Neural networks can be useful as they can learn from human fitness choices. However, they do not model time well and tend to over-fit the training set [5] and fail to generalise (i.e. they will class pieces they already know as good and not recognise good new pieces). It is, however, possible that neural networks could play a useful part in an integrated approach to automatic fitness evaluation.

3.4 Standard MIDI files

MIDI stands for Musical Instrument Digital Interface. It is a serial interconnection standard for electronic musical instruments. It is used for both performance and recording purposes. The standard defines a keyboard mapping to 127 pitches [30] (see figure 3.14).

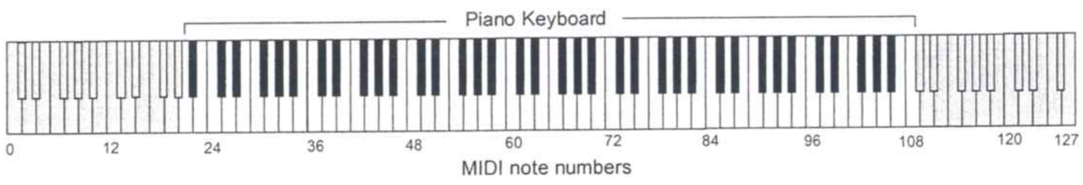


Figure 3.14: Full MIDI range shown on a keyboard

A steam of MIDI messages can be stored, along with related timing information, to form a MIDI sequence. This is the representation of a piece of music stored as a set of commands such as note-on, note-off, and program-change. The Standard MIDI file [37] is a file standard designed to provide a way of transferring time-stamped MIDI data between different programs or different computers.

MIDI files contain one or more MIDI streams, with time information for each event. Other information such as song, sequence and track structures, tempo and time signature is supported.

Some numbers in MIDI files are represented in a form called variable-length quantity. These numbers are represented with 7 bits per byte, most significant bits first. All the bytes except the last have bit 7 set; the last byte has bit 7 left clear.

A standard MIDI file is made up of chunks. Each chunk has a 4-character type and a 32-bit length. Numbers are stored most significant byte first (as the format was originally designed with Atari and Apple computers which use big-endian Motorola processors). This means a length of 6 is stored as 00 00 00 06.

MIDI files contain two kinds of chunks: header chunks and track chunks. A header chunk provides a small amount of information pertaining to the file as a whole. A track chunk contains a sequential stream of MIDI data which may contain information for up to 16 MIDI channels.

A MIDI file always starts with a header chunk, which is followed by one or more track chunks. (see figure 3.15)

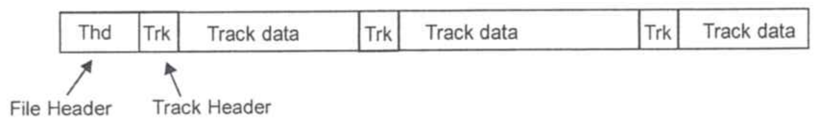


Figure 3.15: MIDI file structure

A track chunk is made up of a track chunk header and a series of track events. Each event has a delta-time before it. A delta-time is the amount of time between the previous event and the next event. Events are either MIDI messages (including system exclusive) or meta-events.

Meta-events are non-MIDI information useful to this format or to sequencers. They include time signature, key signature tempo events and text events such as lyrics.

4 A system for composing Jazz by evolution

This section describes the system that will be used to compose pieces of jazz using evolutionary computing techniques. It includes a brief overview of the entire system, a detailed look at the abstraction and genetic representation of pieces and an explanation of the language defined for the genetic program. The section concludes with an example of how a real piece of music might appear in this representation.

4.1 System Overview

The system consists of the population of pieces, the fitness test and three functional blocks: the Generator, the Breeder and the Renderer. Figure 4.1 shows the relationships between the parts of the system.

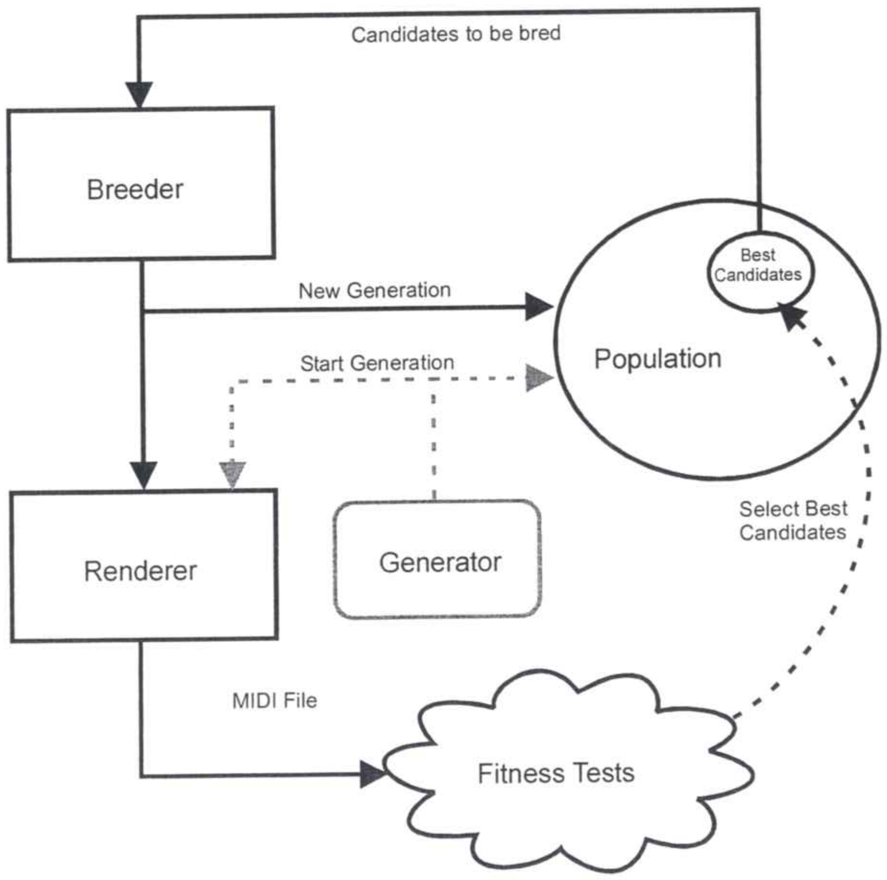


Figure 4.1: System Overview

The population is the set of pieces that make up the current generation g(i). The Generator produces a set of pieces which forms the starting generation g(0). It uses several different stochastic techniques (outlined in section 5.3) to generate new pieces.

The Breeder is the part of the system that breeds pieces together. The parent candidates from the current generation g(i) are bred together to create a new set of pieces which forms the next generation g(i+1).

The Renderer converts pieces from their genetic representations to MIDI files. The MIDI files can be scored out and played using a MIDI sequencer program (see Section 5.5). This makes it possible to apply the fitness criterion and allocate a fitness value to each piece. These fitness values are fed back in to the system to determine which candidates from the current population will be bred on to the next generation.

4.2 An Abstract Model of a Jazz Piece

Jazz standards (see section 3.1) are usually written out as a melody line (the head) and a chord chart that describes the harmonic structure of the piece. These two sets of information are normally all that is needed to describe a piece fully enough for a performer or a group to give a convincing rendition of it. As the purpose of the composition system is to create new pieces of Jazz music, it should produce pieces with two parts: the melody line and the chord progression.

As discussed in section 3.3, a successful composition system must function at several different temporal levels. An evolutionary composition system should therefore be capable of evolution at each of these levels. A model of a piece is needed that caters for the separate levels in such a way as to allow each one to evolve.

Motives are the underlying building blocks of the melody part. The better a piece's motives are, the better the melody line will be. Thus, the model must allow evolution of the motivic material that the piece is based on. Phrases are made up of groups of motives and variations of those motives. Evolution of the phrase structure therefore must also be allowed in the system.

The harmonic structure of a piece is built up of short chord progressions such as cadences and temporary modulations. The model must allow these progressions to evolve. Harmonic phrases are built from groups of these progression patterns and are generally longer than melodic phrases. Hence, as with melody, the system must allow evolution of the harmonic phrase structure.

The higher temporal levels in the piece are made up of groups of phrases. Since a group of phrases can be viewed as a large, subdivided group of motives, each of the higher levels is modelled by the evolving phrase structure (see figure 4.2 overleaf).

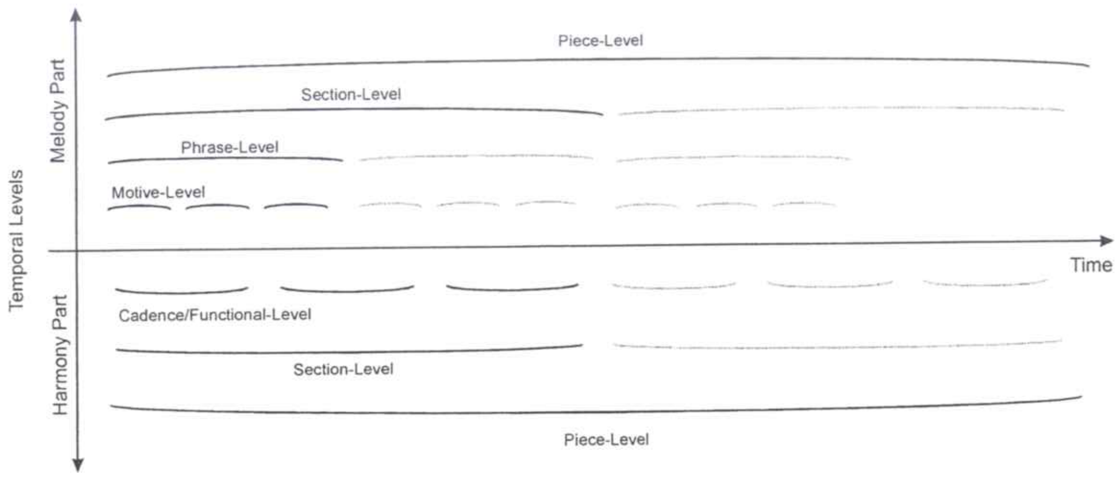


Figure 4.2: Stylised model for different temporal levels in composition

Using this model, the evolutionary system needs to have a genetic representation capable of evolving two independent musical models simultaneously. That is to say the motives and chord progressions are one type of model and the other type is phrase structure. A motive can not be bred with a phrase nor vice versa; the two are independent but each one needs the other to have any meaning. As with the notion of evolutionary algorithms in general, the solution to this problem is inspired by nature:

All living cells have their own DNA. Cells contain organelles: the units that perform cell functions. The nucleus is the largest cell organelle; it contains the main part of the cell DNA. Some other types of organelles contain their own DNA, which is separate from that of the nucleus. Figure 4.3 shows a typical animal cell containing a nucleus and several Mitochondria (a type of organelle).

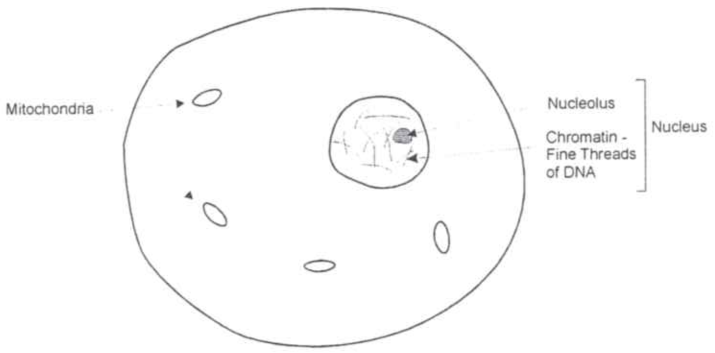


Figure 4.3: A typical animal cell

Mitochondria contain their own circular DNA molecule. When cells breed, the crossover of Mitochondria DNA occurs separately from that of the main cell DNA [13, 20]. The two exhibit a symbiosis that resembles the relationship between the models for motive and phrase evolution. A system using two separate evolutionary algorithms running in parallel can therefore be used to represent the model. This representation is presented in section 4.3.

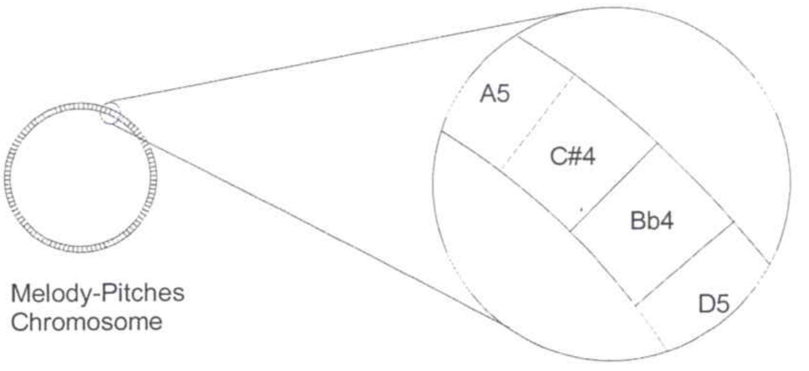
4.3 Genetic Representation of Pieces

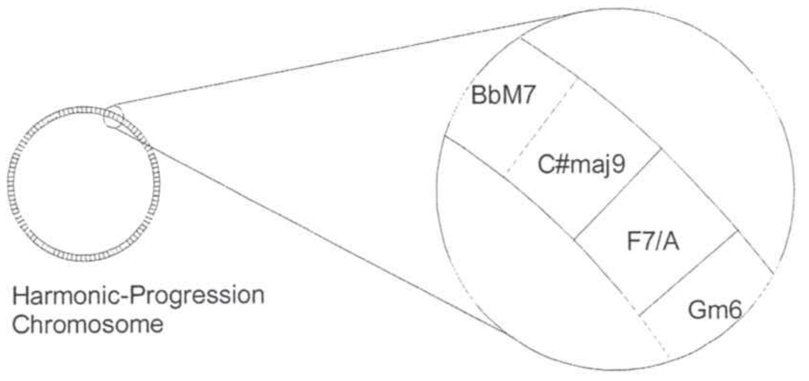
A Genetic Algorithm is used to evolve motive and harmony material. Motive material is represented as a sequence of pitches and a sequence of rhythm patterns. Harmonic material is represented as a series of chords and a sequence of rhythm of patterns. These representations form the 'chromosomes' of the GA and contain the melodic and harmonic information for the whole piece. The GA refines melodic and harmonic material through iterations of the system. A good section of one of the chromosomes, a nice sequence of intervals for example, stands a high chance of survival according to the schema theory (see section 3.2.1).

A Genetic Program is used to evolve the phrase structure of the piece. The Genetic Program consists of a string of 'operators' as opposed to the tree structure more commonly used in Genetic Programming [21]. The operators determine from which parts of the GA chromosomes to take information. Other operators handle variation of motives and chord progressions such as inversion and transposition (see Section 4.4).

The GP string is analogous to the DNA in the cell nucleus and the GA chromosomes are analogous to the DNA of the Mitochondria (from the example in the Section 4.1). The GA chromosomes continue the analogy further in that they are actually circular arrays that hold the motive, harmony and rhythm information (Mitochondria DNA is circular). This ensures that a motive will wrap round to the start if it extends past the end of the array. Figure 4.4 (overleaf) shows the structure of the GA chromosomes for motive material, harmonic progression material and rhythm material.

The chromosome for melody information is a sequence of note pitch-values. The chromosome for harmony is a sequence of chords. The chromosomes for rhythm are sequences of rhythm patterns. The rhythm chromosomes for melody and harmony are identical to each other.





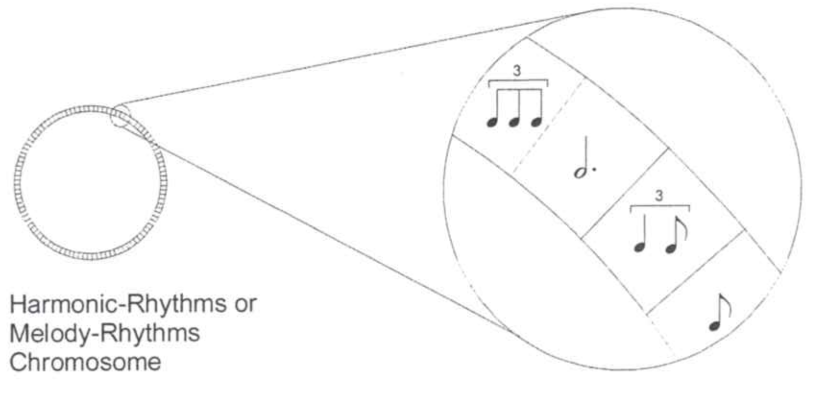


Figure 4.4: Structure of GA chromosomes.

The Genetic Program is stored as a comma-delimited string of characters and numbers. These characters are the functional blocks and operators of the language (see Section 4.4) and the numbers are parameters of those blocks and operators. Figure 4.5 shows how the different parts of the piece representation relate to one another.

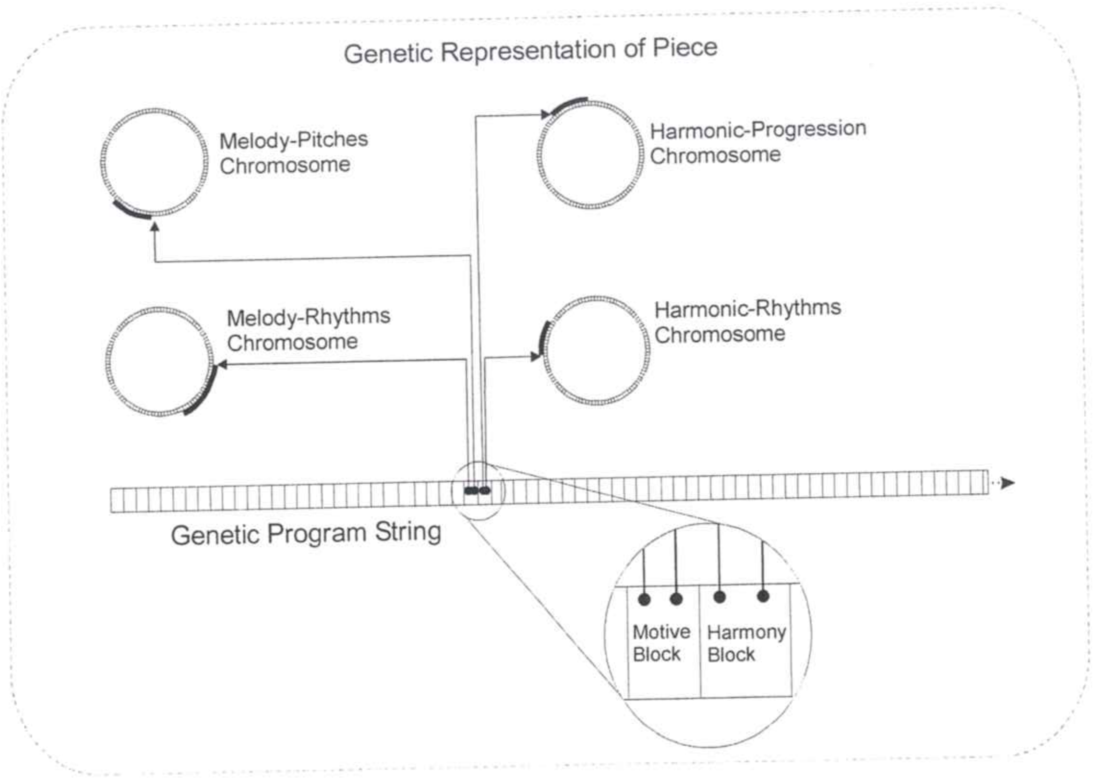

Genetic Representation of Piece
Melody-Pitches Chromosome
Harmonic-Progression Chromosome
Melody-Rhythms Chromosome
Harmonic-Rhythms Chromosome
Genetic Program String
Motive Block
Harmony Block

Figure 4.5: The relationships between different parts of the representation

To complete the piece, the representation must also store initial values for transpose, dynamic and tempo (see Section 4.4). These are stored separately from the genetic material of the GA and GP.

4.4 A Music Language for Genetic Programming

The language for genetic programming written for this composition system consists of two functional types: Harmony and Motive, and five operators: Transpose, Retrograde, Invert, Dynamic and Speed. Motive and Harmony blocks are the units from which the melody line and chord part are built respectively. The five operators defined in the language perform operations on these blocks.

The Harmony block type, H, associates a set of rhythms from the harmonic-rhythm chromosome with a short sequence of chords from the harmonic-progression chromosome to form a short section of the chord part. The parameters of the harmony operator are*duration, rhythm* and *progression.* The *duration* of a harmony block is its length in semiquaver beats, *rhythm* is the index of the start of the rhythm sequence on the harmonic-rhythm chromosome and *progression* is the index of the start of the chord progression on the harmonic-progression chromosome (see figure 4.6).

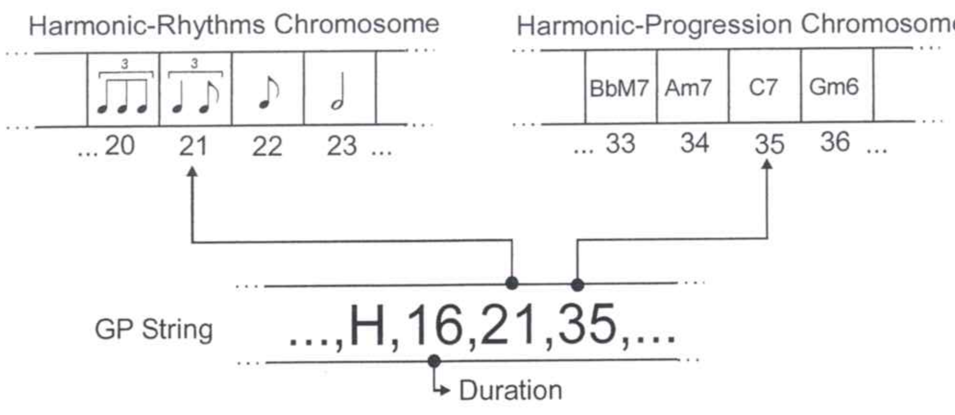


Figure 4.6: Harmony Block

The Motive block type, M, associates a set of rhythms from the melody-rhythms chromosome and a sequence of pitches from the melody-pitches chromosome to form a short section of the melody part. The parameters of the motive operator are *duration, rhythm* and *pitch*. The *duration* of a motive block is its length in semiquaver beats, *rhythm* is the index of the start of the rhythm sequence on the melody-rhythms chromosome and *pitch* is the index of the start of the sequence of pitches on the melody-pitches chromosome (see figure 4.7).

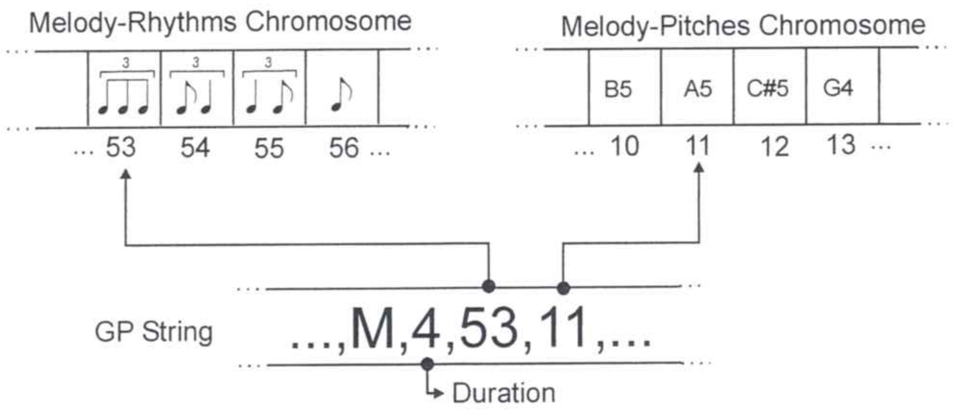


Figure 4.7: Motive Block

The Transpose operator, T, causes a section of the melody part or chord part to be transposed. It has a single parameter, *amount,* which is the number of semitones by which to transpose the section. The first block after a transpose command determines which part will be affected. If a transpose appears before a motive block, the pitches of that block and the following motive blocks will be transposed by that value. Likewise, if a transpose appears before a harmony block, its chord roots and subsequent harmony blocks will be transposed.

The Retrograde operator, R, causes the order of pitches or chords in the first block after itto be reversed (see figure 4.8).

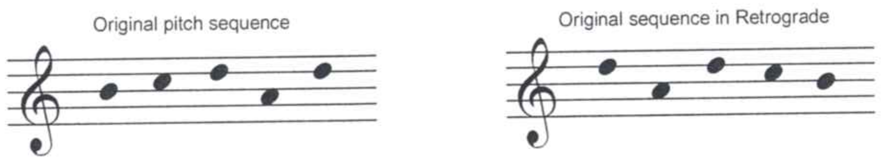


Figure 4.8: Retrograde

The Invert operator, I, causes the pitches of the first following motive block to be inverted around the first note of that block (see figure 4.9).

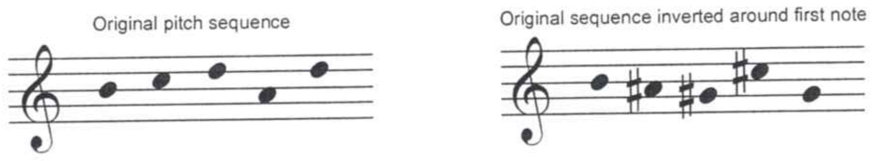


Figure 4.9: Inversion

The Dynamic operator, D, sets the dynamic of a section of the piece. As with the transpose operator, the part it affects is determined by which type of block occurs first after the dynamic. It has one parameter, *marking,* which is a numeric value that sets the velocity (how loud they are played) of notes in the melody or chord part.

The Speed operator, S, sets a new tempo value in the piece (S for speed has been used instead of T for tempo to avoid clashing with the transpose operator). It has one parameter, *tempo,* which is the value of the new tempo in crotchet beats per minute.

Harmony blocks will, in general, be longer than motive blocks as the harmonic progression of a piece is slower than the variation in the melody line. To give the melody and chord parts a basic level of coherence, the two parts are synchronised at the start of each harmony block. This ensures that two blocks close together in the program string will occur close together (in time) in the resulting piece. One harmony block may have several motive blocks associated with it but not vice versa.

As an example, consider the string:

H,32,6,43,M,12,24,53,I,M,16,32,3,T,5,H,24,40,53…

The blocks and operators in this string are:

H(32,6,43)	Harmony block, duration = 32 semiquaver beats harmonic-rhythm index = 6 harmonic-progression index = 43
M(12,24,53)	Motive block duration = 12 semiquaver beats melodic-rhythm index = 24 melodic-pitches index = 53
I	Invert operator
M(16,32,3)	Motive block - inverted duration = 16 semiquaver beats melodic-rhythm index = 32 melodic-pitches index = 3
T(5)	Transpose operator, amount = 5 semitones
H(24,40,53)	Harmony block, - chord roots transposed up 5 semitones duration = 24 semiquaver beats harmonic-rhythm index = 40 harmonic-progression index = 53

Figure 4.10 shows the structure of the music resulting from the example string in relation to bars of 4/4 time. The two motive blocks in the string are associated with the first harmony block.

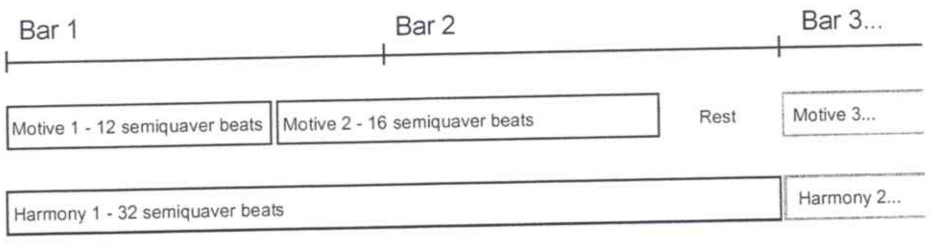

Bar 1
Bar 2
Bar 3...
Motive 1 - 12 semiquaver beats
Motive 2 - 16 semiquaver beats
Rest
Motive 3...
Harmony 1 - 32 semiquaver beats
Harmony 2...

Figure 4.10: Example structure of piece

4.5 Example Piece 'Daisy'

An existing piece of music is analysed in this section so that a genetic representation of it can be 'backwards engineered' from the analysis. The piece chosen for analysis was is nursery rhyme "Daisy" (a small homage to HAL9000, the computer from Arthur C Clarke's 2001: A space Odyssey).

Figure 4.11 shows a transcription of the piece. The top stave is the melody line and the second stave is the chord part showing the chord names and the rhythms of the chord changes. The harmony of the piece has been deliberately 'jazzified' to demonstrate the system's representation of jazz chords.

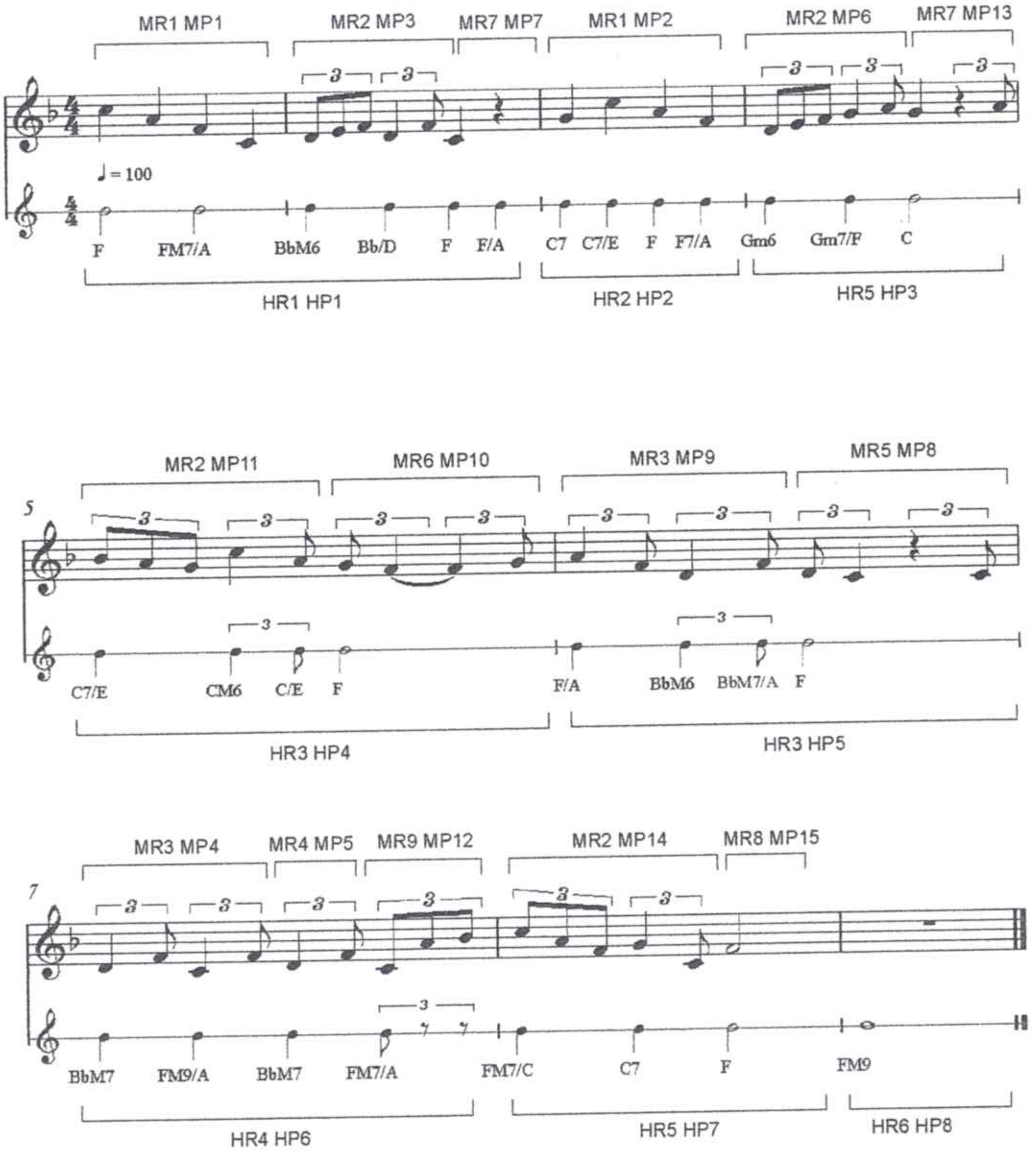


Figure 4.11: Transcription of 'Daisy'

The melody and harmony parts have been split up in to small sections that will form the motive and harmony blocks of the GP. Each motive has a rhythm pattern and note sequence associated with it.

Several motives in the melody line share the same rhythm patterns. Bar 2, for example, starts with triplet quavers followed by a swung quaver-crotchet pair. The same pattern, MR2 (melody rhythm pattern-2) appears at the start of bars 4, 5 and 8. A pattern used more than once only needs to be stored once on the melody-rhythms chromosome. Rhythm patterns can overlap on the chromosome, as the patterns do not affect each other. Figure 4.12 shows the full rhythm sequence for the melody-rhythms chromosome and where each rhythm pattern appears on it.

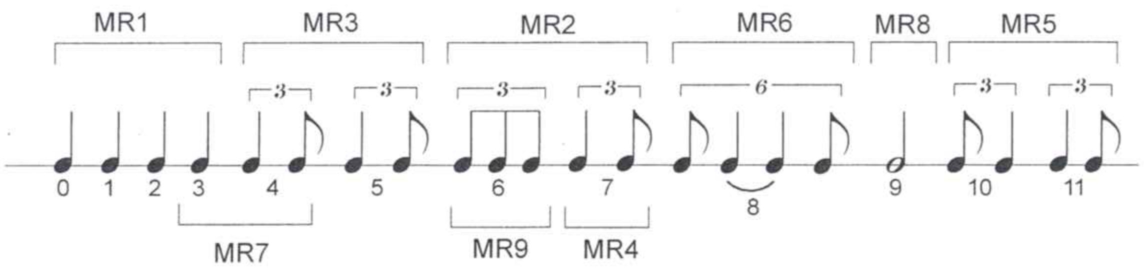


Figure 4.12: melody rhythms

Each motive block uses a pitch sequence that appears somewhere in the melody-pitches chromosome. These sequences can overlap on the chromosome. Most of the motive block pitch sequences are mapped directly on to the chromosome. Figure 4.13 shows the full pitch sequence for the melody-pitches chromosome, where each pattern occurs and which operators have been applied to them.

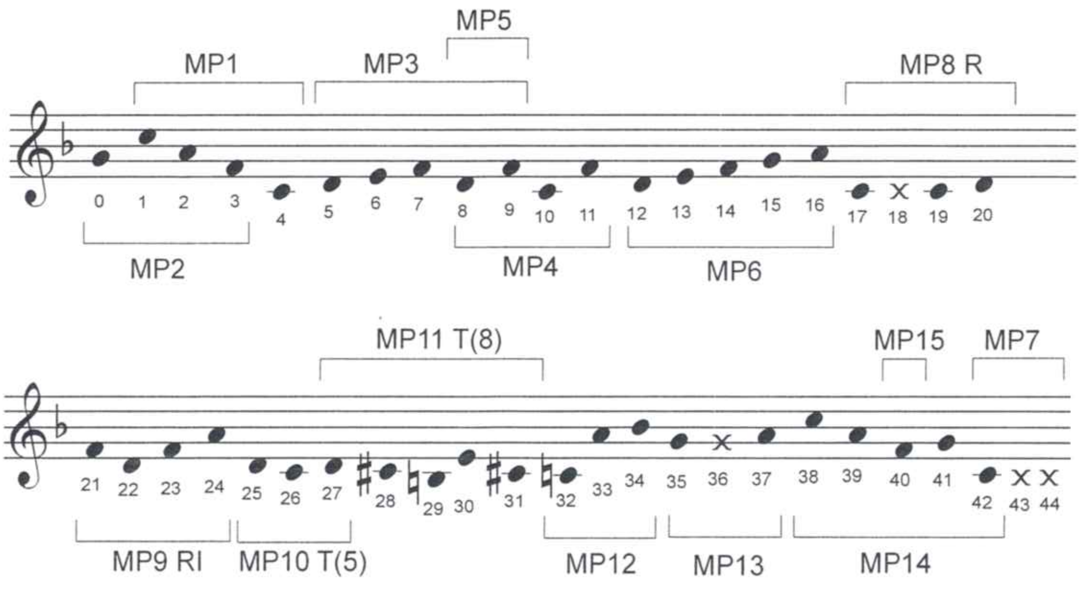


Figure 4.13: melody-pitches

MP9 (melody pitch sequence 9) is reversed on the pitches chromosome. This allows for the demonstration of the Retrograde operator in the GP. Likewise, MP10 and MP11 are transposed (by 5 and 8 semitones respectively) to demonstrate the Transpose operator. Notes to be played as rests (pitch numbers 18, 36, 43 and 44) are displayed as crosses.

The harmony line has been analysed in the same fashion. Figure 4.14 shows the rhythm sequence for the harmonic-rhythms chromosome. Note that HR5 occurs twice (bars 4 and 8), as does HR3 (bars 5 and 6).

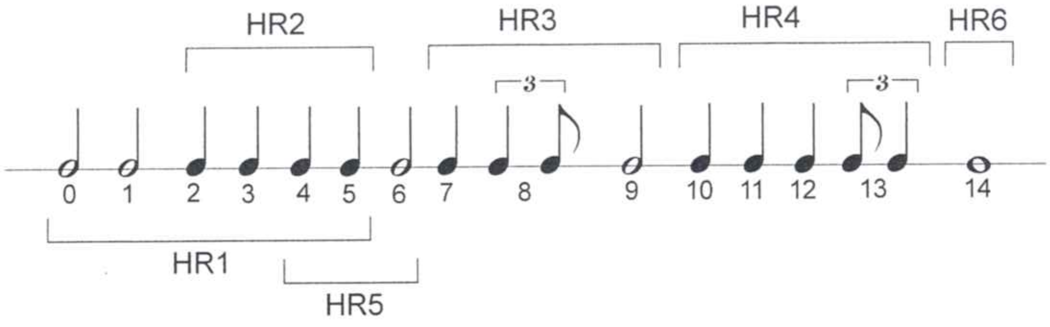


Figure 4.14: harmonic rhythms

Each Harmony block uses a chord progression from the harmonic-progression chromosome. Figure 4.15 shows how the chords are arranged on the chromosome (A chord to beplayed as a rest is shown as an X). To demonstrate how the GP operators affect harmony blocks, HP4 is transposed by three semitones and HP6 is in reverse order.

HP1	FM7	0
	FM7/A	1
	BbM6	2
	BbM6/G	3
	F	4
	F/A	5
HP2	C7	6
	C7/E	7
	F	8
	F/A	9
HP3	Gm6	10
	Gm7/F	11
	C	12
HP4 T(3)	A7/C#	13
	Am6	14
	A/C#	15
	D	16
HP5	F/A	17
	BbM6	18
	BbM7/A	19
	F	20
	F#DIM7/A	21
HP6 R	X	22
	FM7/A	23
	BbM7	24
	Fmaj9/A	25
	BbM7	26
HP7	FM7/C	27
	C7	28
	F	29
	Fmaj9	30

Figure 4.15: harmonic-progression

With the four GA chromosomes defined, it is now possible to write the Genetic Program string that describes the piece. The first block is a Harmony block, two bars long, which uses the rhythm sequence HR1 and the progression sequence HP1. Two bars of 4/4 time is 32 semiquaver beats long. HR1 starts at point 0 on the harmonic-rhythm chromosome and HP1 starts at point 0 on the harmonic-progression chromosome. The first command in the string is therefore:

H,32,0,0

Three motive blocks occur in the duration of the first harmony block. The first of these lasts for a bar (16 semiquaver beats), using MR1 and MP1 (with start points 0 on the melody-rhythms and 1 on the melody-pitches chromosomes respectively. The length of the second motive block is 8 semiquavers and the third is 4. The second motive block uses MR2 (melody-rhythms index 6) and MP3 (melody-pitches index 5) and the third uses MR7 and MP7 (indexes 3 and 42 respectively). Therefore, GP string is now:

H,32,0,0,M,16,0,1,M,8,6,5,M,4,3,42

The rest of the string is coded in the same way as outlined above. The first motive block in bar 5 is transposed by 8 semitones therefore the string at that point is:

…,**T**,**8**,M,8,6,27,…

The first motive block in bar 6 is reversed. The code at that point in the string is:

…,**R**,M,8,4,21,…

This hand-coded representation of 'Daisy' was used to test the Renderer (see Section 5.5). A score of the MIDI file produced (daisy.mid), along with the full code for the piece and the full listing of each GA chromosome array (in the output text file daisy.txt) is shown in Appendix Section 11.3.

5 System Implementation

This section describes the system implemented to generate and breed pieces of music. The system has been split in to separate parts with defined roles. Each part will be dealt with in turn.

The important algorithms and some short code extracts are discussed in this section. The full program code can be found on the CD.

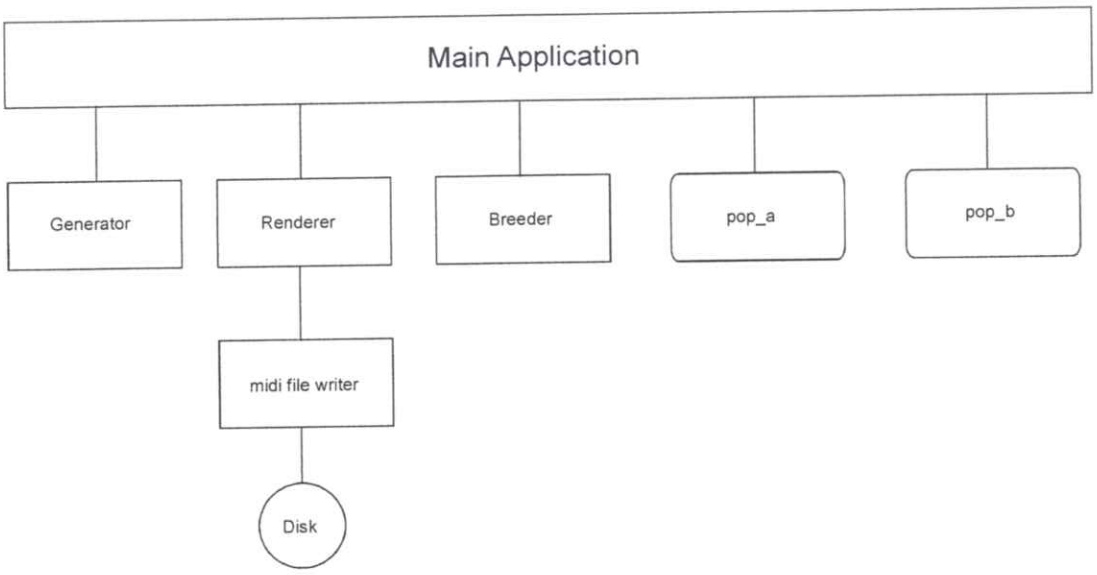


Figure 5.1: Object Model of System

Figure 5.1 shows the overall structure of the system. The Generator, Breeder and Renderer and Midi File Writer are instances of the classes described in sections 5.2, 5.3, 5.4 and 5.5 respectively. pop_a and pop_b are arrays of pieces. These arrays hold the current and new populations at any one time. On each iteration g(i) of the system, it replaces the old generation g(i-1) with the new generation g(i+1) (see figure 5.2).

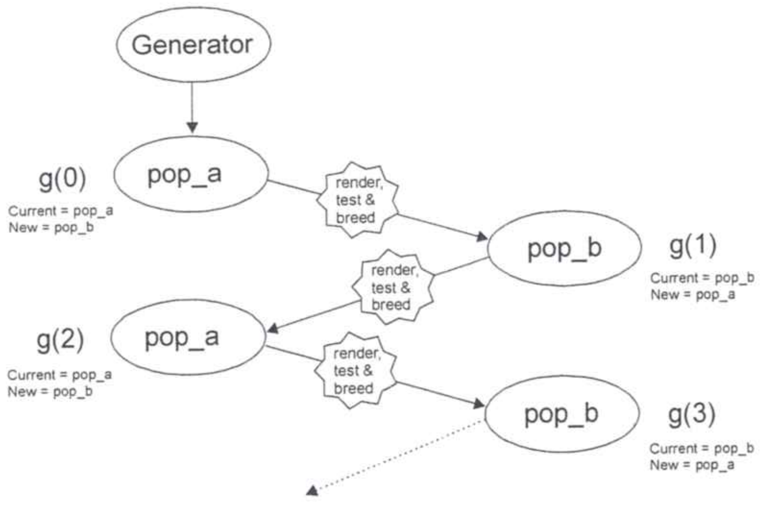


Figure 5.2: Diagram of swapping generations

Swapping generations between the arrays is achieved by using three pointers: current, new and spare (see figure 5.s on previous page). After each cycle, the functions of the arrays are swapped thus:

spare = current

current = new

new = spare

On initialisation, current points to pop_a and new points at pop_b. The Generator populates the current generation array to form the first generation g(0). The members of g(0) are then rendered as MIDI files, assessed and allocated a fitness value. These fitness values are fed back in to the system and used to select the parent candidates for the next generation g(1) (see figure 5.2). The render-test-breed cycle is carried on for each subsequent iteration.

Tournament selection [22] has been used to choose the parents of the next generation. Two individuals are selected randomly from the population (see figure 5.3). The pair are compared and the fittest of the two goes on to be the first parent P1. Another pair of individuals is then chosen randomly from the population (P1 is no longer available for this selection) and the fittest of these two becomes parent P2.

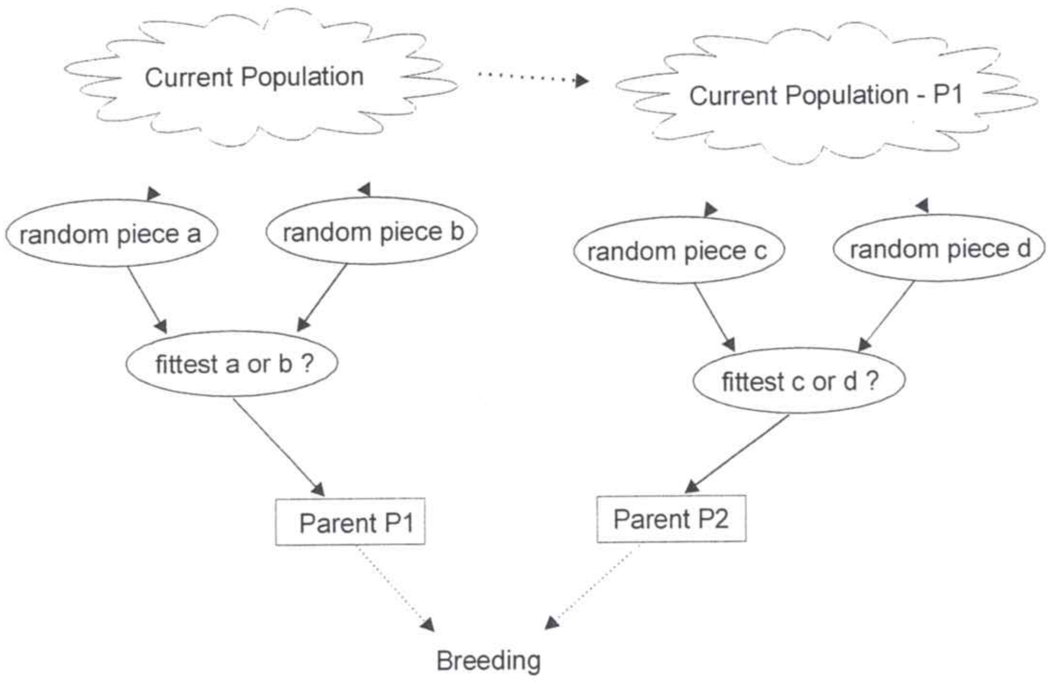


Figure 5.3: Tournament selection

5.1 Notes, Chords and Rhythm Figures

The GA chromosomes of a piece are arrays of data structures. They use three structure types: `NOTE`, `CHORD` and `FIGURE` (These are defined in piece.h on the CD).

5.1.1 Note Structure

```
typedef struct
{
int value;
bool rest;
}NOTE;
```

The `NOTE` structure (above) holds two pieces of information: `value` and `rest`. Value is the MIDI pitch value of the note from 0 to 127 (see Section 3.4). Rest is a Boolean variable that determines whether the `NOTE` will be rendered as a pitched note or as a rest.

5.1.2 Chord Structure

```
typedef struct
{
int type;
int extension;
int root;
int inversion;
bool rest;

}CHORD;
```

The `CHORD` structure (above) holds five pieces of information: type, extension, root, inversion and rest. Type is the chord type (for example M, M7, m6). The system currently understands thirteen basic chord types mapped to the numbers 0 to 12. Extension is the type of extension added to the chord (for example 9, dim11). The system currently understands seven extension types (including 0 – no extension) mapped to the numbers 0 to 6. Root is the MIDI pitch value of the root note of the chord. Inversion is the value which determines which note of the chord to be played as the bass note.

Rest is a Boolean variable that determines whether the `CHORD` will be rendered as a chord or as a rest.

5.1.3 Figure Structure

```
typedef struct
{
int pattern;
int duration;

}FIGURE;
```

The `FIGURE` structure (above) defines a rhythm pattern to be played and is used for both melody and rhythm. It holds two pieces of information: pattern and duration. Pattern is the type of rhythm to be played (e.g. a single note, triplet or a swung pair etc). The system currently understands eleven patterns, mapped to values 0 to 11 (see table 5.1). Duration is number of semiquaver beats that a pattern is played over. For example, a triplet pattern with a duration of four semiquavers (one crotchet) would be played as triplet quavers. However, the same triplet pattern with a duration of twelve semiquavers (a dotted minim duration) would be played as three crotchets (see figure 5.4).

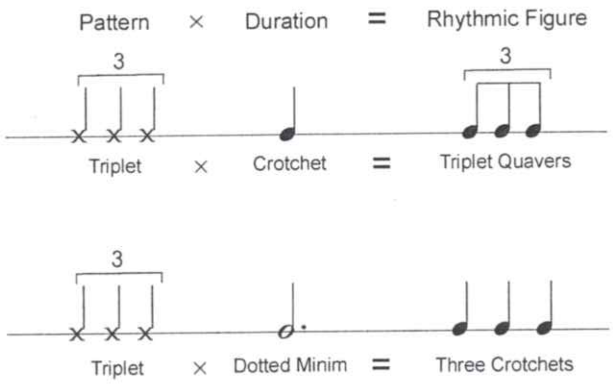

Pattern × Duration = Rhythmic Figure
3
Triplet × Crotchet = Triplet Quavers
Triplet × Dotted Minim = Three Crotchets

Figure 5.4: Rhythm Figure example

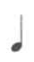
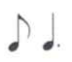
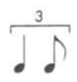
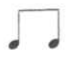
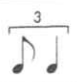
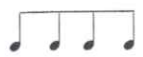
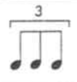
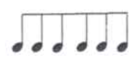
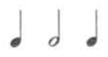
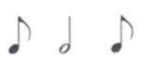
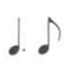
	0		6
	1		7
	2		8
	3		9
	4		10
	5		

Table 5.1: Rhythm Patterns

5.2 Formal Piece Structure

Each piece is an instance of the `gp_piece` class (see piece.h and piece.cpp on the CD).

The gp_piece class contains the GA chromosome data, the Genetic Program, the piece name and the initial values which, together, describe a piece entirely. The class also contains the names of its parent pieces for tracing lines of inheritance back through generations.

The GA information is stored in four arrays (see below). An array of `NOTE`s, `melody_pitches`, stores the pitches of the melody line and an array of `FIGURE`s, `melody_rhythms`, stores the rhythmic patterns for the melody line. An array of `CHORD`s, `harmonic_progression`, stores the chord progressions for the harmony part and another array of `FIGURE`s, `harmonic_rhythms`, stores the rhythmic patterns for the harmony part.

```
NOTE melody_pitches[100];            array of notes
FIGURE melody_rhythms[100];          array of rhythm figures
CHORD harmonic_progression[100];     array of chords
FIGURE harmonic_rhythms[100];        array of rhythm figures
```

The Genetic Program string is stored in a character array: `gp_string` (see below). It is treated as a null-terminated string by the system.

```
char gp_string[1000]; character array to hold gp string
```

The initial values of tempo (in crotchet beats per minute), transpose and dynamic are held as integers.

```
int tempo;          initial tempo (bpm)
int transpose;      initial value of transpose
int dynamic;        initial value of dynamic
```

The name of the piece is stored as a null terminated string, filename. `parent_a` and `parent_b` are the filenames of the piece's parents.

```
char filename[30];     filename (null terminated string)
char parent_a[30];     parent P1 name
char parent_b[30];     parent P2 name
```

The gp_piece class has one function: `save_piece_data()`. This function outputs the contents of the piece to a formatted text file (for an example an output text file, see Daisy.txt in Appendix 11.3).

5.3 Generator

The `gp_generator` class generates the values for the pieces that make up generation g(0) (see generator.h and generator.cpp on the CD). The gp_generator class has a very simple interface; there are two functions: `set_parameters` and `generate` (see below).

```
set_parameters(int length);
generate(gp_piece *new_piece, int number);
```

`set_parameters` simply sets the approximate length of piece (in beats) to be generated. The `generate` function chooses values for the GA chromosomes and GP string of the piece passed to it. Number sets the number X of the piece (i.e. filename = g0xX). The values for the GA and GP are generated using several different stochastic methods.

The first method employed is a simple random-number generator. This is used to generate values such as dynamic and transpose giving a simple rectangular distribution from 0 to an upper limit. Values such as melody pitches and chord roots are generated with the random number generator then added to an offset value.

The second method employed is to take a random value from a discrete probability distribution. This is implemented with an array of values that forms the distribution and a random number generator to choose which value to take. The example below shows a distribution that is skewed to favour lower numbers (see figure 5.5 for distribution that is formed). Values for rests in the melody and harmony parts are determined this way.

```
distribution[12] = {1,1,1,1,2,2,2,3,3,4,5,6};

value = distribution[random(11)];
```

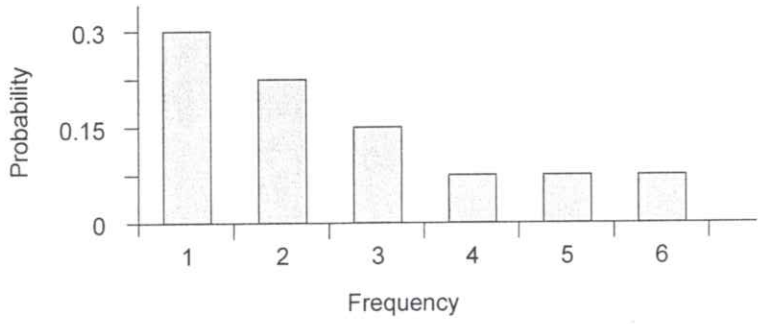


Figure 5.5: Distribution Example

The final method employed is Markov models (see section 3.3.1). A third-order Markov chain is used to generate the Genetic Program string. The state transitions for the GP string Markov model is shown in figure 5.6 (see overleaf) as a labelled, directed graph. The graph is so heavily interconnected that a "dummy" state *Other* has been added so that it may be split up for clarity.

The Markov model is implemented using a probability distribution array as before. The difference is that this array is altered at run time depending on which state is occupied. The states which alter the array are H and M (Harmony and Melody respectively). When a the H state is reached it alters the next-state probabilities of H and M to favour M. When the M state is reached for a third time after the last H state, it alters the next-state probabilities of H and M to favour H (see pseudo code below).

```
distribution[13] = {0,0,0,0,1,1,1,1,2,3,4,5,6}; start distribution

motives = 3          number of motive blocks since last harmony block initialised to 3

begin

switch(distribution[random(12))
{
        case 0: Harmony
                motives=0  reset motives
                distribution[1 through 5]=1  compared to harmony block
                break

        case 1: Motive
                motives++ increment motives
                if(motives>=3)  when motives equals three...
                {
                distribution[1 through 5]=0  ...Alter probability to favour H
                motives=0 reset motives
                }
                break

        case 3 to 6: Other operators – these do not alter the distribution
}

loop to beginning
```

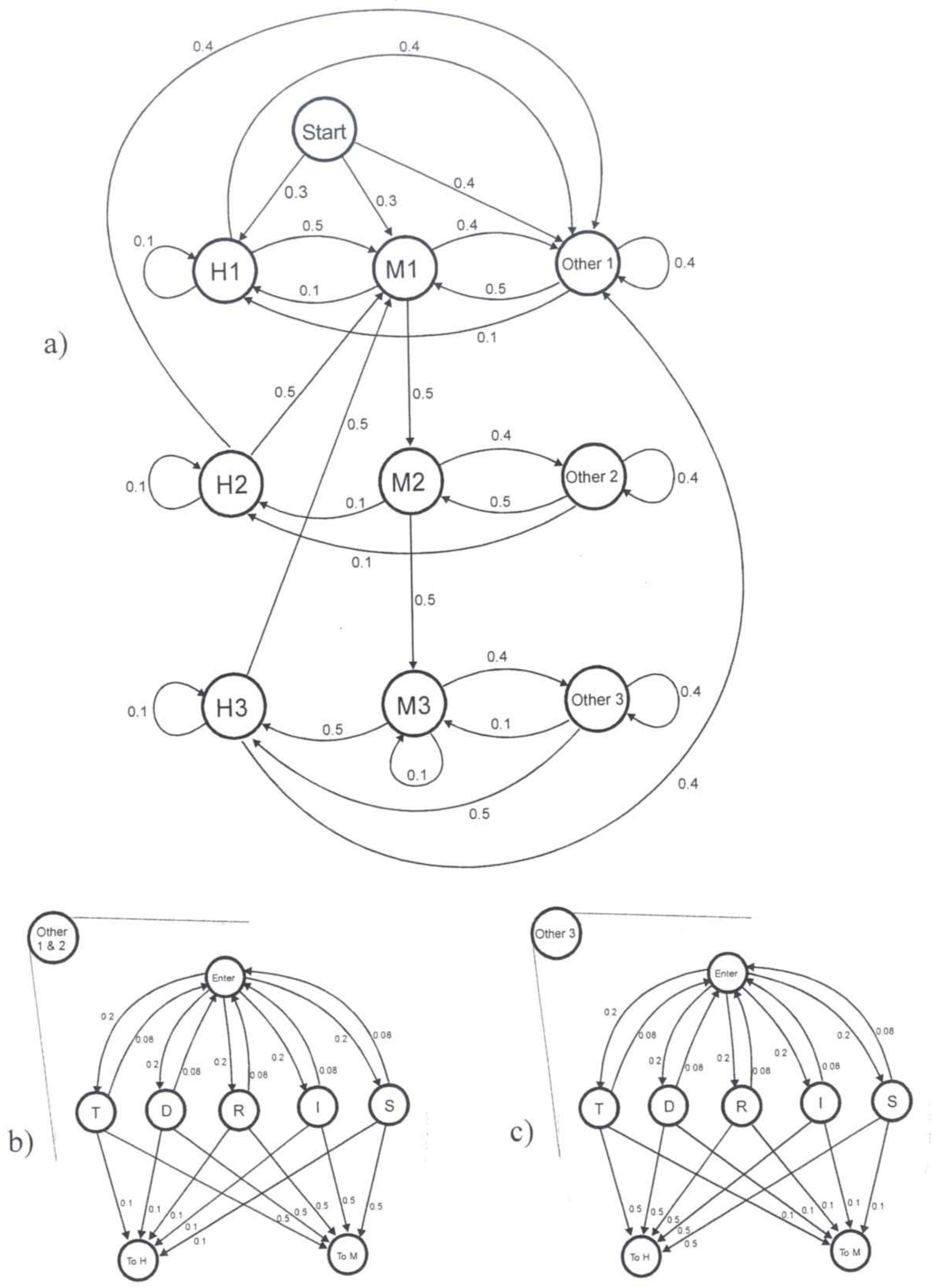


Figure 5.6: Third-Order Markov Model a) shows main graph, b) shows 'Other' for orders 1 & 2 c) shows 'Other' for order 3.

5.4 Breeder

The gp_breeder class takes two parent pieces and breeds them together to produce two children (See breeder.h and breeder.cpp on the CD). It performs crossover on the GA chromosomes and on the GP string. The breeder does not currently implement mutation.

The breeder randomly allocates one parent's initial values to one child and the other parent's values to the other child. This is implemented in code using the ternary operator [1, 31], which has been used extensively in the breeder class (see pseudo code below).

```
i = random(1); pick value for i (0 or 1)

set initial values:

child_a->initial_v = i ? parent_a->initial_v : parent_b->
initial_v;

child_b->initial_v = i ? parent_b->initial_v : parent_a-
>initial_v;
```

The breeding of the GA chromosome uses the ternary operator again. It uses a crossover flag and a dummy array that is the same length as the chromosome that stores the crossover points. The crossover points are marked in the crossover array at random positions. The number of crossover points is set using the `set_rates` function. When a crossover point is reached in the array, the crossover flag is toggled. The current element of a child will be set to equal the current element of parent 1 or parent 2 depending on the value of the crossover flag (see figure 5.7).

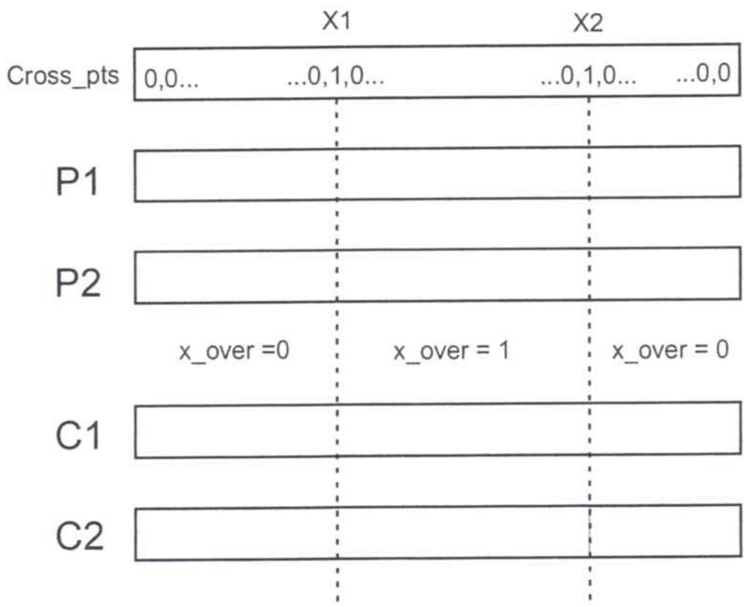


Figure 5.7: GA crossover

In the case of Harmony and Motive blocks, a function is called which creates the required block type (these functions are discussed later in this section). When the current block or operator has been decoded the next one in the string is found and the cycle continues until the null terminator the end of the string is found.

5.5.2 Decoding the GA chromosomes

When a harmony or motive block operator is found in the GP string, a function is called which handles the decoding of the GA chromosomes and sends the relevant data to the file-writer object.

The rhythm information for both types of block is decoded using the function `get_rhythms`. This function decodes `FIGURE`s from a rhythm chromosome and puts the decoded absolute note-lengths into an array. The algorithm works through the chromosome until the accumulated value of the note-lengths found is greater than or equal to the number of beats required (see figure 5.9).

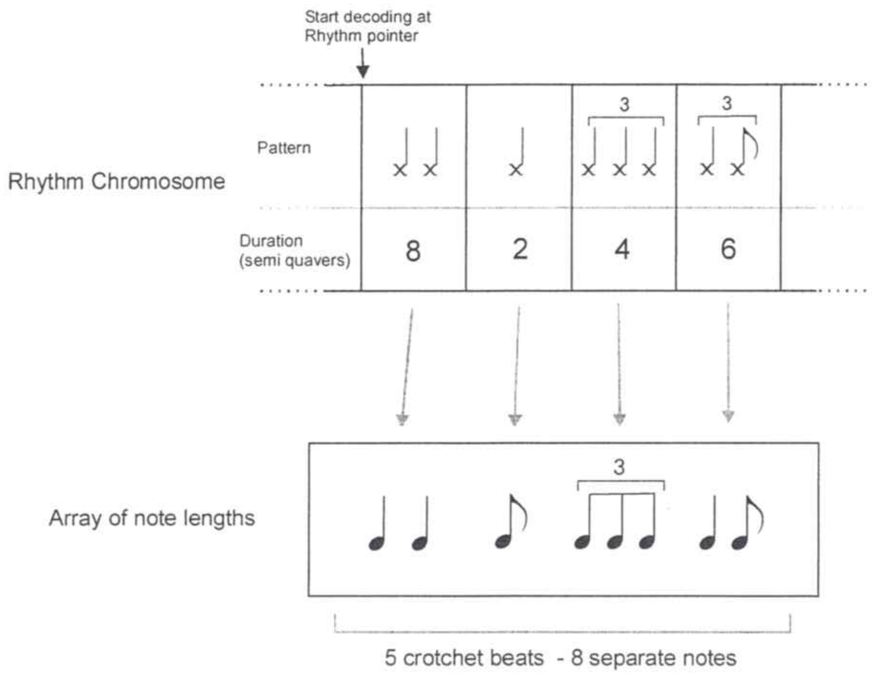


Figure 5.9: Decoding Rhythms

The Motive function calls the get_rhythms function to decode the required part of the melody_rhythms chromosome. The number of notes needed from the melody_pitches chromosome is the number of note-lengths returned by get_rhythms. The `NOTE`s are taken from the melody_pitches chromosome one by one and passed to the file writer object along with the values for transpose and dynamic (see fig 5.10 a) using the write_note function (see Section 5.6). If Retrograde is true then the notes are taken from the same area on the melody_pitches chromosome but in reverse order (figure 5.10 b). The Retrograde operator does not affect the rhythm pattern.

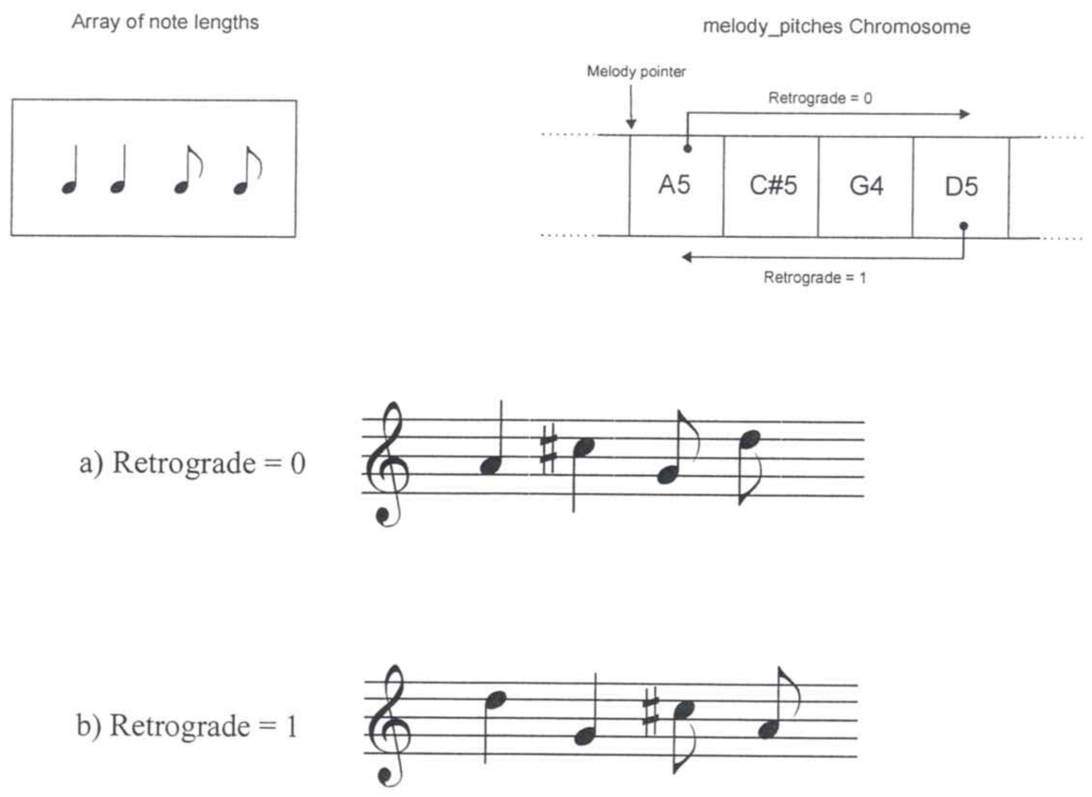


Figure 5.10: Decoding Melody GA a) Normal, b) Retrograde

If Invert (see Section 4.3) is true, the notes are all inverted around the first note of the motive (see psuedo code below).

```
if (invert) note.value = 2*first_note - note.value;
```

The decoding of the Harmony GA parts is performed in the same way as the Melody GA decoding described above. The Harmony block function calls the synchronize_time function of the file writer (explained in Section 5.6) then decodes the GA information. The chords are taken from the harmonic_progression chromosome and passed to the file writer using the write_chord function (chord decoding is performed by the file writer – See Section 5.6).

5.6 Midi File Writer

The midi_file_writer class (see fwriter.h and fwriter.cpp on CD) creates a simple interface for converting the genetic representation of a piece to a standard MIDI file (see Section 3.4). The class has five public functions: `save_file`, `write_note`, `write_chord`, `synchronize_time` and `write_tempo`.

The file created is a type 1 MIDI file [37] with four tracks: Melody, Chords, Bass Line and a dummy track which holds initialisation information and tempo changes. The tracks in a type 1 MIDI file are stored in sequence in the file. Because of this, the track data is stored in large character arrays (one array for each track) so that the tracks may be written to independent of each other. When the piece has been fully rendered, the save_file function is called. This function builds the MIDI file by appending the relevant header data then outputting each array to the file (see figure 5.11).

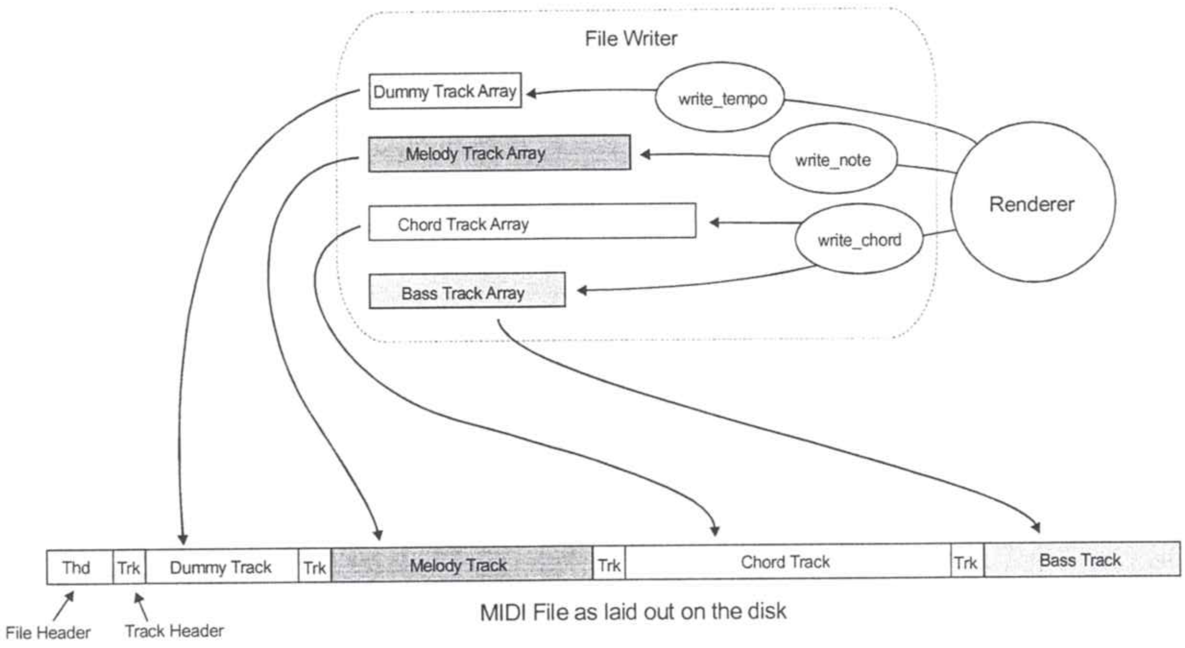


Figure 5.11: Relationship between arrays and Midi File

The write_note function is used write a MIDI note event to the melody track. If the note passed is a rest, no note event is added. The lengths of consecutive rests are accumulated and the total rest time is inserted as a delta-time at the start of the next note to be played. If the note passed to it is not a rest, the function inserts a delta-time equal to the time since the end of the last note (zero if the previous note was not a rest) and a note-on event. It then inserts a delta-time for the note-length and a note-on event with velocity zero (a note-on event with velocity zero is equivalent to a note-off event). The complete MIDI message that is added to the melody array takes the form:

```
XX-XX 90 NN VV XX-XX 90 NN 00
```

where:

`XX-XX`	*variable length delta-time = time since end of last note*
`90`	*note-on header on chan 1*
`NN`	*note value*
`VV`	*velocity value*
`XX-XX`	*variable length delta-time = note length*
`90`	*note-on chan 1*
`NN`	*note value*
`00`	*velocity 0 to end note*

The write_chord function decodes a `CHORD` structure then writes a chord and a lyric event, containing the chord name, to the chords track and a bass note to the bass track. It handles rests in the same way as the write_note function described above.

An array, chord_notes, is used to store all the notes in a chord. The chord-type is determined and the note values that make up that chord are added to the chord_notes array. The chord in the pseudo code example below is a Major 6th chord. This chord is made up of the root, the major 3rd (four semitones above the root), the 5th (seven semitones above the root) and the major 6th (9 semitones above the root). The variable, notes, records how many notes there are in the chord_notes array. The chord extension is found in the same way; in the example the major 9th (14 semitone above the root) is added to the chord.

```
switch(chord->type)
{

        …case  M6
                chord_notes[0] = root + 0;  notes in M6: 1st, 3rd, 5th and
6th
                chord_notes[1] = root + 4;
                chord_notes[2] = root + 7;
                chord_notes[3] = root + 9;
                notes = 4;                  there are now 4 notes in array
                name_string += "M6"         add type to name-string

                switch(chord->extension)    find chord extension
                {
                        …
                        case 9th
                                chord_notes[4] = root + 14;         9th
                                notes++;                    increment notes
                                name_string += "(9)"    add "(9)" to name-
string
                        other extensions
                }

        other chords…
}
```

When the notes of the chord have been added to the chord_notes array, the inversion value of the CHORD is used to determine which note to play in the bass part. If the value of inversion is less than or equal to the number of notes in chord_notes, the note in that element of the array is played as the bass note; if it is not, the root note is used. The bass note is played in the octave below the chord voicing.

The note-on messages for each note in the chord are added to the chord-track array along with a lyric event containing the chord name. The chord's duration is then added as a variable length delta-time followed by note-off messages for each note in the chord. The bass note-on, duration and note-off messages are then added to the bass-track array.

The synchronize_time function synchronises the melody track to the harmony parts (the bass and chord tracks are synchronised all the time as they are always written together). The function works by determining which track lags the other and adding a rest value equal to the time difference to the lagging track (see figure 5.12).

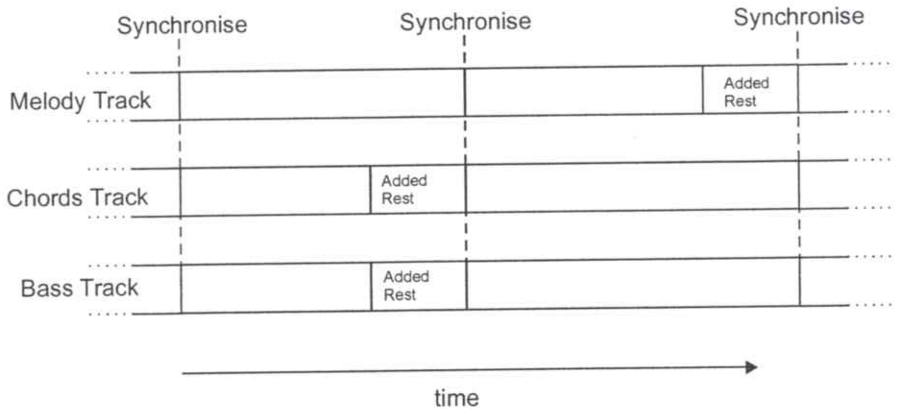


Figure 5.12: Synchronising tracks

The write_tempo function writes a new tempo-change event to a fourth, dummy track. The function inserts a delta-time in to the dummy track that is equal to the time between the last tempo change and the last note played in the melody track. The tempo-change event is then added to the track.

5.7 Fitness Test

The fitness test currently implemented in the system is that of a human judge marking pieces to a set of criteria and giving each piece a final fitness grade. Figure 5.13 shows the mark sheet that is used for a test.

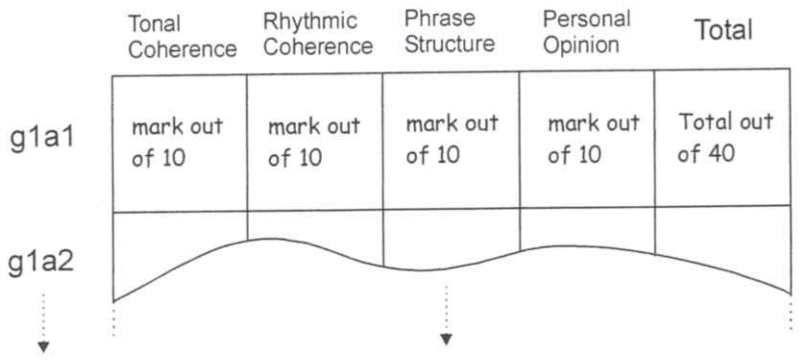


Figure 5.13: Mark sheet

The fitness test comprises of four criteria: Tonal Coherence, Rhythmic Coherence, Phrase Structure and Personal Opinion.

Tonal Coherence is a measure of how tonally pleasing or acceptable the piece is to the listener. A piece with many dissonant chords and an apparently random melody line would be given a low score. Likewise, a piece that exhibits pleasant chords and melodic material will be given a high score.

Rhythmic Coherence is a measure of how rhythmically pleasing or engaging the piece is to the listener. A piece that exhibits totally random rhythmic patterns would be given a low score. Whereas a piece containing pleasing rhythm patterns would gain a high score.

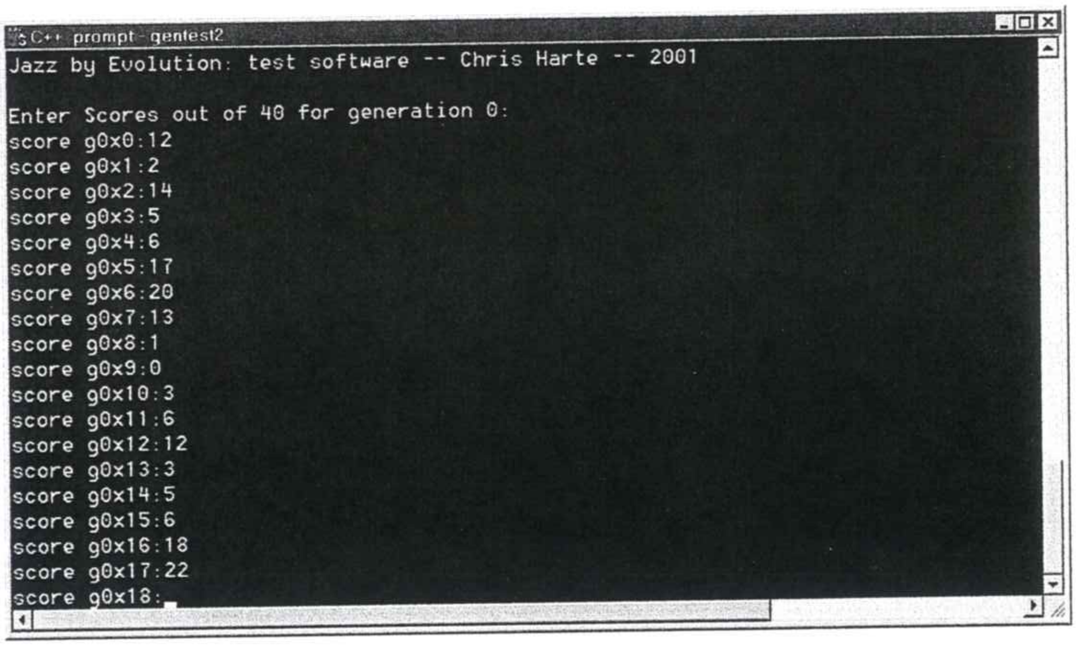
Phrase Structure is a measure of how structured the piece appears to be to the listener. Thus, a piece that is one long sequence of notes and chords with no apparent structure scores badly. Whereas a piece that appears to be built of definite phrases will score well.

Personal Opinion is, as it says, the personal opinion of the listener. This category is an attempt to mark the "tingle factor", i.e. that which makes one piece good and another, equally musically correct, boring. If the listener likes the piece it will score well, if they don't it will score badly.

Each category is scored out of ten. The marks are added together to give a total out of forty which is fed back in to the system as the fitness function for the piece.

5.8 User Interface

A rudimentary user interface has been implemented to test the system. The system prompts the user for the fitness score for each piece from a generation in turn (see screenshot in figure 5.14). The choice is then given to go on to generate another generation or to leave the program.



```
Jazz by Evolution: test software -- Chris Harte -- 2001

Enter Scores out of 40 for generation 0:
score g0x0:12
score g0x1:2
score g0x2:14
score g0x3:5
score g0x4:6
score g0x5:17
score g0x6:20
score g0x7:13
score g0x8:1
score g0x9:0
score g0x10:3
score g0x11:6
score g0x12:12
score g0x13:3
score g0x14:5
score g0x15:6
score g0x16:18
score g0x17:22
score g0x18:_
```

Figure 5.14: Screenshot of initial user interface

6 Evaluation

This section gives an evaluation of the project's progress and initial test results. It begins with a brief look at the progress of the project in terms of time compared to the plan from the initial report. There then follows a description of the testing philosophy applied to writing the software. The section concludes with the initial results and analyses of experiments with the software.

6.1 Project Progress

Figure 6.1 (overleaf) shows a Gannt chart of the project with two sets of timings on it. The first set (in light grey) of timings is those set out in the planning section of the initial report (this can be found on the CD in the 'other documents' area). The second set (in dark grey) are those which more closely describe the actual progress of the project. The flow of the project has been consistent with the plan; the estimated completion times for certain areas, however, were underestimated in the initial report. One major factor in the project's actual progress was the time needed for learning C++. This was not considered an issue of great importance at the outset of the project as a good familiarity was already held with the C programming language. However, the crossover time from one language to the other was longer than predicted which had the cumulative effect of putting the project approximately four weeks behind schedule. This has prevented a full, conclusive set of results being gathered.

6.2 Testing

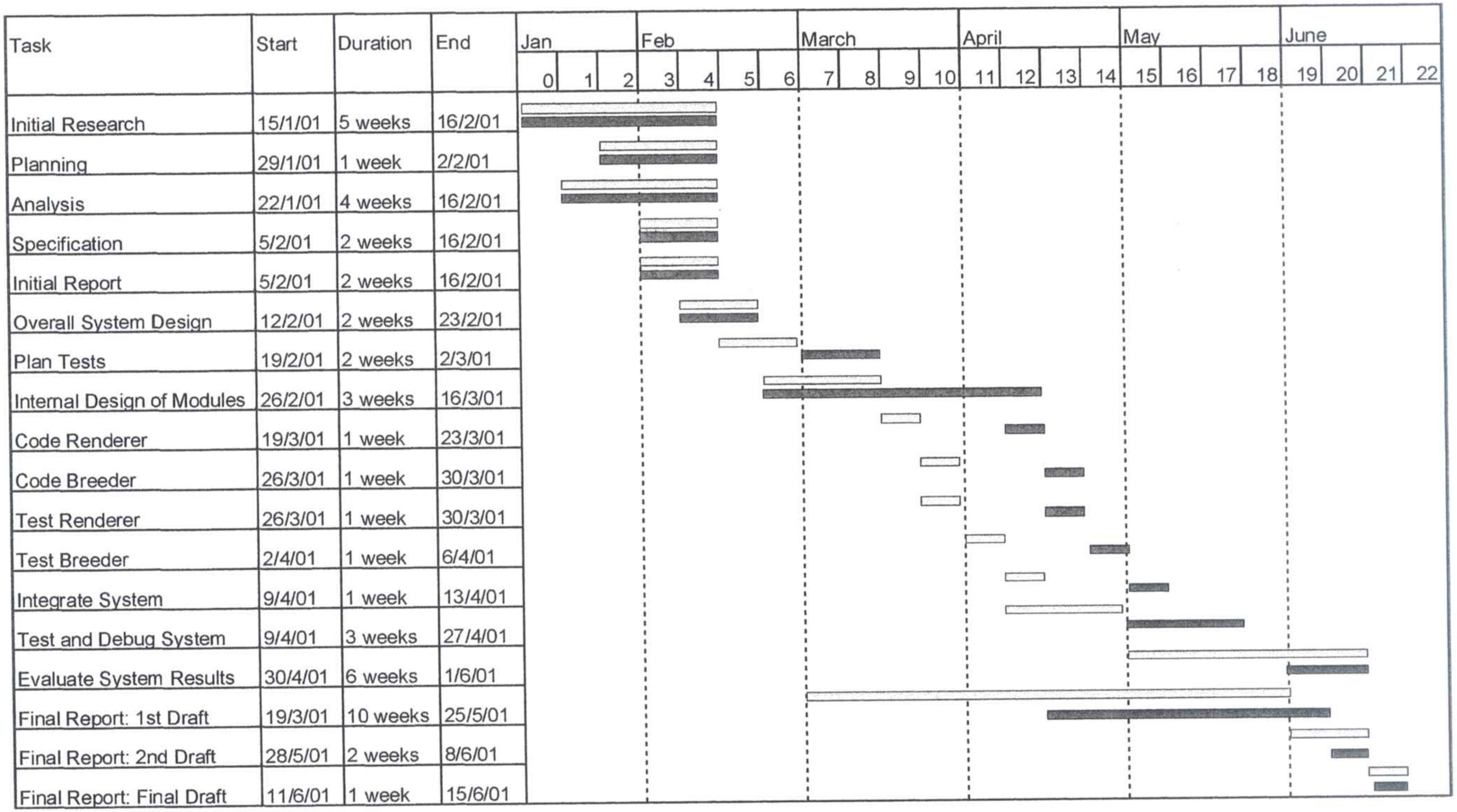
The software for the system was written in stages. At each stage, the class being written was tested using a 'test bed' program, which simulated the conditions under which the class would be run in the full system (these test programs can be found in the 'code\testprograms' folder on the CD). As described in section 4.5, the main test of the software was the rendering of a 'hand coded' piece of music. This was to prove that the Renderer would produce the correct MIDI file for a given `gp_piece`. The coded representation of the rendered piece, 'Daisy', is shown in the form of its output text file in Appendix 11.3. The score of the MIDI file for the piece is shown in Appendix 11.2. Both of these files can be found on the CD in the 'examples' section.

6.3 Experimental Results

Due to the time constraints imposed on the project it is only possible to show preliminary results for the system's performance. The experiments detailed here have been run using comparatively small population sizes over a few generations. The time-consuming nature of the experiment has prohibited the gathering of results for larger populations over more generations.

The initial test runs of the system used a population size of ten to show whether the system was breeding the pieces together properly. Test run 7 was the first to give useful results. The scores for the initial test runs were given simply on my own subjective judgement.

Run 7 had a few particularly good pieces (comparatively speaking) in consecutive generations. To see if there was a relationship between them, the full inheritance tree of the run was traced out using the parent information in the output text files.



Task	Start	Duration	End
Initial Research	15/1/01	5 weeks	16/2/01
Planning	29/1/01	1 week	2/2/01
Analysis	22/1/01	4 weeks	16/2/01
Specification	5/2/01	2 weeks	16/2/01
Initial Report	5/2/01	2 weeks	16/2/01
Overall System Design	12/2/01	2 weeks	23/2/01
Plan Tests	19/2/01	2 weeks	2/3/01
Internal Design of Modules	26/2/01	3 weeks	16/3/01
Code Renderer	19/3/01	1 week	23/3/01
Code Breeder	26/3/01	1 week	30/3/01
Test Renderer	26/3/01	1 week	30/3/01
Test Breeder	2/4/01	1 week	6/4/01
Integrate System	9/4/01	1 week	13/4/01
Test and Debug System	9/4/01	3 weeks	27/4/01
Evaluate System Results	30/4/01	6 weeks	1/6/01
Final Report: 1st Draft	19/3/01	10 weeks	25/5/01
Final Report: 2nd Draft	28/5/01	2 weeks	8/6/01
Final Report: Final Draft	11/6/01	1 week	15/6/01

Figure 6.1: Gantt Chart of Project. Light lines show predicted progress, dark lines show actual progress

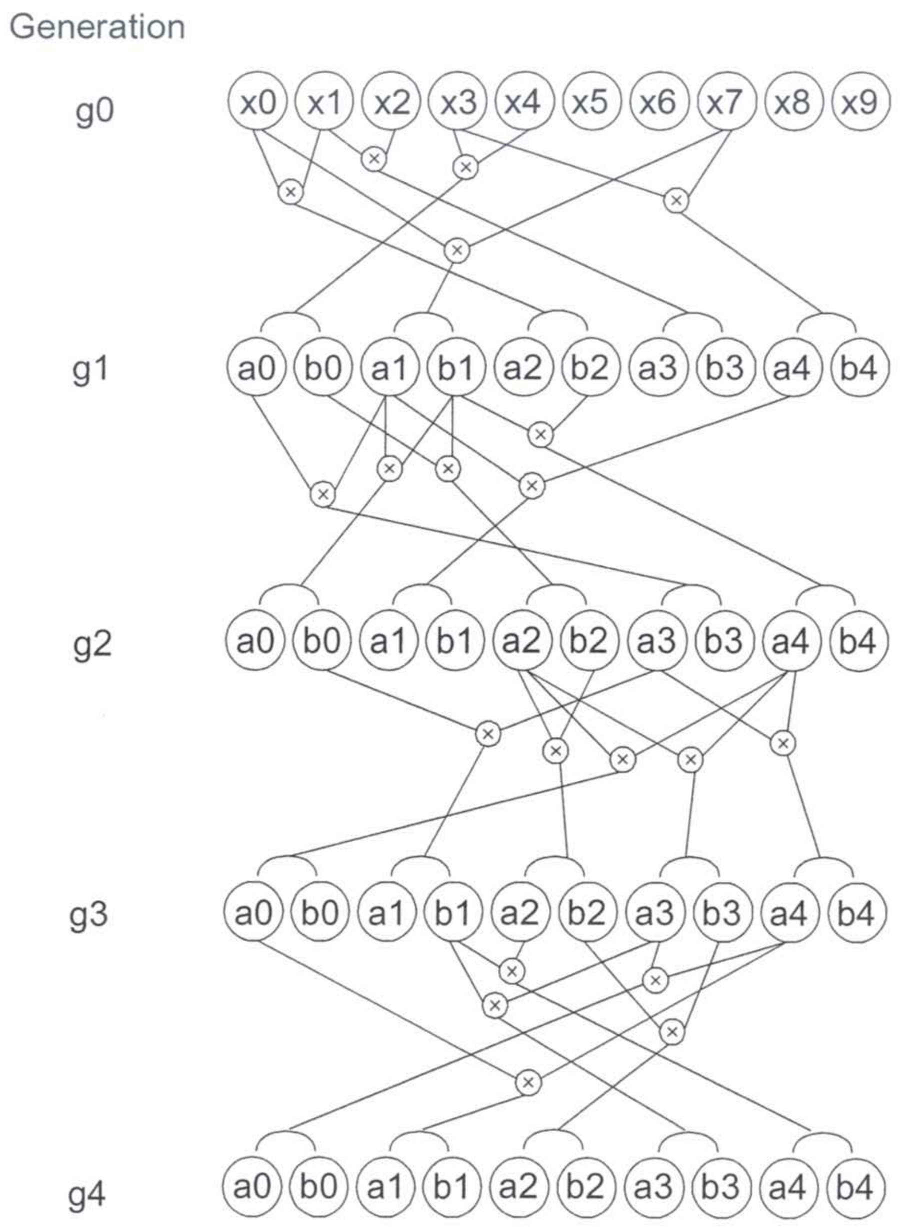


Figure 6.2: Inheritance tree for run 7. The crossover of two parents to produce a pair of children is shown as a small circle with a cross in it

Figure 6.2 shows the full inheritance tree for run 7. The particular individuals of interest were g1b0, g1a1 and g1b1 from generation 1, g2b1, g2b2 from generation 2, g3a3 from generation 3 and g4b4 from generation 4 (these files can be found in the 'results\run7' folder on the CD).

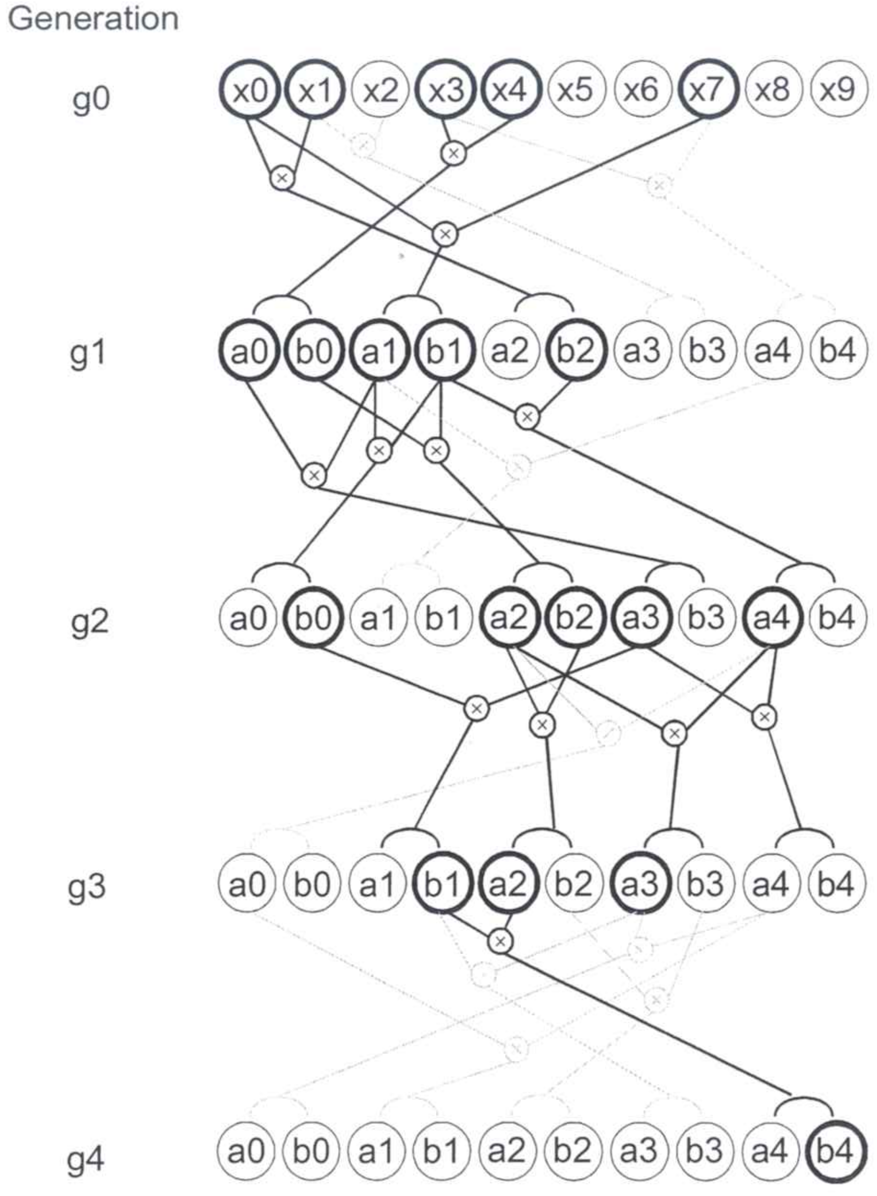


Figure 6.3: Inheritance tree for run 7 with the ancestry of 'fittest' candidates traced

Figure 6.3 shows the tree again with the ancestry of these particular files traced out. It can be seen that g1b1 plays an important role in the tree, being parent to four of the fit files in the next generation.

The initial runs were surprising in that the average fitness for the pieces in the population dropped steadily over each generation, effectively dying out after four or five generations (see the graph in figure 6.4).

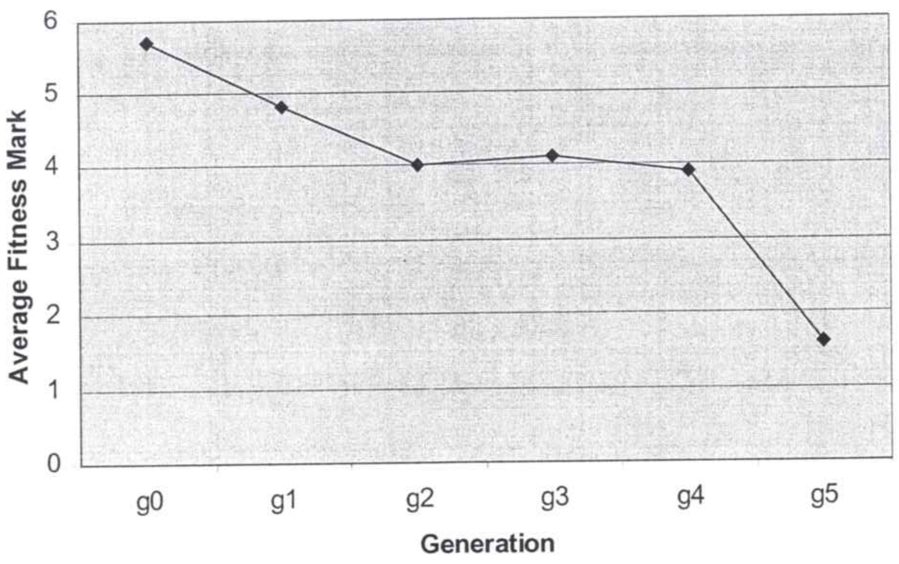


Figure 6.4: Graph showing average scores for each generation of run 7

On analysis of the text files for the evolved pieces, the reason for this decline in fitness became clear. As stated before, in section 5.4, the original plan was to let the crossover point of the GP fall on any point in the string. This would mean the operation of crossover could split function blocks in the string. Initially, this was seen as an interesting way in which new numbers and functions being arranged by the evolutionary process might provide diversity. In practice, however, the crossover operation was too disruptive to the functions and caused the programs to become syntactically incorrect so that they became impossible to render. To illustrate this, File extract 6.1 shows the GP for g0x0 of run 7. The generated code is syntactically correct and it is possible to see the structure of the piece simply by inspection. Compare this to file extract 6.2 (overleaf), taken from g5a0 of the same run. In this extract there are only two renderable functions (H,28,7,11 and H,7,183,2) and even these are badly fragmented.

```
Genetic Program:

H,20,92,14,I,M,4,21,14,M,2,20,91,M,4,80,78,M,12,91,53,M,4,74,6
9,H,20,84,29,H,24,1,77,M,8,18,6,H,48,1,86,
```

File Extract 6.1: GP string from g0x0 of run 7

```
Genetic Program:
H,20,9M,,,2HD,,2,7H,,28,7,,,,,11,,M,,4M12M97,,42H,9I,H,7,,183,
,2
```

File Extract 6.2: Disrupted functions in the GP string from g5x0 of run 7

The solution to this problem was to make the crossover operation split strings only at points in between functions. The method used to perform crossover in this way is discussed in section 5.4.

Once the amendments had been made to the crossover process, the test was enlarged to a have a population size of twenty. Two friends were persuaded to take time out to do the experiment. The test requires approximately half an hour per generation, as the pieces are each around a minute long. This makes testing even a small number of generations several hours work.

The initial results of these experiments were interesting. The average marks across the first run in each case dropped initially then appeared to start rising again (see figures 6.5 and 6.6). It must be noted that this is only an observation of the results recorded. The number of generations is too small in both cases to be sure the trend would continue. In both cases the judges insisted that the material was becoming subjectively better at the end of the test despite the values for the average marks. It is therefore suggested that a lack of familiarity with the test will account for the high scores in the first generation of each.

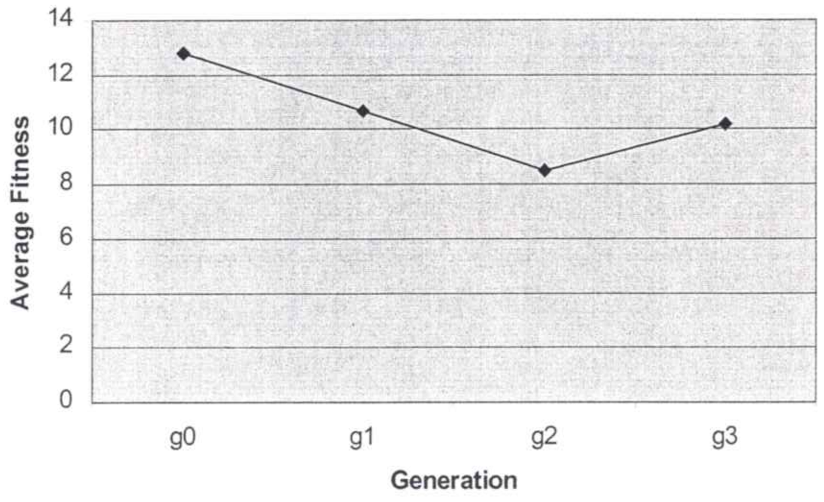


Figure 6.5: Graph of average marks over run-J1

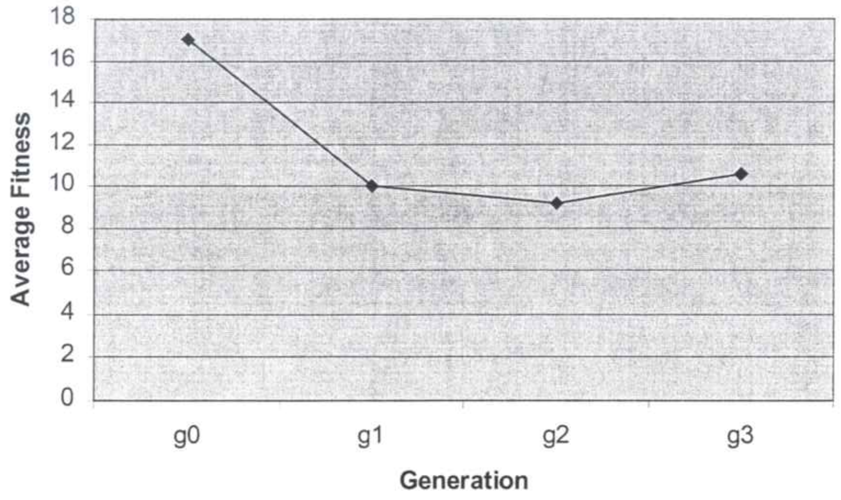


Figure 6.6: Graph of average marks over run-S1

When one of the judges did the experiment for a second time, the results were much more encouraging. This time was familiar with the test and the marks given were representative of that. The graph in figure 6.7 shows a definite upward trend in the average marks over the subject's second run.

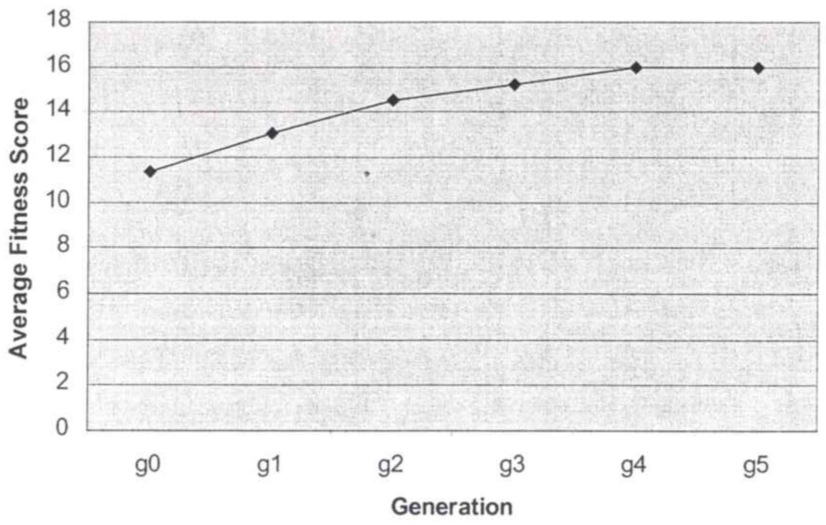


Figure 6.7: Graph of average marks over run-S2

During this run the judge noticed a particular musical idea which appeared in several of the fittest pieces over the different generations. This was the short chord sequence: F G#M6 BbM6(9)/C (see figure 6.8) which brought to mind a track off 'Nuyorican Soul' (an extract of which can be found as a .wav file on the CD in the 'examples' folder).

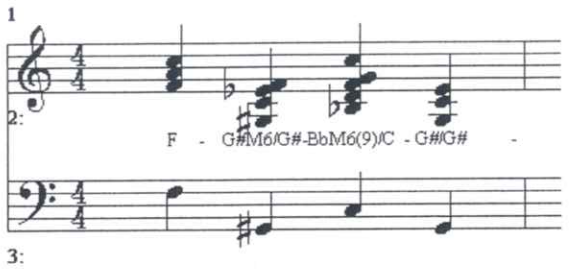


Figure 6.8: Chord sequence as it appears in the opening of piece g3a1 from run S2

This sequence first appears in generation 3 where it occurs in two pieces: g3a1 and g3a5. It was noted that the chords appeared twice in g3a5 (see figures 6.9 and 6.10).

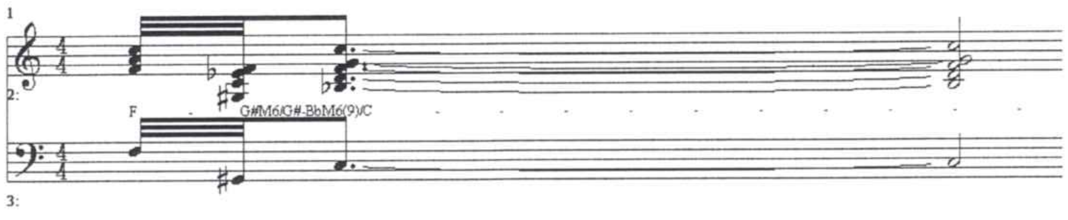


Figure 6.9: first occurrence in g3a5

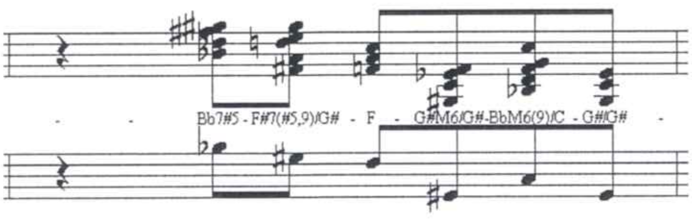


figure 6.10: second occurrence in g3a5

On closer inspection it was found that the sequence also appears again in g3a1 but this time it is transposed up, starting on the chord of D (see figure 6.11) hence the progression becomes D FM6 GM6(9)/A.

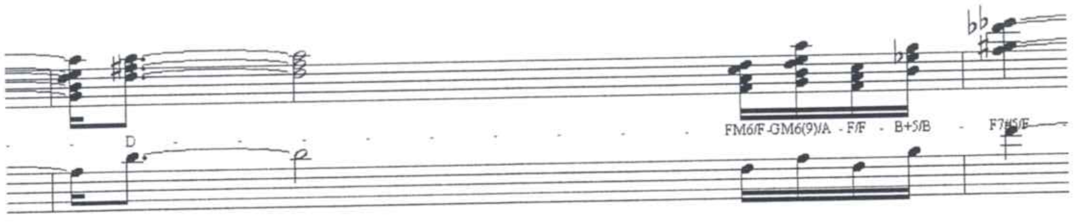


Figure 6.11: second occurrence, transposed, in g3a1

The sequence appears in generation 4 in pieces g4a6 and g4a7. In both files it has been transposed, starting on G (see figure 6.12).

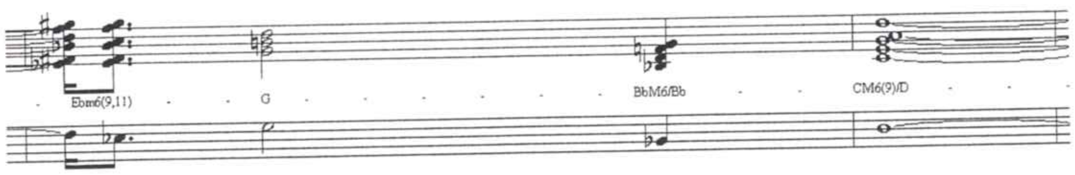


Figure 6.12: Occurrence of sequence in g4a6 – bars 8/9

The fifth generation has a further two pieces in which the pattern occurs. The sequence in g5a0 can be seen in figure 6.13.

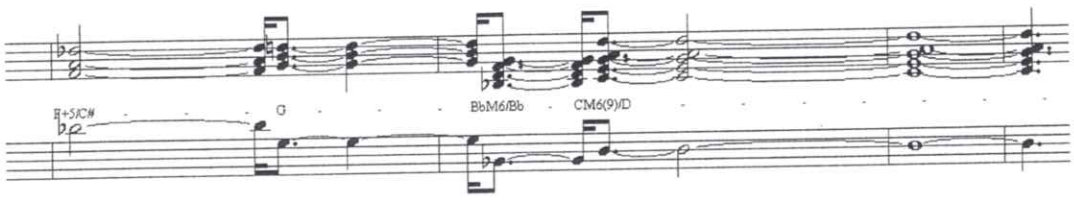


Figure 6.13: Occurrence of sequence in g5a0 – bars 7/8

The progression was not noticed in any pieces from generations 0, 1 and 2. To determine where it came from the code of g3a1 was analysed. The GP string for g3a1 is shown in file extract X.

a)

```
Genetic Program:
H,32,43,10,T,10,H,32,34,8,M,8,13,58,M,4,69,74,D,10,M,4,54,47,H
,24,11,10,S,88,H,8,85,76,H,20,91,18
```

b)

Index	Harmonic-Rhythms	Harmonic-Progression: Type, extension, root, inversion, rest
10	(2 , 11)	(0 , 1 , 64 , 0 , 0)
11	(0 , 11)	(1 , 3 , 55 , 4 , 0)
12	(8 , 1)	(1 , 2 , 57 , 4 , 0)

[File Extract 6.3: a) GP string from piece g3a1 of run S2, b) point on harmonic-progression chromosome where the chord sequence lies]

As the chord sequence opened the piece, it was easy to determine the point where it was located on the harmonic-progression chromosome (i.e. point 10). The parents of g3a1 were g2b7 and g2a2. The sequence itself came from the harmonic-progression chromosome of g2a2. The harmony block that makes it appear in the music came from g2b7. This shows that the sequence was not a visible part any of the pieces generated initially. It emerged due to crossover between g2b7 and g2a2 after the second generation.

These results are encouraging as it could be argued that this is an example of schema theory in operation. That is to say, the short chord progression forms part of a particularly fit schema.

7 Further Work

This section outlines those areas of the project that require further work and those that may merit further investigation. These are presented as a series of miniature project specifications to provide starting points for possible future projects.

7.1 Mutation in the System

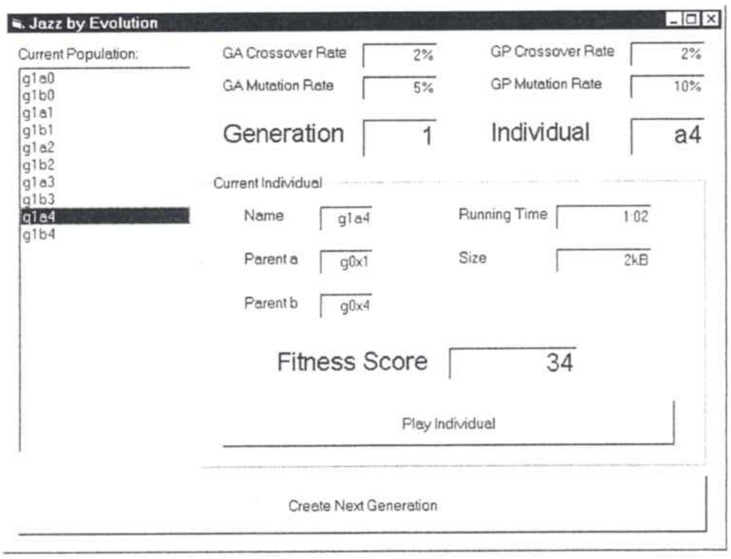
The current system does not implement mutation at either the GA or GP levels. Mutation at the GA level would consist of picking a random point on a chromosome and altering its value. It is suggested that the system of selecting a new value from a statistical distribution around the current value be employed. This approach fits neatly with the music representation, as a fairly good area on the chromosome could, with a small change, potentially become highly fit.

Mutation in the GP should also follow this scheme. A random character (or number) from the string should be replaced with another character chosen from a statistical distribution around the current value.

The points for mutation code to be added are commented in the breeder functions `gp_breed` and `ga_breed` (see breeder.cpp on the CD).

7.2 Windows User Interface

The current system test software has a rudimentary DOS user interface. The system would be made more user-friendly if a new user interface was coded for use in the Windows environment. At the present time, the user must play each MIDI file using an external piece of sequencing software. A useful addition to a new user interface would be to link a MIDI file player to the system software itself. Figure 7.1 shows a possible layout of a Windows interface for Jazz by Evolution.


Jazz by Evolution

Current Population:
g1a0
g1b0
g1a1
g1b1
g1a2
g1b2
g1a3
g1b3
g1a4
g1b4

GA Crossover Rate: 2%
GP Crossover Rate: 2%
GA Mutation Rate: 5%
GP Mutation Rate: 10%
Generation: 1
Individual: a4

Current Individual
Name: g1a4
Running Time: 1:02
Parent a: g0x1
Size: 2kB
Parent b: g0x4

Fitness Score: 34

Figure 7.1: Jazz by Evolution: Mock up of windows interface

7.3 *Splitting the Test in time*

A problem, highlighted when users did the experiment, is the necessity for the test run to be completed all in one go. At present, there is no facility for saving the current generation of pieces so that the experiment may be continued at another time. To test the system properly for large population sizes over many generations such a facility must be added, as there is a limit to how long one human judge can spend on the test at any one time.

Serialisation of the `gp_piece` class into a file, and the ability to read such files back, would make much longer experiments possible as judges could then pick the test up, from where they left it, at their convenience.

7.4 *Musical Fitness Tests*

The system currently uses a human judge, working to a set of subjective criteria, to grade the fitness of each piece. This means, for example, to test a generation of twenty pieces, each about a minute in length, will take almost half an hour; to test ten generations of a hundred individuals will take over twenty four hours. The current fitness test becomes prohibitively long for experiments with larger populations over more generations. It is necessary to test large population sizes over many generations to gather significant, meaningful results. A machine fitness test that removes, or at least diminishes, the requirements of a human judge is therefore necessary to make such experiments practical. Several approaches to a machine-fitness test are now considered:

7.4.1 Automatic Elimination of Unsuitable Individuals

For some of the individuals produced by the system, it is possible to identify them as being poor simply by inspection. An unsuitable individual may be a MIDI file containing only one or two notes or even none at all (not enough to constitute a piece of music). Clearly, for such a file, it is unnecessary to apply the subjective human fitness criteria, as it cannot possibly score well.

An automatic test function that identifies these poor individuals may be devised to filter out such pieces in order to speed the human judgement process up. There are many ways in which such a test may be implemented.

An initial approach might be to check the size of the individual's MIDI file. Each MIDI message (see Section 5.6) will be approximately 14 bytes in length, each chunk header around 12 bytes, each Meta-event around 16 bytes and each track-end event 3 bytes. Therefore, an approximate, minimum acceptable length for a MIDI file output by the system can be defined. Consider a track containing one melody note, one 4-note chord (plus a lyric event of the chord name) and one bass note:

File header:	12 bytes	
Track 0 header:	12 bytes	(Dummy Track)
Track 0 data:	$2 \times 16 + 3 = 35$ bytes	(Key and time signature events plus track-end)
Track 1 header:	12 bytes	(Melody Track)

Track 1 data:	16 +14 + 3 = 33 bytes	(initial tempo, one note and track-end)
Track 2 header:	12 bytes	(Chord Track)
Track 2 data:	4 × 14 + 16 + 3 = 75 bytes	(4 notes in a chord, lyric event and track-end)
Track 3 header:	12 bytes	(Bass Track)
Track 3 data:	14 + 3 = 17 bytes	(One bass note plus track-end)

Total number of bytes = 12 +12 +35 +12 +33 +12 +75 +12 +17 = 220 bytes

The file described above obviously does not constitute a piece of music worth judging with the human test. As a result we can see that a MIDI file produced by the system must be more than 220 bytes in length so any MIDI file that is shorter should be ignored.

A MIDI file containing several chords but no notes in the melody line would easily pass the file size test, though it is still an unsuitable individual. Therefore, a further development of this idea would be to check each track individually. Each track has a set minimum length (for example, the lengths derived above) so if any track falls short, the file can be rejected.

7.4.2 Codifying Musical and Jazz Rules

It may be possible to code many of the musical rules described in section 3.3.3 into rule-based tests. The Harmonic-preference rule, for example, [24] could be coded so that a test runs through the chord progressions of a piece and assigns a mark to each chord transition according to the distance moved between the chords on the line of fifths. The average of these marks would be an indication of how acceptable the harmonic scheme of the whole piece is.

The music produced by the system may not be in the same meter as the file suggests (i.e. the MIDI file defaults to 4/4 at 120bpm but the music may not be). To apply the strong-beat rule, the piece could be recorded as an audio file then digital signal-processing techniques (fast auto-correlation for example) could be applied to find where the meter really lies and how strong it is. The average power on the strong-beats should be higher than that of weak-beats.

7.4.3 Neural Networks and Pattern Matching

Although previous research has shown neural networks to be poor at handling time, they are very good at pattern matching. A possible application of a neural network as part of a larger, integrated fitness evaluation could be to recognise patterns in the GA chromosomes. Consider the case of the melody-pitches chromosome. As described in section 4.3, the melody-pitches chromosome stores a sequence of pitches which is the source material for motives.

A set of equivalent pitch sequences could be derived from jazz melodies and solos (Charlie Parker solos for example). These might be used as a training set for a neural network which could be used to recognise potentially good material in the melody-pitch chromosome of a piece.

Another area where pattern matching could be used would be to identify structures in the genetic program. Repeated or similar structure in the GP string may give a clue to how much reuse of material occurs in the piece. There are techniques that would spot such patterns in the music itself but as there is already an abstract representation available (the string itself) it would seem sensible to use it. A pattern-matching test on the GP would also be capable of finding transformations of material faster than testing the MIDI file, as the transformation operators are part of the coded string as well.

7.4.4 An Integrated Approach to the Musical Fitness Test

Many of the fitness tests used in evolutionary systems for music composition try to apply a general test to decide whether the music is 'good' or 'bad'. Biles' neural network fitness function [5], for example, is trained to imitate the human mentor's responses, but is unsuccessful. This is because the system tries to model the responses of the mentor without modelling the perception and reasoning behind those responses.

We listen to many different elements of the music at many different levels. The human fitness test described in section 5.7 is split in to four categories which is an attempt to give some of these different elements separate marks in order to make the test more fair. A possible approach to the fitness test problem may be to have several different fitness algorithms, which test the different elements in parallel and provide an overall fitness mark for the piece. These tests could include any or all of the approaches mentioned earlier in this section. A possible topology for such a test system is shown in figure 7.2.

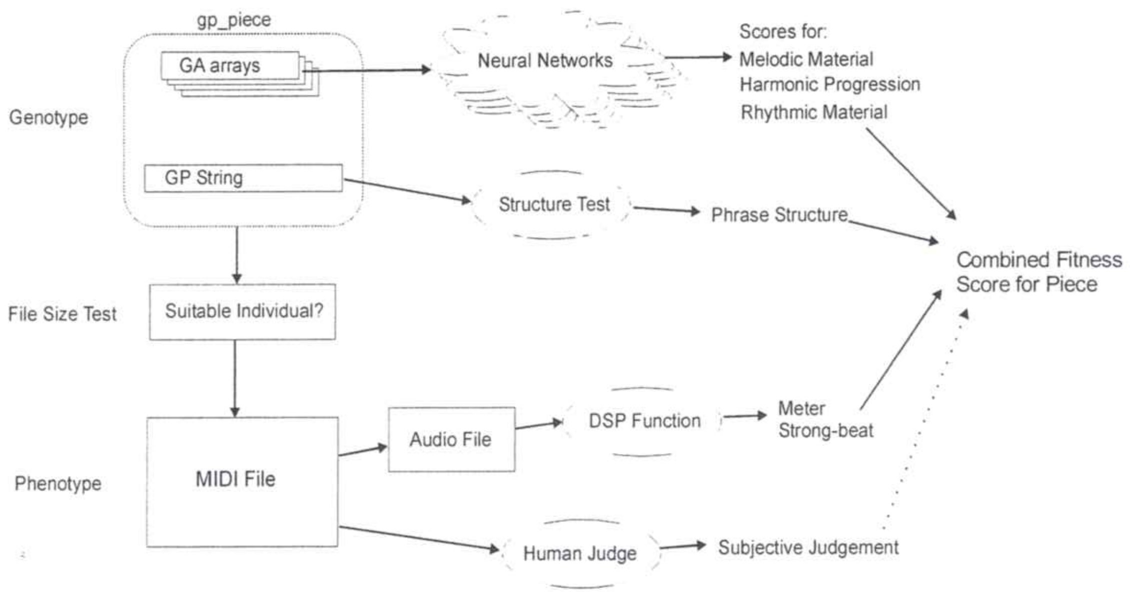


Figure 7.2: Multiple fitness tests

A human judge is still necessary to act as a control for the tests. However, this approach could potentially alleviate the fitness bottleneck by guiding the judge to the potentially best pieces. If such a test scheme was found to give good results, full control could be handed to it.

7.5 *Improving the Renderer's performance skills*

The Renderer only has one voicing for each particular chord type. A real jazz pianist uses many different voicings [10] when playing. The MIDI files that are produced by the Renderer would sound more natural if it could play a wider range of voicings (see figure 7.3).

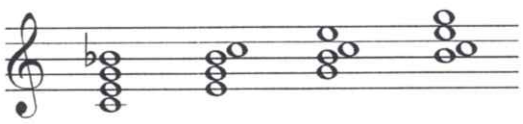


Figure 7.3: Different voicings of a C7 chord; thepresent Renderer can only play the fist one

The choice of voicing depends on many factors including the preceding chord and its voicing, the next chord and its possible voicings and what is happening in the melody line.

A level of artificial intelligence must be added to the Renderer for it to create chord parts that use different chord voicings. A possible solution is to use a rule-based approach and hard code a model of the rules a jazz musician instinctively follows when playing. Some of these rules can be found in jazz texts [10, 11]. Where others are less well defined, an element of probability will have to be added to the decision process to complete the model.

7.6 *Music File Compression*

During the course of the project, it was noted that the genetic representation, developed for the system, might have unforeseen benefits in compressing music. It was suggested that the representation of a piece, if written to a disk, would take up less space than its equivalent MIDI file.

To test this hypothesis quickly, consider the example piece 'Daisy' analysed in section 4.5. The MIDI file of this piece (see Appendix 11.2) is 1,803 bytes long. If the genetic representation of the piece can be stored in fewer, there is a strong case for further development of the idea. To find the number of bytes required to store the representation, the analysis must be reviewed.

The analysis in Section 4.5 produces 11 `FIGURE`s in melody-rhythms, 44 `NOTE`s in melody-pitches, 14 `FIGURE`s in harmonic-rhythms and 30 `CHORD`s in harmonic-progression.

`FIGURE`s, `NOTE`s and `CHORD`s are structures that are built from `int` and `bool` data types as defined in C++. An `int` is two bytes long and a `bool` is one bit. For the sake of argument, let us imagine that a `bool` takes up one byte in a file on disk. A `NOTE` consists of one `int` and one `bool` value so requires three bytes of disk space. A `FIGURE` consists of two int values which is four bytes. A `CHORD` consists of four `int` values and one `bool` which is nine bytes. The total number of bytes required to store the GA information is therefore:

`FIGUREs`:	(11+14) × 4 = 100
`NOTEs`:	44 × 3 =132
`CHORDs`:	30 × 9 = 270
Total:	100 + 132 + 270 = 502 bytes

The GP of 'daisy' is 244 characters long. Each character is one byte long therefore adding this to the GA total gives:

Total required space: 502 + 244 = 746 bytes

Even taking the need for a file header in to consideration, this is less than half the size of the equivalent MIDI file.

The GP data can be compressed further. The data is stored as a comma-delimited character string. This means that, allowing for numbers of more than one digit, just under half of the 244 characters are commas and are therefore constitute redundancy in the string. Thus, if all the commas are removed and the numerical values stored as integers then there are further possible space savings.

This is the result for a piece that has been analysed by hand and, as such, is unlikely to be the most efficient coding. Techniques such as fractal compression and pattern matching could be employed to generate, automatically, the sequences of notes, figures and chords and the GP itself.

The representation is limited in that it currently describes only one melody part and one chord sequence. It would be possible to include more melody parts simply by adding more motive-style functions to the GP language. The chord and rhythm vocabulary of the system can also be extended easily; using `int` gives 65,535 possible values.

8 Final Conclusions

The aim of this project was to investigate ways in which evolutionary algorithms could be applied to the composition of Jazz music and, if feasible, implement a system that would perform this task.

The disciplines that the project combines, namely *Jazz*, *Evolutionary Computing Techniques* and *Algorithmic Composition* have been thoroughly researched and are presented in Section 3 of the report. A brief explanation of Standard MIDI Files is also given as these were used in the implementation of the composition system.

A design for the Jazz composition system has been presented in Section 4 of the report. The design started with the definition of an Abstract Model for a Jazz piece. A two-level genetic representation for the model, using both Genetic Algorithms and Genetic Programming techniques, was then proposed and demonstrated by encoding a real piece of music.

A two-level evolutionary composition system has been implemented in C++. The various algorithms and techniques used in the implementation of the system have been outlined in Section 5 of the report.

The initial implementation of the system was tested and the design was found to have a flaw in the way it treated crossover of Genetic Programs. This problem was overcome and the current system has been demonstrated to function properly in several separate test runs. The results of these tests have been analysed and presented in Section 6. The initial findings presented here are encouraging but are not conclusive.

From the initial test runs, the system has been shown to be capable of producing passable, if slightly avant garde, Jazz. It is possible, with greater population sizes and longer test runs, that the system might be able to produce more conventionally acceptable pieces. The major factor limiting the success of the system is the fact that a human judge is still required to evaluate the fitness function. This has made the evolutionary cycle of the system very slow. As a result of this, the subject of a machine fitness test and how it may be approached has been discussed as further work in Section 7.

Other areas of further work have been identified and discussed as potential starting points for future projects. Most are extensions to the system, such as the implementation of the mutation operator and the programming of a Windows interface for the system.

One additional, interesting area of possible future work that has come to light is the apparent ability of the genetic representation to store the information describing a piece more efficiently than its corresponding MIDI file. The last part of Section 7 discusses the potential use of the representation for music file compression showing, through a thought experiment, that it is an idea that merits further investigation.

9 Acknowledgements

I would like to thank:

Andy Tyrrell and Andy Hunt for their enthusiastic and helpful supervision of my project,

Sarah Moore and Julia Sheppard for performing the experiment,

Simon Taylor for proof reading my report,

Mark Slaymaker, Justen Hyde and Oliver Hancock for their help in providing valuable leads for my initial research,

and finally, the guys from the BioInspired Office for putting up with me for Six months.

10 References

[1] Allcock D. 1992 *Illustrating C* Cambridge University Press.

[2] Alpern, A. 1995. "Techniques for Algorithmic Composition of Music" Available on the World Wide Web at: http://hamp.hampshire.edu/~adaF92/algocomp/algocomp95.html

[3] Biles, J. A., 1994, "GenJam: A Genetic Algorithm for Generating Jazz Solos", *Proceedings of the 1994 International Computer Music Conference,* pp.131-137, San Fransisco: International Computer Music Association.

[4] Biles, J. A., 1995, "GenJam Populi: Training an IGA via audience-mediated performance", *Proceedings of the 1995 International Computer Music Conference,* pp.347-348, San Fransisco: International Computer Music Association.

[5] Biles, J. A., Anderson, P. G., and Loggi, L. W. 1996 "Neural network fitness functions for a musical GA" *Proceedings of the International ICSC Symposium on Intelligent Industrial Automation (IIA'96) and Soft Computing (SOCO'96),* pp.B39-B44). Reading, UK:ICSC Academic Press.

[6] Burton, A. R., and Vladimirova, T., "Generation of Musical Sequences with Genetic Techniques", *Computer Music Journal,* 23:4, pp.59-73.

[7] Beckert, D. "Algorithmic Composition" Available on the World Wide Web at: www.digitale-medien.de/beckert

[8] Chomsky, N 1957. *Syntactic Structures.* The Hague. Mouton Publishers.

[9] Cope, D. 1993. "Algorithmic Composition [re]Defined." *ICMC '93* pp. 23-5

[10] Coker, J., 1967, *Improvising Jazz,* Simon & Schuster, Inc.

[11] Collier, G. 1975, *Jazz: A Student's and Teacher's Guide.* Cambridge University Press

[12] Goldberg, D. E. 1989, *Genetic Algorithms in Search, Optimisation, and Machine Learning.* Addison-Wesley

[13] Green, N. P. O., Stout, G. W., and Taylor, D. J. 1984, *Biological Science 1: Organisms, Energy and Environment* Ed. R. Soper, Cambridge University Press.

[14] Hancock, O. 2000. *Music Technology: the state of the art* MSc report Dept. of Electronics, University of York.

[15] Holland, J. H., 1975, *Adaptation in Natural and Artificial Systems,* Ann Arbor, University of Michigan Press (Second Edition: MIT Press, 1992)

[16] Huron, D., 1992 "Design Principles in Computer-Based Music Representation", *Computer Representations and Models in Music,* Academic Press, pp. 5-39.

[17] Jacob, B. L., 1995 "Composing with Genetic Algorithms", *Proceedings of the International Computer Music Conference,* Banff Alberta.

[18] Jacob, B. L. "Algorithmic Composition as a Model of Creativity" Available on the World Wide Web at: http://www.eecs.umich.edu/~blj/algorithmic_composition/

[19] Johanson, B., and Poli, R., *GP-Music: An Interactive Genetic Programming System for Music Generation with Automated Fitness Raters,* " Available on the World Wide Web at: http://graphics.stanford.edu/~bjohanso/gp-music/

[20] Jones R. A., and Karp A., 1986 *Introducing Genetics* John Murray Publishers Ltd.

[21] Koza, J. R., 1992, *Genetic Programming, On the Programming of Computers By Means of Natural Selection,* Cambridge MA: MIT Press/Bradford Books

[22] Langdon, W. B., 1998, *Genetic Programming and Data Structures* Kluwer Academic Publishers.

[23] Lee, L. S., "The Perception of Metrical Structure: Experimental Evidence and a Model", *Representing Musical Structure,* Howell, P., West, R., and Cross, I. (Eds.), Academic Press, pp. 33-127

[24] Lerdahl, F., and Jackendoff, R. 1983, *A Generative Theory of Tonal Music,* Cambridge, Ma.: The MIT Press.

[25] Lerdahl, F., 1991, "Underlying Musical Schemata", *Representing Musical Structure,* Howell, P., West, R., and Cross, I. (Eds.), Academic Press, pp.273-290.

[26] Levy, S., 1992, *Artificial Life,* Penguin Books.

[27] Lones, M. 1998 *Evolutionary Genetic Models and the Line Labelling problem: A Review of the Literature* 3rd Year Project, Dept. Computer Science, University of York.

[28] McAlpine, K., Miranda, E., and Hoggar, S. 1999 "Making Music with Algorithms: a Case-Study System", *Computer Music Journal*, 23:2, pp. 19-30

[29] Michalewicz, Z, 1992, *Genetic Algorithms + Data Structures = Evolution Progams* Springer

[30] Milano, D. (Ed.) 1987, *Mind over MIDI* GPI Publications

[31] Schildt, H. 2000, *C/C++ Programmers Reference, Second Edition* Osborne, McGraw Hill

[32] Schuller G. 1968. *Early Jazz: Its roots and musical development* Oxford University Press.

[33] Spector, L., and Alpern, A., 1995, "Induction and Recapitulation of Deep Musical Structure", *Proceedings of the IJCAI-95 Workshop on Artificial Intelligence and Music.*

[34] Stearns, M. W. 1956. *The Story of Jazz* Oxford University Press

[35] Temperley, D., and Sleator, D., 1999 "Modelling Meter and Harmony: A Preference Rule Approach", *Computer Music Journal,* 23:1, pp. 10-27.

[36] Thywissen, K. 1999 "GeNotator: An environment for exploring the application of evolutionary techniques in computer-assisted composition." *Organised Sound* 4:2 pp. 127-33.

[37] *The MIDI file format,* 1998, Available on the World Wide Web at: http://ourworld.compuserve.com/homepages/mark_clay/midi.htm

[38] Todd, P. M., and Werner, G. M., 1998 "Frankensteinian Methods for Evolutionary Music Composition", *Musical networks:Parallel distributed perception and performance.* Griffith, N. and Todd, P. M. (Eds.), Cambridge, MA: MIT Press/Bradford Books

[39] Waschka II, R. 1999. "Avoiding the Fitness 'Bottleneck': Using Genetic Algorithms to Compose Orchestral Music" *ICMC 1999* pp. 201-3

[40] Xenakis, I. 1971 Formalized Music: Thought and Mathematics in Music, Pendragon Press

11 Appendices

11.1 A Guide to the CD

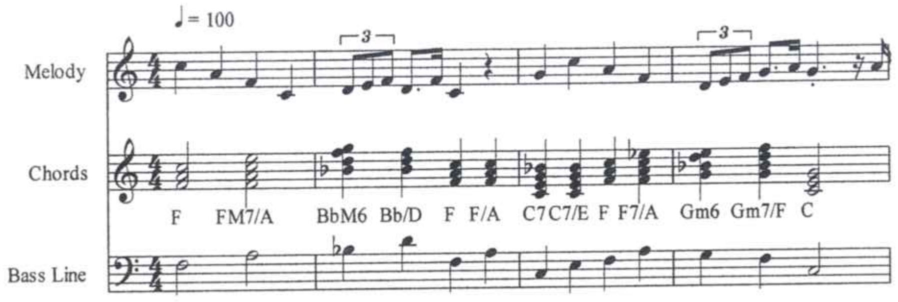
Please load Welcome.html

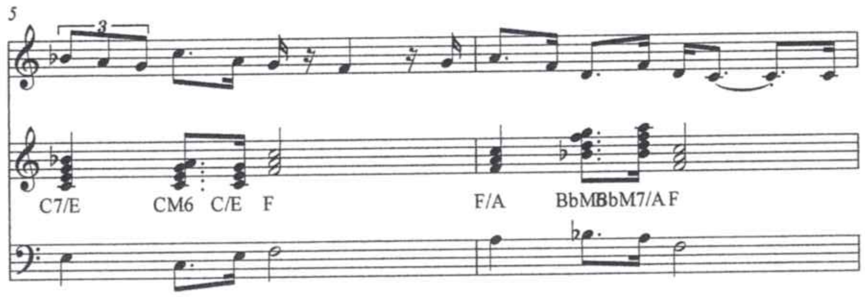
CD

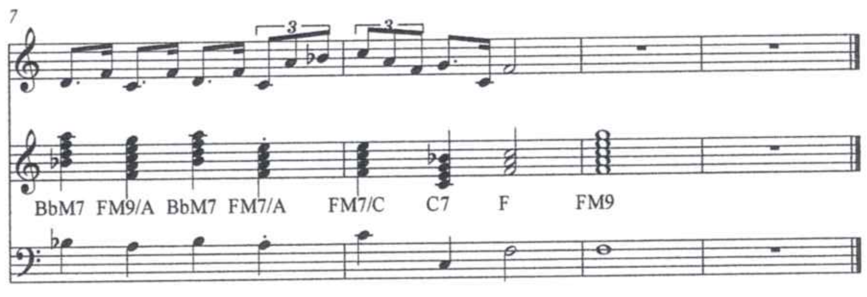
- **Examples**
  - Contains: Daisy examples and 'Nuyorican Soul' extract
- **Code**
  - Contains all code for the System
- **Results**
  - Contains results from the tests
- **Report**
  - Final Report Word Files
- **Other Documents**
  - Initial Report
    Pseudo Code and
    Spreadsheets

11.2 Test Score 'Daisy.mid'

The notation package that daisy.mid was loaded into shows the crotchet-quaver pairs as dotted-quaver-semiquaver pairs. The actual rhythms are correct as can be heard from the file on the CD.







11.3 Test File 'Daisy.txt'

```
Name:        daisy
□
P1:   parent1
□
P2:   parent2


□
Initial Values:
□


Tempo: 100
□
Transpose: 0
□
Dynamic: 60
□

□
Index Melody Rhythms:             Melody Pitches:
0      (0 , 4)            (67 , 0)
1      (0 , 4)            (72 , 0)
2      (0 , 4)            (69 , 0)
3      (0 , 4)            (65 , 0)
4      (1 , 4)            (60 , 0)
5      (1 , 4)            (62 , 0)
6      (3 , 4)            (64 , 0)
7      (1 , 4)            (65 , 0)
8      (10 , 4)           (62 , 0)
9      (0 , 8)            (65 , 0)
10     (2 , 4)            (60 , 0)
11     (1 , 4)            (65 , 0)
12     (0 , 4)            (62 , 0)
13     (0 , 4)            (64 , 0)
14     (0 , 4)            (65 , 0)
15     (0 , 4)            (67 , 0)
16     (0 , 4)            (69 , 0)
17     (0 , 4)            (60 , 0)
18     (0 , 4)            (72 , 1)
19     (0 , 4)            (60 , 0)
20     (0 , 4)            (62 , 0)
21     (0 , 1)            (65 , 0)
22     (0 , 1)            (62 , 0)
23     (0 , 1)            (65 , 0)
24     (0 , 1)            (69 , 0)
25     (0 , 1)            (62 , 0)
26     (0 , 1)            (60 , 0)
27     (0 , 1)            (62 , 0)
28     (0 , 1)            (61 , 0)
29     (0 , 1)            (59 , 0)
```

```
30    (0 , 1)          (64 , 0)
31    (0 , 1)          (61 , 0)
32    (0 , 1)          (60 , 0)
33    (0 , 1)          (69 , 0)
34    (0 , 1)          (70 , 0)
35    (0 , 1)          (67 , 0)
36    (0 , 1)          (72 , 1)
37    (0 , 1)          (69 , 0)
38    (0 , 1)          (72 , 0)
39    (0 , 1)          (69 , 0)
40    (0 , 1)          (65 , 0)
41    (0 , 1)          (67 , 0)
42    (0 , 1)          (60 , 0)
43    (0 , 1)          (60 , 0)
44    (0 , 1)          (0 , 0)
45    (0 , 1)          (0 , 0)
46    (0 , 1)          (0 , 0)
47    (0 , 1)          (0 , 0)
48    (0 , 1)          (0 , 0)
49    (0 , 1)          (0 , 0)
50    (0 , 1)          (0 , 0)
51    (0 , 1)          (0 , 0)
52    (0 , 1)          (0 , 0)
53    (0 , 1)          (0 , 0)
54    (0 , 1)          (0 , 0)
55    (0 , 1)          (0 , 0)
56    (0 , 1)          (0 , 0)
57    (0 , 1)          (0 , 0)
58    (0 , 1)          (0 , 0)
59    (0 , 1)          (0 , 0)
60    (0 , 1)          (0 , 0)
61    (0 , 1)          (0 , 0)
62    (0 , 1)          (0 , 0)
63    (0 , 1)          (0 , 0)
64    (0 , 1)          (0 , 0)
65    (0 , 1)          (0 , 0)
66    (0 , 1)          (0 , 0)
67    (0 , 1)          (0 , 0)
68    (0 , 1)          (0 , 0)
69    (0 , 1)          (0 , 0)
70    (0 , 1)          (0 , 0)
71    (0 , 1)          (0 , 0)
72    (0 , 1)          (0 , 0)
73    (0 , 1)          (0 , 0)
74    (0 , 1)          (0 , 0)
75    (0 , 1)          (0 , 0)
76    (0 , 1)          (0 , 0)
77    (0 , 1)          (0 , 0)
78    (0 , 1)          (0 , 0)
79    (0 , 1)          (0 , 0)
80    (0 , 1)          (0 , 0)
81    (0 , 1)          (0 , 0)
82    (0 , 1)          (0 , 0)
83    (0 , 1)          (0 , 0)
```

```
84     (0 , 1)              (0 , 0)
85     (0 , 1)              (0 , 0)
86     (0 , 1)              (0 , 0)
87     (0 , 1)              (0 , 0)
88     (0 , 1)              (0 , 0)
89     (0 , 1)              (0 , 0)
90     (0 , 1)              (0 , 0)
91     (0 , 1)              (0 , 0)
92     (0 , 1)              (0 , 0)
93     (0 , 1)              (0 , 0)
94     (0 , 1)              (0 , 0)
95     (0 , 1)              (0 , 0)
96     (0 , 1)              (0 , 0)
97     (0 , 1)              (0 , 0)
98     (0 , 1)              (0 , 0)
99     (0 , 1)              (0 , 0)

Index Harmonic Rhythms: Harmonic Progression:
0      (0 , 8)              (0 , 0 , 65 , 0 , 0)
1      (0 , 8)              (2 , 0 , 65 , 1 , 0)
2      (0 , 4)              (1 , 0 , 70 , 0 , 0)
3      (0 , 4)              (0 , 0 , 70 , 1 , 0)
4      (0 , 4)              (0 , 0 , 65 , 0 , 0)
5      (0 , 4)              (0 , 0 , 65 , 1 , 0)
6      (0 , 8)              (7 , 0 , 60 , 0 , 0)
7      (0 , 4)              (7 , 0 , 60 , 1 , 0)
8      (1 , 4)              (0 , 0 , 65 , 0 , 0)
9      (0 , 8)              (7 , 0 , 65 , 1 , 0)
10     (0 , 4)              (4 , 0 , 67 , 0 , 0)
11     (0 , 4)              (5 , 0 , 67 , 3 , 0)
12     (0 , 4)              (0 , 0 , 60 , 0 , 0)
13     (2 , 4)              (7 , 0 , 57 , 1 , 0)
14     (0 , 16)             (1 , 0 , 57 , 0 , 0)
15     (0 , 4)              (0 , 0 , 57 , 1 , 0)
16     (0 , 4)              (0 , 0 , 62 , 0 , 0)
17     (0 , 4)              (0 , 0 , 65 , 1 , 0)
18     (0 , 4)              (1 , 0 , 70 , 0 , 0)
19     (0 , 4)              (2 , 0 , 70 , 3 , 0)
20     (0 , 4)              (0 , 0 , 65 , 0 , 0)
21     (0 , 1)              (11 , 0 , 66 , 1 , 0)
22     (0 , 1)              (2 , 0 , 65 , 1 , 1)
23     (0 , 1)              (2 , 0 , 65 , 1 , 0)
24     (0 , 1)              (2 , 0 , 70 , 0 , 0)
25     (0 , 1)              (2 , 2 , 65 , 1 , 0)
26     (0 , 1)              (2 , 0 , 70 , 0 , 0)
27     (0 , 1)              (2 , 0 , 65 , 2 , 0)
28     (0 , 1)              (7 , 0 , 60 , 0 , 0)
29     (0 , 1)              (0 , 0 , 65 , 0 , 0)
30     (0 , 1)              (2 , 2 , 65 , 0 , 0)
31     (0 , 1)              (0 , 0 , 0 , 0 , 1)
32     (0 , 1)              (0 , 0 , 0 , 0 , 1)
33     (0 , 1)              (0 , 0 , 0 , 0 , 1)
34     (0 , 1)              (0 , 0 , 0 , 0 , 1)
35     (0 , 1)              (0 , 0 , 0 , 0 , 1)
```

```
36    (0 , 1)          (0 , 0 , 0 , 0 , 1)
37    (0 , 1)          (0 , 0 , 0 , 0 , 1)
38    (0 , 1)          (0 , 0 , 0 , 0 , 1)
39    (0 , 1)          (0 , 0 , 0 , 0 , 1)
40    (0 , 1)          (0 , 0 , 0 , 0 , 1)
41    (0 , 1)          (0 , 0 , 0 , 0 , 1)
42    (0 , 1)          (0 , 0 , 0 , 0 , 1)
43    (0 , 1)          (0 , 0 , 0 , 0 , 1)
44    (0 , 1)          (0 , 0 , 0 , 0 , 1)
45    (0 , 1)          (0 , 0 , 0 , 0 , 1)
46    (0 , 1)          (0 , 0 , 0 , 0 , 1)
47    (0 , 1)          (0 , 0 , 0 , 0 , 1)
48    (0 , 1)          (0 , 0 , 0 , 0 , 1)
49    (0 , 1)          (0 , 0 , 0 , 0 , 1)
50    (0 , 1)          (0 , 0 , 0 , 0 , 1)
51    (0 , 1)          (0 , 0 , 0 , 0 , 1)
52    (0 , 1)          (0 , 0 , 0 , 0 , 1)
53    (0 , 1)          (0 , 0 , 0 , 0 , 1)
54    (0 , 1)          (0 , 0 , 0 , 0 , 1)
55    (0 , 1)          (0 , 0 , 0 , 0 , 1)
56    (0 , 1)          (0 , 0 , 0 , 0 , 1)
57    (0 , 1)          (0 , 0 , 0 , 0 , 1)
58    (0 , 1)          (0 , 0 , 0 , 0 , 1)
59    (0 , 1)          (0 , 0 , 0 , 0 , 1)
60    (0 , 1)          (0 , 0 , 0 , 0 , 1)
61    (0 , 1)          (0 , 0 , 0 , 0 , 1)
62    (0 , 1)          (0 , 0 , 0 , 0 , 1)
63    (0 , 1)          (0 , 0 , 0 , 0 , 1)
64    (0 , 1)          (0 , 0 , 0 , 0 , 1)
65    (0 , 1)          (0 , 0 , 0 , 0 , 1)
66    (0 , 1)          (0 , 0 , 0 , 0 , 1)
67    (0 , 1)          (0 , 0 , 0 , 0 , 1)
68    (0 , 1)          (0 , 0 , 0 , 0 , 1)
69    (0 , 1)          (0 , 0 , 0 , 0 , 1)
70    (0 , 1)          (0 , 0 , 0 , 0 , 1)
71    (0 , 1)          (0 , 0 , 0 , 0 , 1)
72    (0 , 1)          (0 , 0 , 0 , 0 , 1)
73    (0 , 1)          (0 , 0 , 0 , 0 , 1)
74    (0 , 1)          (0 , 0 , 0 , 0 , 1)
75    (0 , 1)          (0 , 0 , 0 , 0 , 1)
76    (0 , 1)          (0 , 0 , 0 , 0 , 1)
77    (0 , 1)          (0 , 0 , 0 , 0 , 1)
78    (0 , 1)          (0 , 0 , 0 , 0 , 1)
79    (0 , 1)          (0 , 0 , 0 , 0 , 1)
80    (0 , 1)          (0 , 0 , 0 , 0 , 1)
81    (0 , 1)          (0 , 0 , 0 , 0 , 1)
82    (0 , 1)          (0 , 0 , 0 , 0 , 1)
83    (0 , 1)          (0 , 0 , 0 , 0 , 1)
84    (0 , 1)          (0 , 0 , 0 , 0 , 1)
85    (0 , 1)          (0 , 0 , 0 , 0 , 1)
86    (0 , 1)          (0 , 0 , 0 , 0 , 1)
87    (0 , 1)          (0 , 0 , 0 , 0 , 1)
88    (0 , 1)          (0 , 0 , 0 , 0 , 1)
89    (0 , 1)          (0 , 0 , 0 , 0 , 1)
```

```
90     (0 , 1)          (0 , 0 , 0 , 0 , 1)
91     (0 , 1)          (0 , 0 , 0 , 0 , 1)
92     (0 , 1)          (0 , 0 , 0 , 0 , 1)
93     (0 , 1)          (0 , 0 , 0 , 0 , 1)
94     (0 , 1)          (0 , 0 , 0 , 0 , 1)
95     (0 , 1)          (0 , 0 , 0 , 0 , 1)
96     (0 , 1)          (0 , 0 , 0 , 0 , 1)
97     (0 , 1)          (0 , 0 , 0 , 0 , 1)
98     (0 , 1)          (0 , 0 , 0 , 0 , 1)
99     (0 , 1)          (0 , 0 , 0 , 0 , 1)

Genetic Program:
H,32,0,0,D,100,M,16,0,1,M,8,6,5,M,4,3,42,H,16,2,6,M,16,0,0,H,16,4,10
,M,8,6,12,M,8,3,35,T,3,H,16,7,13,T,8,M,8,6,27,T,5,M,8,8,25,T,0,H,16,
7,17,T,0,R,M,8,4,21,R,M,8,10,17,R,H,16,10,22,M,8,4,8,M,4,7,8,M,4,6,3
2,H,16,4,27,M,8,6,38,M,8,9,40,H,16,14,30
```